PART III

Layers

In This Part:

- **Layers and Tiers**
- **Presentation Layer Guidelines**
- **Business Layer Guidelines**
- **Data Access Layer Guidelines**
- **Service Layer Guidelines**

Chapter 9: Layers and Tiers

Objectives

- Learn how to divide your applications into separate physical and logical parts.
- Learn the difference between logical layers and physical tiers.
- Learn about services that you can use to expose logic on layers.
- Learn about the components commonly encountered in layers and tiers.
- Learn about applications that support multiple client types.
- Learn how to choose an appropriate functional layout for your applications.

Overview

This chapter discusses the overall structure for applications, in terms of the logical grouping of components into separate layers or tiers that communicate with each other and with other clients and applications. Layers are concerned with the logical division of components and functionality, and take no account of the physical location of components on different servers or in different locations. Tiers are concerned with the physical distribution of components and functionality on separate servers, computers, networks, and remote locations. Although both layers and tiers use the same set of names (presentation, business, service, and data), remember that only tiers imply a physical separation. It is quite common to locate more than one layer on the same physical machine. You can think of the term “tier” as referring to physical distribution patterns such as two-tier, three-tier, and n-tier.

Layers

Layers are the logical groupings of the software components that make up the application or service. They help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sublayers, with each sublayer performing a specific type of task. By identifying the generic types of components that exist in most solutions, you can construct a meaningful map of an application or service, and then use this map as a blueprint for your design.

Splitting an application into separate layers that have distinct roles and functionalities helps you to maximize maintainability of the code, optimize the way that the application works when deployed in different ways, and provide a clear delineation between locations where certain technology or design decisions must be made.

Presentation, Business, and Data Services

At the highest and most abstract level, the logical architecture view of any system can be considered to be a set of cooperating services grouped into the following layers, as shown in Figure 1.

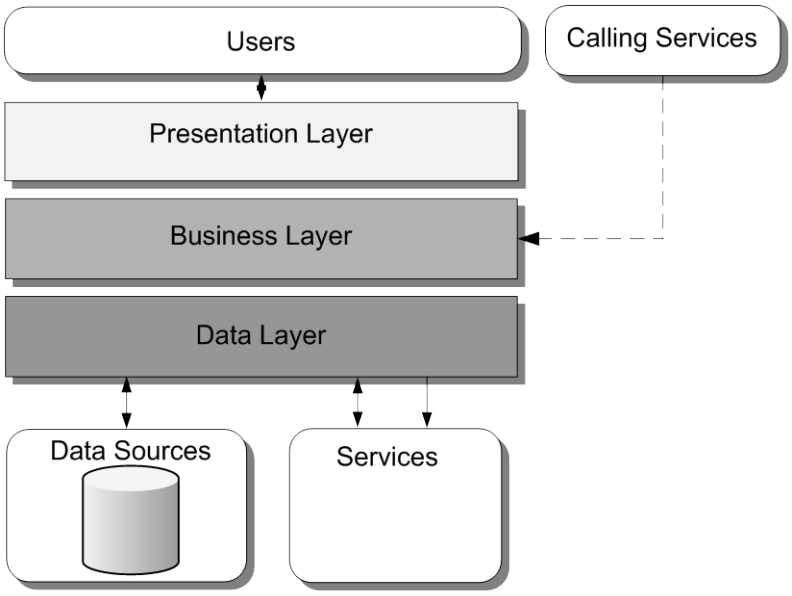


Figure 1 The logical architecture view of a layered system

The sections of the application design shown in Figure 1 can be thought of as three basic sets of services:

- **Presentation services**. These are the user-oriented services responsible for managing user interaction with the system, and generally consist of components located within the presentation layer. They provide a common bridge into the core business logic encapsulated in the business services.
- **Business services**. These services implement the core functionality of the system, and encapsulate the relevant business logic. They generally consist of components located within the business layer, which may expose service interfaces that other callers can use.
- **Data services**. These services provide access to data that is hosted within the boundaries of the system, and data exposed by other back-end systems; perhaps accessed through services. The data layer exposes data to the business layer through generic interfaces designed to be convenient for use by business services.

Components

Each layer of an application will contain a series of components that implement the functionality for that layer. These components should be cohesive and loosely coupled to simplify reuse and maintenance. Figure 2 shows the types of components commonly found in each layer.

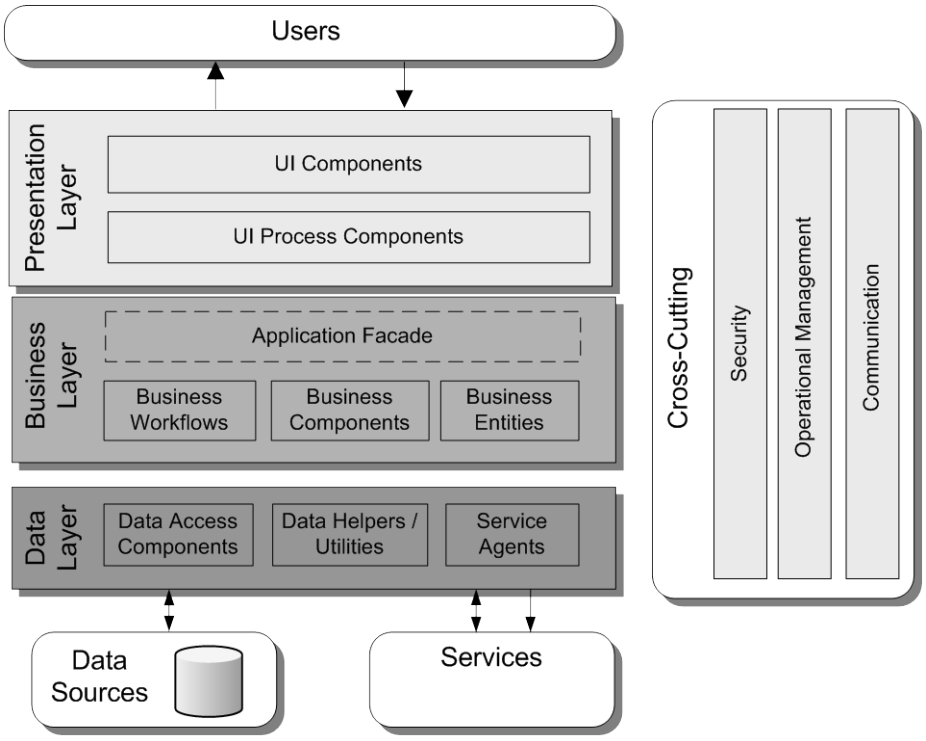


Figure 2 Types of components commonly found in each layer

The components shown in Figure 2 are described in the following sections.

Presentation Layer Components

Presentation layer components implement the functionality required to allow users to interact with the application. The following types of components are commonly found in the presentation layer:

- **User interface (UI) components**. These components provide the mechanism for users to interact with the application. They format data and render it for display, and acquire and validate data entered by users.
- **UI process components**. To help synchronize and orchestrate user interactions, it can be useful to drive the process using separate UI process components. This prevents the process flow and state management logic from being hard-coded into the UI elements themselves, and allows you to reuse the same basic user interaction patterns in other user interfaces.

Business Layer Components

Business layer components implement the core functionality of the system, and encapsulate the relevant business logic. The following types of components are commonly found in the business layer:

- **Application façade.** This is an optional feature that you can use to combine multiple business operations into single messaged-based operations. This feature is useful when you locate the presentation layer components on a separate physical tier from the business layer components, allowing you to optimize use of the communication method that connects them.

- **Business components**. These components implement the business logic of the application. Regardless of whether a business process consists of a single step or an orchestrated workflow, your application is likely to require components that implement business rules and perform business tasks.
- **Business workflows**. After the UI components collect the required data from the user and pass it to the business layer, the application can use this data to perform a business process. Many business processes involve multiple steps that must be performed in the correct order, and may interact with each other through an orchestration. Business workflow components define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.
- **Business entity components**. Business entities are used to pass data between components. The data represents real-world business entities, such as products or orders. The business entities that the application uses internally are usually data structures, such as DataSets, DataReaders, or Extensible Markup Language (XML) streams, but they can also be implemented using custom object-oriented classes that represent the real-world entities that your application will handle.

Data Layer Components

Data layer components provide access to data that is hosted within the boundaries of the system, and data exposed by other back-end systems. The following types of components are commonly found in the data layer:

- **Data access components**. These components abstract the logic required to access the underlying data stores. Doing so centralizes data access functionality and makes the application easier to configure and maintain.
- **Data helper and utility components**. Most data access tasks require common logic that can be extracted and implemented in separate reusable helper components. This helps to reduce the complexity of the data access components and centralizes the logic, which simplifies maintenance. Other tasks that are common across data layer components, and not specific to any set of components, may be implemented as separate utility components. Both helper and utility components can often be reused in other applications.
- **Service agents**. When a business component must use functionality provided by an external service, you might need to implement code to manage the semantics of communicating with that particular service. Service agents isolate the idiosyncrasies of calling diverse services from your application, and can provide additional services such as basic mapping between the format of the data exposed by the service and the format your application requires.

Cross-Cutting Components

Many tasks carried out by the code of an application are required in more than one layer. Cross-cutting components implement specific types of functionality that can be accessed from components in any layer. The following are common types of cross-cutting components:

- **Components for implementing security**. These may include components that perform authentication, authorization, and validation.

- **Components for implementing operational management tasks**. These may include components that implement exception handling policies, logging, performance counters, configuration, and tracing.
- **Components for implementing communication**. These may include components that communicate with other services and applications.

Services and Layers

From a high-level perspective, a service-based solution can be seen as being composed of multiple services, each communicating with the others by passing messages. Conceptually, the services can be seen as components of the overall solution. However, internally, each service is made up of software components, just like any other application, and these components can be logically grouped into presentation, business, and data services. Other applications can make use of the services without being aware of the way they are implemented. The principles discussed in the previous sections on the layers and components of an application apply equally to service-based solutions.

Services Layer

When an application will act as the provider of services to other applications, as well as implementing features to support clients directly, a common approach is to use a services layer that exposes the functionality of the application, as shown in Figure 3.

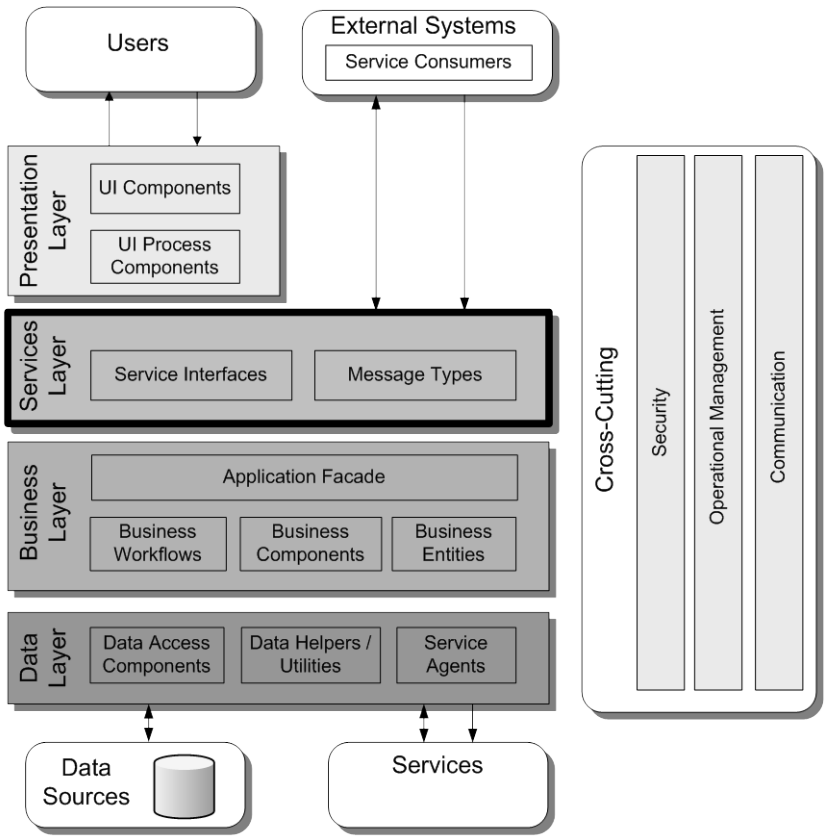


Figure 3 Incorporating a services layer in an application

The following section describes the components usually found in the services layer.

Services Layer Components

Services layer components provide other clients and applications with a way to access business logic in the application, and make use of the functionality of the application by passing messages to and from it over a communication channel. The following types of components are commonly found in the services layer:

- **Service interfaces**. Services expose a service interface to which all inbound messages are sent. The definition of the set of messages that must be exchanged with a service in order for the service to perform a specific business task constitutes a contract. You can think of a service interface as a façade that exposes the business logic implemented in the service to potential consumers.
- **Message types**. When exchanging data across the service layer, data structures are wrapped by message structures that support different types of operations. For example, you might have a Command message, a Document message, or another type of message. These message types are the "message contracts" for communication between service consumers and providers.

Multi-Client Application Scenario

Applications often must support different types of clients. In this scenario, the application will usually expose services to external systems, as well as directly supporting local clients, as shown in Figure 4.

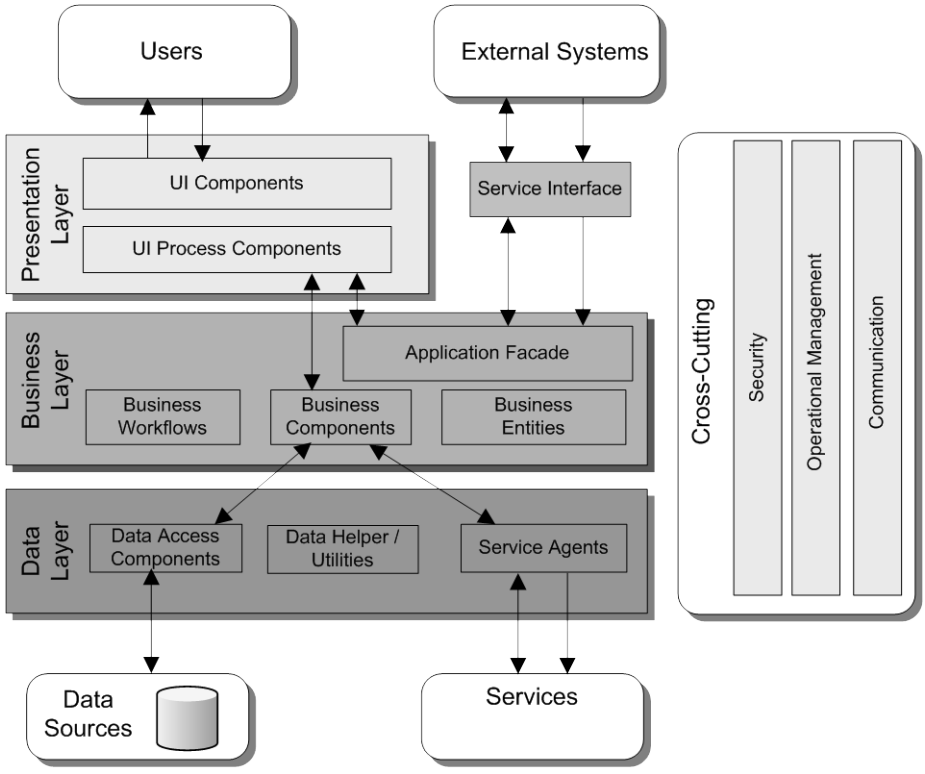


Figure 4 The multi-client application scenario

In this scenario, local and known client types can access the application through the presentation layer, which communicates either directly to the components in the business layer or through an application façade in the business layer if the communication methods require composition of functionality. Meanwhile, external clients and other systems can treat the application as an "application server" and make use of its functionality by communicating with the business layer through service interfaces.

Business Entities Used by Data and Business Services

There are many cases where business entities must be accessible to components and services in both the business layer and the data layer. For example, business entities can be mapped to the data source and accessed by business components. However, you should still separate business logic from data access logic. You can achieve this by moving business entities into a separate assembly that can be shared by both the business services and data services assemblies, as shown in Figure 5. This is similar to using a dependency inversion pattern, where business entities are decoupled from the business and data layer so that both business and data layers are dependent on business entities as a shared contract.

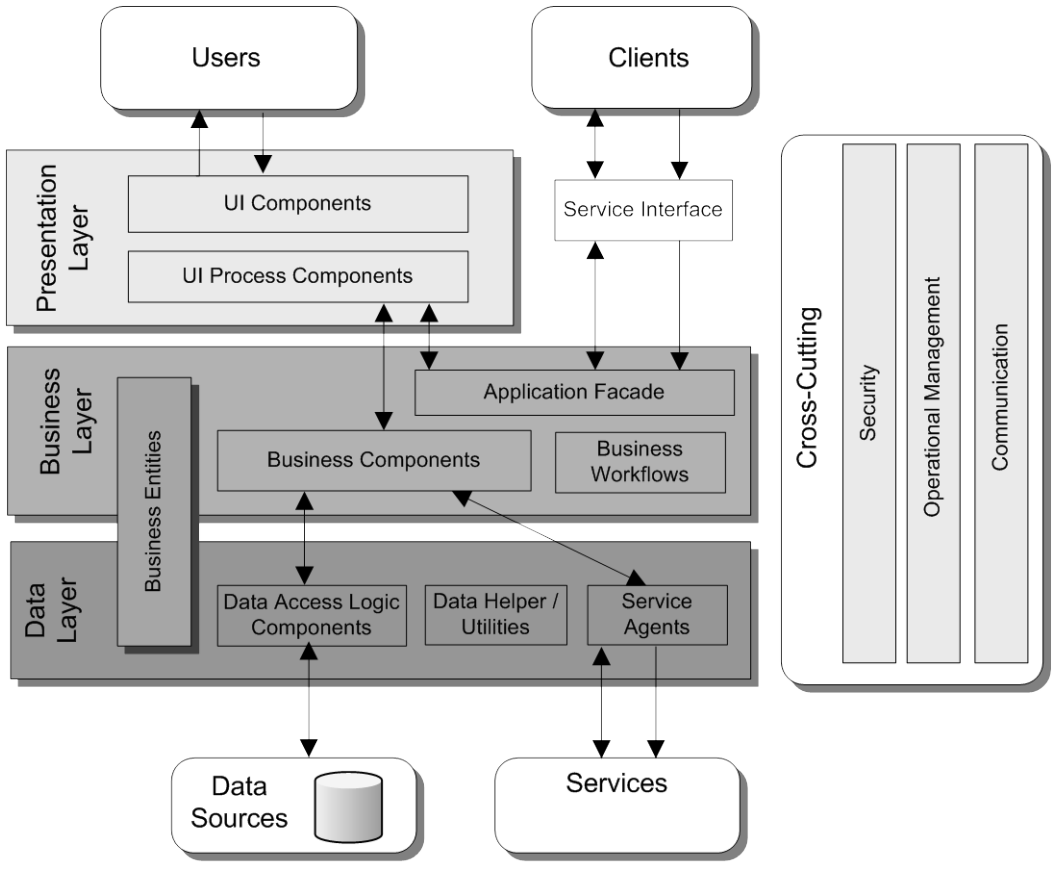


Figure 5 Business entities used by data and business services

Choosing Layers for Your Application

Use a layered approach to improve the maintainability of your application and make it easier to scale out when necessary to improve performance. Keep in mind that a layered approach adds complexity and can impact your initial development time. Be smart about adding layers, and don't add them if you don't need them. Use the following guidelines to help you decide on the layering requirements for your application:

- If your application does not expose a UI, such as a service application, you do not require a presentation layer.
- If your application does not contain business logic, you may not require a business layer.
- If your application does not expose services, you do not require a services layer.
- If your application does not access data, you do not require a data layer.
- Only distribute components where this is necessary. Common reasons for implementing distributed deployment include security policies, physical constraints, shared business logic, and scalability.
- In Web applications, deploy business components that are used synchronously by user interfaces or user process components in the same physical tier as the user interface to maximize performance and ease operational management, unless there are security implications that require a trust boundary between them.
- In rich client applications where the UI processing occurs on the desktop, you may prefer to deploy components that are used synchronously by UIs or user process components in a separate business tier for security reasons, and to ease operational management.
- Deploy service agent components on the same tier as the code that calls the components, unless there are security implications that require a trust boundary between them.
- Deploy asynchronous business components, workflow components, and business services on a separate physical tier where possible.
- Deploy business entities on the same physical tier as the code that uses them.

Tiers

Tiers represent the physical separation of the presentation, business, services, and data functionality of your design across separate computers and systems. Common tiered design patterns are two-tier, three-tier, and n-tier. The following sections explore each of these scenarios.

Two-Tier

The two-tier pattern represents a basic structure with two main components, a client and a server. In this scenario, the client and server may exist on the same machine, or may be located on two different machines. Figure 6 illustrates a common Web application scenario where the client interacts with a Web server located in the client tier. This tier contains the presentation layer logic and any required business layer logic. The Web application communicates with a separate machine that hosts the database tier, which contains the data layer logic.

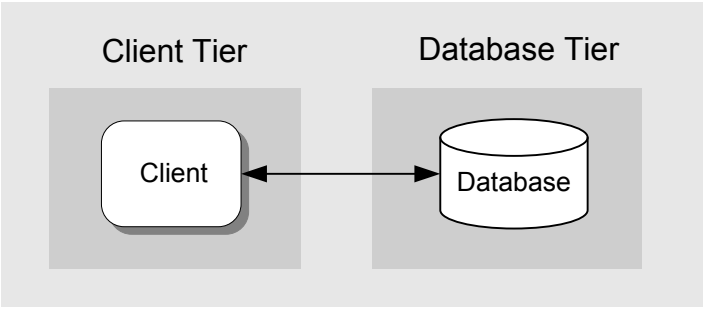


Figure 6 The two-tier deployment pattern

Three-Tier

In a three-tier design, the client interacts with application software deployed on a separate server, and the application server interacts with a database that is also located on a separate server. This is a very common pattern for most Web applications and Web services. Figure 7 illustrates the three-tier deployment pattern.

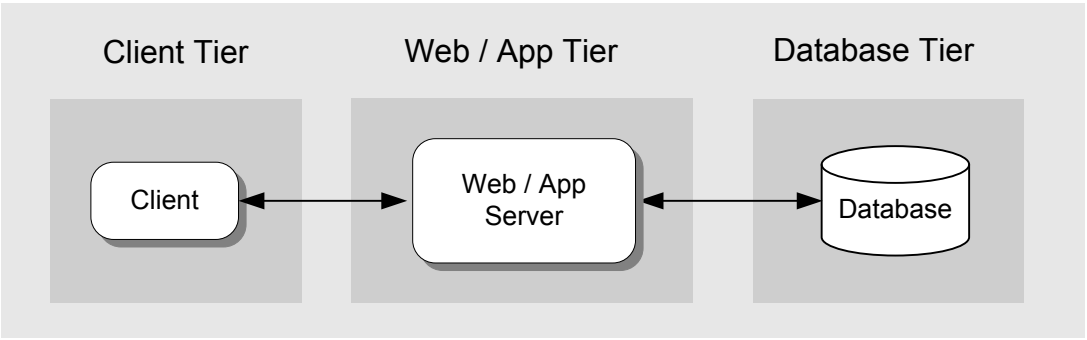


Figure 7 The three-tier deployment pattern

N-Tier

In this scenario, the Web server (which contains the presentation layer logic) is physically separated from the application server that implements the business logic. This usually occurs for security reasons, where the Web server is deployed within a perimeter network and accesses the application server located on a different subnet through a firewall. It is also common to implement a firewall between the client and the Web tier. Figure 8 illustrates the n-tier deployment pattern.

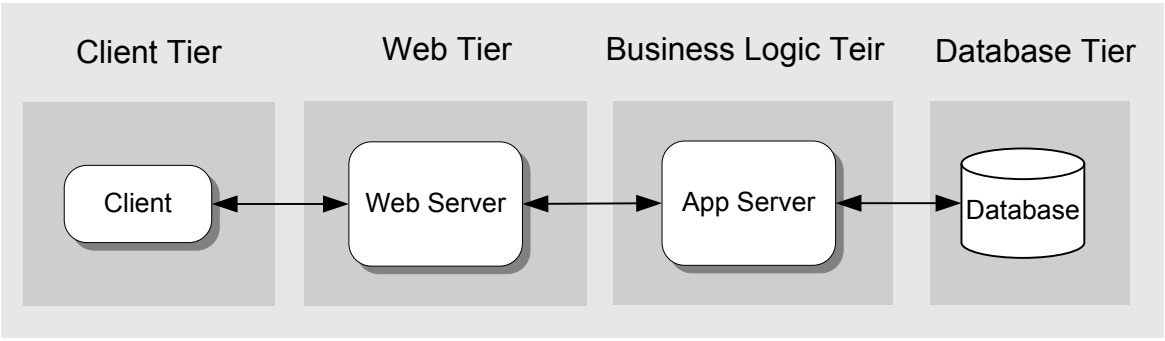


Figure 8 The n-tier deployment pattern

Choosing Tiers for Your Application

Placing your layers on separate physical tiers can help performance by distributing the load across multiple servers. It can also help with security by segregating more sensitive components and layers onto different networks or on the Internet versus an intranet. Keep in mind that adding tiers increases the complexity of your deployment, so don't add more tiers than you need.

In most cases, you should locate all of the application code on the same server, using a single-tier approach. Whenever communications must cross physical boundaries, performance is affected because the data must be serialized. However, in some cases you might need to split functionality across servers, because of security or organizational constraints. To mitigate the effects of serialization, depending on where servers are located, you can usually choose communication protocols that are optimized for performance.

Consider the client/server or two-tier pattern if:

- You are developing a client that must access an application server.
- You are developing a stand-alone client that accesses an external database.

Consider the three-tier pattern if:

- You are developing an intranet-based application, where all servers are located within a private network.
- You are developing an Internet-based application, and security requirements do not restrict implementation of business logic on the public-facing Web or application server.

Consider the N-tier pattern if:

- Security requirements dictate that business logic cannot be deployed to the perimeter network.
- You have application code that makes heavy use of resources on the server, and you want to offload that functionality to another server.

Chapter 10: Presentation Layer Guidelines

Objectives

- Understand how the presentation layer fits into typical application architecture.
- Understand the components of the presentation layer.
- Learn the steps for designing the presentation layer.
- Learn the common issues faced while designing the presentation layer.
- Learn the key guidelines for designing the presentation layer.
- Learn the key patterns and technology considerations for designing the presentation layer.

Overview

The presentation layer contains the components that implement and display the user interface and manage user interaction. This layer includes controls for user input and display, in addition to components that organize user interaction. Figure 1 shows how the presentation layer fits into a common application architecture.

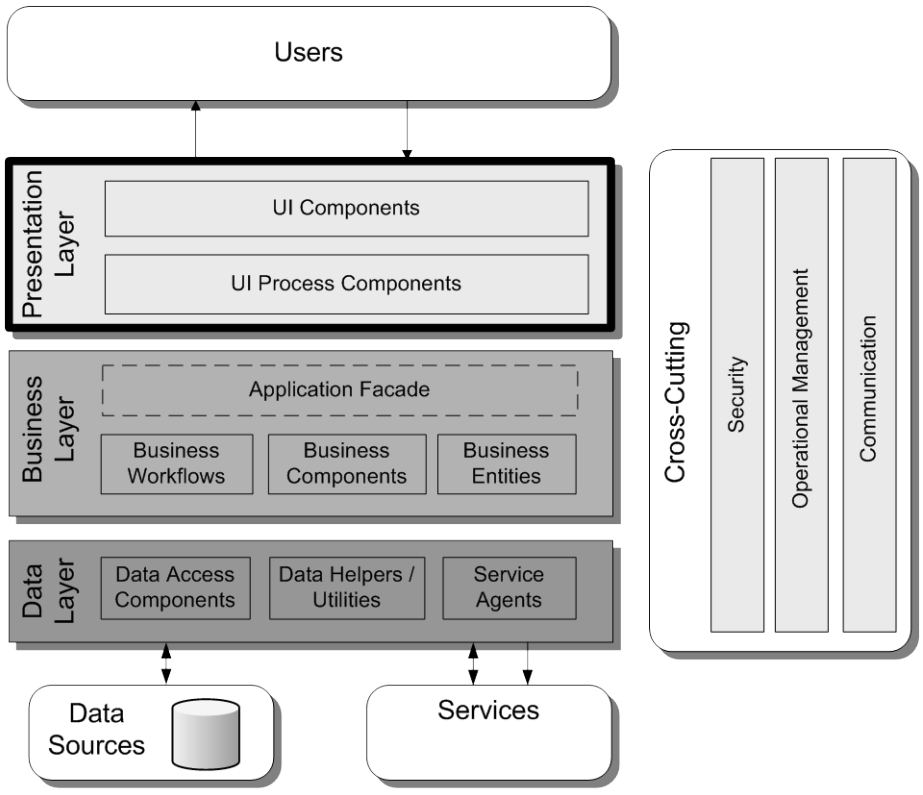


Figure 1 A typical application showing the presentation layer and the components it may contain

Presentation Layer Components

- **User interface (UI) components**. User interface components provide a way for users to interact with the application. They render and format data for users. They also acquire and validate data input by the user.
- **User process components**. User process components synchronize and orchestrate user interactions. Separate user process components may be useful if you have a complicated UI. Implementing common user interaction patterns as separate user process components allows you to reuse them in multiple UIs.

Approach

The following steps describe the process you should adopt when designing the presentation layer for your application. This approach will ensure that you consider all of the relevant factors as you develop your architecture:

1. **Identify your client type**. Choose a client type that satisfies your requirements and adheres to the infrastructure and deployment constraints of your organization. For instance, if your users are on mobile devices and will be intermittently connected to the network, a mobile rich client is probably your best choice.
2. **Determine how you will present data**. Choose the data format for your presentation layer and decide how you will present the data in your UI.
3. **Determine your data-validation strategy**. Use data-validation techniques to protect your system from untrusted input.
4. **Determine your business logic strategy**. Factor out your business logic to decouple it from your presentation layer code.
5. **Determine your strategy for communication with other layers**. If your application has multiple layers, such as a data access layer and a business layer, determine a strategy for communication between your presentation layer and other layers.

Design Considerations

There are several key factors that you should consider when designing your presentation layer. Use the following principles to ensure that your design meets the requirements for your application, and follows best practices:

- **Choose the appropriate UI technology.** Determine if you will implement a rich (smart) client, a Web client, or a rich Internet application (RIA). Base your decision on application requirements, and on organizational and infrastructure constraints.
- **Use the relevant patterns.** Review the presentation layer patterns for proven solutions to common presentation problems.
- **Design for separation of concerns.** Use dedicated UI components that focus on rendering and display. Use dedicated presentation entities to manage the data required to present your views. Use dedicated UI process components to manage the processing of user interaction.

- **Consider human interface guidelines.** Review your organization's guidelines for UI design. Review established UI guidelines based on the client type and technologies that you have chosen.
- **Adhere to user-driven design principles.** Before designing your presentation layer, understand your customer. Use surveys, usability studies, and interviews to determine the best presentation design to meet your customer's requirements.

Presentation Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 Presentation Layer Frame

Category	Common issues
Caching	• Caching volatile data. • Caching unencrypted sensitive data. • Incorrect choice of caching store. • Failing to choose a suitable caching mechanism for use in a Web farm. • Assuming that data will still be available in the cache – it may have expired and been removed.
Composition	• Failing to consider use of patterns and libraries that support dynamic layout and injection of views and presentation at runtime. • Using presentation components that have dependencies on support classes and services instead of considering patterns that support run-time dependency injection. • Failing to use the Publish/Subscribe pattern to support events between components. • Failing to properly decouple the application as separate modules that can be added easily.
Exception Management	• Failing to catch unhandled exceptions. • Failing to clean up resources and state after an exception occurs. • Revealing sensitive information to the end user. • Using exceptions to control application flow. • Catching exceptions you do not handle. • Using custom exceptions when not necessary.
Input	• Failing to design for intuitive use, or implementing overly complex interfaces. • Failing to design for accessibility. • Failing to design for different screen sizes and resolutions. • Failing to design for different device and input types, such as mobile devices, touch-screen, and pen and ink–enabled devices.

Category	Common issues
Layout	• Using an inappropriate layout style for Web pages. • Implementing an overly complex layout. • Failing to choose appropriate layout components and technologies. • Failing to adhere to accessibility and usability guidelines and standards. • Implementing an inappropriate workflow interface. • Failing to support localization and globalization.
Navigation	• Inconsistent navigation. • Duplication of logic to handle navigation events. • Using hard-coded navigation. • Failing to manage state with wizard navigation.
Presentation Entities	• Defining entities that are not necessary. • Failing to implement serialization when necessary.
Request Processing	• Blocking the UI during long-running requests. • Mixing processing and rendering logic. • Choosing an inappropriate request-handling pattern.
User Experience	• Displaying unhelpful error messages. • Lack of responsiveness. • Overly complex user interfaces. • Lack of user personalization. • Lack of user empowerment. • Designing inefficient user interfaces.
UI Components	• Creating custom components that are not necessary. • Failing to maintain state in the Model-View-Controller (MVC) pattern. • Choosing inappropriate UI components.
UI Process Components	• Implementing UI process components when not necessary. • Implementing the wrong design patterns. • Mixing business logic with UI process logic. • Mixing rendering logic with UI process logic.
Validation	• Failing to validate all input. • Failure to validate on the server for security concerns. • Failing to correctly handle validation errors. • Not identifying business rules that are appropriate for validation. • Failing to log validation failures.

Caching

Caching is one of the best mechanisms you can use to improve application performance and UI responsiveness. Use data caching to optimize data lookups and avoid network round trips. Cache the results of expensive or repetitive processes to avoid unnecessary duplicate processing.

Consider the following guidelines when designing your caching strategy:

- Do not cache volatile data.
- Consider using ready-to-use cache data when working with an in-memory cache. For example, use a specific object instead of caching raw database data.
- Do not cache sensitive data unless you encrypt it.
- If your application is deployed in Web farm, avoid using local caches that need to be synchronized; instead, consider using a transactional resource manager such as Microsoft SQL Server® or a product that supports distributed caching.
- Do not depend on data still being in your cache. It may have been removed.

Composition

Consider whether your application will be easier to develop and maintain if the presentation layer uses independent modules and views that are easily composed at run time. Composition patterns support the creation of views and the presentation layout at run time. These patterns also help to minimize code and library dependencies that would otherwise force recompilation and redeployment of a module when the dependencies change. Composition patterns help you to implement sharing, reuse, and replacement of presentation logic and views.

Consider the following guidelines when designing your composition strategy:

- Avoid using dynamic layouts. They can be difficult to load and maintain.
- Be careful with dependencies between components. For example, use abstraction patterns when possible to avoid issues with maintainability.
- Consider creating templates with placeholders. For example, use the Template View pattern to compose dynamic Web pages in order to ensure reuse and consistency.
- Consider composing views from reusable modular parts. For example, use the Composite View pattern to build a view from modular, atomic component parts.
- If you need to allow communication between presentation components, consider implementing the Publish/Subscribe pattern. This will lower the coupling between the components and improve testability.

Exception Management

Design a centralized exception-management mechanism for your application that catches and throws exceptions consistently. Pay particular attention to exceptions that propagate across layer or tier boundaries, as well as exceptions that cross trust boundaries. Design for unhandled exceptions so they do not impact application reliability or expose sensitive information.

Consider the following guidelines when designing your exception management strategy:

- Use user-friendly error messages to notify users of errors in the application.
- Avoid exposing sensitive data in error pages, error messages, log files, and audit files.
- Design a global exception handler that displays a global error page or an error message for all unhandled exceptions.
- Differentiate between system exceptions and business errors. In the case of business errors, display a user-friendly error message and allow the user to retry the operation. In the case

of system exceptions, check to see if the exception was caused by issues such as system or database failure, display a user-friendly error message, and log the error message, which will help in troubleshooting.
- Avoid using exceptions to control application logic.

Input

Design a user input strategy based on your application input requirements. For maximum usability, follow the established guidelines defined in your organization, and the many established industry usability guidelines based on years of user research into input design and mechanisms.

Consider the following guidelines when designing your input collection strategy:
- Use forms-based input controls for normal data-collection tasks.
- Use a document-based input mechanism for collecting input in Microsoft Office–style documents.
- Implement a wizard-based approach for more complex data collection tasks, or for input that requires a workflow.
- Design to support localization by avoiding hard-coded strings and using external resources for text and layout.
- Consider accessibility in your design. You should consider users with disabilities when designing your input strategy; for example, implement text-to-speech software for blind users, or enlarge text and images for users with poor sight. Support keyboard-only scenarios where possible for users who cannot manipulate a pointing device.

Layout

Design your UI layout so that the layout mechanism itself is separate from the individual UI components and UI process components. When choosing a layout strategy, consider whether you will have a separate team of designers building the layout, or whether the development team will create the UI. If designers will be creating the UI, choose a layout approach that does not require code or the use of development-focused tools.

Consider the following guidelines when designing your layout strategy:
- Use templates to provide a common look and feel to all of the UI screens.
- Use a common look and feel for all elements of your UI to maximize accessibility and ease of use.
- Consider device-dependent input, such as touch screens, ink, or speech, in your layout. For example, with touch-screen input you will typically use larger buttons with more spacing between them than you would with mouse or keyboard inputs.
- When building a Web application, consider using Cascading Style Sheets (CSS) for layout. This will improve rendering performance and maintainability.
- Use design patterns, such as Model-View-Presenter (MVP), to separate the layout design from interface processing.

Navigation

Design your navigation strategy so that users can navigate easily through your screens or pages, and so that you can separate navigation from presentation and UI processing. Ensure that you display navigation links and controls in a consistent way throughout your application to reduce user confusion and hide application complexity.

Consider the following guidelines when designing your navigation strategy:

- Use well-known design patterns to decouple the UI from the navigation logic where this logic is complex.
- Design toolbars and menus to help users find functionality provided by the UI.
- Consider using wizards to implement navigation between forms in a predictable way.
- Determine how you will preserve navigation state if the application must preserve this state between sessions.
- Consider using the Command Pattern to handle common actions from multiple sources.

Presentation Entities

Use presentation entities to store the data you will use in your presentation layer to manage your views. Presentation entities are not always necessary; use them only if your datasets are sufficiently large and complex to require separate storage from the UI controls.

Consider the following guidelines when designing presentation entities:

- Determine if you require presentation entities. Typically, you may require presentation entities only if the data or the format to be displayed is specific to the presentation layer.
- If you are working with data-bound controls, consider using custom objects, collections, or datasets as your presentation entity format.
- If you want to map data directly to business entities, use a custom class for your presentation entities.
- Do not add business logic to presentation entities.
- If you need to perform data type validation, consider adding it in your presentation entities.

Request Processing

Design your request processing with user responsiveness in mind, as well as code maintainability and testability.

Consider the following guidelines when designing request processing:

- Use asynchronous operations or worker threads to avoid blocking the UI for long-running actions.
- Avoid mixing your UI processing and rendering logic.
- Consider using the Passive View pattern (a variant of MVP) for interfaces that do not manage a lot of data.
- Consider using the Supervising Controller pattern (a variant of MVP) for interfaces that manage large amounts of data.

User Experience

Good user experience can make the difference between a usable and unusable application. Carry out usability studies, surveys, and interviews to understand what users require and expect from your application, and design with these results in mind.

Consider the following guidelines when designing for user experience:

- When developing a rich Internet application (RIA), avoid synchronous processing where possible.
- When developing a Web application, consider using Asynchronous JavaScript and XML (AJAX) to improve responsiveness and to reduce post backs and page reloads.
- Do not design overloaded or overly complex interfaces. Provide a clear path through the application for each key user scenario.
- Design to support user personalization, localization, and accessibility.
- Design for user empowerment. Allow the user to control how he or she interacts with the application, and how it displays data to them.

UI Components

UI components are the controls and components used to display information to the user and accept user input. Be careful not to create custom controls unless it is necessary for specialized display or data collection.

Consider the following guidelines when designing UI components:

- Take advantage of the data-binding features of the controls you use in the UI.
- Create custom controls or use third-party controls only for specialized display and data-collection tasks.
- When creating custom controls, extend existing controls if possible instead of creating a new control.
- Consider implementing designer support for custom controls to make it easier to develop with them.
- Consider maintaining the state of controls as the user interacts with the application instead of reloading controls with each action.

UI Process Components

UI process components synchronize and orchestrate user interactions. UI processing components are not always necessary; create them only if you need to perform significant processing in the presentation layer that must be separated from the UI controls. Be careful not to mix business and display logic within the process components; they should be focused on organizing user interactions with your UI.

Consider the following guidelines when designing UI processing components:

- Do not create UI process components unless you need them.

- If your UI requires complex processing or needs to talk to other layers, use UI process components to decouple this processing from the UI.
- Consider dividing UI processing into three distinct roles: Model, View, and Controller/Presenter, by using the MVC or MVP pattern.
- Avoid business rules, with the exception of input and data validation, in UI processing components.
- Consider using abstraction patterns, such as dependency inversion, when UI processing behavior needs to change based on the run-time environment.
- Where the UI requires complex workflow support, create separate workflow components that use a workflow system such as Windows Workflow or a custom mechanism.

Validation

Designing an effective input and data-validation strategy is critical to the security of your application. Determine the validation rules for user input as well as for business rules that exist in the presentation layer.

Consider the following guidelines when designing your input and data validation strategy:

- Validate all input data on the client side where possible to improve interactivity and reduce errors caused by invalid data.
- Do not rely on client-side validation only. Always use server-side validation to constrain input for security purposes and to make security-related decisions.
- Design your validation strategy to constrain, reject, and sanitize malicious input.
- Use the built-in validation controls where possible, when working with .NET Framework.
- In Web applications, consider using AJAX to provide real-time validation.

Pattern Map

Key patterns are organized by key categories, as detailed in the Presentation Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern Map

Category	Relevant patterns
Caching	• Cache Dependency • Page Cache
Composition	• Composite View • Transform View • Two-step View
Exception Management	• Exception Shielding
Layout	• Template View
Navigation	• Front Controller • Page Controller • Command Pattern
Presentation Entities	• Entity Translator

Category	Relevant patterns
User Experience	• Asynchronous Callback • Chain of Responsibility
UI Processing Components	• Model-View-Controller (MVC) • Passive View • Presentation Model • Supervising Controller

- For more information on the *Page Cache* pattern, see "Enterprise Solution Patterns Using Microsoft .NET" at http://msdn.microsoft.com/en-us/library/ms998469.aspx
- For more information on the Model-View-Controller (MVC), Page Controller, Front Controller, Template View, Transform View, and Two-Step View patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on the Composite View, Supervising Controller, and Presentation Model patterns, see "Patterns in the Composite Application Library" at http://msdn.microsoft.com/en-us/library/cc707841.aspx
- For more information on the Chain of responsibility and Command pattern, see "data & object factory" at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Asynchronous Callback pattern, see "Creating a Simplified Asynchronous Call Pattern for Windows Forms Applications" at http://msdn.microsoft.com/en-us/library/ms996483.aspx
- For more information on the Exception Shielding and Entity Translator patterns, see "Useful Patterns for Services" at http://msdn.microsoft.com/en-us/library/cc304800.aspx

Pattern Descriptions

- **Asynchronous Callback.** Execute long-running tasks on a separate thread that executes in the background, and provide a function for the thread to call back into when the task is complete.
- **Cache Dependency.** Use external information to determine the state of data stored in a cache.
- **Chain of Responsibility.** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- **Composite View**. Combine individual views into a composite representation.
- **Command Pattern.** Encapsulate request processing in a separate command object with a common execution interface.
- **Entity Translator.** An object that transforms message data types into business types for requests, and reverses the transformation for responses.
- **Exception Shielding.** Prevent a service from exposing information about its internal implementation when an exception occurs.
- **Front Controller**. Consolidate request handling by channeling all requests through a single handler object, which can be modified at run time with decorators.
- **Model-View-Controller**. Separate the UI code into three separate units: Model (data), View (interface), and Presenter (processing logic), with a focus on the View. Two variations on

this pattern include Passive View and Supervising Controller, which define how the View interacts with the Model.
- **Page Cache.** Improve the response time for dynamic Web pages that are accessed frequently but change less often and consume a large amount of system resources to construct.
- **Page Controller**. Accept input from the request and handle it for a specific page or action on a Web site.
- **Passive View**. Reduce the view to the absolute minimum by allowing the controller to process user input and maintain the responsibility for updating the view.
- **Presentation Model**. Move all view logic and state out of the view, and render the view through data-binding and templates.
- **Supervising Controller**. A variation of the MVC pattern in which the controller handles complex logic, in particular coordinating between views, but the view is responsible for simple view-specific logic.
- **Template View**. Implement a common template view, and derive or construct views using this template view.
- **Transform View**. Transform the data passed to the presentation tier into HTML for display in the UI.
- **Two-Step View**. Transform the model data into a logical presentation without any specific formatting, and then convert that logical presentation to add the actual formatting required.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology. The guidelines also contain suggestions for common patterns that are useful for specific types of application and technology.

Mobile Applications

Consider the following guidelines when designing a mobile application:
- If you want to build full-featured connected, occasionally connected, and disconnected executable applications that run on a wide range of Microsoft Windows®–based devices, consider using the Microsoft Windows Compact Framework.
- If you want to build connected applications that require Wireless Application Protocol (WAP), compact HTML (cHTML), or similar rendering formats, consider using ASP.NET Mobile Forms and Mobile Controls.
- If you want to build applications that support rich media and interactivity, consider using Microsoft Silverlight® for Mobile.

Rich Client Applications

Consider the following guidelines when designing a rich client application:
- If you want to build applications with good performance and interactivity, and have design support in Microsoft Visual Studio®, consider using Windows Forms.

- If you want to build applications that fully support rich media and graphics, consider using Windows Presentation Foundation (WPF).
- If you want to build applications that are downloaded from a Web server and then execute on the client, consider using XAML Browser Applications (XBAP).
- If you want to build applications that are predominantly document-based, or are used for reporting, consider designing a Microsoft Office Business Application.
- If you decide to use Windows Forms and you are designing composite interfaces, consider using the Smart Client Software Factory.
- If you decide to use WPF and you are designing composite interfaces, consider using the Composite Application Guidance for WPF.
- If you decide to use WPF, consider using the Presentation Model (Model-View-ViewModel) pattern.
- If you decide to use WPF, consider using WPF Commands to communicate between your View and your Presenter or ViewModel.
- If you decide to use WPF, consider implementing the Presentation Model pattern by using DataTemplates over User Controls to give designers more control.

Rich Internet Applications (RIA)

Consider the following guidelines when designing an RIA:

- If you want to build browser-based, connected applications that have broad cross-platform reach, are highly graphical, and support rich media and presentation features, consider using Silverlight.
- If you decide to use Silverlight, consider using the Presentation Model (Model-View-ViewModel) pattern.

Web Applications

Consider the following guidelines when designing a Web application:

- If you want to build applications that are accessed through a Web browser or specialist user agent, consider using ASP.NET.
- If you want to build applications that provide increased interactivity and background processing, with fewer page reloads, consider using ASP.NET with AJAX.
- If you want to build applications that include islands of rich media content and interactivity, consider using ASP.NET with Silverlight controls.
- If you are using ASP.NET and want to implement a control-centric model with separate controllers and improved testability, consider using the ASP.NET MVC Framework.
- If you are using ASP.NET, consider using master pages to simplify development and implement a consistent UI across all pages.

patterns & practices Solution Assets

- Web Client Software Factory at http://msdn.microsoft.com/en-us/library/bb264518.aspx
- Smart Client Software Factory at http://msdn.microsoft.com/en-us/library/aa480482.aspx

- Composite Application Guidance for WPF at http://msdn.microsoft.com/en-us/library/cc707819.aspx
- Smart Client - Composite UI Application Block at http://msdn.microsoft.com/en-us/library/aa480450.aspx

Additional Resources

- *For more information, see Microsoft Inductive User Interface Guidelines* at http://msdn.microsoft.com/en-us/library/ms997506.aspx.
- *For more information, see User Interface Control Guidelines* at http://msdn.microsoft.com/en-us/library/bb158625.aspx.
- *For more information, see User Interface Text Guidelines* at http://msdn.microsoft.com/en-us/library/bb158574.aspx.
- *For more information, see Design and Implementation Guidelines for Web Clients* at http://msdn.microsoft.com/en-us/library/ms978631.aspx.
- *For more information, see Web Presentation Patterns* at http://msdn.microsoft.com/en-us/library/ms998516.aspx.

Chapter 11: Business Layer Guidelines

Objectives

- Understand how the business layer fits into the overall application architecture.
- Understand the components of the business layer.
- Learn the steps for designing these components.
- Learn about the common issues faced when designing the business layer.
- Learn the key guidelines for designing the business layer.
- Learn the key patterns and technology considerations.

Overview

This chapter describes the design process for business layers, and contains key guidelines that cover the important aspects you should consider when designing business layers and business components. These guidelines are organized into categories that include designing business layers and implementing appropriate functionality such as security, caching, exception management, logging, and validation. These categories represent the key areas where mistakes occur most often in business layer design. Figure 1 shows how the business layer fits into typical application architecture.

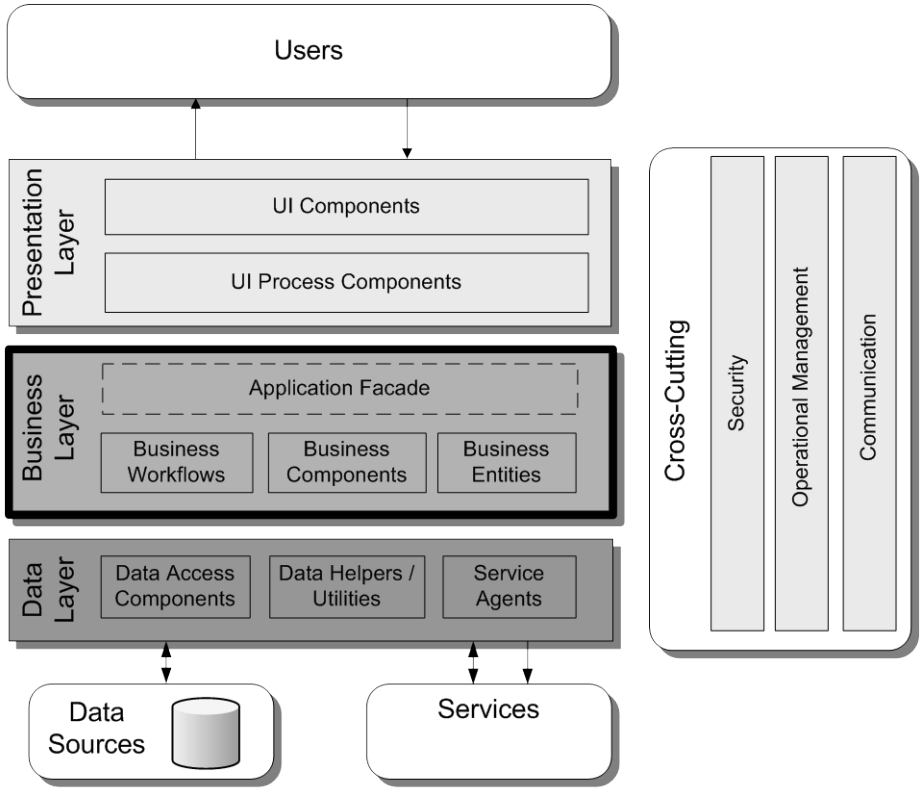


Figure 1 A typical application showing the business layer and the components it may contain

Key Business Components

The following list explains the roles and responsibilities of the main components within the business layer:

- **Application façade** (optional). An application façade combines multiple business operations into a single message-based operation. You might access the application façade from the presentation layer by using different communication technologies.
- **Business components**. Within the business layer there are different components that provide business services, such as processing business rules and interacting with data access components. For example, you might have a business component that implements the transaction script pattern, which allows you to execute multiple operations within a single component used to manage the transaction. Another business component might be used to process requests and apply business rules.
- **Business entities**. Business components used to pass data between other components are considered business entities. The data can represent real-world business entities, such as products and orders, or database entities, such as tables and views. The business entities that an application uses internally can be implemented using custom objects that represent real-world or database entities your application has to work with. Alternatively, business entities can be implemented using data structures such as DataSets and Extensible Markup Language (XML) documents.
- **Business workflows**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

Approach

When designing a business layer, you must also take into account the design requirements for the main constituents of the layer, such as business components, business entities, and business workflow components. This section briefly explains the main activities involved in designing each of the components and the business layer itself. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your business layer:**
    a. Identify the consumers of your business layer.
    b. Determine how you will expose your business layer.
    c. Determine the security requirements for your business layer.
    d. Determine the validation requirements and strategy for your business layer.
    e. Determine the caching strategy for your business layer.
    f. Determine the exception-management strategy for your business layer.
2. **Design your business components**:
    a. Identify business components your application will use.
    b. Make key decisions about location, coupling, and interactions for business components.
    c. Choose appropriate transaction support.
    d. Identify how your business rules are handled.

    e. Identify patterns that fit the requirements.
3. **Design your business entity components**:
    a. Identify common data formats for the business entities.
    b. Choose the data format.
    c. Optionally, choose a design for your custom objects.
    d. Optionally, determine how you will serialize the components.
4. **Design your workflow components:**
    a. Identify workflow style using scenarios.
    b. Choose an authoring mode.
    c. Determine how rules will be handled.
    d. Choose a workflow solution.
    e. Design business components to support workflow.

Design Considerations

When designing a business layer, the goal of a software architect is to minimize the complexity by separating tasks into different areas of concern. For example, business processing, business workflow, and business entities all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not include code related to other areas of concern.

Consider the following guidelines when designing the business layer:

- **Decide if you need a separate business layer**. It is always a good idea to use a separate business layer where possible to improve the maintainability of your application.
- **Identify the responsibilities of your business layer**. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation.
- **Do not mix different types of components in your business layer**. Use a business layer to decouple business logic from presentation and data access code, and to simplify the testing of business logic.
- **Reuse common business logic**. Use a business layer to centralize common business logic functions and promote reuse.
- **Identify the consumers of your business layer**. This will help to determine how you expose your business layer. For example, if your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.
- **Reduce round trips when accessing a remote business layer**. If you are using a message-based interface, consider using coarse-grained packages for data, such as Data Transfer Objects. In addition, consider implementing a remote façade for the business layer interface.
- **Avoid tight coupling between layers**. Use abstraction when creating an interface for the business layer. The abstraction can be implemented using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation layer and the business layer.

Business Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 Business Layer Frame

Category	Common issues
Authentication	• Applying authentication in a business layer when not required. • Designing a custom authentication mechanism. • Failing to use single-sign-on where appropriate.
Authorization	• Using incorrect granularity for roles. • Using impersonation and delegation when not required. • Mixing authorization code and business processing code.
Business Components	• Overloading business components, by mixing unrelated functionality. • Mixing data access logic within business logic in business components. • Not considering the use of message-based interfaces to expose business components.
Business Entities	• Using the Domain Model when not appropriate. • Choosing incorrect data formats for your business entities. • Not considering serialization requirements.
Caching	• Caching volatile data. • Caching too much data in the business layer. • Failing to cache data in a ready-to-use format. • Caching sensitive data in unencrypted form.
Coupling and Cohesion	• Tight coupling across layers. • No clear separation of concerns within the business layer. • Failing to use a message-based interface between layers.
Concurrency and Transactions	• Not preventing concurrent access to static data that is not read-only. • Not choosing the correct data concurrency model. • Using long-running transactions that hold locks on data.
Data Access	• Accessing the database directly from the business layer. • Mixing data access logic within business logic in business components.

Category	Common issues
Exception Management	• Revealing sensitive information to the end user. • Using exceptions to control application flow. • Not logging sufficient detail from exceptions. • Failing to appropriately notify users with useful error messages.
Logging and Instrumentation	• Failing to add adequate instrumentation to business components. • Failing to log system-critical and business-critical events. • Not suppressing logging failures.
Service Interface	• Breaking the service interface. • Implementing business rules in the service interface. • Failing to consider interoperability requirements.
Validation	• Relying on validation that occurs in the presentation layer. • Failure to validate for length, range, format and type. • Not reusing the validation logic.
Workflows	• Not considering application management requirements. • Choosing an incorrect workflow pattern. • Not considering how to handle all exception states. • Choosing an incorrect workflow technology.

Authentication

Designing an effective authentication strategy for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attacks.

Consider the following guidelines when designing an authentication strategy:

- Only authenticate users in the business layer if it is shared by other applications. If the business layer will be used only by a presentation layer or a service layer on the same tier, avoid authentication in the business layer.
- If your business layer will be used in multiple applications, using separate user stores, consider implementing a single-sign-on mechanism.
- Only flow the caller's identity to the business layer if you need to authenticate based on the original caller's ID.
- If the presentation and business layers are deployed on the same machine and you need to access resources based on the original caller's ACL permissions, consider using impersonation.
- If the presentation and business layers are deployed to separate machines and you need to access resources based on the original caller's ACL permissions, consider using delegation. Only use delegation if it is absolutely necessary, as many environments do not allow delegation. Instead, authenticate the user at the boundary and use trusted subsystems in subsequent calls to lower layers.

- If using Web services, consider using IP Filtering to restrict web service being called only from the presentation layer.

Authorization

Designing an effective authorization strategy for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges.

Consider the following guidelines when designing an authorization strategy:

- Protect resources by applying authorization to callers based on their identity, account groups, or roles.
- Consider using role-based authorization for business decisions.
- Consider using resource-based authorization for system auditing.
- Consider using claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.
- Avoid using impersonation and delegation because it can significantly affect performance and scaling. It is generally more expensive to impersonate a client on a call than to make the call directly.

Business Components

Business components implement business rules in diverse patterns, and accept and return simple or complex data structures. Your business components should expose functionality in a way that is agnostic to the data stores and services required to perform the work. Compose your business components in meaningful and transactionally consistent ways. Designing business components is an important task. If you fail to design business components correctly, the result is likely to be code that is impossible to maintain.

Consider the following guidelines when designing business components:

- Avoid mixing data access logic and business logic within your business components.
- Design components to be highly cohesive. In other words, you should not overload business components by adding unrelated or mixed functionality.
- If you want to keep business rules separate from business data, consider using business process components to implement your business rules.
- If your application has volatile business rules, store them in a rules engine.
- If the business process involves multiple steps and long-running transactions, consider using workflow components.

Business Entities

Business entities store data values and expose them through properties; they provide stateful programmatic access to the business data and related functionality. Therefore, designing or

choosing appropriate business entities is vitally important for maximizing the performance and efficiency of your business layer.

Consider the following guidelines when designing business entities:

- Choose appropriate data formats for your business entities. For smaller data-driven applications consider using DataSets, and for document-centric data consider using XML for the data format. For other types of applications, consider using custom objects instead.
- Consider the analysis requirements and complexity associated with a Domain Model design before choosing to use it for business entities. A Domain Model is very good for handling complex business rules, and works best with a stateful application such as a rich client.
- If the tables in the database represent business entities, consider using the Table Module pattern.
- Consider the serialization requirements of your business entities. For example, if you are storing business entities in a central location for state management, or passing business entities across process or network boundaries, they will need to support serialization.
- Minimize the number of calls made across physical tiers. For example, use the Data Transfer Object (DTO) pattern.

Caching

Designing an appropriate caching strategy for your business layer is important for the performance and responsiveness of your application. Use caching to optimize reference data lookups, avoid network round trips, and avoid unnecessary and duplicated processing. As part of your caching strategy, you must decide when and how to load the cache data. To avoid client delays, load the cache asynchronously or by using a batch process.

Consider the following guidelines when designing a caching strategy:

- Consider caching static data that will be reused regularly within the business layer.
- Consider caching data that cannot be retrieved from the database quickly and efficiently.
- Consider caching data in a ready-to-use format within your business layer.
- Avoid caching sensitive data if possible, or design a mechanism to protect sensitive data in the cache.
- Consider how Web farm deployment will affect the design of your business layer caching solution. If a request can be handled by any server in the farm, you will need to support the synchronization of cached data that can change.

Coupling and Cohesion

When designing components for your business layer, ensure that they are highly cohesive, and implement loose coupling between layers. This helps to improve the scalability of your application.

Consider the following guidelines when designing for coupling and cohesion:

- Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).
- Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete components depend on abstractions and not on other concrete components (the principle of Dependency Inversion).
- Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.
- Design for high cohesion. Components should contain only functionality specifically related to that component.
- Avoid mixing data access logic with business logic in your business components.

Concurrency and Transactions

When designing for concurrency and transactions, it is important to identify the appropriate concurrency model and determine how you will manage transactions. You can choose between an optimistic model and a pessimistic model for concurrency. With *optimistic concurrency*, locks are not held on data and updates require code to check, usually against a timestamp, that the data has not changed since it was last retrieved. With *pessimistic concurrency*, data is locked and cannot be updated by another operation until the lock is released.

Consider the following guidelines when designing for concurrency and transactions:

- Consider transaction boundaries, so that retries and composition are possible.
- Where you cannot apply a commit or rollback, or if you use a long-running transaction, implement compensating methods to revert the data store to its previous state in case an operation within the transaction fails.
- Avoid holding locks for long periods; for example, when executing long-running atomic transactions or when locking access to shared data.
- Choose an appropriate transaction isolation level, which defines how and when changes become available to other operations.

Data Access

Designing an effective data-access strategy for your business layer is important for maximizing maintainability and the separation of concerns. Failing to do so can make your application difficult to manage and extend as business requirements change. An effective data-access strategy will allow your business layer to adapt to changes in the underlying data sources. It will also make it easier to reuse functionality and components in other applications.

Consider the following guidelines when designing a data-access strategy:

- Avoid mixing data-access code and business logic within your business components.
- Avoid directly accessing the database from your business layer.
- Consider using a separate data access layer for access to the database.

Exception Management

Designing an effective exception-management solution for your business layer is important for the security and reliability of your application. Failing to do so can leave your application vulnerable to Denial of Service (DoS) attacks, and may allow it to reveal sensitive and critical information about your application. Raising and handling exceptions is an expensive operation, so it is important that your exception management design takes into account the impact on performance.

When designing an exception-management strategy, consider following guidelines:

- Do not use exceptions to control business logic.
- Only catch internal exceptions that you can handle, or if you need to add information. For example, catch data conversion exceptions that can occur when trying to convert null values.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer.
- Design an approach for catching and handling unhandled exceptions.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.

Logging and Instrumentation

Designing a good logging and instrumentation solution for your business layer is important for the security and reliability of your application. Failing to do so can leave your application vulnerable to repudiation threats, where users deny their actions. Log files may also be required to prove wrongdoing in legal proceedings. Auditing is generally considered most authoritative if the log information is generated at the precise time of resource access, and by the same routine that accesses the resource. Instrumentation can be implemented using performance counters and events. System-monitoring tools can use this instrumentation, or other access points, to provide administrators with information about the state, performance, and health of an application.

Consider the following guidelines when designing a logging and instrumentation strategy:

- Centralize logging and instrumentation for your business layer.
- Include instrumentation for system-critical and business-critical events in your business components.
- Do not store business-sensitive information in the log files.
- Ensure that a logging failure does not affect normal business layer functionality.
- Consider auditing and logging all access to functions within business layer.

Service Interface

When the business layer is deployed to a separate tier, or when implementing the business layer for a service, you must consider the guidelines for service interfaces. When designing a

service interface, you must consider the granularity of service operations and interoperability requirements. Generally, services should provide coarse-grained operations that reduce round trips between the service and service consumer. In addition, you should use common data formats for the interface schema that can be extended without affecting consumers of the service.

Consider the following guidelines when designing a service interface:

- Design your service interfaces in such a way that changes to the business logic do not affect the interface.
- Do not implement business rules in a service interface or in the service implementation layer.
- Design service interfaces for maximum interoperability with other platforms and services by using common protocols and data formats.
- Design the service to expose schema and contract information only, and make no assumptions on how the service will be used.
- Choose an appropriate transport protocol. For example, choose named pipes or shared memory when the service and service consumer are on the same physical machine, TCP when a service is accessed by consumers within the same network, or HTTP for services exposed over the Internet.

Validation

Designing an effective validation solution for your business layer is important for the security and reliability of your application. Failure to do so can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attacks. There is no comprehensive definition of what constitutes a valid input or malicious input. In addition, how your application uses input influences the risk of the exploit.

Consider the following guidelines when designing a validation strategy:

- Validate all input and method parameters within the business layer, even when input validation occurs in the presentation layer.
- Centralize your validation approach, if it can be reused.
- Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious.
- Validate input data for length, range, format, and type.

Workflows

Workflow components are used only when your application must support a series of tasks that are dependent on the information being processed. This information can be anything from data checked against business rules to human interaction. When designing workflow components, it is important to consider how you will manage the workflows, and to understand the available options.

Consider the following guidelines when designing a workflow strategy:

- Implement workflows within components that involve a multi-step or long-running process.
- Choose an appropriate workflow style depending on the application scenario.
- Handle fault conditions within workflows, and expose suitable exceptions.
- If the component must execute a specified set of steps sequentially and synchronously, consider using the pipeline pattern.
- If the process steps can be executed asynchronously in any order, consider using the event pattern.

Deployment Considerations

When deploying a business layer, you must consider performance and security issues within the production environment.

Consider the following guidelines when deploying a business layer:

- Deploy the business layer to the same physical tier as the presentation or service layer in order to maximize application performance.
- If you must support a remote business layer, consider using the TCP protocol to improve application performance.
- Consider using Internet Protocol Security (IPSec) to protect data passed between physical tiers for all business layers for all applications.
- Consider using Secure Sockets Layer (SSL) encryption to protect calls from business layer components to remote Web services.

Pattern Map

Key patterns are organized by the key categories detailed in the Business Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern Map

Category	Relevant patterns
Business Components	• Application Façade • Chain of Responsibility • Command
Business Entities	• Domain Model • Entity Translator • Table Module
Concurrency and Transactions	• Capture Transaction Details • Coarse-Grained Lock • Implicit Lock • Optimistic Offline Lock • Pessimistic Offline Lock • Transaction Script

Data Access	• Active Record • Data Mapper • Query Object • Repository • Row Data Gateway • Table Data Gateway
Workflows	• Data-driven workflow • Human workflow • Sequential workflow • State-driven workflow

- For more information on the Domain Model, Table Module, Coarse-Grained Lock, Implicit Lock, Transaction Script, Active Record, Data Mapper, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data Gateway patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on the Façade, Chain of Responsibility, and Command patterns, see "data & object factory" at http://www.dofactory.com/Patterns/Patterns.aspx
- For more information on the Entity Translator pattern, see "Useful Patterns for Services" at http://msdn.microsoft.com/en-us/library/cc304800.aspx
- For more information on the Capture Transaction Details pattern, see "Data Patterns" at http://msdn.microsoft.com/en-us/library/ms998446.aspx
- For more information on the Data-Driven Workflow, Human Workflow, Sequential Workflow, and State-Driven Workflow, see "Windows Workflow Foundation Overview" at http://msdn.microsoft.com/en-us/library/ms734631.aspx and "Workflow Patterns" at http://www.workflowpatterns.com/

Pattern Descriptions

- **Active Record.** Include a data access object within a domain entity.
- **Application Façade.** Centralize and aggregate behavior to provide a uniform service layer.
- **Capture Transaction Details**. Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.
- **Chain of Responsibility**. Avoid coupling the sender of a request to its receiver by allowing more than one object to handle the request.
- **Coarse Grained Lock**. Lock a set of related objects with a single lock.
- **Command**. Encapsulate request processing in a separate command object with a common execution interface.
- **Data Mapper**. Implement a mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.
- **Data-driven Workflow**. A workflow that contains tasks whose sequence is determined by the values of data in the workflow or the system.
- **Domain Model**. A set of business objects that represents the entities in a domain and the relationships between them.

- **Entity Translator**. An object that transforms message data types to business types for requests, and reverses the transformation for responses.
- **Human Workflow**. A workflow that involves tasks performed manually by humans.
- **Implicit Lock**. Use framework code to acquire locks on behalf of code that accesses shared resources.
- **Optimistic Offline Lock**. Ensure that changes made by one session do not conflict with changes made by another session.
- **Pessimistic Offline Lock**. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.
- **Query Object**. An object that represents a database query.
- **Repository**. An in-memory representation of a data source that works with domain entities.
- **Row Data Gateway**. An object that acts as a gateway to a single record in a data source.
- **Sequential Workflow**. A workflow that contains tasks that follow a sequence, where one task is initiated after completion of the preceding task.
- **State-driven Workflow**. A workflow that contains tasks whose sequence is determined by the state of the system.
- **Table Data Gateway**. An object that acts as a gateway to a table or view in a data source and centralizes all of the select, insert, update, and delete queries.
- **Table Module**. A single component that handles the business logic for all rows in a database table or view.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology and implement transaction support:

- If you require workflows that automatically support secure, reliable, transacted data exchange, a broad choice of transport and encoding options, and provide built-in persistence and activity tracking, consider using Windows Workflow (WF).
- If you require workflows that implement complex orchestrations and support reliable store and forward messaging capabilities, consider using Microsoft BizTalk® Server.
- If you must interact with non-Microsoft systems, perform electronic data interchange (EDI) operations, or implement Enterprise Service Bus (ESB) patterns, consider using the ESB Guidance for BizTalk Server.
- If your business layer is confined to a single Microsoft Office SharePoint® site and does not require access to information in other sites, consider using Microsoft Office SharePoint Server (MOSS). Note that MOSS is not suitable for multiple-site scenarios.
- If you are designing transactions that span multiple data sources, consider using a transaction scope (System.Transaction) to manage the entire transaction.

Additional Resources

- For more information, see *Concurrency Control* at http://msdn.microsoft.com/en-us/library/ms978457.aspx.
- For more information, see *Integration Patterns* at http://msdn.microsoft.com/en-us/library/ms978729.aspx.

Chapter 12: Data Access Layer Guidelines

Objectives

- Understand how the data layer fits into the application architecture.
- Understand the components of the data layer.
- Learn the steps for designing these components.
- Learn the common issues faced when designing the data layer.
- Learn the key guidelines for designing the data layer.
- Learn the key patterns and technology considerations for designing the data access layer.

Overview

This chapter describes the key guidelines for designing the data layer of an application. The guidelines are organized by category and cover the common issues encountered, and mistakes commonly made, when designing the data layer. Figure 1. shows how the data layer fits into typical application architecture.

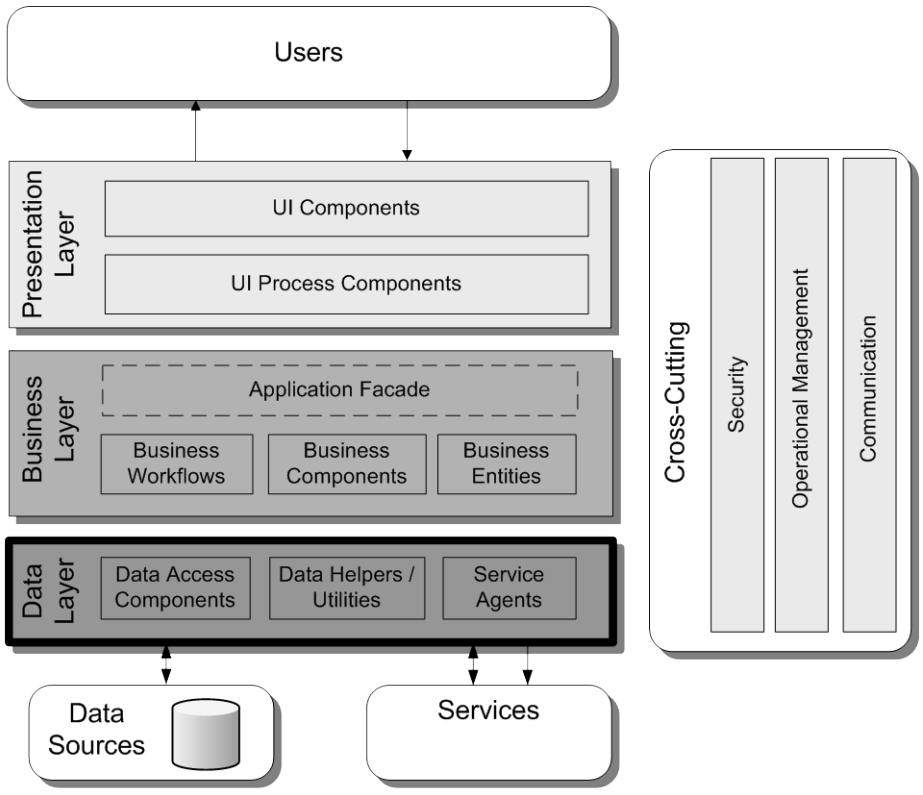


Figure 1 A typical application showing the data layer and the components it may contain

Data Layer Components

- **Data access logic components**. Data access components abstract the logic necessary to access your underlying data stores. Doing so centralizes the data access functionality, which makes the application easier to configure and maintain.
- **Data helpers / utilities**. Helper functions and utilities assist in data manipulation, data transformation, and data access within the layer. They consist of specialized libraries and/or custom routines especially designed to maximize data access performance and reduce the development requirements of the logic components and the service agent parts of the layer.
- **Service agents**. When a business component must use functionality exposed by an external service, you might need to create code that manages the semantics of communicating with that service. Service agents isolate your application from the idiosyncrasies of calling diverse services, and can provide additional services such as basic mapping between the format of the data exposed by the service and the format your application requires.

Approach

A correct approach to designing the data layer will reduce development time and assist in maintenance of the data layer after the application is deployed. This section briefly outlines an effective design approach for the data layer. Perform the following key activities in each of these areas when designing your data layer:

1. **Create an overall design for your data access layer:**
    a. Identify your data source requirements.
    b. Determine your data access approach.
    c. Choose how to map data structures to the data source.
    d. Determine how to connect to the data source.
    e. Determine strategies for handling data source errors.
2. **Design your data access components**:
    a. Enumerate the data sources that you will access.
    b. Decide on the method of access for each data source.
    c. Determine whether helper components are required or desirable to simplify data access component development and maintenance.
    d. Determine relevant design patterns. For example, consider using the Table Data Gateway, Query Object, Repository, and other patterns.
3. **Design your data helper components:**
    a. Identify functionality that could be moved out of the data access components and centralized for reuse.
    b. Research available helper component libraries.
    c. Consider custom helper components for common problems such as connection strings, data source authentication, monitoring, and exception processing.
    d. Consider implementing routines for data access monitoring and testing in your helper components.
    e. Consider the setup and implementation of logging for your helper components.

4. **Design your service agents:**
    a. Use the appropriate tool to add a service reference. This will generate a proxy and the data classes that represent the data contract from the service.
    b. Determine how the service will be used in your application. For most applications, you should use an abstraction layer between the business layer and the data access layer, which will provide a consistent interface regardless of the data source. For smaller applications, the business layer, or even the presentation layer, may access the service agent directly.

Design Guidelines

The following design guidelines provide information about different aspects of the data access layer that you should consider. Follow these guidelines to ensure that your data access layer meets the requirements of your application, performs efficiently and securely, and is easy to maintain and extend as business requirements change.

- **Choose the data access technology**. The choice of an appropriate data access technology will depend on the type of data you are dealing with, and how you want to manipulate the data within the application. Certain technologies are better suited for specific scenarios. Refer to the Data Access Technology Matrix found later in this guide. It discusses these options and enumerates the benefits and considerations for each data access technology.
- **Use abstraction to implement a loosely coupled interface to the data access layer**. This can be accomplished by defining interface components, such as a gateway with well-known inputs and outputs, which translate requests into a format understood by components within the layer. In addition, you can use interface types or abstract base classes to define a shared abstraction that must be implemented by interface components.
- **Consider consolidating data structures**. If you are dealing with table-based entities in your data access layer, consider using Data Transfer Objects (DTOs) to help you organize the data into unified structures. In addition, DTOs encourage coarse-grained operations while providing a structure that is designed to move data across different boundary layers.
- **Encapsulate data access functionality within the data access layer**. The data access layer hides the details of data source access. It is responsible for managing connections, generating queries, and mapping application entities to data source structures. Consumers of the data access layer interact through abstract interfaces using application entities such as custom objects, DataSets, DataReaders, and XML. Other application layers that access the data access layer will manipulate this data in more complex ways to implement the functionality of the application. Separating concerns in this way assists in application development and maintenance.
- **Decide how to map application entities to data source structures**. The type of entity you use in your application is the main factor in deciding how to map those entities to data source structures.
- **Decide how you will manage connections**. As a rule, the data access layer should create and manage all connections to all data sources required by the application. You must choose an appropriate method for storing and protecting connection information that conforms to application and security requirements.

- **Determine how you will handle data exceptions**. The data access layer should catch and (at least initially) handle all exceptions associated with data sources and CRUD (Create, Read, Update, and Delete) operations. Exceptions concerning the data itself, and data source access and timeout errors, should be handled in this layer and passed to other layers only if the failures affect application responsiveness or functionality.
- **Consider security risks**. The data access layer should protect against attacks that try to steal or corrupt data, and protect the mechanisms used to gain access to the data source. It should also use the "least privilege" design approach to restrict privileges to only those needed to perform the operations required by the application. If the data source itself has the ability to limit privileges, security should be considered and implemented in the data access layer as well as in the source.
- **Reduce round trips**. Consider batching commands into a single database operation.
- **Consider performance and scalability objectives**. Scalability and performance objectives for the data access layer should be taken into account during design. For example, when designing an Internet-based merchant application, data layer performance is likely to be a bottleneck for the application. When data layer performance is critical, use profiling to understand and then limit expensive data operations.

Data Layer Frame

There are several common issues that you must consider as your develop your design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 Data Layer Frame

Category	Common issues
BLOB	• Improperly storing BLOBs in the database instead of the file system. • Using an incorrect type for BLOB data in the database. • Searching and manipulating BLOB data.
Batching	• Failing to use batching to reduce database round trips. • Holding onto locks for excessive periods when batching.
Connections	• Improper configuration of connection pooling. • Failing to handle connection timeouts and disconnections. • Performing transactions that span multiple connections. • Holding connections open for excessive periods. • Using individual identities instead of a trusted subsystem to access the database.
Data Format	• Choosing the wrong data format. • Failing to consider serialization requirements. • Not mapping objects to a relational data store.
Exception Management	• Not handling data access exceptions. • Failing to shield database exceptions from the original caller. • Failing to log critical exceptions.

Queries	• Using string concatenation to build queries. • Mixing queries with business logic. • Not optimizing the database for query execution.
Stored Procedures	• Using an incorrect strategy to pass parameters to stored procedures. • Formatting data for display to users in stored procedures. • Not considering how dynamic SQL in stored procedures can impact performance, security, and maintainability.
Transactions	• Using the incorrect isolation level. • Using exclusive locks, which can cause contention and deadlocks. • Allowing long-running transactions to block access to data.
Validation	• Failing to validate and constrain data fields. • Not handling NULL values. • Not filtering for invalid characters.
XML	• Not considering how to handle extremely large XML data sets. • Not choosing the appropriate technology for XML to relational database interaction. • Failure to set up proper indexes on applications that do heavy querying with XML • Failing to validate XML inputs using schemas.

BLOB

A BLOB is a binary large object. When data is stored and retrieved as a single stream of data, it can be considered to be a BLOB. A BLOB may have structure within it, but that structure is not apparent to the database that stores it or the data layer that reads and writes it. Databases can store the BLOB data or can store pointers to them within the database. The BLOB data is usually stored in a file system if not stored directly in the database. BLOBs are typically used to store image data, but can also be used to store binary representations of objects.

Consider the following guidelines when designing for BLOBs:

- Store BLOB data in a database only when it is not practical to store it on the disk.
- Consider using BLOBs to simplify synchronization of large binary objects between servers.
- Consider whether you need to search the BLOB data. If so, create and populate other searchable database fields instead of parsing the BLOB data.
- When retrieving the BLOB, cast it to the appropriate type for manipulation within your business or presentation layer.
- Do not consider storing the BLOB in the database when using buffered transmission.

Batching

Batching database commands can improve the performance of your data layer. Each request to the database execution environment incurs an overhead. Batching can reduce the total overhead by increasing throughput and decreasing latency. Batching similar queries can improve performance because the database caches and can reuse a query execution plan for a similar query.

Consider the following guidelines when designing batching:

- Consider using batched commands to reduce round trips to the database and minimize network traffic.
- Batch similar queries for maximum benefit. Batching dissimilar or random queries provides less reduction in overhead.
- Consider using batched commands and a DataReader to load or copy multiple sets of data.
- When loading large volumes of file-based data into the database, consider using bulk copy utilities.
- Do not consider placing locks on long-running batch commands.

Connections

Connections to data sources are a fundamental part of the data layer. All data source connections should be managed by the data layer. Creating and managing connections uses valuable resources in both the data layer and the data source. To maximize performance, follow guidelines for creating, managing, and closing connections

Consider the following guidelines when designing for data layer connections:

- In general, open connections as late as possible and close them as early as possible.
- To maximize the effectiveness of connection pooling, consider using a trusted subsystem security model and avoid impersonation if possible.
- Perform transactions through a single connection where possible.
- For security reasons, avoid using a System or User Data Source Name (DSN) to store connection information.
- Design retry logic to manage the situation where the connection to the data source is lost or times out.

Data Format

Data formats and types are important in order to properly interpret the raw bytes stored in the database and transferred by the data layer. Choosing the appropriate data format provides interoperability with other applications, and facilitates serialized communications across different processes and physical machines. Data format and serialization are also important in order to allow the storage and retrieval of application state by the business layer.

Consider the following guidelines when designing your data format:

- In most cases, you should use custom data or business entities for improved application maintainability. This will require additional code to map the entities to database operations. However, new object/relational mapping (O/RM) solutions are available to reduce the amount of custom code required.
- Consider using XML for interoperability with other systems and platforms or when working with data structures that can change over time.
- Consider using DataSets for disconnected scenarios in simple CRUD-based applications.
- Understand the serialization and interoperability requirements of your application.

Exception Management

Design a centralized exception-management strategy so that exceptions are caught and thrown consistently in your data layer. If possible, centralize exception-handling logic in your database helper components. Pay particular attention to exceptions that propagate through trust boundaries and to other layers or tiers. Design for unhandled exceptions so they do not result in application reliability issues or exposure of sensitive application information.

Consider the following guidelines when designing your exception-management strategy:

- Determine exceptions that should be caught and handled in the data access layer. Deadlocks, connection issues, and optimistic concurrency checks can often be resolved at the data layer.
- Consider implementing a retry process for operations where data source errors or timeouts occur, where it is safe to do so.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer.
- Design an approach for catching and handling unhandled exceptions.
- Design an appropriate logging and notification strategy for critical errors and exceptions that does not reveal sensitive information.

Object Relational Mapping Considerations

When designing an object oriented (OO) application, consider the impedance mismatch between the OO model and the relational model that makes it difficult to translate between them. For example, encapsulation in OO designs, where fields are hidden, contradicts the public nature of properties in a database. Other examples of impedance mismatch include differences in the data types, structural differences, transactional differences, and differences in how data is manipulated. The two common approaches to handling the mismatch are data access design patterns such as Repository, and O/RM tools. A common model associated with OO design is the Domain Model, which is based on modeling entities after objects within a domain. As a result, the term “domain” represents an object-oriented design in the following guidelines.

Consider the following guidelines when designing for object relational mapping:

- Consider using or developing a framework that provides a layer between domain entities and the database.
- If you are working in a Greenfield environment, where you have full control over the database schema, choose an O/RM tool that will generate a schema to support the object model and provide a mapping between the database and domain entities.
- If you are working in a Brownfield environment, where you must work with an existing database schema, consider tools that will help you to map between the domain model and relational model.
- If you are working with a smaller application or do not have access to O/RM tools, implement a common data access pattern such as Repository. With the Repository pattern, the repository objects allow you to treat domain entities as if they were located in memory.

- When working with Web applications or services, group entities and support options that will partially load domain entities with only the required data. This allows applications to handle the higher user load required to support stateless operations, and limit the use of resources by avoiding holding initialized domain models for each user in memory.

Queries

Queries are the primary data manipulation operations for the data layer. They are the mechanism that translates requests from the application into create, retrieve, update and delete (CRUD) actions on the database. As queries are so essential, they should be optimized to maximize database performance and throughput.

When using queries in your data layer, consider the following guidelines:

- Use parameterized SQL statements and typed parameters to mitigate security issues and reduce the chance of SQL injection attacks succeeding.
- When it is necessary to build queries dynamically, ensure that you validate user input data used in the query.
- Do not use string concatenation to build dynamic queries in the data layer.
- Consider using objects to build queries. For example, implement the Query Object pattern or use the object support provided by ADO.NET.
- When building dynamic SQL, avoid mixing business-processing logic with logic used to generate the SQL statement. Doing so can lead to code that is very difficult to maintain and debug.

Stored Procedures

In the past, stored procedures represented a performance improvement over dynamic SQL statements. However, with modern database engines, performance is no longer a major factor. When considering the use of stored procedures, the primary factors are abstraction, maintainability, and your environment. This section contains guidelines to help you design your application when using stored procedures. For guidance on choosing between using stored procedures and dynamic SQL statements, see the section that follows.

When it comes to security and performance, the primary guidelines are to use typed parameters and avoid dynamic SQL within the stored procedure. Parameters are one of the factors that influence the use of cached query plans instead of rebuilding the query plan from scratch. When parameter types and the number of parameters change, new query execution plans are generated, which can reduce performance.

Consider the following guidelines when designing stored procedures:

- Use typed parameters as input values to the procedure and output parameters to return single values.
- Use parameter or database variables if it is necessary to generate dynamic SQL within a stored procedure.
- Consider using XML parameters for passing lists or tabular data.

- Design appropriate error handling and return errors that can be handled by the application code.
- Avoid the creation of temporary tables while processing data. However, if temporary tables need to be used, consider creating them in-memory rather than on disk.

Stored Procedures vs. Dynamic SQL

The choice between stored procedures and dynamic SQL focuses primarily on the use of SQL statements dynamically generated in code instead of SQL implemented within a stored procedure in the database. When choosing between stored procedures and dynamic SQL, you must consider the abstraction requirements, maintainability, and environment constraints.

The main advantages of stored procedures are:

- They provide an abstraction layer to the database, which can minimize the impact on application code when the database schema changes.
- Security is easier to implement and manage because you can restrict access to everything except the stored procedure.

The main advantages of dynamic SQL statements are:

- You can take advantage of fine-grained security features supported by most databases.
- They require less in terms of specialist skills than stored procedures.
- They are easier to debug than stored procedures.

Consider the following guidelines when choosing between stored procedures and dynamic SQL:

- If you have a small application that has a single client and few business rules, dynamic SQL is often the best choice.
- If you have a larger application that has multiple clients, consider how you can achieve the required abstraction. Decide where that abstraction should exist: at the database in the form of stored procedures, or in the data layer of your application in the form of data access patterns or O/RM products.
- If you want to minimize code changes when the database schema changes, consider using stored procedures to provide an abstraction layer. Changes associated with normalization or schema optimization will often have no affect on application code. If a schema change does affect inputs and outputs in a procedure, application code is affected; however, the changes are limited to clients of the stored procedure.
- Consider the resources you have for development of the application. If you do not have resources that are intimately familiar with database programming, consider tools or patterns that are more familiar to your development staff.
- Consider debugging support. Dynamic SQL is easier for application developers to debug.
- When considering dynamic SQL, you must understand the impact that changes to database schemas will have on your application. As a result, you should implement an abstraction in the data access layer to decouple business components from the generation of database queries. Several patterns, such as Query Object and Repository, can be used to provide this abstraction.

Transactions

A *transaction* is an exchange of sequential information and associated actions that are treated as an atomic unit in order to satisfy a request and ensure database integrity. A transaction is only considered complete if all information and actions are complete, and the associated database changes are made permanent. Transactions support undo (rollback) database actions following an error, which helps to preserve the integrity of data in the database.

Consider the following guidelines when designing transactions:

- Enable transactions only when you need them. For example, you should not use a transaction for an individual SQL statement because Microsoft SQL Server® automatically executes each statement as an individual transaction.
- Keep transactions as short as possible to minimize the amount of time that locks are held.
- Use the appropriate isolation level. The tradeoff is data consistency versus contention. A high isolation level will offer higher data consistency at the price of overall concurrency. A lower isolation level improves performance by lowering contention at the cost of consistency.
- If using manual or explicit transactions, consider implementing the transaction within a stored procedure.
- Consider the use of multiple active result sets (MARS) in transaction-heavy concurrent applications to avoid potential deadlock issues.

Validation

Designing an effective input and data-validation strategy is critical to the security of your application. Determine the validation rules for data received from other layers and from third-party components, as well as from the database or data store. Understand your trust boundaries so that you can validate any data that crosses these boundaries.

Consider the following guidelines when designing a validation strategy:

- Validate all data received by the data layer from all callers.
- Consider the purpose to which data will be put when designing validation. For example, user input used in the creation of dynamic SQL should be examined for characters or patterns that occur in SQL injection attacks.
- Understand your trust boundaries so that you can validate data that crosses these boundaries.
- Return informative error messages if validation fails.

XML

Extensible Markup Language (XML) is useful for interoperability and for maintaining data structure outside the database. For performance reasons, be careful when using XML for very large amounts of data. If you must handle large amounts of data, use attribute-based schemas instead of element-based schemas. Use schemas to validate the XML structure and content.

Consider the following guidelines when designing for the use of XML:

- Consider using XML readers and writers to access XML-formatted data.
- Consider using an XML schema to define formats and to provide validation for data stored and transmitted as XML.
- Consider using custom validators for complex data parameters within your XML schema.
- Store XML in typed columns in the database, if available, for maximum performance.
- For read-heavy applications that use XML in SQL Server, consider XML indexes.

Manageability Considerations

Manageability is an important factor in your application because a manageable application is easier for administrators and operators to install, configure, and monitor. Manageability also makes it easier to detect, validate, resolve, and verify errors at run time. You should always strive to maximize manageability when designing your application.

Consider the following guidelines when designing for manageability:

- Consider using common interface types or a shared abstraction (Dependency Inversion) to provide an interface to the data access layer.
- Consider the use of custom entities, or decide if other data representations will better meet your requirements. Coding custom entities can increase development costs; however, they also provide improved performance through binary serialization and a smaller data footprint.
- Implement business entities by deriving them from a base class that provides basic functionality and encapsulates common tasks. However, be careful not to overload the base class with unrelated operations, which would reduce the cohesiveness of entities derived from the base class, and cause maintainability and performance issues.
- Design business entities to rely on data access logic components for database interaction. Centralize implementation of all data access policies and related business logic. For example, if your business entities access SQL Server databases directly, all applications deployed to clients that use the business entities will require SQL connectivity and logon permissions.
- Consider using stored procedures to abstract data access from the underlying data schema. However, be careful not to overuse them because this will severely impact code maintenance and reuse and thus the maintainability of your application. A symptom of overuse is large trees of stored procedures that call each other.

Performance Considerations

Performance is a function of both your data layer design and your database design. Consider both together when tuning your system for maximum data throughput.

Consider the following guidelines when designing for performance:

- Use connection pooling and tune performance based on results obtained by running simulated load scenarios.

- Consider tuning isolation levels for data queries. If you are building an application with high-throughput requirements, special data operations may be performed at lower isolation levels than the rest of the transaction. Combining isolation levels can have a negative impact on data consistency, so you must carefully analyze this option on a case-by-case basis.
- Consider batching commands to reduce round trips to the database server.
- Consider using optimistic concurrency with non-volatile data to mitigate the cost of locking data in the database. This avoids the overhead of locking database rows, including the connection that must be kept open during a lock.
- If using a DataReader, use ordinal lookups for faster performance.

Security Considerations

The data layer should protect the database against attacks that try to steal or corrupt data. It should allow only as much access to the various parts of the data source as is required. The data layer should also protect the mechanisms used to gain access to the data source.

Consider the following guidelines when designing for security:

- When using Microsoft SQL Server, consider using Windows authentication with a trusted subsystem.
- Encrypt connection strings in configuration files instead of using a system or user data source name (DSN).
- When storing passwords, use a salted hash instead of an encrypted version of the password.
- Require that callers send identity information to the data layer for auditing purposes.
- If you are using SQL statements, consider the parameterized approach instead of string concatenation to protect against SQL injection attacks.

Deployment Considerations

When deploying a data access layer, the goal of a software architect is to consider the performance and security issues in the production environment.

Consider the following guidelines when deploying the data access layer:

- Locate the data access layer on the same tier as the business layer to improve application performance.
- If you need to support a remote data access layer, consider using the TCP protocol to improve performance.
- You should not locate the data access layer on the same server as the database.

Pattern Map

Key patterns are organized by the key categories detailed in the Data Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern Map

Category	Relevant patterns
General	• Active Record • Data Mapper • Data Transfer Object • Domain Model • Query Object • Repository • Table Data Gateway • Table Module
Batching	• Parallel Processing • Partitioning
Transactions	• Coarse-Grained Lock • Capture Transaction Details • Implicit Lock • Optimistic Offline Lock • Pessimistic Offline Lock • Transaction Script

- For more information on the Domain Model, Table Module, Coarse-Grained Lock, Implicit Lock, Transaction Script, Active Record, Data Mapper, Data Transfer Object, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data Gateway patterns, see "Patterns of Enterprise Application Architecture (P of EAA)" at http://martinfowler.com/eaaCatalog/
- For more information on the Capture Transaction Details pattern, see "Data Patterns" at http://msdn.microsoft.com/en-us/library/ms998446.aspx

Pattern Descriptions

- **Active Record**. Include a data access object within a domain entity.
- **Application Service**. Centralize and aggregate behavior to provide a uniform service layer.
- **Capture Transaction Details**. Create database objects, such as triggers and shadow tables, to record changes to all tables belonging to the transaction.
- **Coarse Grained Lock**. Lock a set of related objects with a single lock.
- **Data Mapper.** Implement a mapping layer between objects and the database structure that is used to move data from one structure to another while keeping them independent.
- **Data Transfer Object**. An object that stores the data transported between processes, reducing the number of method calls required.
- **Domain Model**. A set of business objects that represents the entities in a domain and the relationships between them.

- **Implicit Lock**. Use framework code to acquire locks on behalf of code that accesses shared resources.
- **Optimistic Offline Lock**. Ensure that changes made by one session do not conflict with changes made by another session.
- **Parallel Processing**. Allow multiple batch jobs to run in parallel to minimize the total processing time.
- **Partitioning**. Partition multiple large batch jobs to run concurrently.
- **Pessimistic Offline Lock**. Prevent conflicts by forcing a transaction to obtain a lock on data before using it.
- **Query Object**. An object that represents a database query.
- **Repository**. An in-memory representation of a data source that works with domain entities.
- **Table Data Gateway**. An object that acts as a gateway to a table or view in a data source and centralizes all of the select, insert, update, and delete queries.
- **Table Module**. A single component that handles the business logic for all rows in a database table or view.
- **Transaction Script**. Organize the business logic for each transaction in a single procedure, making calls directly to the database or through a thin database wrapper.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology and techniques depending on the type of application you are designing and the requirements of that application:

- If you require basic support for queries and parameters, consider using ADO.NET objects directly.
- If you require support for more complex data-access scenarios, or need to simplify your data access code, consider using the Enterprise Library Data Access Application Block.
- If you are building a data-driven Web application with pages based on the data model of the underlying database, consider using ASP.NET Dynamic Data.
- If you want to manipulate XML-formatted data, consider using the classes in the **System.Xml** namespace and its subsidiary namespaces.
- If you are using ASP.NET to create user interfaces, consider using a DataReader to access data to maximize rendering performance. DataReaders are ideal for read-only, forward-only operations in which each row is processed quickly.
- If you are accessing Microsoft SQL Server, consider using classes in the ADO.NET SqlClient namespace to maximize performance.
- If you are accessing Microsoft SQL Server 2008, consider using a FILESTREAM for greater flexibility in the storage and access of BLOB data.
- If you are designing an object-oriented business layer based on the Domain Model pattern, consider using the ADO.NET Entity Framework.

patterns & practices Solution Assets

For information about patterns & practices solution assets, see the following resources:

- Enterprise Library - *Data Access Application Block* at http://msdn.microsoft.com/en-us/library/cc309504.aspx
- *Performance Testing Guidance* at http://www.codeplex.com/PerfTesting/Wiki/View.aspx?title=Home

Additional Resources

For more information on general data access guidelines, see the following resources:

- *Typing, storage, reading, and writing BLOBs* at http://msdn.microsoft.com/en-us/library/ms978510.aspx#daag_handlingblobs
- *Using stored procedures instead of SQL statements* at http://msdn.microsoft.com/en-us/library/ms978510.aspx.
- *.NET Data Access Architecture Guide* at http://msdn.microsoft.com/en-us/library/ms978510.aspx.
- *Data Patterns* at http://msdn.microsoft.com/en-us/library/ms998446.aspx.
- *Designing Data Tier Components and Passing Data Through Tiers* at http://msdn.microsoft.com/en-us/library/ms978496.aspx

Chapter 13: Service Layer Guidelines

Objectives

- Understand how the service layer fits into the application architecture.
- Understand the components of the service layer.
- Learn the steps for designing the service layer.
- Learn the common issues faced when designing the service layer.
- Learn the key guidelines for designing the service layer.
- Learn the key patterns and technology considerations for designing the service layer.

Overview

When providing application functionality through services, it is important to separate the service functionality into a separate service layer. Within the service layer, you define the service interface, implement the service interface, and provide translator components that translate data formats between the business layer and external data contracts. One of the more important concepts to keep in mind is that a service should never expose internal entities that are used by the business layer. Figure 1 shows where a service layer fits into the overall design of your application.

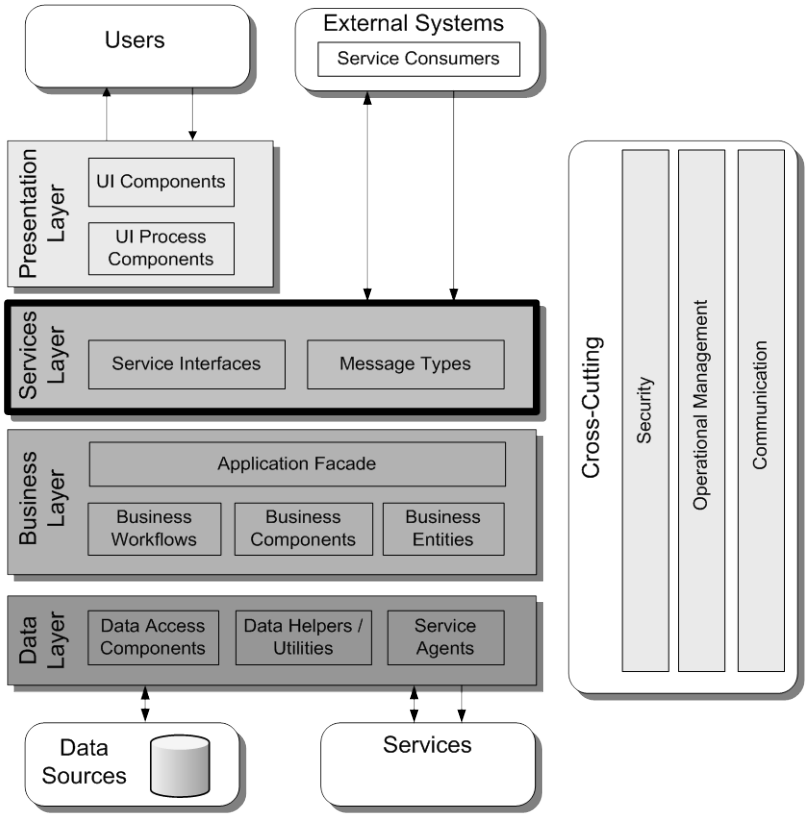


Figure 1 An overall view of a typical application showing the service layer

Service Layer Components

The service layer is made up of the following components:

- **Service interfaces**. Services expose a service interface to which all inbound messages are sent. The definition of the set of messages that must be exchanged with a service, in order for the service to perform a specific business task, constitutes a contract. You can think of a service interface as a façade that exposes the business logic implemented in the service to potential consumers.
- **Message types**. When exchanging data across the service layer, data structures are wrapped by message structures that support different types of operations. For example, you might have a Command message, a Document message, or another type of message. These message types are the "message contracts" for communication between service consumers and providers.

Approach

The approach used to design a service layer starts by defining the service interface, which consists of the contracts that you plan to expose from your service. Once the service interface has been defined, the next step is to design the service implementation; which is used to translate data contracts into business entities and to interact with the business layer.

The following steps can be used when designing a service layer:

- Define the Data and Message contracts that represent the schema used for messages.
- Define the Service contracts that represent operations supported by your service.
- Define the Fault contracts that return error information to consumers of the service.
- Design transformation objects that translate between business entities and data contracts.
- Design the abstraction approach used to interact with the business layer.

Design Considerations

There are many factors that you should consider when designing the service layer. Many of these design considerations relate to proven practices concerned with layered architectures. However, with a service, you must take into account message-related factors. The main thing to consider is that a service uses message-based interaction, which is inherently slower than object-based interaction. In addition, messages passed between a service and a consumer can be routed, modified, or lost, which requires a design that will account for the non-deterministic behavior of messaging.

Consider the following guidelines when designing the service layer:

- **Design services to be application-scoped and not component-scoped**. Service operations should be coarse-grained and focused on application operations. For example, with demographics data, you should provide an operation that returns all of the data in one call. You should not use multiple operations to return subsets of the data with multiple calls.
- **Design entities for extensibility**. In other words, data contracts should be designed so that you can extend them without affecting consumers of the service.

- **Compose entities from standard elements**. When possible, use standard elements to compose the complex types used by your service.
- **Use a layered approach to designing services**. Separate the business rules and data access functions into distinct components where appropriate.
- **Avoid tight coupling across layers**. Use abstraction to provide an interface into the business layer. This abstraction can be implemented by using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation and business layers.
- **Design without the assumption that you know who the client is**. You should not make assumptions about the client, or about how they plan to use the service that you provide.
- **Design only for the service contract**. In other words, you should not implement functionality that is not reflected by the service contract. In addition, the implementation of a service should never be exposed to external consumers.
- **Design to assume the possibility of invalid requests**. You should never assume that all messages received by the service will be valid.
- **Separate functional business concerns from infrastructure operational concerns**. Cross-cutting logic should never be combined with application logic. Doing so can lead to implementations that are difficult to extend and maintain.
- **Ensure that the service can detect and manage repeated messages (idempotency)**. When designing the service, implement well-known patterns to ensure that duplicate messages are not processed.
- **Ensure that the service can manage messages arriving out of order (commutativity)**. If there is a possibility that messages will arrive out of order, implement a design that will store messages and then process them in the correct order.
- **Consider versioning of contracts**. A new version for service contracts mean new operations exposed by the service, whereas for data contracts it means the addition of new schema type definitions.

Service Layer Frame

There are several common issues that you must consider as your develop your service layer design. These issues can be categorized into specific areas of the design. The following table lists the common issues for each category where mistakes are most often made.

Table 1 Service Layer Frame

Area	Key issues
Authentication and Authorization	• Lack of authentication across trust boundaries. • Lack of authorization across trust boundaries. • Granular or improper authorization.
Communication	• Incorrect choice of transport protocol. • Use of a chatty service communication interface. • Failing to protect sensitive data.

Area	Key issues
Exception Management	• Not catching exceptions that can be handled. • Not logging exceptions. • Not dealing with message integrity when an exception occurs.
Messaging Channels	• Choosing an inappropriate message channel • Failing to handle exception conditions on the channel. • Providing access to non-messaging clients.
Message Construction	• Failing to handle time-sensitive message content. • Incorrect message construction for the operation. • Passing too much data in a single message.
Message Endpoint	• Not supporting idempotent operations. • Not supporting commutative operations. • Subscribing to an endpoint while disconnected.
Message Protection	• Not protecting sensitive data. • Failure to use message security to protect messages that cross multiple servers. • Not considering data integrity.
Message Routing	• Not choosing the appropriate router design. • Ability to access a specific item from a message. • Ensuring that messages are handled in the correct order.
Message Transformation	• Performing unnecessary transformations. • Implementing transformations at the wrong point. • Using a canonical model when not necessary.
REST	• Implementing state within the service. • Overusing POST statements. • Putting actions in the URI. • Using hypertext to manage state.
SOAP	• Not choosing the appropriate security model. • Not planning for fault conditions. • Using complex types in the message schema.

Authentication

Designing an effective authentication strategy for your service layer is important for the security and reliability of your application. Failure to design a good authentication strategy can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attacks.

Consider the following guidelines when designing an authentication strategy:

- Identify a suitable mechanism for securely authenticating users.
- Consider the implications of using different trust settings for executing service code.
- Ensure that secure protocols such as Secure Sockets Layer (SSL) are used with Basic authentication, or when credentials are passed as plain text.
- Consider using secure mechanisms such as WS Security with SOAP messages.

Authorization

Designing an effective authorization strategy for your service layer is important for the security and reliability of your application. Failure to design a good authorization strategy can leave your application vulnerable to information disclosure, data tampering, and elevation of privileges.

Consider the following guidelines when designing an authorization strategy:

- Set appropriate access permissions on resources for users, groups, and roles.
- Use URL authorization and/or file authorization when using Windows authentication.
- Where appropriate, restrict access to publicly accessible Web methods by using declarative principle permission demands.
- Execute services under the most restrictive account that is appropriate.

Communication

When designing the communication strategy for your service, the protocol you choose should be based on the deployment scenario your service must support. If the service will be deployed within a closed network, you can use Transmission Control Protocol (TCP) for more efficient communications. If the service will be deployed into a public-facing network, you should choose the HyperText Transfer Protocol (HTTP) protocol.

Consider the following guidelines when designing a communication strategy:

- Determine how to handle unreliable or intermittent communication.
- Consider using dynamic URL behavior with configured endpoints for maximum flexibility.
- Validate endpoint addresses in messages.
- Determine whether you need to make asynchronous calls.
- Determine if you need request-response or duplex communication.
- Decide if message communication must be one-way or two-way.

Exception Management

Designing an effective exception-management strategy for your service layer is important for the security and reliability of your application. Failure to do so can make your application vulnerable to denial of service (DoS) attacks, and can also allow it to reveal sensitive and critical information.
Raising and handling exceptions is an expensive operation, so it is important for the design to take into account the potential impact on performance. A good approach is to design a centralized exception management and logging mechanism, and consider providing access points that support instrumentation and centralized monitoring in order to assist system administrators.

Consider the following guidelines when designing an exception-management strategy:

- Do not use exceptions to control business logic.
- Design a strategy for handling unhandled exceptions.
- Do not reveal sensitive information in exception messages or log files.

- Use SOAP Fault elements or custom extensions to return exception details to the caller.
- Disable tracing and debug-mode compilation for all services, except during development and testing.

Messaging Channels

Communication between a service and its consumers consists of sending data through a channel. In most cases, you will use channels provided by your chosen service infrastructure, such as Windows Communication Foundation (WCF). You must understand which patterns your chosen infrastructure supports, and determine the appropriate channel for interaction with consumers of the service.

Consider the following guidelines when designing message channels:

- Determine appropriate patterns for messaging channels, such as Channel Adapter, Messaging Bus, and Messaging Bridge.
- Determine how you will intercept and inspect the data between endpoints if necessary.

Message Construction

When data is exchanged between a service and consumer, it must be wrapped inside a message. The format of that message is based on the types of operations you need to support. For example, you may be exchanging documents, executing commands, or raising events. When using slow message-delivery channels, you should also consider using expiration information in the message.

Consider the following guidelines when designing a message-construction strategy:

- Determine the appropriate patterns for message constructions, such as Command, Document, Event, and Request-Reply.
- Divide very large quantities of data into smaller chunks, and send them in sequence.
- Include expiration information in messages that are time-sensitive. The service should ignore expired messages.

Message Endpoint

The message endpoint represents the connection that applications use to interact with your service. The implementation of your service interface represents the message endpoint. When designing the service implementation, you must consider the possibility that duplicate or invalid messages can be sent to your service.

Consider the following guidelines when designing message endpoints:

- Determine relevant patterns for message endpoints, such as Gateway, Mapper, Competing Consumers, and Message Dispatcher.
- Determine if you should accept all messages or implement a filter to handle specific messages.

- Design for idempotency in your message interface. Idempotency is the situation where you could receive duplicate messages from the same consumer, but should only handle one. In other words, an idempotent endpoint will guarantee that only one message will be handled, and all duplicate messages will be ignored.
- Design for commutativity in your message interface. Commutativity is related to the order in which messages are received. In some cases, you may need to store inbound messages so that they can be processed in the correct order.
- Design for disconnected scenarios. For instance, you might need to support guaranteed delivery.

Message Protection

When transmitting sensitive data between a service and its consumer, you should design for message protection. You can use transport layer protection or message-based protection. However, in most cases, you should use message-based protection. For example, you should encrypt sensitive sections within a message and use a signature to protect the message from tampering.

Consider the following guidelines when designing message protection:

- If interactions between the service and the consumer are not routed through other services, you can use transport layer security, such as SSL.
- If the message passes through one or more servers, always use message-based protection. In addition, you can also use transport layer security with message-based security. With transport layer security, the message is decrypted and then encrypted at each server it passes through, which represents a security risk.
- Consider using both transport layer and message-based security in your design.
- Use encryption to protect sensitive data in messages.
- Consider using digital signatures to protect messages and parameters from tampering.

Message Routing

A message router is used to decouple a service consumer from the service implementation. There are three main types of routers that you might use: simple, composed, and pattern-based. Simple routers use a single router to determine the final destination of a message. Composed routers combine multiple simple routers to handle more complex message flows. Architectural patterns are used to describe different routing styles based on simple message routers.

Consider the following guidelines when designing message routing:

- Determine relevant patterns for message routing, such as Aggregator, Content-Based Router, Dynamic Router, and Message Filter.
- If sequential messages are sent from a consumer, the router must ensure that they are all delivered to the same endpoint in the required order (commutativity).

- A message router will normally inspect information in the message to determine how to route the message. As a result, you must ensure that the router can access that information.

Message Transformation

When passing messages between a service and consumer, there are many cases where the message must be transformed into a format that the consumer can understand. This normally occurs in cases where non–message-based consumers need to process data from a message-based system. You can use adapters to provide access to the message channel for a non–message-based consumer, and translators to convert the message data into a format that the consumer understands.

Consider the following guidelines when designing message transformation:

- Determine relevant patterns for message transformation, such as Canonical Data Mapper, Envelope Wrapper, and Normalizer.
- Use metadata to define the message format.
- Consider using an external repository to store the metadata.

REST

Representational State Transfer (REST) represents an architecture style for distributed systems. It is designed to reduce complexity by dividing a system into resources. The resources and the operations supported by a resource are represented and exposed as a set of URIs over the HTTP protocol.

Consider the following guidelines when designing REST resources:

- Identify and categorize resources that will be available to clients.
- Choose an approach for resource representation. A good practice would be to use meaningful names for REST starting points and unique identifiers, such as a globally unique identifier (GUID), for specific resource instances. For example, http://www.contoso.com/employee/ represents an employee starting point. http://www.contoso.com/employee/8ce762d5-b421-6123-a041-5fbd07321bac4 uses a GUID that represents a specific employee.
- Decide if multiple representations should be supported for different resources. For example, you can decide if the resource should support an XML, Atom, or JSON format and make it part of the resource request. A resource could be exposed as both http://www.contoso.com/example.atom and http://www.contoso.com/example.json
- Decide if multiple views should be supported for different resources. For example, decide if the resource should support GET and POST operations, or only GET operations.

Service Interface

The service interface represents the contract exposed by your service. When designing a service interface, you should consider boundaries that must be crossed and the type of consumers who

will be accessing your service. For instance, service operations should be coarse-grained and application scoped. One of the biggest mistakes with service interface design is to treat the service as a component with fine-grained operations. This results in a design that requires multiple calls across physical or process boundaries, which are very expensive in terms of performance and latency.

Consider the following guidelines when designing a service interface:
- Consider using a coarse-grained interface to batch requests and minimize the number of calls over the network.
- Design service interfaces in such a way that changes to the business logic do not affect the interface.
- Do not implement business rules in a service interface.
- Consider using standard formats for parameters to provide maximum compatibility with different types of clients.
- Do not make assumptions in your interface design about the way that clients will use the service.
- Do not use object inheritance to implement versioning for the service interface.

SOAP

SOAP is a message-based protocol that is used to implement the message layer of a service. The message is composed of an envelope that contains a header and body. The header can be used to provide information that is external to the operation being performed by the service. For instance, a header may contain security, transaction, or routing information. The body contains contracts, in the form of XML schemas, which are used to implement the service.

Consider the following guidelines when designing SOAP messages:
- Define the schema for the operations that can be performed by a service.
- Define the schema for the data structures passed with a service request.
- Define the schema for the errors or faults that can be returned from a service request.

Deployment Considerations

The service layer can be deployed on the same tier as other layers of the application, or on a separate tier in cases where performance and isolation requirements demand this. However, in most cases the service layer will reside on the same physical tier as the business layer in order to minimize performance impact when exposing business functionality.

Consider the following guidelines when deploying the service layer:
- Deploy the service layer to the same tier as the business layer to improve application performance, unless performance and security issues inherent within the production environment prevent this.
- If the service is located on the same physical tier as the service consumer, consider using the named pipes or shared memory protocols.

- If the service is accessed only by other applications within a local network, consider using TCP for communications.
- If the service is publicly accessible from the Internet, use HTTP for your transport protocol.

Pattern Map

Key patterns are organized by the key categories detailed in the Service Layer Frame in the following table. Consider using these patterns when making design decisions for each category.

Table 2 Pattern Map

Category	Relevant patterns
Communication	• Duplex • Fire and Forget • Reliable Sessions • Request Response
Messaging Channels	• Channel Adapter • Message Bus • Messaging Bridge • Point-to-point Channel • Publish-subscribe Channel
Message Construction	• Command Message • Document Message • Event Message • Request-Reply
Message Endpoint	• Competing Consumer • Durable Subscriber • Idempotent Receiver • Message Dispatcher • Messaging Gateway • Messaging Mapper • Polling Consumer • Selective Consumer • Service Activator • Transactional Client
Message Protection	• Data Confidentiality • Data Integrity • Data Origin Authentication • Exception Shielding • Federation • Replay Protection • Validation

Category	Relevant patterns
Message Routing	• Aggregator • Content-Based Router • Dynamic Router • Message Broker (Hub-and-Spoke) • Message Filter • Process Manager
Message Transformation	• Canonical Data Mapper • Claim Check • Content Enricher • Content Filter • Envelope Wrapper • Normalizer
REST	• Behavior • Container • Entity • Store • Transaction
Service Interface	• Remote Façade
SOAP	• Data Contracts • Fault Contracts • Service Contracts

- For more information on the Duplex and Request Reponse patterns, see “Designing Service Contracts” at http://msdn.microsoft.com/en-us/library/ms733070.aspx
- For more information on the Atomic and Cross-Service Transaction patterns, see “WS-* Specifications” at http://www.ws-standards.com/ws-atomictransaction.asp
- For more information on the Command, Document Message, Event Message, Durable Subscriber, Idempotent Receiver, Polling Consumer, and Transactional Client patterns, see “Messaging Patterns in Service-Oriented Architecture, Part I” at http://msdn.microsoft.com/en-us/library/aa480027.aspx
- For more information on the Data Confidentiality and Data Origin Authentication patterns, see “Chapter 2: Message Protection Patterns” at http://msdn.microsoft.com/en-us/library/aa480573.aspx
- For more information on the Replay Detection, Exception Shielding, and Validation patterns, see “Chapter 5: Service Boundary Protection Patterns” at http://msdn.microsoft.com/en-us/library/aa480597.aspx
- For more information on the Claim Check, Content Enricher, Content Filter, and Envelope Wrapper patterns, see ”Messaging Patterns in Service Oriented Architecture, Part 2” at http://msdn.microsoft.com/en-us/library/aa480061.aspx
- For more information on the Remote Façade pattern, see “P of EAA: Remote Façade” at http://martinfowler.com/eaaCatalog/remoteFacade.html
- For more information on *REST* patterns such as Behavior, Container, and Entity, see “REST Patterns” at http://wiki.developer.mindtouch.com/REST/REST_Patterns

- For more information on the Aggregator, Content-Based Router, Publish-Subscribe, Message Bus, and Point-to-Point patterns, see "Messaging patterns in Service-Oriented Architecture, Part I" at http://msdn.microsoft.com/en-us/library/aa480027.aspx

Pattern Descriptions

- **Aggregator**. A filter that collects and stores individual related messages, combines these messages, and publishes a single aggregated message to the output channel for further processing.
- **Behavior (REST)**. Applies to resources that carry out operations. These resources generally contain no state of their own, and only support the POST operation.
- **Canonical Data Mapper**. Uses a common data format to perform translations between two disparate data formats.
- **Channel Adapter**. A component that can access the application's API or data and publish messages on a channel based on this data, and that can receive messages and invoke functionality inside the application.
- **Claim Check**. Retrieves data from a persistent store when required.
- **Command Message**. Provides a message structure used to support commands.
- **Competing Consumer**. Sets multiple consumers on a single message queue and have them compete for the right to process the messages, which allows the messaging client to process multiple messages concurrently.
- **Container**. Builds on the entity pattern by providing the means to dynamically add and/or update nested resources.
- **Content Enricher**. Enriches messages with missing information obtained from an external data source.
- **Content Filter**. Removes sensitive data from a message and reduces network traffic by removing unnecessary data from a message.
- **Content-Based Router**. Routes each message to the correct consumer based on the contents of the message; such as existence of fields, specified field values, and so on.
- **Data Confidentiality**. Uses message-based encryption to protect sensitive data in a message.
- **Data Contract**. A schema that defines data structures passed with a service request.
- **Data Integrity**. Ensures that messages have not been tampered with in transit.
- **Data Origin Authentication**. Validates the origin of a message as an advanced form of data integrity.
- **Document Message**. A structure used to reliably transfer documents or a data structure between applications.
- **Duplex**. Two-way message communication where both the service and the client send messages to each other independently, irrespective of the use of the one-way or the request/reply pattern.
- **Durable Subscriber**. In a disconnected scenario, messages are saved and then made accessible to the client when connecting to the message channel in order to provide guaranteed delivery.

- **Dynamic Router**. A component that dynamically routes the message to a consumer after evaluating the conditions/rules that the consumer has specified.
- **Entity**. **(REST)**. Resources that can be read with a GET operation, but can only be changed by PUT and DELETE operations.
- **Envelope Wrapper**. A wrapper for messages that contains header information used, for example, to protect, route, or authenticate a message.
- **Event Message**. A structure that provides reliable asynchronous event notification between applications.
- **Exception Shielding**. Prevents a service from exposing information about its internal implementation when an exception occurs.
- **Façade**. Implements a unified interface to a set of operations in order to provide a simplified reduce coupling between systems.
- **Fault Contracts**. A schema that defines errors or faults that can be returned from a service request.
- **Federation**. An integrated view of information distributed across multiple services and consumers.
- **Fire and Forget**. A one-way message communication mechanism used when no response is expected.
- **Idempotent Receiver**. Ensures that a service will only handle a message once.
- **Message Broker (Hub-and-Spoke)**. A central component that communicates with multiple applications to receive messages from multiple sources, determines the correct destination, and route the message to the correct channel.
- **Message Bus**. Structures the connecting middleware between applications as a communication bus that enables the applications to work together using messaging.
- **Message Dispatcher**. A component that sends messages to multiple consumers.
- **Message Filter**. Eliminates undesired messages, based on a set of criteria, from being transmitted over a channel to a consumer.
- **Messaging Bridge**. A component that connects messaging systems and replicates messages between these systems.
- **Messaging Gateway**. Encapsulates message-based calls into a single interface in order to separate it from the rest of the application code.
- **Messaging Mapper**. Transforms requests into business objects for incoming messages, and reverses the process to convert business objects into response messages.
- **Normalizer**. Converts or transforms data into a common interchange format when organizations use different formats.
- **Point-to-point Channel**. Sends a message on a Point-to-Point Channel to ensure that only one receiver will receive a particular message.
- **Polling Consumer**. A service consumer that checks the channel for messages at regular intervals.
- **Process Manager**. A component that enables routing of messages through multiple steps in a workflow.
- **Publish-subscribe Channel**. Creates a mechanism to send messages only to the applications that are interested in receiving the messages without knowing the identity of the receivers.

- **Reliable Sessions**. End-to-end reliable transfer of messages between a source and a destination, regardless of the number or type of intermediaries that separate endpoints.
- **Remote Façade**. Creates a high-level unified interface to a set of operations or processes in a remote subsystem to make that subsystem easier to use, by providing a course-grained interface over fine-grained operations to minimize calls across the network.
- **Replay Protection**. Enforces message idempotency by preventing an attacker from intercepting a message and executing it multiple times.
- **Request Response**. A two-way message communication mechanism where the client expects to receive a response for every message sent.
- **Request-Reply**. Uses separate channels to send the request and reply.
- **Selective Consumer**. The service consumer uses filters to receive messages that match specific criteria.
- **Service Activator**. A service that receives asynchronous requests to invoke operations in business components.
- **Service Contract**. A schema that defines operations that the service can perform.
- **Service Interface**. A programmatic interface that other systems can use to interact with the service.
- **Store (REST)**. Allows entries to be created and updated with PUT.
- **Transaction (REST)**. Resources that support transactional operations.
- **Transactional Client**. A client that can implement transactions when interacting with a service.
- **Validation**. Checks the content and values in messages to protect a service from malformed or malicious content.

Technology Considerations

The following guidelines will help you to choose an appropriate implementation technology for your service layer:

- Consider using ASP.NET Web services (ASMX) for simplicity, but only when a suitable Web server will be available.
- Consider using WCF services for advanced features and support for multiple transport protocols.
- If you are using ASMX and you require message-based security and binary data transfer, consider using Web Service Extensions (WSE).
- If you are using WCF and you want interoperability with non-WCF or non-Windows clients, consider using HTTP transport based on SOAP specifications.
- If you are using WCF and you want to support clients within an intranet, consider using the TCP protocol and binary message encoding with transport security and Windows authentication.
- If you are using WCF and you want to support WCF clients on the same machine, consider using the named pipes protocol and binary message encoding.
- If you are using WCF, consider defining service contracts that use an explicit message wrapper instead of an implicit one. This allows you to define message contracts as inputs

and outputs for your operations, which then allows you to extend the data contracts included in the message contract without affecting the service contract.

Additional Resources

- For more information, see *Enterprise Solution Patterns Using Microsoft .NET* at http://msdn.microsoft.com/en-us/library/ms998469.aspx.
- For more information, see *Web Service Security Guidance* at http://msdn.microsoft.com/en-us/library/aa480545.aspx
- For more information, see *Improving Web Services Security: Scenarios and Implementation Guidance for WCF* at http://www.codeplex.com/WCFSecurityGuide

Reference Chapter 21
Software Architecture in Practice
Third Edition
Len Bass
Paul Clements
Rick Kazman

Architecture Evaluation through Architecture Trade off Analysis Method

Harvinder S Jabbal
SEZG651/SSZG653 Software Architectures

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

SEZG651/ SSZG653
Software Architectures
Module 5-CS 06

Chapter Outline

innovate achieve lead

- Evaluation Factors
- The Architecture Tradeoff Analysis Method (ATAM)
- Lightweight Architecture Evaluation
- Summary

Evaluation Factors

Three Forms of Evaluation

innovate achieve lead

Evaluation by the designer within the design process.

Evaluation by peers within the design process.

Analysis by outsiders once the architecture has been designed.

Evaluation by the Designer

Every time the designer makes a key design decision or completes a design milestone, the chosen and competing alternatives should be evaluated.

Evaluation by the designer is the "test" part of the "generate-and-test" approach to architecture design.

How much analysis? This depends on the importance of the decision. Factors include:

- *The importance of the decision*. The more important the decision, the more care should be taken in making it and making sure it's right.
- *The number of potential alternatives.* The more alternatives, the more time could be spent in evaluating them. Try to eliminate alternatives quickly so that the number of viable potential alternatives is small.
- *Good enough as opposed to perfect.* Many times, two possible alternatives do not differ dramatically in their consequences. In such a case, it is more important to make a choice and move on with the design process than it is to be absolutely certain that the best choice is being made. Again, do not spend more time on a decision than it is worth.

Peer Review

Architectural designs can be peer reviewed, just as code can.

A peer review can be carried out at any point of the design process where a candidate architecture, or at least a coherent reviewable part of one, exists.

Allocate at least several hours and possibly half a day.

Peer Review Steps

innovate achieve lead

Step 1

- *The reviewers determine a number of quality attribute scenarios to drive the review.* These scenarios can be developed by the review team or by additional stakeholders.

Step 2

- *The architect presents the portion of the architecture to be evaluated.* The reviewers individually ensure that they understand the architecture. Questions at this point are specifically for understanding.

Step 3

- *For each scenario, the designer walks through the architecture and explains how the scenario is satisfied.* The reviewers ask questions to determine (a) that the scenario is, in fact, satisfied and (b) whether any of the other scenarios being considered will not be satisfied.

Step 4

- *Potential problems are captured.* Real problems must either must be fixed or a decision must be explicitly made by the designers and the project manager that they are willing to accept the problems and its probability of occurrence.

Analysis by Outsiders

Outside evaluators can cast an objective eye on an architecture.

"Outside" is relative; this may mean

- outside the development project
- outside the business unit where the project resides but within the same company
- outside the company altogether.

Outsiders are chosen because they possess specialized knowledge or experience, or long experience successfully evaluating architectures.

Managers tend to be more inclined to listen to problems uncovered by an outside team.

An outside team tends to be used to evaluate complete architectures.

Contextual Factors for Evaluation

innovate achieve lead

What artifacts are available?

- To perform an architectural evaluation, there must be an artifact that describes the architecture.

Who sees the results?

- Some evaluations are performed with the full knowledge and participation of all of the stakeholders. Others are performed more privately.

Who performs the evaluation?

- Evaluations can be carried out by an individual or a team.

Which stakeholders will participate?

- The evaluation process should provide a method to elicit the goals and concerns that the important stakeholders have regarding the system. Identifying the individuals who are needed and assuring their participation in the evaluation is critical.

What are the business goals?

- The evaluation should answer whether the system will satisfy the business goals.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) has been used for over a decade to evaluate software architectures in domains ranging from automotive to financial to defense.

The ATAM is designed so that evaluators need not be familiar with the architecture or its business goals, the system need not yet be constructed, and there may be a large number of stakeholders.

Participants in the ATAM

The evaluation team.

- External to the project whose architecture is being evaluated.
- Three to five people; a single person may adopt several roles in an ATAM.
- They need to be recognized as competent, unbiased outsiders.

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 may be present (or represented) as well.
- The architect is always included – the architect must willingly participate.

Architecture stakeholders.

- Stakeholders have a vested interest in the architecture performing as advertised.
- Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and, possibly, others.
- Their job is to articulate the specific quality attribute goals that the architecture should meet.
- Expect to enlist 12 to 15 stakeholders for the evaluation of a large enterprise-critical architecture.

ATAM Evaluation Team Roles

innovate achieve lead

Role

Responsibilities

Team leader

- Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; forms evaluation team; sees that final report is produced and delivered (although the writing may be delegated)

Evaluation leader

- Runs evaluation; facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture; facilitates on-site analysis

Scenario scribe

- Writes scenarios on flipchart or whiteboard during scenario elicitation; captures agreed-on wording of each scenario, halting discussion until exact wording is captured

Proceedings scribe

- Captures proceedings in electronic form on laptop or workstation: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and resolution of each scenario when applied to architecture(s); also generates a printed list of adopted scenarios for handout to all participants

Questioner

- Raises issues of architectural interest, usually related to the quality attributes in which he or she has expertise

Outputs of the ATAM

1. A concise presentation of the architecture. The architecture is presented in one hour
2. Articulation of the business goals. Frequently, the business goals presented in the ATAM are being seen by some of the assembled participants for the first time and these are captured in the outputs.
3. Prioritized quality attribute requirements expressed as quality attribute scenarios. These quality attribute scenarios take the form described in Chapter 4.
4. A set of risks and nonrisks.
   - A risk is defined as an architectural decision that may lead to undesirable consequences in light of quality attribute requirements.
   - A nonrisk is an architectural decision that, upon analysis, is deemed safe.
   - The identified risks form the basis for an architectural risk mitigation plan.

Outputs of the ATAM

innovate achieve lead

5.
- A set of risk themes. When the analysis is complete, the evaluation team examines the full set of discovered risks to look for overarching 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 business goals.

6.
- Mapping of architectural decisions to quality requirements. For each quality attribute scenario examined during an ATAM, those architectural decisions that help to achieve it are determined and captured.

7.
- A set of identified sensitivity and tradeoff points. These are architectural decisions that have a marked effect on one or more quality attributes.

Intangible Outputs

There are also *intangible* results of an ATAM-based evaluation. These include

- a sense of community on the part of the stakeholders
- open communication channels between the architect and the stakeholders
- a better overall understanding on the part of all participants of the architecture and its strengths and weaknesses.

While these results are hard to measure,

- they are no less important than the others and often are the longest-lasting.

Phases of the ATAM

Phase	Activity	Participants	Typical duration
0	Partnership and preparation: Logistics, planning, stakeholder recruitment, team formation	Evaluation team leadership and key project decision-makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation: Steps 1-6	Evaluation team and project decision-makers	1-2 days followed by a hiatus of 2-3 weeks
2	Evaluation: Steps 7-9	Evaluation team, project decision makers, stakeholders	2 days
3	Follow-up: Report generation and delivery, process improvement	Evaluation team and evaluation client	1 week

Step 1: Present the ATAM

The evaluation leader presents the ATAM to the assembled project representatives.

This time is used to explain the process that everyone will be following, to answer questions, and to set the context and expectations for the remainder of the activities.

Using a standard presentation, the leader describes the ATAM steps in brief and the outputs of the evaluation.

Step 2: Present Business Drivers

innovate achieve lead

Everyone involved in the evaluation needs to understand the context for the system and the primary business drivers motivating its development.

In this step, a project decision maker (ideally the project manager or the system's customer) presents a system overview from a business perspective.

The presentation should describe the following:

- The system's most important functions
- Any relevant technical, managerial, economic, or political constraints
- The business goals and context as they relate to the project
- The major stakeholders
- The architectural drivers (that is, the architecturally significant requirements)

Step 3: Present the Architecture

The lead architect (or architecture team) makes a presentation describing the architecture.

The architect covers technical constraints such as operating system, hardware, or middleware prescribed for use, and other systems with which the system must interact.

The architect describes the architectural approaches (or patterns, or tactics, if the architect is fluent in that vocabulary) used to meet the requirements.

The architect's presentation should convey the essence of the architecture and not stray into ancillary areas or delve too deeply into the details of just a few aspects.

The architect should present the views that he or she found most important during the creation of the architecture and the views that help to reason about the most important quality attribute concerns of the system.

Step 4: Identify Architectural Approaches

innovate achieve lead

The ATAM focuses on analyzing an architecture by understanding its architectural approaches, especially patterns and tactics.

By now, the evaluation team will have a good idea of what patterns and tactics the architect used in designing the system.

- They will have studied the architecture documentation
- They will have heard the architect's presentation in step 3.
- The team should also be adept at spotting approaches not mentioned explicitly

The evaluation team simply catalogs the patterns and tactics that have been identified.

The list is publicly captured and will serve as the basis for later analysis.

Step 5: Generate Utility Tree

The quality attribute goals are articulated in detail via a quality attribute utility tree.

Utility trees serve to make the requirements concrete by defining precisely the relevant quality attribute requirements that the architects were working to provide.

The important quality attribute goals for the architecture under consideration were named in step 2.

In this step, the evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals.

These are expressed as scenarios, which populate the leaves of the utility tree.

The scenarios are assigned a rank of importance (High, Medium, Low).

Step 6: Analyze Architectural Approaches

The evaluation team examines the highest-ranked scenarios one at a time; the architect is asked to explain how the architecture supports each one.

Evaluation team members—especially the questioners—probe for the architectural approaches that the architect used to carry out the scenario.

Along the way, the evaluation team documents the relevant architectural decisions and identifies and catalogs their risks, nonrisks, sensitivity points, and tradeoffs. Examples:

- Risk: The frequency of heartbeats affects the time in which the system can detect a failed component. Some assignments will result in unacceptable values of this response.
- Sensitivity point: The number of simultaneous database clients will affect the number of transactions that a database can process per second.
- Tradeoff: The heartbeat frequency determines the time for detecting a fault. Higher frequency leads to better availability but consumes more processing time and communication bandwidth (potentially reducing performance).

These, in turn, may catalyze a deeper analysis.

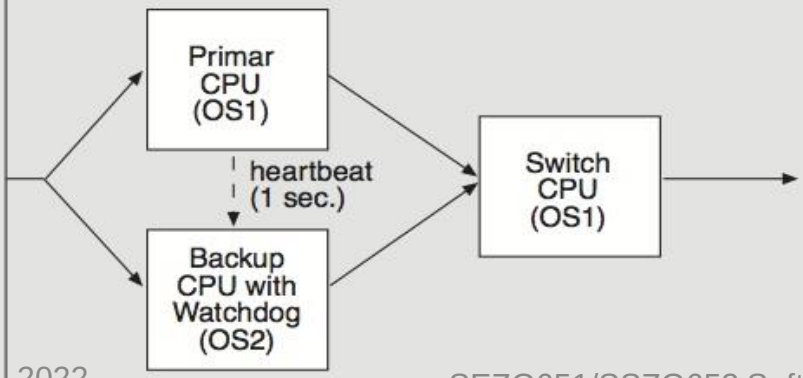
The analysis is not meant to be comprehensive. The key is to elicit sufficient architectural information to establish a link between the architectural decisions made and the quality attribute requirements that need to be satisfied.

innovate achieve lead

Example of an Analysis

Scenario #: A12	Scenario: Detect and recover from HW failure of main switch.
Attribute(s)	Availability
Environment	Normal operations
Stimulus	One of the CPUs fails
Response	0.999999 availability of switch

Architectural decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14

Reasoning	Ensures no common mode failure by using different hardware and operating system (see Risk 8) Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog Watchdog is simple and has proved reliable Availability requirement might be at risk due to lack of backup data channel ... (see Risk 9)
Architecture diagram	

Step 7: Brainstorm and Prioritize Scenarios

The stakeholders brainstorm scenarios that are operationally meaningful with respect to the stakeholders' individual roles.

- A maintainer will likely propose a modifiability scenario
- A user will probably come up with a scenario that expresses useful functionality or ease of operation
- A quality assurance person will propose a scenario about testing the system or being able to replicate the state of the system leading up to a fault.

The purpose of scenario brainstorming is to take the pulse of the larger stakeholder community: to understand what system success means for them.

Once the scenarios have been collected, they are prioritized by voting.

The list of prioritized scenarios is compared with those from the utility tree exercise.

- If they agree, it indicates good alignment between what the architect had in mind and what the stakeholders actually wanted.
- If additional driving scenarios are discovered—and they usually are—this may itself be a risk, if the discrepancy is large. This would indicate that there was some disagreement in the system's important goals between the stakeholders and the architect.

Step 8: Analyze Architectural Approaches

In this step the evaluation team performs the same activities as in step 6, using the highest-ranked, newly generated scenarios.

The evaluation team guides the architect in the process of carrying out the highest ranked new scenarios.

The architect explains how relevant architectural decisions contribute to realizing each one.

This step might cover the top 5-10 scenarios, as time permits.

Step 9: Present Results

The evaluation team confers privately to group risks into risk themes, based on some common underlying concern or systemic deficiency.

- For example, a group of risks about inadequate or out-of-date documentation might be grouped into a risk theme stating that documentation is given insufficient consideration.
- A group of risks about the system's inability to function in the face of various hardware and/or software failures might lead to a risk theme about insufficient attention to backup capability or providing high availability.

For each risk theme, the evaluation team identifies which of the business drivers listed in step 2 are affected.

- This elevates the risks that were uncovered to the attention of management, who cares about the business drivers

Step 9: Present Results

The collected information from the evaluation is summarized and presented to stakeholders.

The following outputs are presented:

- The architectural approaches documented
- The set of scenarios and their prioritization from the brainstorming
- The utility tree
- The risks discovered
- The nonrisks documented
- The sensitivity points and tradeoff points found
- Risk themes and the business drivers threatened by each one

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Lightweight Architectural Evaluation

Lightweight Architectural Evaluation

An ATAM is a substantial undertaking.

- It requires some 20 to 30 person-days of effort from an evaluation team, plus even more for the architect and stakeholders.
- Investing this amount of time makes sense on a large and costly project, where the risks of making a major mistake in the architecture are unacceptable.

We have developed a Lightweight Architecture Evaluation method, based on the ATAM, for smaller, less risky projects.

- May take place in a single day, or even a half-day meeting.
- May be carried out entirely by members internal to the organization.
- Of course this lower level of scrutiny and objectivity may not probe the architecture as deeply.

Because the participants are all internal to the organization and fewer in number than for the ATAM, giving everyone their say and achieving a shared understanding takes much less time.

The steps and phases of a Lightweight Architecture Evaluation can be carried out more quickly.

Typical Agenda: 4-6 Hours

innovate achieve lead

Step	Time	Notes
1. Present the ATAM	0 hours	Participants already familiar with process.
2. Present business drivers	0.25 hours	The participants are expected to understand the system and its business goals and their priorities. A brief review ensures that these are fresh in everyone's mind and that there are no surprises.
3. Present architecture	0.5 hours	All participants are expected to be familiar with the system. A brief overview of the architecture, using at least module and C&C views, is presented. 1-2 scenarios are traced through these views.
4. Identify architectural approaches	0.25 hours	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5. Generate QA utility tree	0.5- 1.5 hours	Scenarios might exist: part of previous evaluations, part of design, part of requirements elicitation. Put these in a tree. Or, a utility tree may already exist.
6. Analyze architectural approaches	2-3 hours	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.
7. Brainstorm scenarios	0 hours	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8. Analyze architectural approaches	0 hours	This step is also omitted, since all analysis is done in step 6.
9. Present results	0.5 hours	At the end of an evaluation, the team reviews the existing and newly discovered risks, nonrisks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.

Summary

If a system is important enough for you to explicitly design its architecture, then that architecture should be evaluated.

The number of evaluations and the extent of each evaluation may vary from project to project.

- A designer should perform an evaluation during the process of making an important decision.
- Lightweight evaluations can be performed several times during a project as a peer review exercise.

The ATAM is a comprehensive method for evaluating software architectures. It works by having project decision makers and stakeholders articulate a precise list of quality attribute requirements (in the form of scenarios) and by illuminating the architectural decisions relevant to carrying out each high-priority scenario. The decisions can then be understood in terms of risks or non-risks to find any trouble spots in the architecture.

Lightweight Architecture Evaluation, based on the ATAM, provides an inexpensive, low-ceremony architecture evaluation that can be carried out in an afternoon.

Thank you……..

Credits

- **Chapter Reference from Text T1: 21**
- Slides have been adapted from Authors Slides Software Architecture in Practice – Third Ed.
  - Len Bass
  - Paul Clements
  - Rick Kazman

19

Architecture, Implementation, and Testing

You don't make progress by standing on the sidelines, whimpering and complaining. You make progress by implementing ideas.
—Shirley Hufstedler

Although this is a book about software architecture—you've noticed that by now, no doubt—we need to remind ourselves from time to time that architecture is not a goal unto itself, but only the means to an end. Building systems from the architecture is the end game, systems that have the qualities necessary to meet the concerns of their stakeholders.

This chapter covers two critical areas in system-building—implementation and testing—from the point of view of architecture. What is the relationship of architecture to implementation (and vice versa)? What is the relationship of architecture to testing (and vice versa)?

19.1 Architecture and Implementation

Architecture is intended to serve as the blueprint for implementation. The sidebar "Potayto, Potahto . . ." makes the point that architectures and implementations rely on different sets of vocabulary, which results in development tools usually serving one community or the other fairly well, but not both. Frequently the implementers are so engrossed in their immediate task at hand that they make

implementation choices that degrade the modular structure of the architecture, for example.

This leads to one of the most frustrating situations for architects. It is very easy for code and its intended architecture to drift apart; this is sometimes called "architecture erosion." This section talks about four techniques to help keep the code and the architecture consistent.

Embedding the Design in the Code

A key task for implementers is to faithfully execute the prescriptions of the architecture. George Fairbanks, in *Just Enough Architecture*, prescribes using an "architecturally-evident coding style." Throughout the code, implementers can document the architectural concept or guidance that they're reifying. That is, they can "embed" the architecture in their implementations. They can also try to localize the implementation of each architectural element, as opposed to scattering it across different implementation entities.

This practice is made easier if implementers (consistently across a project) adopt a set of conventions for how architectural concepts "show up" in code. For example, identifying the layer to which a code unit belongs will make it more likely that implementers and maintainers will respect (and hence not violate) the layering.

Frameworks

"Framework" is a terribly overused term, but here we mean a reusable set of libraries or classes for a software system. "Library" and "class" are implementation-like terms, but frameworks have broad architectural implications—they are a place where architecture and implementation meet. The classes (in an object-oriented framework) are appropriate to the application domain of the system that is being constructed. Frameworks can range from small and straightforward (such as ones that provide a set of standard and commonly used data types to a system) to large and sophisticated. For example, the AUTomotive Open System ARchitecture (AUTOSAR) is a framework for automotive software, jointly developed by automobile manufacturers, suppliers, and tool developers.

Frameworks that are large and sophisticated often encode architectural interaction mechanisms, by encoding how the classes (and the objects derived from them) communicate and synchronize with each other. For example, AUTOSAR is an architecture and not (just) an architecture framework.

A framework amounts to a substantial (in some cases, enormous) piece of reusable software, and it brings with it all of the advantages of reuse: saving time and cost, avoiding a costly design task, encoding domain knowledge, and decreasing the chance of errors from individual implementers coding the same thing differently and erroneously. On the other hand, frameworks are difficult to

design and get correct. Adopting a framework means investing in a selection process as well as training, and the framework may not provide all the functionality that you require. The learning curve for a framework is often extremely steep. A framework that provides a complete set of functionality for implementing an application in a particular domain is called a "platform."

Code Templates

A template provides a structure within which some architecture-specific functionality is achieved, in a consistent fashion system-wide. Many code generators, such as user interface builders, produce a template into which a developer inserts code, although templates can also be provided by the development environment.

Suppose that an architecture for a high-availability system prescribes that every component that implements a critical responsibility must use a failover technique that switches control to a backup copy of itself in case a fault is detected in its operation.

The architecture could, and no doubt would, describe the failover protocol. It might go something like this:

In the event that a failure is detected in a critical-application component, a switchover occurs as follows:

1. A secondary copy, executing in parallel in background on a different processor, is promoted to the new primary.
2. The new primary reconstitutes with the application's clients by sending them a message that means, essentially: The operational unit that was serving you has had a failure. Were you waiting for anything from us at the time? It then proceeds to service any requests received in response.
3. A new secondary is started to serve as a backup for the new primary.
4. The newly started secondary announces itself to the new primary, which starts sending it messages as appropriate to keep it up to date while it is executing in background.

If failure is detected within a secondary, a new one is started on some other processor. It coordinates with its primary and starts receiving state data.

Even though the primary and secondary copies are never doing the same thing at the same time (the primary is performing its duty and sending state updates to its backups, and the secondaries are waiting to leap into action and accepting state updates), both components come from identical copies of the same source code.

To accomplish this, the coders of each critical component would be expected to implement that protocol. However, a cleverer way is to give the coder a code template that contains the tricky failover part as boilerplate and contains fill-in-the-blank sections where coders can fill in the implementation for the functionality that is unique to each application. This template could be embedded in

the development environment so that when the developer specifies that the module being developed is to support a failover protocol, the template appears as the initial code for the module.

An example of such a template, taken from an air traffic control system, is illustrated in Figure 19.1. The structure is a continuous loop that services incoming events. If the event is one that causes the application to take a normal (non-fault-tolerance-related) action, it carries out the appropriate action, followed by an update of its backup counterparts' data so that the counterpart can take over if necessary. Most applications spend most of their time processing normal events. Other events that may be received involve the transfer (transmission and reception) of state and data updates. Finally, there is a set of events that involves both the announcement that this unit has become the primary and requests from clients for services that the former (now failed) primary did not complete.

Using a template has architectural implications: it makes it simple to add new applications to the system with a minimum of concern for the actual workings of the fault-tolerant mechanisms designed into the approach. Coders and maintainers of applications do not need to know about message-handling mechanisms except abstractly, and they do not need to ensure that their applications are fault tolerant—that has been handled architecturally.

Code templates have implications for reliability: once the template is debugged, then entire classes of coding errors across the entire system disappear. But in the context of this discussion, templates represent a true common ground where the architecture and the implementation come together in a consistent and useful fashion.

Keeping Code and Architecture Consistent

Code can drift away from architecture in a depressingly large number of ways. First, there may be no constraints imposed on the coders to follow the architecture. This makes no apparent sense, for why would we bother to invest in an architecture if we aren't going to use it to constrain the code? However, this happens more often than you might think. Second, some projects use the published architecture to start out, but when problems are encountered (either technical or schedule-related), the architecture is abandoned and coders scramble to field the system as best they can. Third (and perhaps most common), after the system has been fielded, changes to it are accomplished with code changes only, but these changes affect the architecture. However, the published architecture is not updated to guide the changes, nor updated afterward to keep up with them.

One simple method to remedy the lack of updating the architecture is to not treat the published architecture as an all-or-nothing affair—it's either all correct or all useless. Parts of the architecture may become out of date, but it will help enormously if those parts are marked as "no longer applicable" or "to be revised." Conscientiously marking sections as out of date keeps the architecture

documentation a living document and (paradoxically) sends a stronger message about the remainder: it is still correct and can still be trusted.

```
terminate:= false
initialize application/application protocols
ask for current state (image request)
Loop
Get_event
Case Event_Type is
-- "normal" (non-fault-tolerant-related) requests to
-- perform actions; only happens if this unit is the
-- current primary address space
when X => Process X
Send state data updates to other address spaces
when Y => Process Y
Send state data updates to other address spaces
...
when Terminate_Directive => clean up resources; terminate
          := true
when State_Data_Update => apply to state data
-- will only happen if this unit is a secondary address
-- space, receiving the update from the primary after it
-- has completed a "normal" action sending, receiving
-- state data
when Image_Request => send current state data to new
          address space
when State_Data_Image => Initialize state data
when Switch_Directive => notify service packages of
          change in rank
-- these are requests that come in after a PAS/SAS
-- switchover; they report services that they had
-- requested from the old (failed) PAS which this unit
-- (now the PAS) must complete. A, B, etc. are the names
-- of the clients.
when Recon_from_A => reconstitute A
when Recon_from_B => reconstitute B
...
when others => log error
end case
exit when terminate
end loop
```

FIGURE 19.1 A code template for a failover protocol. "Process X" and "Process Y" are placeholders for application-specific code.

In addition, strong management and process discipline will help prevent erosion. One way is to mandate that changes to the system, no matter when they occur, are vetted through the architecture first. The alternatives for achieving code alignment with the architecture include the following:

- *Sync at life-cycle milestone.* Developers change the code until the end of some phase, such as a release or end of an iteration. At that point, when the schedule pressure is less, the architecture is updated.
- *Sync at crisis.* This undesirable approach happens when a project has found itself in a technical quagmire and needs architectural guidance to get itself going again.
- *Sync at check-in.* Rules for the architecture are codified and used to vet any check-in. When a change to the code "breaks" the architecture rules, key project stakeholders are informed and then either the code or the architecture rules must be modified. This process is typically automated by tools.

These alternatives can work only if the implementation follows the architecture mostly, departing from it only here and there and in small ways. That is, it works when syncing the architecture involves an update and not a wholesale overhaul or do-over.

Potayto, Potahto, Tomayto, Tomahto— Let's Call the Whole Thing Off!

One of the most vexing realities about architecture-based software development is the gulf between architectural and implementation *ontologies*, the set of concepts and terms inherent in an area. Ask an architect what concepts they work with all day, and you're likely to hear things like modules, components, connectors, stakeholders, evaluation, analysis, documentation, views, modeling, quality attributes, business goals, and technology roadmaps.

Ask an implementer the same question, and you likely won't hear any of those words. Instead you'll hear about objects, methods, algorithms, data structures, variables, debugging, statements, code comments, compilers, generics, operator overloading, pointers, and build scripts.

This is a gap in language that reflects a gap in concepts. This gap is, in turn, reflected in the languages of the tools that each community uses. UML started out as a way to model object-oriented designs that could be quickly converted to code—that is, UML is conceptually "close" to code. Today it is a *de facto* architecture description language, and likely the most popular one. But it has no built-in concept for the most ubiquitous of architectural concepts, the layer. If you want to represent layers in UML, you have to adopt some convention to do it. Packages stereotyped as <<layer>>, associated with stereotyped <<allowed to use>> dependencies do the trick. But it *is* a trick, a workaround for a language deficiency. UML has "connectors," two of

them in fact. But they are a far cry from what architects think of as connectors. Architectural connectors can and do have rich functionality. For instance, an enterprise service bus (ESB) in a service-oriented architecture handles routing, data and format transformation, technology adaptation, and a host of other work. It is most natural to depict the ESB as a connector tying together services that interact with each other through it. But UML connectors are impoverished things, little more than bookkeeping mechanisms that have no functionality whatsoever. The delegation connector in UML exists merely to associate the ports of a parent component with ports of its nested children, to send inputs from the outside into a child's input port, and outputs from a child to the output port of the parent. And the assembly connector simply ties together one component's "requires" interface with another's "provides" interface. These are no more than bits of string to tie two components together. To represent a true architectural connector in UML, you have to adopt a convention—another workaround—such as using simple associations tagged with explanatory annotations, or abandon the architectural concept completely and capture the functionality in another component.

Part of the concept gap between architecture and implementation is inevitable. Architectures, after all, are abstractions of systems and their implementations. Back in Chapter 2, we said that was one of the valuable properties of architecture: you could build many different systems from one. And that's what an abstraction is: a one-to-many mapping. One abstraction, many instances; one architecture, many implementations. That architecture is an abstraction of implementation is almost its whole point: architecture lets us achieve intellectual control over a system without having to capture, let alone master, all of the countless and myriad truths about its implementation.

And here comes the gap again: All of those truths about its implementation are what coders produce for a living, without which the system remains but an idea. Architects, on the other hand, dismiss all of that reality by announcing that they are not interested in implementation "details."

Can't we all get along?

We could. There is nothing inherently impossible about a language that embraces architectural as well as coding concepts, and several people have proposed some. But UML is beastly difficult to change, and programming language purveyors all seem to focus their attention down on the underlying machine and not up to the architecture that is directing the implementation.

Until this gap is resolved, until architects and coders (and their tools) speak the same conceptual language, we are likely to continue to deal with the most vexing result of this most vexing reality: writing code (or introducing a code change) that ignores the architecture is the easiest thing in the world.

The good news is that even though architecture and implementation speak different languages, they aren't languages from different planets. Concepts in one ontology usually correspond pretty well to concepts in another. Frameworks are an area where the languages enjoy a fair amount of overlap. So are interfaces. These constructs live on the cusp of the two domains, and provide hope that we might one day speak the same language.

—PCC

19.2 Architecture and Testing

What is the relationship between architecture and testing? One possible answer is "None," or "Not much." Testing can be seen as the process of making sure that a software system meets its requirements, that it brings the necessary functionality (endowed with the necessary quality attributes) to its user community. Testing, seen this way, is simply connected to requirements, and hardly connected to architecture at all. As long as the system works as expected, who cares what the architecture is? Yes, the architecture played the leading role in *getting* the system to work as expected, thank you very much, but once it has played that role it should make a graceful exit off the stage. Testers work with requirements: Thanks, architecture, but we'll take it from here.

Not surprisingly, we don't like that answer. This is an impoverished view of testing, and in fact an unrealistic one as well. As we'll see, architecture *cannot help* but play an important role in testing. Beyond that, though, we'll see that architecture can help make testing less costly and more effective when embraced in testing activities. We'll also see what architects can do to help testers, and what testers can do to take advantage of the architecture.

Levels of Testing and How Architecture Plays a Role in Each

There are "levels" of testing, which range from testing small, individual pieces in isolation to an entire system.

- *Unit testing* refers to tests run on specific pieces of software. Unit testing is usually a part of the job of implementing those pieces. In fact, unit tests are typically written by developers themselves. When the tests are written before developing the unit, this practice is known as test-driven development.

  Certifying that a unit has passed its unit tests is a precondition for delivery of that unit to integration activities. Unit tests test the software in a standalone fashion, often relying on "stubs" to play the role of other units with which the tested unit interacts, as those other units may not yet be available. Unit tests won't usually catch errors dealing with the interaction between elements—that comes later—but unit tests provide confidence that each of the system's building blocks is exhibiting as much correctness as is possible on its own.

  A unit corresponds to an architectural element in one of the architecture's module views. In object-oriented software, a unit might correspond to a class. In a layered system, a unit might correspond to a layer, or a part of a layer. Most often a unit corresponds to an element at the leaf of a module decomposition tree.

  Architecture plays a strong role in unit testing. First, it defines the units: they are architectural elements in one or more of the module views. Second, it defines the responsibilities and requirements assigned to each unit.

Modifiability requirements can also be tested at unit test time. How long it will take to make specified changes can be tested, although this is seldom done in practice. If specified changes take too long for the developers to make, imagine how long they will take when a new and separate maintenance group is in charge without the intimate knowledge of the modules.

Although unit testing goes beyond architecture (tests are based on nonarchitectural information such as the unit's internal data structures, algorithms, and control flows), they cannot begin their work without the architecture.

- *Integration testing* tests what happens when separate software units start to work together. Integration testing concentrates on finding problems related to the interfaces between elements in a design. Integration testing is intimately connected to the specific increments or subsets that are planned in a system's development.

  The case where only one increment is planned, meaning that integration of the entire system will occur in a single step, is called "big bang integration" and has largely been discredited in favor of integrating many incrementally larger subsets. Incremental integration makes locating errors much easier, because any new error that shows up in an integrated subset is likely to live in whatever new parts were added this time around.

  At the end of integration testing, the project has confidence that the pieces of software work together correctly and provide at least some correct system-wide functionality (depending on how big a subset of the system is being integrated). Special cases of integration testing are these:

  - System testing, which is a test of all elements of the system, including software and hardware in their intended environment
  - Integration testing that involves third-party software

  Once again, architecture cannot help but play a strong role in integration testing. First, the increments that will be subject to integration testing must be planned, and this plan will be based on the architecture. The uses view is particularly helpful for this, as it shows what elements must be present for a particular piece of functionality to be fielded. That is, if the project requires that (for example) in the next increment of a social networking system users will be able to manage photographs they've allowed other users to post in their own member spaces, the architect can report that this new functionality is part of the `user_permissions` module, which will use a new part of the `photo_sharing` module, which in turn will use a new structure in the master `user_links` database, and so forth. Project management will know, then, that all of the software must be ready for integration at the same time.

  Second, the interfaces between elements are part of the architecture, and those interfaces determine the integration tests that are created and run.

  Integration testing is where runtime quality attribute requirements can be tested. Performance and reliability testing can be accomplished. A

sophisticated test harness is useful for performing these types of tests. How long does an end-to-end synchronization of a local database with a global database take? What happens if faults are injected into the system? What happens when a process fails? All of these conditions can be tested at integration time.

Integration testing is also the time to test what happens when the system runs for an extended period. You could monitor resource usage during the testing and look for resources that are consumed but not freed. Does your pool of free database connections decrease over time? Then maybe database connections should be managed more aggressively. Does the thread pool show signs of degradation over time? Ditto.

- *Acceptance testing* is a kind of system testing that is performed by users, often in the setting in which the system will run. Two special cases of acceptance testing are alpha and beta testing. In both of these, users are given free rein to use the system however they like, as opposed to testing that occurs under a preplanned regimen of a specific suite of tests. Alpha testing usually occurs in-house, whereas beta testing makes the system available to a select set of end users under a "User beware" proviso. Systems in beta test are generally quite reliable—after all, the developing organization is highly motivated to make a good first impression on the user community—but users are given fair warning that the system might not be bug-free or (if "bug-free" is too lofty a goal) at least not up to its planned quality level.

  Architecture plays less of a role in acceptance testing than at the other levels, but still an important one. Acceptance testing involves stressing the system's quality attribute behavior by running it at extremely heavy loads, subjecting it to security attacks, depriving it of resources at critical times, and so forth. A crude analogy is that if you want to bring down a house, you can whale away at random walls with a sledgehammer, but your task will be accomplished much more efficiently if you consult the architecture first to find which of the walls is holding up the roof. (The point of testing is, after all, to "bring down the house.")

Overlaying all of these types of testing is regression testing, which is testing that occurs after a change has been made to the system. The name comes from the desire to uncover old bugs that might resurface after a change, a sign that the software has "regressed" to a less mature state. Regression testing can occur at any of the previously mentioned levels, and often consists of rerunning the bank of tests and checking for the occurrence of old (or for that matter, new) faults.

Black-Box and White-Box Testing

Testing (at any level) can be "black box" or "white box." Black-box testing treats the software as an opaque "black box," not using any knowledge about the

internal design, structure, or implementation. The tester's only source of information about the software is its requirements.

Architecture plays a role in black-box testing, because it is often the architecture document where the requirements for a piece of the system are described. An element of the architecture is unlikely to correspond one-to-one with a requirement nicely captured in a requirements document. Rather, when the architect creates an architectural element, he or she usually assigns it an amalgamation of requirements, or partial requirements, to carry out. In addition, the interface to an element also constitutes a set of "requirements" for it—the element must happily accept the specified parameters and produce the specified effect as a result. Testers performing black-box testing on an architectural element (such as a major subsystem) are unlikely to be able to do their jobs using only requirements published in a requirements document. They need the architecture as well, because the architecture will help the tester understand what portions of the requirements relate to the specified subsystem.

White-box testing makes full use of the internal structures, algorithms, and control and data flows of a unit of software. Tests that exercise all control paths of a unit of software are a primary example of white-box testing. White-box testing is most often associated with unit testing, but it has a role at higher levels as well. In integration testing, for example, white-box testing can be used to construct tests that attempt to overload the connection between two components by exploiting knowledge about how a component (for example) manages multiple simultaneous interactions.

Gray-box testing lies, as you would expect, between black and white. Testers get to avail themselves of some, but not all, of the internal structure of a system. For example, they can test the interactions between components but not employ tests based on knowledge of a component's internal data structures.

There are advantages and disadvantages with each kind of testing. Black-box testing is not biased by a design or implementation, and it concentrates on making sure that requirements are met. But it can be inefficient by (for example) running many unit tests that a simple code inspection would reveal to be unnecessary. White-box testing often keys in on critical errors more quickly, but it can suffer from a loss of perspective by concentrating tests to make the implementation break, but not concentrating on the software delivering full functionality under all points in its input space.

Risk-based Testing

Risk-based testing concentrates effort on areas where risk is perceived to be the highest, perhaps because of immature technologies, requirements uncertainty, developer experience gaps, and so forth. Architecture can inform risk-based testing by contributing categories of risks to be considered. Architects can identify areas where architectural decisions (if wrong) would have a widespread impact, where

architectural requirements are uncertain, quality attributes are demanding on the architecture, technology selections risky, or third-party software sources unreliable. Architecturally significant requirements are natural candidates for risk-based test cases. If the architecturally significant requirements are not met, then the system is unacceptable, by definition.

Test Activities

Testing, depending on the project, can consume from 30 to 90 percent of a development's schedule and budget. Any activity that gobbles resources as voraciously as that doesn't just happen, of course, but needs to be planned and carried out purposefully and as efficiently as possible. Here are some of the activities associated with testing:

- *Test planning.* Test activities have to be planned so that appropriate resources can be allocated. "Resources" includes time in the project schedule, labor to run the tests, and technology with which the testing will be carried out. Technology might include test tools, automatic regression testers, test script builders, test beds, test equipment or hardware such as network sniffers, and so forth.
- *Test development.* This is an activity in which the test procedures are written, test cases are chosen, test datasets are created, and test suites are scripted. The tests can be developed either before or after development. Developing the tests prior to development and then developing a module to satisfy the test is a characteristic of test-first development.
- *Test execution.* Here, testers apply the tests to the software and capture and record errors.
- *Test reporting and defect analysis.* Testers report the results of specific tests to developers, and they report overall metrics about the test results to the project's technical management. The analysis might include a judgment about whether the software is ready for release. Defect analysis is done by the development team usually along with the customer, to adjudicate disposition of each discovered fault: fix it now, fix it later, don't worry about it, and so on.
- *Test harness creation.* One of the architect's common responsibilities is to create, along with the architecture, a set of test harnesses through which elements of the architecture may be conveniently tested. Such test harnesses typically permit setting up the environment for the elements to be tested, along with controlling their state and the data flowing into and out of the elements.

Once again, architecture plays a role and informs each of these activities; the architect can contribute useful information and suggestions for each. For test planning, the architecture provides the list of software units and incremental

subsets. The architect can also provide insight as to the complexity or, if the software does not yet exist, the expected complexity of each of the software units. The architect can also suggest useful test technologies that will be compatible with the architecture; for example, Java's ability to support assertions in the code can dramatically increase software testability, and the architect can provide arguments for or against adopting that technology. For test development, the architecture can make it easy to swap datasets in and out of the system. Finally, test reporting and defect analysis are usually reported in architectural terms: *this* element passed all of its tests, but *that* element still has critical errors showing. *This* layer passed the delivery test, but *that* layer didn't. And so forth.

The Architect's Role

Here are some of the things an architect can do to facilitate quality testing. First and foremost, the architect can design the system so that it is highly testable. That is, the system should be designed with the quality attribute of testability in mind. Applying the cardinal rule of architecture ("Know your stakeholders!"), the architect can work with the test team (and, to the extent they have a stake in testing, other stakeholders) to establish what is needed. Together, they can come up with a definition of the testability requirements using scenarios, as described in Chapter 10. Testability requirements are most likely to be a concern of the developing organization and not so much of the customer or users, so don't expect to see many testing requirements in a requirements document. Using those testability requirements, the testability tactics in Chapter 10 can be brought to bear to provide the testability needed.

In addition to designing for testability, the architect can also do these other things to help the test effort:

- Insure that testers have access to the source code, design documents, and the change records.
- Give testers the ability to control and reset the entire dataset that a program stores in a persistent database. Reverting the database to a known state is essential for reproducing bugs or running regression tests. Similarly, loading a test bed into the database is helpful. Even products that don't use databases can benefit from routines to automatically preload a set of test data. One way to achieve this is to design a "persistence layer" so that the whole program is database independent. In this way, the entire database can be swapped out for testing, even using an in-memory database if desired.
- Give testers the ability to install multiple versions of a software product on a single machine. This helps testers compare versions, isolating when a bug was introduced. In distributed applications, this aids testing deployment configurations and product scalability. This capability could require configurable communication ports and provisions for avoiding collisions over resources such as the registry.

As a practical matter, the architect cannot afford to ignore the testing process because if, after delivery, something goes seriously wrong, the architect will be one of the first people brought in to diagnose the problem. In one case we heard about, this involved flying to the remote mountains of Peru to diagnose a problem with mining equipment.

19.3 Summary

Architecture plays a key role in both implementation and testing. In the implementation phase, letting future readers of the code know what architectural constructs are being used, using frameworks, and using code templates all make life easier both at implementation time and during maintenance.

During testing the architecture determines what is being tested at which stage of development. Development quality attributes can be tested during unit test and runtime quality attributes can be tested during integration testing.

Testing, as with other activities in architecture-based development, is a cost/benefit activity. Do not spend as much time testing for faults whose consequences are small and spend the most time testing for faults whose consequences are serious. Do not neglect testing for faults that show up after the system has been executing for an extended period.

19.4 For Further Reading

George Fairbanks gives an excellent treatment of architecture and implementation in Chapter 10 of his book *Just Enough Software Architecture*, which is entitled "The Code Model" [Fairbanks 10].

Mary Shaw long ago recognized the conceptual gap between architecture and implementation and wrote about it eloquently in her article "Procedure Calls Are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status" [Shaw 94]. In it she pointed out the disparity between rich connectors available in architecture and the impoverished subroutine call that is the mainstay of nearly every programming language.

Details about the AUTOSAR framework can be found at www.autosar.org.

Architecture-based testing is an active field of research. [Bertolino 96b], [Muccini 07], [Muccini 03], [Eickelman 96], [Pettichord 02], and [Binder 94] specifically address designing systems so that they are more testable. In fact, the three bullets concerning the architect's role in Section 19.2 are drawn from Pettichord's work.

Voas [Voas 95] defines testability, identifies its contributing factors, and describes how to measure it.

Bertolino extends Voas's work and ties testability to dependability [Bertolino 96].

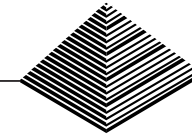
Finally, Baudry et al. have written an interesting paper that examines the testability of well-known design patterns [Baudry 03].

19.5 Discussion Questions

1. In a distributed system each computer will have its own clock. It is difficult to perfectly synchronize those clocks. How will this complicate making performance measures of distributed systems? How would you go about testing that the performance of a particular system activity is adequate?
2. Plan and implement a modification to a module. Ask your colleagues to do the same modification independently. Now compare your results to those of your colleagues. What is the mean and the standard deviation for the time it takes to make that modification?
3. List some of the reasons why an architecture and a code base inevitably drift apart. What processes and tools might address this gap? What are their costs and benefits?
4. Most user interface frameworks work by capturing events from the user and by establishing callbacks or hooks to application-specific functionality. What limitations do these architectural assumptions impose on the rest of the system?
5. Consider building a test harness for a large system. What quality attributes should this harness exhibit? Create scenarios to concretize each of the quality attributes.
6. Testing requires the presence of a test oracle, which determines the success (or failure) of a test. For scalability reasons, the oracle must be automatic. How can you ensure that your oracle is correct? How do you ensure that its performance will scale appropriately? What process would you use to record and fix faults in the testing infrastructure?
7. In embedded systems faults often occur "in the field" and it is difficult to capture and replicate the state of the system that led to its failure. What architectural mechanisms might you use to solve this problem?
8. In integration testing it is a bad idea to integrate everything all at once (big bang integration). How would you use architecture to help you plan integration increments?

This page intentionally left blank

20



Architecture Reconstruction and Conformance

It was six men of Indostan / To learning much inclined,
Who went to see the Elephant / (Though all of them were blind),
That each by observation / Might satisfy his mind.

The First approach'd the Elephant, / And happening to fall
Against his broad and sturdy side, / At once began to bawl:
"God bless me! but the Elephant / Is very like a wall!"

The Second, feeling of the tusk, / Cried, — "Ho! what have we here
So very round and smooth and sharp? / To me 'tis mighty clear
This wonder of an Elephant / Is very like a spear!"

The Third approached the animal, / And happening to take
The squirming trunk within his hands, / Thus boldly up and spake:
"I see," quoth he, "the Elephant / Is very like a snake!"

The Fourth reached out his eager hand, / And felt about the knee.
"What most this wondrous beast is like / Is mighty plain," quoth he,
"'Tis clear enough the Elephant / Is very like a tree!"

The Fifth, who chanced to touch the ear, / Said: "E'en the blindest man
Can tell what this resembles most; / Deny the fact who can,
This marvel of an Elephant / Is very like a fan!"

The Sixth no sooner had begun / About the beast to grope,
Then, seizing on the swinging tail / That fell within his scope,
"I see," quoth he, "the Elephant / Is very like a rope!"

And so these men of Indostan / Disputed loud and long,
Each in his own opinion / Exceeding stiff and strong,
Though each was partly in the right, / And all were in the wrong!
—"The Blind Men and the Elephant," by John Godfrey Saxe

Throughout this book we have treated architecture as something largely under your control and shown how to make architectural decisions to achieve the goals and requirements in place for a system under development. But there is another side to the picture. Suppose you have been given responsibility for a system that already exists, but you do not know its architecture. Perhaps the architecture was never recorded by the original developers, now long gone. Perhaps it was recorded but the documentation has been lost. Or perhaps it was recorded but the documentation is no longer synchronized with the system after a series of changes. How do you maintain such a system? How do you manage its evolution to maintain the quality attributes that its architecture (whatever it may be) has provided for us?

This chapter surveys techniques that allow an analyst to build, maintain, and understand a representation of an existing architecture. This is a process of reverse engineering, typically called architecture reconstruction. Architecture reconstruction is used, by the architect, for two main purposes:

- To document an architecture where the documentation never existed or where it has become hopelessly out of date
- To ensure conformance between the as-built architecture and the as-designed architecture.

In architecture reconstruction, the "as-built" architecture of an implemented system is reverse-engineered from existing system artifacts.

When a system is initially developed, its architectural elements are mapped to specific implementation elements: functions, classes, files, objects, and so forth. This is forward engineering. When we reconstruct those architectural elements, we need to apply the inverses of the original mappings. But how do we go about determining these mappings? One way is to use automated and semiautomated extraction tools; the second way is to probe the original design intent of the architect. Typically we use a combination of both techniques in reconstructing an architecture.

In practice, architecture reconstruction is a tool-intensive activity. Tools extract information about the system, typically by scouring the source code, but they may also analyze other artifacts as well, such as build scripts or traces from running systems. But architectures are abstractions—they can not be seen in the low-level implementation details, the programming constructs, of most systems. So we need tools that aid in building and aggregating the abstractions that we need, as architects, on top of the ground facts that we develop, as developers. If our tools are usable and accurate, the end result is an architectural representation that aids the architect in reasoning about the system. Of course, if the original architecture and its implementation are "spaghetti," the reconstruction will faithfully expose this lack of organization.

Architecture reconstruction tools are not, however, a panacea. In some cases, it may not be possible to generate a useful architectural representation. Furthermore, not all aspects of architecture are easy to automatically extract. Consider

this: there is no programming language construct in any major programming language for "layer" or "connector" or other architectural elements; we can't simply pick these out of a source code file. Similarly, architectural patterns, if used, are typically not explicitly documented in code.

Architecture reconstruction is an interpretive, interactive, and iterative process involving many activities; it is not automatic. It requires the skills and attention of both the reverse-engineering expert and, in the best case, the architect (or someone who has substantial knowledge of the architecture). And whether the reconstruction is successful or not, there is a price to pay: the tools come with a learning curve that requires time to climb.

20.1 Architecture Reconstruction Process

Architecture reconstruction requires the skillful application of tools, often with a steep learning curve. No single tool does the entire job. For one reason, there is often diversity in the number of implementation languages and dialects in which a software system is implemented—a mature MRI scanner or a legacy banking application may easily comprise more than ten different programming and scripting languages. No tool speaks every language.

Instead we are inevitably led to a "tool set" approach to support architecture reconstruction activities. And so the first step in the reconstruction process is to set up the workbench.

An architecture reconstruction workbench should be open (making it easy to integrate new tools as required) and provide an integration framework whereby new tools that are added to the tool set do not impact the existing tools or data unnecessarily.

Whether or not an explicit workbench is used, the software architecture reconstruction process comprises the following phases (each elaborated in a subsequent section):

1. *Raw view extraction.* In the raw view extraction phase, raw information about the architecture is obtained from various sources, primarily source code, execution traces, and build scripts. Each of these sets of raw information is called a view.[1]
2. *Database construction.* The database construction phase involves converting the raw extracted information into a standard form (because the various extraction tools may each produce their own form of output). This standardized form of the extracted views is then used to populate a reconstruction

1. This use of the term "view" is consistent with our definition in Chapter 18: "a representation of a set of system elements and relations among them."

database. When the reconstruction process is complete, the database will be used to generate authoritative architecture documentation.

3. *View fusion and manipulation.* The view fusion phase combines the various views of the information stored in the database. Individual views may not contain complete or fully accurate information. View fusion can improve the overall accuracy. For example, a static view extracted from source code might miss dynamically bound information such as calling relationships. This could then be combined with a dynamic view from an execution trace, which will capture all dynamically bound calling information, but which may not provide complete coverage. The combination of these views will provide higher quality information than either could provide alone. Furthermore, view creation and fusion is typically associated with some expert interpretation and manipulation. For example, an expert might decide that a group of elements should be aggregated together to form a *layer*.
4. *Architecture analysis.* View fusion will result in a set of *hypotheses* about the architecture. These hypotheses take the form of architectural elements (such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct, and that is the function of the analysis step. Some of these hypotheses might be disproven, requiring additional view extraction, fusion, and manipulation.

The four phases of architecture reconstruction are iterative. Figure 20.1 depicts the major tasks of architecture reconstruction and their relationships and outputs. Solid lines represent data flow and dashed lines represent human interaction.

All of these activities are greatly facilitated by engaging people who are familiar with the system. They can provide insights about what to look for—that is, what views are amenable to extraction—and provide a guided approach to view fusion and analysis. They can also point out or explain exceptions to the design rules (which will show up as violations of the hypotheses during the analysis phase). If the experts are long gone, reconstruction is still possible, but it may well require more backtracking from incorrect initial guesses.

20.2 Raw View Extraction

Raw view extraction involves analyzing a system's existing design and implementation artifacts to construct one or more models of it. The result is a set of information that is used in the view fusion activity to construct more-refined views of the system that directly support the goals of the reconstruction, goals such as these:

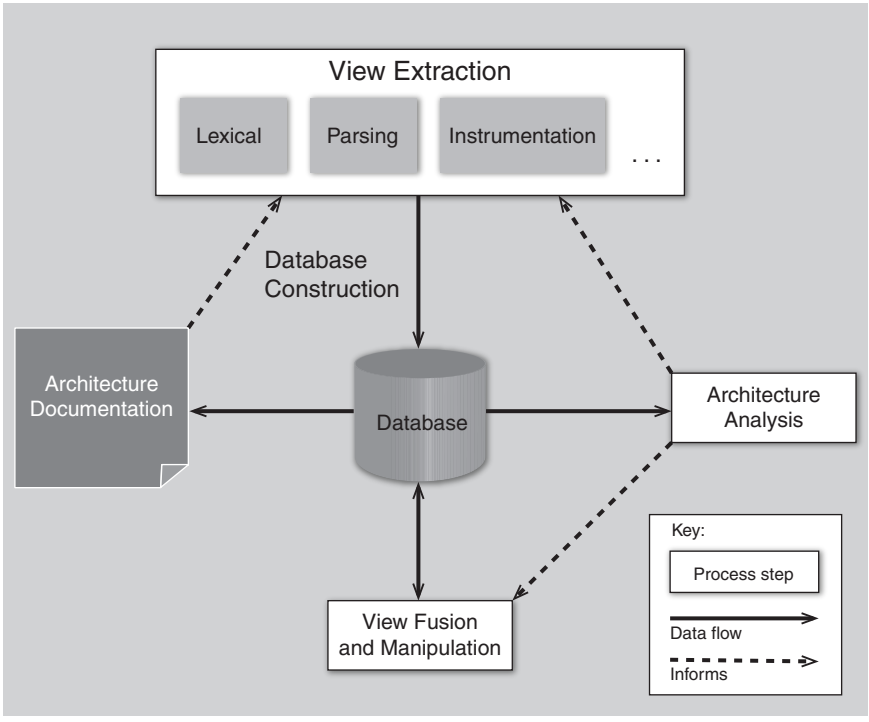


FIGURE 20.1 Architecture reconstruction process

- Extracting and representing a target set of architectural views, to support the overall architecture documentation effort.
- Answering specific questions about the architecture. For example, "What components are potentially affected if I choose to rewrite component X?" or "How can I refactor my layering to remove cyclic dependencies?"

The raw view extraction process is a blend of the ideal (what information do you want to discover about the architecture that will most help you meet the goals of your reconstruction effort?) and the practical (what information can your available tools actually extract and present?).

From the source artifacts (code, header files, build files, and so on) and other artifacts (e.g., execution traces), you can identify and capture the elements of interest within the system (e.g., files, functions, variables) and their relationships to obtain several base system views. Table 20.1 shows a typical list of the elements and several relationships among them that might be extracted.

TABLE 20.1 Examples of Extracted Elements and Relations

Source Element	Relation	Target Element	Description
File	`includes`	File	C preprocessor `#include` of one file by another
File	`contains`	Function	Definition of a function in a file
File	`defines_var`	Variable	Definition of a variable in a file
Directory	`contains`	Directory	Directory contains a subdirectory
Directory	`contains`	File	Directory contains a file
Function	`calls`	Function	Static function call
Function	`access_read`	Variable	Read access on a variable
Function	`access_write`	Variable	Write access on a variable

Each of the relationships between the elements gives different information about the system:

- The `calls` relationship between functions helps us build a call graph.
- The `includes` relationship between the files gives us a set of dependencies between system files.
- The `access_read` and `access_write` relationships between functions and variables show us how data is used. Certain functions may write a set of data and others may read it. This information is used to determine how data is passed between various parts of the system. We can determine whether or not a global data store is used or whether most information is passed through function calls.
- Certain elements or subsystems may be stored in particular directories, and capturing relations such as `dir_contains_file` and `dir_contains_dir` is useful when trying to identify elements later.
- If the system to be reconstructed is object oriented, classes and methods are added to the list of elements to be extracted, and relationships such as `class_is_subclass_of_class` and `class_contains_method` are extracted and used.

Information obtained can be categorized as either static or dynamic. Static information is obtained by observing only the system artifacts, while dynamic information is obtained by observing how the system runs. The goal is to fuse both to create more accurate system views.

If the architecture of the system changes at runtime, that runtime configuration should be captured and used when carrying out the reconstruction. For

example, in some systems a configuration file is read in by the system at startup, or a newly started system examines its operating environment, and certain elements are executed or connections are made as a result.

Another reason to capture dynamic information is that some architecturally relevant information may not exist in the source artifacts because of late binding. Examples of late binding include the following:

- Polymorphism
- Function pointers
- Runtime parameterization
- Plug-ins
- Service interactions mediated by brokers

Further, the precise topology of a system may not be determined until runtime. For example, in peer-to-peer systems, service-oriented architectures, and cloud computing, the topology of the system is established dynamically, depending on the availability, loading, and even dynamic pricing of system resources. The topology of such systems cannot be directly recovered from their source artifacts and hence cannot be reverse-engineered using static extraction tools.

Therefore, it may be necessary to use tools that can generate dynamic information about the system (e.g., profiling tools, instrumentation that generates runtime traces, or aspects in an aspect-oriented programming language that can monitor dynamic activity). Of course, this requires that such tools be available on the platforms on which the system executes. Also, it may be difficult to collect the results from code instrumentation. For example, embedded systems often have no direct way to output such information.

Table 20.2 summarizes some of the common categories of tools that might be used to populate the views loaded into the reconstruction database.

Tools to analyze design models, build files, and executables can also be used to extract further information as required. For instance, build files include information on module or file dependencies that exist within the system, and this information may not be reflected in the source code, or anywhere else.

An additional activity that is often required prior to loading a raw view into the database is to prune irrelevant information. For example, in a C code base there may be several `main()` routines, but only one of those (and its resulting call graph) will be of concern for analysis. The others may be for test harnesses and other utility functions. Similarly if you are building or using libraries that are operating-system specific, you may only be interested in a specific OS (e.g., Linux) and thus want to discard the libraries for other platforms.

TABLE 20.2 Tool Categories for Populating Reconstructed Architecture Views

Tool	Static or Dynamic	Description
Parsers	Static	Parsers analyze the code and generate internal representations from it (for the purpose of generating machine code). It is possible to save this internal representation to obtain a view.
Abstract Syntax Tree (AST) Analyzers		AST analyzers do a similar job to parsers, but they build an explicit tree representation of the parsed information. We can build analysis tools that traverse the AST and output selected pieces of architecturally relevant information in an appropriate format.
Lexical Analyzers		Lexical analyzers examine source artifacts purely as strings of lexical elements or tokens. The user of a lexical analyzer can specify a set of code patterns to be matched and output. Similarly, a collection of ad hoc tools such as grep and Perl can carry out pattern matching and searching within the code to output some required information. All of these tools—code-generating parsers, AST-based analyzers, lexical analyzers, and ad hoc pattern matchers—are used to output static information.
Profilers	Dynamic	Profiling and code coverage analysis tools can be used to output information about the code as it is being executed, and usually do not involve adding new code to the system.
Code Instrumentation Tools		Code instrumentation, which has wide applicability in the field of testing, involves adding code to the system to output specific information while the system is executing. Aspects, in an aspect-oriented programming language, can serve the same purpose and have the advantage of keeping the instrumentation code separate from the code being monitored.

20.3 Database Construction

Some of the information extracted from the raw view extraction phase, while necessary for the process of reconstruction, may be too specific to aid in architectural understanding. Consider Figure 20.2. In this figure we show a set of facts extracted from a code base consisting of classes and methods, and inclusion and calling relations. Each element is plotted on a grid and each relation is drawn as a

line between the elements. This view, while accurate, provides no insight into the overarching abstractions or coarse-grained structures present in the architecture.

Thus we need to manipulate such raw views, to collapse information (for example, hiding methods inside class definitions), and to show abstractions (for example, showing all of the connections between business objects and user interface objects, or identifying distinct layers).

It is helpful to use a database to store the extracted information because the amount of information being stored is large, and the manipulations of the data are tedious and error-prone if done manually. Some reverse-engineering tools, such as Lattix, SonarJ, and Structure101, fully encapsulate the database, and so the user of the tool need not be concerned with its operation. However, those who are using a suite of tools together—a workbench—will need to choose a database and decide on internal representations of the views.

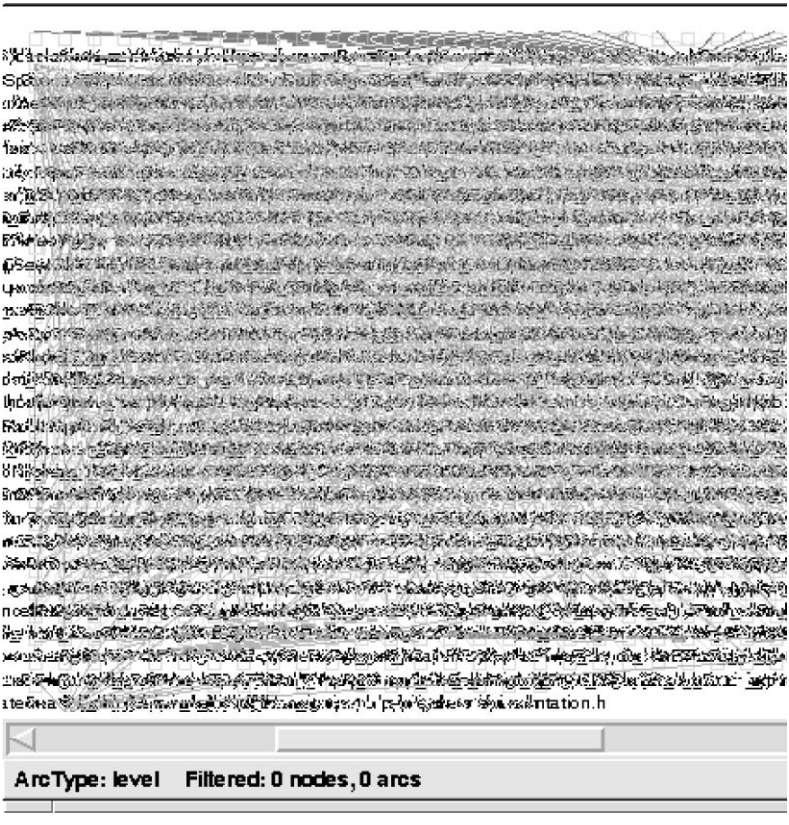


FIGURE 20.2 A raw extracted view: white noise

20.4 View Fusion

Once the raw facts have been extracted and stored in a database, the reconstructor can now perform view fusion. In this phase, the extracted views are manipulated to create *fused* views. Fused views combine information from one or more extracted views, each of which may contain specialized information. For example, a static call view might be fused with a dynamic call view. One might want to combine these two views because a static call view will show all explicit calls (where method A calls method B) but will miss calls that are made via late binding mechanisms. A dynamically extracted call graph will never miss a call that is made during an execution, but it suffers the "testing" problem: it will only report results from those paths through the system that are traversed during its execution. So a little-used part of the system—perhaps for initialization or error recovery—might not show up in the dynamic view. Therefore we fuse these two views to produce a more complete and more accurate graph of system relationships.

The process of creating a fused view is the process of creating a hypothesis about the architecture and a visualization of it to aid in analysis. These hypotheses result in new aggregations that show various abstractions or clusterings of the elements (which may be source artifacts or previously identified abstractions). By interpreting these fused views and analyzing them, it is possible to produce hypothesized architectural views of the system. These views can be interpreted, further refined, or rejected. There are no universal completion criteria for this process; it is complete when the architectural representation is sufficient to support the analysis needs of its stakeholders.

For example, Figure 20.3 shows the early results of interacting with the tool SonarJ. SonarJ first extracts facts from a set of source code files (in this case, written in Java) and lets you define a set of layers and vertical slices through those layers in a system. SonarJ will then instantiate the user-specified definitions of layers and slices and populate them with the extracted software elements.

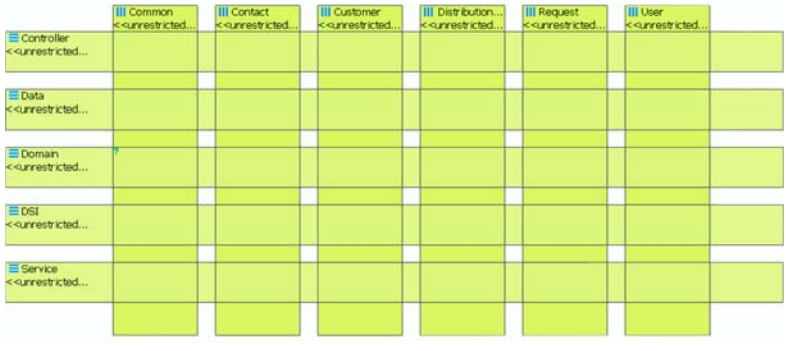


FIGURE 20.3 Hypothesized layers and vertical slices

In the figure there are five layers: Controller, Data, Domain, DSI, and Service. And there are six vertical slices defined that span these layers: Common, Contact, Customer, Distribution, Request, and User. At this point, however, there are no relationships between the layers or vertical slides shown—this is merely an enumeration of the important system abstractions.

20.5 Architecture Analysis: Finding Violations

Consider the following situation: You have designed an architecture but you have suspicions that the developers are not faithfully implementing what you developed. They may do this out of ignorance, or because they have differing agendas for the system, or simply because they were rushing to meet a deadline and ignored any concern not on their critical path. Whatever the root cause, this divergence of the architecture and the implementation spells problems for you, the architect. So how do you test and ensure conformance to the design?

There are two major possibilities for maintaining conformance between code and architecture:

- *Conformance by construction.* Ensuring consistency by construction—that is, automatically *generating* a substantial part of the system based on an architectural specification—is highly desirable because tools can guarantee conformance. Unfortunately, this approach has limited applicability. It can only be applied in situations where engineers can employ specific architecture-based development tools, languages, and implementation strategies. For systems that are composed of existing parts or that require a style of architecture or implementation outside those supported by generation tools, this approach does not apply. And this is the vast majority of systems.
- *Conformance by analysis.* This technique aims to ensure conformance by analyzing (reverse-engineering) system information to flag nonconforming elements, so that they can be fixed: brought into conformance. When an implementation is sufficiently constrained so that modularization and coding patterns can be identified with architectural elements, this technique can work well. Unfortunately, however, the technique is limited in its applicability. There is an inherent mismatch between static, code-based structures such as classes and packages (which are what programmers see) and the runtime structures, such as processes, threads, clients, servers, and databases, that are the essence of most architectural descriptions. Further complicating this analysis, the actual runtime structures may not be known or established until the program executes: clients and servers may come and go dynamically, components not under direct control of the implementers may be dynamically loaded, and so forth.

We will focus on the second option: conformance by analysis.

In the previous step, view fusion gave us a set of hypotheses about the architecture. These hypotheses take the form of architectural elements (sometimes aggregated, such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct—to see if they conform with the architect's intentions. That is the function of the analysis step.

Figure 20.4 shows the results of adding relationships and constraints to the architecture initially created in Figure 20.3. These relationship and constraints are information added by the architect, to reflect the design intent. In this example, the architect has indicated the relationships between the layers of Figure 20.3. These relationships are indicated by the directed lines drawn between the layers (and vertical slices). Using these relationships and constraints, a tool such as SonarJ is able to automatically detect and report violations of the layering in the software.

We can now see that the Data layer (row 2 in Figure 20.4) can access, and hence depends on, the DSI layer. We can further see that it may not access, and has no dependencies on, Domain, Service, or Controller (rows 1, 3, and 5 in the figure).

In addition we can see that the JUnit component in the "External" component is defined to be inaccessible. This is an example of an architectural constraint that is meant to pervade the entire system: no portion of the application should depend upon JUnit, because this should only be used by test code.

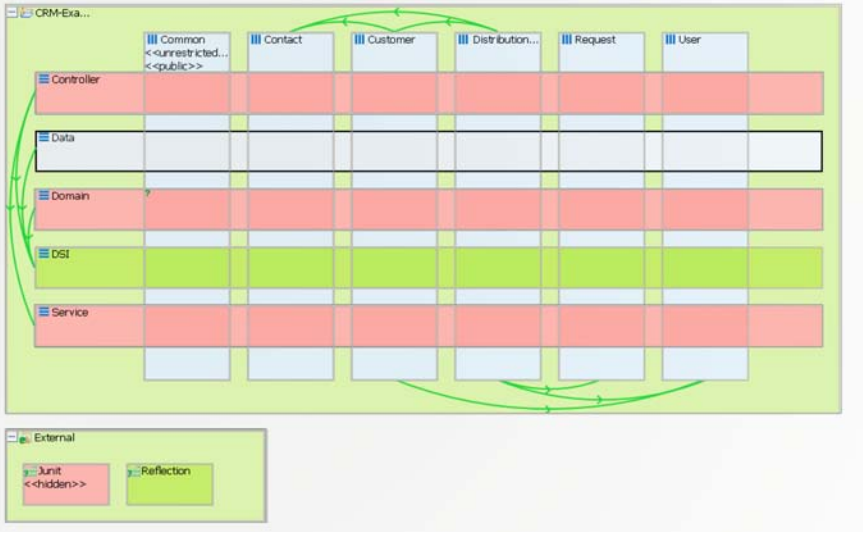


FIGURE 20.4 Layers, vertical slices, relationships, and constraints

Figure 20.5 shows an example of an architecture violation of the previous restriction. This violation is found by SonarJ by searching through its database, applying the user-defined patterns, and finding violations of those patterns. In this figure you can see an arc between the Service layer and JUnit. This arc is highlighted to indicate that this is an illegal dependency and an architectural violation. (This figure also shows some additional dependencies, to external modules.)

Architecture reconstruction is a means of testing the conformance to such constraints. The preceding example showed how these constraints might be detected and enforced using static code analysis. But static analysis is primarily useful for understanding module structures. What if one needed to understand runtime information, as represented by C&C structures?

In the example given in Figure 20.6, an architecture violation was discovered via dynamic analysis, using the research DiscoTect system. In this case an analysis of the runtime architecture of the Duke's Bank application—a simple Enterprise JavaBeans (EJB) banking application created by Sun Microsystems as a demonstration of EJB functionality—was performed. The code was "instrumented" using AspectJ; instrumentation aspects were woven into the compiled bytecode of the EJB application. These aspects emitted events when methods entered or exited and when objects were constructed.

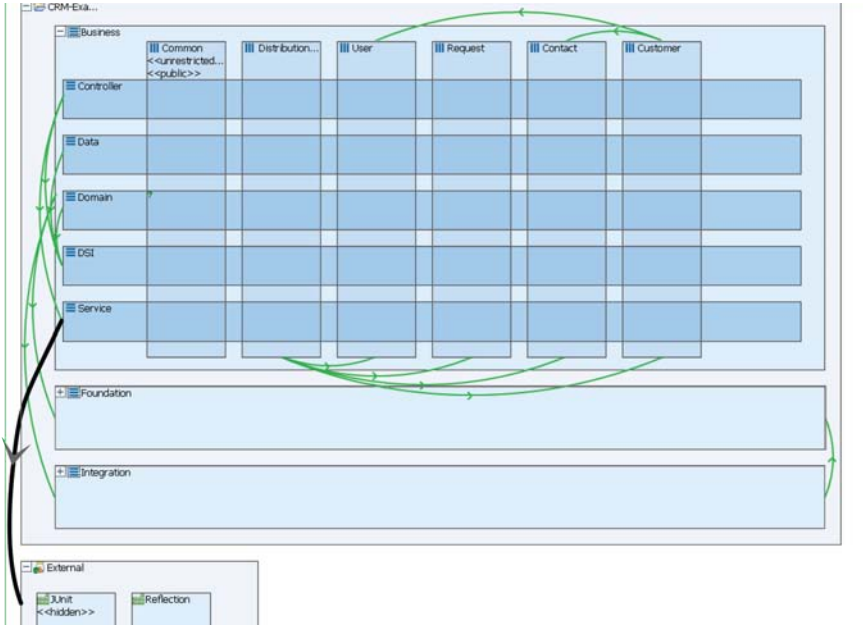


FIGURE 20.5 Highlighting an architecture violation

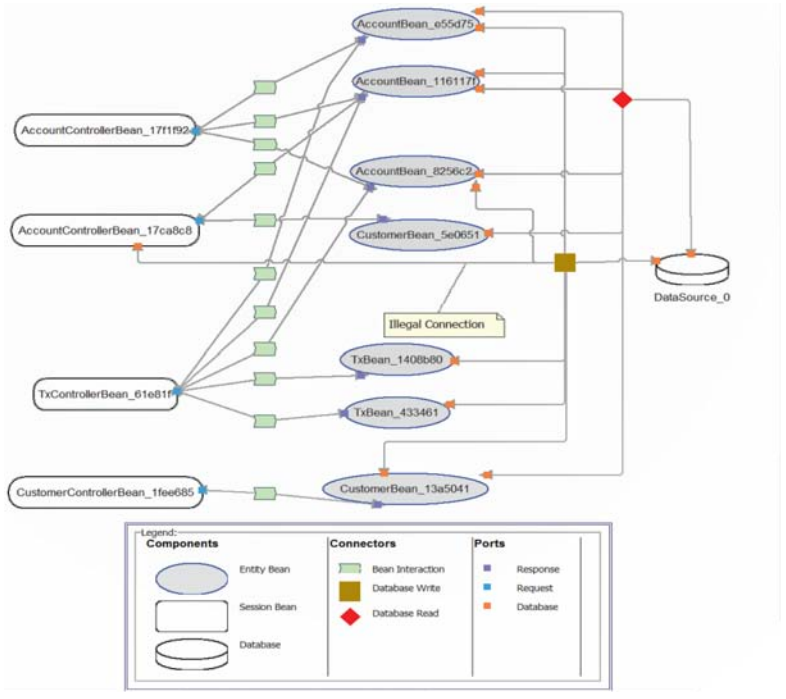


FIGURE 20.6 An architecture violation discovered by dynamic analysis

Figure 20.6 shows that a "database write" connector was discovered in the dynamic analysis of the architecture. Sun's EJB specification and its documented architecture of Duke's Bank forbid such connections. All database access is supposed to be managed by entity beans, and only by entity beans. Such architectural violations are difficult to find in the source code—often just a single line of code is involved—and yet can substantially affect the quality attributes of the resulting system.

20.6 Guidelines

The following are a set of guidelines for the reconstruction process:

- Have a goal and a set of objectives or questions in mind before undertaking an architecture reconstruction project. In the absence of these, a lot of effort could be spent on extracting information and generating architecture views that may not be helpful or serve any useful purpose.

- Obtain some representation, however coarse, of the system before beginning the detailed reconstruction process. This representation serves several purposes, including the following:
  - It identifies what information needs to be extracted from the system.
  - It guides the reconstructor in determining what to look for in the architecture and what views to generate.

  Identifying layers is a good place to start.
- In many cases, the existing documentation for a system may not accurately reflect the system as it is implemented. Therefore it may be necessary to disregard the existing documentation and use it only to generate the high-level views of the system, because it should give an indication of the high-level concepts.
- Tools can support the reconstruction effort and shorten the reconstruction process, but they cannot do an entire reconstruction effort automatically. The work involved in the effort requires the involvement of people (architects, maintainers, and developers) who are familiar with the system. It is important to get these people involved in the effort at an early stage as it helps the reconstructor get a better understanding of the system being reconstructed.

20.7 Summary

Architecture reconstruction and architecture conformance are crucial tools in the architect's toolbox to ensure that a system is built the way it was designed, and that it evolves in a way that is consistent with its creators' intentions. All nontrivial long-lived systems evolve: the code and the architecture both evolve. This is a good thing. But if the code evolves in an ad hoc manner, the result will be the big ball of mud, and the system's quality attributes will inevitably suffer. The only defense against this erosion is consistent attention to architecture quality, which implies the need to maintain architecture conformance.

The results of architectural reconstruction can be used in several ways:

- If no documentation exists or if it is seriously out of date, the recovered architectural representation can be used as a basis for documenting the architecture, as discussed in Chapter 18.
- It can be used to recover the as-built architecture, or to check conformance against an "as-designed" architecture. Conformance checking assures us that our developers and maintainers have followed the architectural edicts set forth for them and are not eroding the architecture by breaking down abstractions, bridging layers, compromising information hiding, and so forth.

- The reconstruction can be used as the basis for analyzing the architecture or as a starting point for reengineering the system to a new desired architecture.
- Finally, the representation can be used to identify elements for reuse or to establish an architecture-based software product line (see Chapter 25).

The software architecture reconstruction process comprises the following phases:

1. *Raw view extraction.* In the raw view extraction phase, raw information about the architecture is obtained from various sources, primarily source code, execution traces, and build scripts. Each of these sets of raw information is called a view.
2. *Database construction.* The database construction phase involves converting the extracted information into a standard form (because the various extraction tools may each produce their own form of output) and populating a reconstruction database with this information.
3. *View fusion.* The view fusion phase combines views of the information stored in the database.
4. *Architecture analysis.* View fusion has given us a set of hypotheses about the architecture. These hypotheses take the form of architectural elements (sometimes aggregated, such as layers) and the constraints and relationships among them. These hypotheses need to be tested to see if they are correct, and that is the function of the analysis step.

20.8 For Further Reading

The Software Engineering Institute (SEI) has developed two reconstruction workbenches: Dali and Armin. Dali was our first attempt at creating a workbench for architecture recovery and conformance [Kazman 99]. Armin, a complete rewrite and rethink of Dali, is described in [O'Brien 03].

Both Armin and Dali were primarily focused on module structures of an architecture. A later tool, called DiscoTect, was aimed at discovering C&C structures. This is described in [Schmerl 06].

Many other architecture reverse-engineering tools have been created. A few of the notable ones created in academia are [van Deursen 04], [Murphy 01], and [Storey 97].

In addition there are a number of commercial architecture extraction and reconstruction tools that have been slowly gaining market acceptance in the past decade. Among these are the following:

- SonarJ (www.hello2morrow.com)
- Lattix (www.lattix.com)

- Understand (www.scitools.com)

Cai et al. [Cai 2011] compellingly demonstrate the need for architecture conformance testing in an experimental study that they conducted, wherein they found that software engineering students, given UML designs for a variety of relatively simple systems, violate those designs over 70 percent of the time.

Finally, the set of guidelines presented in this chapter for how to go about reconstructing an architecture was excerpted from [Kazman 02].

20.9 Discussion Questions

1. Suppose that for a given system you wanted to extract the architectural structures (as discussed in Chapter 1) listed in the table rows below. For each row, fill in each column to appraise each strategy listed in the columns. "VH" (very high) means the strategy would be very effective at extracting this structure; "VL" means it would be very ineffective; "H," M," and "L" have the obvious in-between values.

		Reconstruction Strategies		
Architectural Structures Interviewing experts on the system		**Analyzing structure of source code files**	**Static analysis of source code**	**Dynamic analysis of system's execution**
Module structures	Decomposition			
	Uses			
	Layers			
	Class			
	Data model			
C&C structures	Service (for SOA systems)			
	Concurrency			
Allocation structures	Deployment			
	Implementation			
	Work assignment			

2. Recall that in layered systems, the relationship among layers is *allowed to use*. Also recall that it is possible for one piece of software to use another piece without actually calling it—for example, by depending on it leaving some shared resource in a usable state. Does this interpretation change your answer above for the "Uses" and "Layers" structures?

3. What inferences can you make about a system's module structures from examining a set of behavioral traces gathered dynamically?

4. Suppose you believe that the architecture for a system follows a broker pattern. What information would you want to extract from the source code to confirm or refute this hypothesis? What behavioral or interaction pattern would you expect to observe at runtime?

5. Suppose you hypothesize that a system makes use of particular tactics to achieve a particular quality attribute. Fill in the columns of the table below to show how you would go about verifying your hypothesis. (Begin by filling in column 1 with a particular tactic for the named quality attribute.)

	Reconstruction Strategies			
Tactics for…	**Interviewing experts on the system**	**Analyzing structure of source code files**	**Static analysis of source code**	**Dynamic analysis of system's execution**
Availability				
Interoperability				
Modifiability				
Performance				
Security				
Testability				
Usability				

6. Suppose you want to confirm that developers and maintainers had remained faithful to an architecture over the lifetime of the system. Describe the reconstruction and/or auditing processes you would undertake.

21

Architecture Evaluation

Fear cannot be banished, but it can be calm and without panic; it can be mitigated by reason and evaluation.
—Vannevar Bush

We discussed analysis techniques in Chapter 14. Analysis lies at the heart of architecture evaluation, which is the process of determining if an architecture is fit for the purpose for which it is intended. Architecture is such an important contributor to the success of a system and software engineering project that it makes sense to pause and make sure that the architecture you've designed will be able to provide all that's expected of it. That's the role of evaluation. Fortunately there are mature methods to evaluate architectures that use many of the concepts and techniques you've already learned in previous chapters of this book.

21.1 Evaluation Factors

Evaluation usually takes one of three forms:

- Evaluation by the designer within the design process
- Evaluation by peers within the design process
- Analysis by outsiders once the architecture has been designed

Evaluation by the Designer

Every time the designer makes a key design decision or completes a design milestone, the chosen and competing alternatives should be evaluated using the analysis techniques of Chapter 14. Evaluation by the designer is the "test" part of the "generate-and-test" approach to architecture design that we discussed in Chapter 17.

How much analysis? This depends on the importance of the decision. Obviously, decisions made to achieve one of the driving architectural requirements should be subject to more analysis than others, because these are the ones that will shape critical portions of the architecture. But in all cases, performing analysis is a matter of cost and benefit. Do not spend more time on a decision than it is worth, but also do not spend less time on an important decision than it needs. Some specific considerations include these:

- *The importance of the decision*. The more important the decision, the more care should be taken in making it and making sure it's right.
- *The number of potential alternatives.* The more alternatives, the more time could be spent in evaluating them. Try to eliminate alternatives quickly so that the number of viable potential alternatives is small.
- *Good enough as opposed to perfect.* Many times, two possible alternatives do not differ dramatically in their consequences. In such a case, it is more important to make a choice and move on with the design process than it is to be absolutely certain that the best choice is being made. Again, do not spend more time on a decision than it is worth.

Peer Review

Architectural designs can be peer reviewed just as code can be peer reviewed. A peer review can be carried out at any point of the design process where a candidate architecture, or at least a coherent reviewable part of one, exists. There should be a fixed amount of time allocated for the peer review, at least several hours and possibly half a day. A peer review has several steps:

1. *The reviewers determine a number of quality attribute scenarios to drive the review.* Most of the time these scenarios will be architecturally significant requirements, but they need not be. These scenarios can be developed by the review team or by additional stakeholders.
2. *The architect presents the portion of the architecture to be evaluated.* (At this point, comprehensive documentation for it may not exist.) The reviewers individually ensure that they understand the architecture. Questions at this point are specifically for understanding. There is no debate about the decisions that were made. These come in the next step.
3. *For each scenario, the designer walks through the architecture and explains how the scenario is satisfied.* (If the architecture is already documented, then the reviews can use it to assess for themselves how it satisfies the scenario.) The reviewers ask questions to determine two different types of information. First, they want to determine that the scenario is, in fact, satisfied. Second, they want to determine whether any of the other scenarios being considered will not be satisfied because of the decisions made in the portion of the architecture being reviewed.

4. *Potential problems are captured.* The list of potential problems forms the basis for the follow-up of the review. If the potential problem is a real problem, then it either must be fixed or a decision must be explicitly made by the designers and the project manager that they are willing to accept the problem and its probability of occurrence.

If the designers are using the ADD process described in Chapter 17, then a peer review can be done at the end of step 3 of each ADD iteration.

Analysis by Outsiders

Outside evaluators can cast an objective eye on an architecture. "Outside" is relative; this may mean outside the development project, outside the business unit where the project resides but within the same company; or outside the company altogether. To the degree that evaluators are "outside," they are less likely to be afraid to bring up sensitive problems, or problems that aren't apparent because of organizational culture or because "we've always done it that way."

Often, outsiders are chosen because they possess specialized knowledge or experience, such as knowledge about a quality attribute that's important to the system being examined, or long experience in successfully evaluating architectures.

Also, whether justified or not, managers tend to be more inclined to listen to problems uncovered by an outside team hired at considerable cost. (This can be understandably frustrating to project staff who may have been complaining about the same problems to no avail for months.)

In principle, an outside team may evaluate a completed architecture, an incomplete architecture, or a portion of an architecture. In practice, because engaging them is complicated and often expensive, they tend to be used to evaluate complete architectures.

Contextual Factors

For peer reviews or outside analysis, there are a number of contextual factors that must be considered when structuring an evaluation. These include the artifacts available, whether the results are public or private, the number and skill of evaluators, the number and identity of the participating stakeholders, and how the business goals are understood by the evaluators.

- *What artifacts are available?* To perform an architectural evaluation, there must be an artifact that describes the architecture. This must be located and made available. Some evaluations may take place after the system is operational. In this case, recovery tools as described in Chapter 20 may be used both to assist in discovering the architecture and to test that the as-built system conforms to the as-designed system.

- *Who sees the results?* Some evaluations are performed with the full knowledge and participation of all of the stakeholders. Others are performed more privately. The private evaluations may be done for a variety of reasons, ranging from corporate culture to (in one case we know about) an executive wanting to determine which of a collection of competitive systems he should back in an internal dispute about the systems.
- *Who performs the evaluation?* Evaluations can be carried out by an individual or a team. In either case, the evaluator(s) should be highly skilled in the domain and the various quality attributes for which the system is to be evaluated. And for carrying out evaluation methods with extensive stakeholder involvement, excellent organizational and facilitation skills are a must.
- *Which stakeholders will participate?* The evaluation process should provide a method to elicit the goals and concerns that the important stakeholders have regarding the system. Identifying the individuals who are needed and assuring their participation in the evaluation is critical.
- *What are the business goals?* The evaluation should answer whether the system will satisfy the business goals. If the business goals are not explicitly captured and prioritized prior to the evaluation, then there should be a portion of the evaluation dedicated to doing so.

21.2 The Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) has been used for over a decade to evaluate software architectures in domains ranging from automotive to financial to defense. The ATAM is designed so that evaluators need not be familiar with the architecture or its business goals, the system need not yet be constructed, and there may be a large number of stakeholders.

Participants in the ATAM

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

- *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 21.1 for a description of these roles, along with a set of desirable characteristics for each. A single person may adopt several roles in an ATAM.) 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 to grind.

- *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 may be present (or represented) as well. The architect is always included—a cardinal rule of architecture evaluation is that the architect must willingly participate.
- *Architecture stakeholders.* Stakeholders have a vested interest in the architecture performing as advertised. They are the ones whose ability to do their job hinges on the 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 listed in Chapter 3. 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 12 to 15 stakeholders for the evaluation of a large enterprise-critical architecture. Unlike the evaluation team and the project decision makers, stakeholders do not participate in the entire exercise.

TABLE 21.1 ATAM Evaluation Team Roles

Role	Responsibilities
Team Leader	Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; forms evaluation team; sees that final report is produced and delivered (although the writing may be delegated)
Evaluation Leader	Runs evaluation; facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture; facilitates on-site analysis
Scenario Scribe	Writes scenarios on flipchart or whiteboard during scenario elicitation; captures agreed-on wording of each scenario, halting discussion until exact wording is captured
Proceedings Scribe	Captures proceedings in electronic form on laptop or workstation: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and resolution of each scenario when applied to architecture(s); also generates a printed list of adopted scenarios for handout to all participants
Questioner	Raises issues of architectural interest, usually related to the quality attributes in which he or she has expertise

Outputs of the ATAM

As in any testing process, a large benefit derives from preparing for the test. In preparation for an ATAM exercise, the project's decision makers must prepare the following:

1. *A concise presentation of the architecture.* 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.
2. *Articulation of the business goals.* Frequently, the business goals presented in the ATAM are being seen by some of the assembled participants for the first time, and these are captured in the outputs. This description of the business goals survives the evaluation and becomes part of the project's legacy.

The ATAM uses prioritized quality attribute scenarios as the basis for evaluating the architecture, and if those scenarios do not already exist (perhaps as a result of a prior requirements capture exercise or ADD activity), they are generated by the participants as part of the ATAM exercise. Many times, ATAM participants have told us that one of the most valuable outputs of ATAM is this next output:

3. *Prioritized quality attribute requirements expressed as quality attribute scenarios.* These quality attribute scenarios take the form described in Chapter 4. These also survive past the evaluation and can be used to guide the architecture's evolution.

The primary output of the ATAM is a set of issues of concern about the architecture. We call these risks:

4. *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 attribute requirements. Similarly, a nonrisk is an architectural decision that, upon analysis, is deemed safe. The identified risks form the basis for an architectural risk mitigation plan.
5. *A set of risk themes.* When the analysis is complete, the evaluation team examines the full set of discovered risks to look for overarching 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 business goals.

Finally, along the way, other information about the architecture is discovered and captured:

6. *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 attribute scenario examined during an ATAM, those

architectural decisions that help to achieve it are determined and captured. This can serve as a statement of rationale for those decisions.

7. *A set of identified sensitivity and tradeoff points*. These are architectural decisions that have a marked effect on one or more quality attributes.

The outputs of the ATAM are used to build a final written report that recaps the method, summarizes the proceedings, captures the scenarios and their analysis, and catalogs the findings.

There are intangible results of an ATAM-based evaluation. These include a palpable sense of community on the part of the stakeholders, open communication channels between the architect and the stakeholders, and a better overall understanding on the part of all participants of the architecture and its strengths and weaknesses. While these results are hard to measure, they are no less important than the others and often are the longest-lasting.

Phases of the ATAM

Activities in an ATAM-based evaluation are spread out over four phases:

- In phase 0, "Partnership and Preparation," the evaluation team leadership and the key project decision makers informally meet to work out the details of the exercise. The project representatives brief the evaluators about the project so that the team can be supplemented by people who possess the appropriate expertise. Together, the two groups agree on logistics, such as the time and place of meetings, who brings the flipcharts, and who supplies the donuts and coffee. They also agree on a preliminary list of stakeholders (by name, not just role), and they negotiate on when the final report is to be delivered and to whom. They deal with formalities such as a statement of work or nondisclosure agreements. The evaluation team examines the architecture documentation to gain an understanding of the architecture and the major design approaches that it comprises. Finally, the evaluation team leader explains what information the manager and architect will be expected to show during phase 1, and helps them construct their presentations if necessary.
- Phase 1 and phase 2 are the evaluation phases, where everyone gets down to the business of analysis. By now the evaluation team will have studied the architecture documentation and will have a good idea of what the system is about, the overall architectural approaches taken, and the quality attributes that are of paramount importance. During phase 1, the evaluation team meets with the project decision makers (for one to two days) to begin information gathering and analysis. For phase 2, the architecture's stakeholders join the proceedings and analysis continues, typically for two days. Unlike the other phases, phase 1 and phase 2 comprise a set of specific steps; these are detailed in the next section.

TABLE 21.2 ATAM Phases and Their Characteristics

Phase	Activity	Participants	Typical Duration
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 and project decision makers	1–2 days followed by a hiatus of 1–3 weeks
2	Evaluation (continued)	Evaluation team, project decision makers, and stakeholders	2 days
3	Follow-up	Evaluation team and evaluation client	1 week

Source: Adapted from [Clements 01b].

- Phase 3 is follow-up, in which the evaluation team produces and delivers a written final report. It is first circulated to key stakeholders to make sure that it contains no errors of understanding, and after this review is complete it is delivered to the person who commissioned the evaluation.

Table 21.2 shows the four phases of the ATAM, who participates in each one, and an approximate timetable.

Steps of the Evaluation Phases

The ATAM analysis phases (phase 1 and phase 2) consist of nine steps. Steps 1 through 6 are carried out in phase 1 with the evaluation team and the project's decision makers: typically, the architecture team, project manager, and project sponsor. In phase 2, with all stakeholders present, steps 1 through 6 are summarized and steps 7 through 9 are carried out.

Table 21.3 shows a typical agenda for the first day of phase 1, which covers steps 1 through 5. Step 6 in phase 1 is carried out the next day.

Step 1: Present the ATAM. The first step calls for the evaluation leader to present the ATAM to the assembled project representatives. This time is used to explain the process that everyone will be following, to answer questions, and to set the context and expectations for the remainder of the activities. Using a standard presentation, the leader describes the ATAM steps in brief and the outputs of the evaluation.

Step 2: Present the Business Drivers. Everyone involved in the evaluation—the project representatives as well as the evaluation team members—needs to understand the context for the system and the primary business drivers

motivating its development. In this step, a project decision maker (ideally the project manager or the system's customer) presents a system overview from a business perspective. The presentation should describe the following:

- The system's most important functions
- Any relevant technical, managerial, economic, or political constraints
- The business goals and context as they relate to the project
- The major stakeholders
- The architectural drivers (that is, the architecturally significant requirements)

Step 3: Present the Architecture. Here, the lead architect (or architecture team) makes a presentation describing the architecture at an appropriate level of detail. The "appropriate level" depends on several factors: how much of the architecture has been designed and documented; how much time is available; and the nature of the behavioral and quality requirements.

In this presentation the architect covers technical constraints such as operating system, hardware, or middleware prescribed for use, and other systems with which the system must interact. Most important, the architect describes the architectural approaches (or patterns, or tactics, if the architect is fluent in that vocabulary) used to meet the requirements.

To make the most of limited time, the architect's presentation should have a high signal-to-noise ratio. That is, it should convey the essence of the architecture and not stray into ancillary areas or delve too deeply into the details of just a few aspects. Thus, it is extremely helpful to brief the architect beforehand (in phase 0) about the information the evaluation team requires. A template such as the one in the sidebar can help the architect prepare the presentation. Depending on the architect, a dress rehearsal can be included as part of the phase 0 activities.

TABLE 21.3 Agenda for Day 1 of the ATAM

Time	Activity
0830 – 1000	Introductions; Step 1: Present the ATAM
1000 – 1100	Step 2: Present Business Drivers
1100 – 1130	Break
1130 – 1230	Step 3: Present Architecture
1230 – 1330	Lunch
1330 – 1430	Step 4: Identify Architectural Approaches
1430 – 1530	Step 5: Generate Utility Tree
1530 – 1600	Break
1600 – 1700	Step 5: Generate Utility Tree (continued)

Architecture Presentation (Approximately 20 slides; 60 Minutes)

Driving architectural requirements, the measurable quantities you associate with these requirements, and any existing standards/models/approaches for meeting these (2–3 slides)

Important architectural information (4–8 slides):

- Context diagram—the system within the context in which it will exist. Humans or other systems with which the system will interact.
- Module or layer view—the modules (which may be subsystems or layers) that describe the system's decomposition of functionality, along with the objects, procedures, functions that populate these, and the relations among them (e.g., procedure call, method invocation, callback, containment).
- Component-and-connector view—processes, threads along with the synchronization, data flow, and events that connect them.
- Deployment view—CPUs, storage, external devices/sensors along with the networks and communication devices that connect them. Also shown are the processes that execute on the various processors.

Architectural approaches, patterns, or tactics employed, including what quality attributes they address and a description of how the approaches address those attributes (3–6 slides):

- Use of commercial off-the-shelf (COTS) products and how they are chosen/integrated (1–2 slides).
- Trace of 1 to 3 of the most important use case scenarios. If possible, include the runtime resources consumed for each scenario (1–3 slides).
- Trace of 1 to 3 of the most important change scenarios. If possible, describe the change impact (estimated size/difficulty of the change) in terms of the changed modules or interfaces (1–3 slides).
- Architectural issues/risks with respect to meeting the driving architectural requirements (2–3 slides).
- Glossary (1 slide).

Source: Adapted from [Clements 01b].

As may be seen in the presentation template, we expect architectural views, as described in Chapters 1 and 18, to be the primary vehicle for the architect to convey the architecture. Context diagrams, component-and-connector views, module decomposition or layered views, and the deployment view are useful in almost every evaluation, and the architect should be prepared to show them. Other views can be presented if they contain information relevant to the architecture at hand, especially information relevant to achieving important quality attribute goals.

As a rule of thumb, the architect should present the views that he or she found most important during the creation of the architecture and the views that help to reason about the most important quality attribute concerns of the system.

During the presentation, the evaluation team asks for clarification based on their phase 0 examination of the architecture documentation and their knowledge of the business drivers from the previous step. They also listen for and write down any architectural tactics or patterns they see employed.

Step 4: Identify Architectural Approaches. The ATAM focuses on analyzing an architecture by understanding its architectural approaches. As we saw in Chapter 13, architectural patterns and tactics are useful for (among other reasons) the known ways in which each one affects particular quality attributes. A layered pattern tends to bring portability and maintainability to a system, possibly at the expense of performance. A publish-subscribe pattern is scalable in the number of producers and consumers of data. The active redundancy tactic promotes high availability. And so forth.

By now, the evaluation team will have a good idea of what patterns and tactics the architect used in designing the system. They will have studied the architecture documentation, and they will have heard the architect's presentation in step 3. During that step, the architect is asked to explicitly name the patterns and tactics used, but the team should also be adept at spotting ones not mentioned.

In this short step, the evaluation team simply catalogs the patterns and tactics that have been identified. The list is publicly captured by the scribe for all to see and will serve as the basis for later analysis.

Step 5: Generate Quality Attribute Utility Tree. In this step, the quality attribute goals are articulated in detail via a quality attribute utility tree. Utility trees, which were described in Chapter 16, serve to make the requirements concrete by defining precisely the relevant quality attribute requirements that the architects were working to provide.

The important quality attribute goals for the architecture under consideration were named in step 2, when the business drivers were presented, but not to any degree of specificity that would permit analysis. Broad goals such as "modifiability" or "high throughput" or "ability to be ported to a number of platforms" establish important context and direction, and provide a backdrop against which subsequent information is presented. However, they are not specific enough to let us tell if the architecture suffices. Modifiable in what way? Throughput that is how high? Ported to what platforms and in how much time?

In this step, the evaluation team works with the project decision makers to identify, prioritize, and refine the system's most important quality attribute goals. These are expressed as scenarios, as described in Chapter 4, which populate the leaves of the utility tree.

Step 6: Analyze Architectural Approaches. Here the evaluation team examines the highest-ranked scenarios (as identified in the utility tree) one at a time; the architect is asked to explain how the architecture supports each one. Evaluation team members—especially the questioners—probe for the architectural approaches that the architect used to carry out the scenario. Along the way, the evaluation team documents the relevant architectural decisions and identifies and catalogs their risks, nonrisks, sensitivity points, and tradeoffs. For well-known approaches, the evaluation team asks how the architect overcame known weaknesses in the approach or how the architect gained assurance that the approach sufficed. The goal is for the evaluation team to be convinced that the instantiation of the approach is appropriate for meeting the attribute-specific requirements for which it is intended.

Scenario walkthrough leads to a discussion of possible risks, nonrisks, sensitivity points, or tradeoff points. For example:

- The frequency of heartbeats affects the time in which the system can detect a failed component. Some assignments will result in unacceptable values of this response—these are risks.
- The number of simultaneous database clients will affect the number of transactions that a database can process per second. Thus, the assignment of clients to the server is a sensitivity point with respect to the response as measured in transactions per second.
- The frequency of heartbeats determines the time for detection of a fault. Higher frequency leads to improved availability but will also consume more processing time and communication bandwidth (potentially leading to reduced performance). This is a tradeoff.

These, in turn, may catalyze a deeper analysis, depending on how the architect responds. For example, if the architect cannot characterize the number of clients and cannot say how load balancing will be achieved by allocating processes to hardware, there is little point in a sophisticated performance analysis. If such questions can be answered, the evaluation team can perform at least a rudimentary, or back-of-the-envelope, analysis to determine if these architectural decisions are problematic vis-à-vis the quality attribute requirements they are meant to address.

The analysis is not meant to be comprehensive. The key is to elicit sufficient architectural information to establish some link between the architectural decisions that have been made and the quality attribute requirements that need to be satisfied.

Figure 21.1 shows a template for capturing the analysis of an architectural approach for a scenario. As shown, based on the results of this step, the evaluation team can identify and record a set of sensitivity points and tradeoffs, risks, and nonrisks.

At the end of step 6, the evaluation team should have a clear picture of the most important aspects of the entire architecture, the rationale for key design decisions, and a list of risks, nonrisks, sensitivity points, and tradeoff points.

At this point, phase 1 is concluded.

Scenario #: A12	Scenario: Detect and recover from HW failure of main switch.
Attribute(s)	Availability
Environment	Normal operations
Stimulus	One of the CPUs fails
Response	0.999999 availability of switch

Architectural decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14

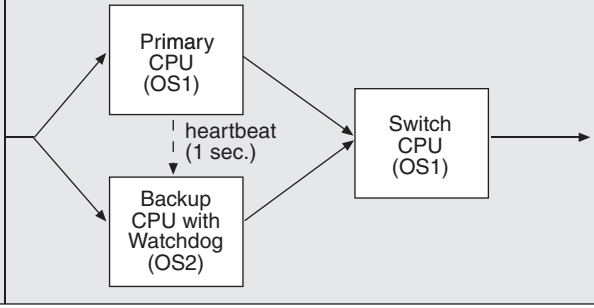
Reasoning	Ensures no common mode failure by using different hardware and operating system (see Risk 8) Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog Watchdog is simple and has proved reliable Availability requirement might be at risk due to lack of backup data channel ... (see Risk 9)
Architecture diagram	

FIGURE 21.1 Example of architecture approach analysis (adapted from [Clements 01b])

Hiatus and Start of Phase 2. The evaluation team summarizes what it has learned and interacts informally (usually by phone) with the architect during a hiatus of a week or two. More scenarios might be analyzed during this period, if desired, or questions of clarification can be resolved.

Phase 2 is attended by an expanded list of participants with additional stakeholders attending. To use an analogy from programming: Phase 1 is akin to when you test your own program, using your own criteria. Phase 2 is when you give your program to an independent quality assurance group, who will likely subject your program to a wider variety of tests and environments.

In phase 2, step 1 is repeated so that the stakeholders understand the method and the roles they are to play. Then the evaluation leader recaps the results of steps 2 through 6, and shares the current list of risks, nonrisks, sensitivity points, and tradeoffs. Now the stakeholders are up to speed with the evaluation results so far, and the remaining three steps can be carried out.

Step 7: Brainstorm and Prioritize Scenarios. In this step, the evaluation team asks the stakeholders to brainstorm scenarios that are operationally meaningful with respect to the stakeholders' individual roles. A maintainer will likely propose a modifiability scenario, while a user will probably come up with a scenario that expresses useful functionality or ease of operation, and a quality assurance person will propose a scenario about testing the system or being able to replicate the state of the system leading up to a fault.

While utility tree generation (step 5) is used primarily to understand how the architect perceived and handled quality attribute architectural drivers, the purpose of scenario brainstorming is to take the pulse of the larger stakeholder community: to understand what system success means for them. Scenario brainstorming works well in larger groups, creating an atmosphere in which the ideas and thoughts of one person stimulate others' ideas.

Once the scenarios have been collected, they must be prioritized, for the same reasons that the scenarios in the utility tree needed to be prioritized: the evaluation team needs to know where to devote its limited analytical time. First, stakeholders are asked to merge scenarios they feel represent the same behavior or quality concern. Then they vote for those they feel are most important. Each stakeholder is allocated a number of votes equal to 30 percent of the number of scenarios,[1] rounded up. So, if there were 40 scenarios collected, each stakeholder would be given 12 votes. These votes can be allocated in any way that the stakeholder sees fit: all 12 votes for 1 scenario, 1 vote for each of 12 distinct scenarios, or anything in between.

The list of prioritized scenarios is compared with those from the utility tree exercise. If they agree, it indicates good alignment between what the architect had in mind and what the stakeholders actually wanted. If additional driving scenarios are discovered—and they usually are—this may itself be a risk, if the discrepancy is large. This would indicate that there was some disagreement in the system's important goals between the stakeholders and the architect.

1. This is a common facilitated brainstorming technique.

Step 8: Analyze Architectural Approaches. After the scenarios have been collected and prioritized in step 7, the evaluation team guides the architect in the process of carrying out the highest ranked scenarios. The architect explains how relevant architectural decisions contribute to realizing each one. Ideally this activity will be dominated by the architect's explanation of scenarios in terms of previously discussed architectural approaches.

In this step the evaluation team performs the same activities as in step 6, using the highest-ranked, newly generated scenarios.

Typically, this step might cover the top five to ten scenarios, as time permits.

Step 9: Present Results. In step 9, the evaluation team groups risks into risk themes, based on some common underlying concern or systemic deficiency. For example, a group of risks about inadequate or out-of-date documentation might be grouped into a risk theme stating that documentation is given insufficient consideration. A group of risks about the system's inability to function in the face of various hardware and/or software failures might lead to a risk theme about insufficient attention to backup capability or providing high availability.

For each risk theme, the evaluation team identifies which of the business drivers listed in step 2 are affected. Identifying risk themes and then relating them to specific drivers brings the evaluation full circle by relating the final results to the initial presentation, thus providing a satisfying closure to the exercise. As important, it elevates the risks that were uncovered to the attention of management. What might otherwise have seemed to a manager like an esoteric technical issue is now identified unambiguously as a threat to something the manager is on record as caring about.

The collected information from the evaluation is summarized and presented to stakeholders. This takes the form of a verbal presentation with slides. The evaluation leader recapitulates the steps of the ATAM and all the information collected in the steps of the method, including the business context, driving requirements, constraints, and architecture. Then the following outputs are presented:

- The architectural approaches documented
- The set of scenarios and their prioritization from the brainstorming
- The utility tree
- The risks discovered
- The nonrisks documented
- The sensitivity points and tradeoff points found
- Risk themes and the business drivers threatened by each one

". . . but it was OK."

Years of experience have taught us that no architecture evaluation exercise ever goes completely by the book. And yet for all the ways that an exercise might go terribly wrong, for all the details that can be overlooked, for all the fragile egos that can be bruised, and for all the high stakes that are on the table, we have never had an architecture evaluation exercise spiral out of control. Every single one has been a success, as measured by the feedback we gather from clients.

While they all turned out successfully, there were a few memorable cliffhangers.

More than once, we began an architecture evaluation only to discover that the development organization had no architecture to be evaluated. Sometimes there was a stack of class diagrams or vague text descriptions masquerading as an architecture. Once we were promised that the architecture would be ready by the time the exercise began, but in spite of good intentions, it wasn't. (We weren't always so prudent about pre-exercise preparation and qualification. Our current diligence was a result of experiences like these.) But it was OK. In cases like these, the evaluation's main results included the articulated set of quality attributes, a "whiteboard" architecture sketched during the exercise, plus a set of documentation obligations on the architect. In all cases, the client felt that the detailed scenarios, the analysis we were able to perform on the elicited architecture, plus the recognition of what needed to be done, more than justified the exercise.

A couple of times we began an evaluation only to lose the architect in the middle of the exercise. In one case, the architect resigned between preparation and execution of the evaluation. This was an organization in turmoil and the architect simply got a better offer in a calmer environment elsewhere. Normally we don't proceed without the architect, but it was OK. In this case the architect's apprentice stepped in. A little additional prework to prepare him, and we were all set. The evaluation went off as planned, and the preparation that the apprentice did for the exercise helped mightily to prepare him to step into the architect's shoes.

Once we discovered halfway through an ATAM exercise that the architecture we had prepared to evaluate was being jettisoned in favor of a new one that nobody had bothered to mention. During step 6 of phase 1, the architect responded to a problem raised by a scenario by casually mentioning that "the new architecture" would not suffer from that deficiency. Everyone in the room, stakeholders and evaluators alike, looked at each other in the puzzled silence that followed. "What new architecture?" I asked blankly, and out it came. The developing organization (a contractor for the U.S. military, which had commissioned the evaluation), had prepared a new architecture for the system, to handle the more stringent requirements they knew were coming in the future. We called a timeout, conferred with the architect and the client, and decided to continue the exercise using the new architecture as the subject instead of the old. We backed up to step 3 (the architecture presentation), but everything else on the table—business drivers, utility

tree, scenarios—still were completely valid. The evaluation proceeded as before, and at the conclusion of the exercise our military client was extremely pleased at the knowledge gained.

In perhaps the most bizarre evaluation in our experience, we lost the architect midway through phase 2. The client for this exercise was the project manager in an organization undergoing a massive restructuring. The manager was a pleasant gentleman with a quick sense of humor, but there was an undercurrent about him that said he was not to be crossed. The architect was being reassigned to a different part of the organization in the near future; this was tantamount to being fired from the project, and the manager said he wanted to establish the quality of the architecture before his architect's awkward departure. (We didn't find any of this out until after the evaluation.) When we set up the ATAM exercise, the manager suggested that the junior designers attend. "They might learn something," he said. We agreed. As the exercise began, our schedule (which was very tight to begin with) kept being disrupted. The manager wanted us to meet with his company's executives. Then he wanted us to have a long lunch with someone who could, he said, give us more architectural insights. The executives, it turned out, were busy just now, and so could we come back and meet with them a bit later? By now, phase 2 was thrown off schedule by so much that the architect, to our horror, had to leave to fly back to his home in a distant city. He was none too happy that his architecture was going to be evaluated without him. The junior designers, he said, would never be able to answer our questions. Before his departure, our team huddled. The exercise seemed to be teetering on the brink of disaster. We had an unhappy departing architect, a blown schedule, and questionable expertise available. We decided to split our evaluation team. One half of the team would continue with phase 2 using the junior designers as our information resource. The second half of the team would continue with phase 2 by telephone the next day with the architect. Somehow we would make the best of a bad situation.

Surprisingly, the project manager seemed completely unperturbed by the turn of events. "It will work out, I'm sure," he said pleasantly, and then retreated to confer with various vice presidents about the reorganization.

I led the team interviewing the junior designers. We had never gotten a completely satisfactory architecture presentation from the architect. Discrepancies in the documentation were met with a breezy "Oh, well, that's not how it really works." So I decided to start over with ATAM step 3. We asked the half dozen or so designers what their view of the architecture was. "Could you draw it?" I asked them. They looked at each other nervously, but one said, "I think I can draw part of it." He took to the whiteboard and drew a very reasonable component-and-connector view. Someone else volunteered to draw a process view. A third person drew the architecture for an important offline part of the system. Others jumped in to assist.

As we looked around the room, everyone was busy transcribing the whiteboard pictures. None of the pictures corresponded to anything we had seen in the documentation so far. "Are these diagrams documented

anywhere?" I asked. One of the designers looked up from his busy scribbling for a moment to grin. "They are now," he said.

As we proceeded to step 8, analyzing the architecture using the scenarios previously captured, the designers did an astonishingly good job of working together to answer our questions. Nobody knew everything, but everybody knew something. Together in a half day, they produced a clear and consistent picture of the whole architecture that was much more coherent and understandable than anything the architect had been willing to produce in two whole days of pre-exercise discussion. And by the end of phase 2, the design team was transformed. This erstwhile group of information-starved individuals with limited compartmentalized knowledge became a true architecture team. The members drew out and recognized each others' expertise. This expertise was revealed and validated in front of everyone—and most important, in front of their project manager, who had slipped back into the room to observe. There was a look of supreme satisfaction on his face. It began to dawn on me that—you guessed it—it was OK.

It turned out that this project manager knew how to manipulate events and people in ways that would have impressed Machiavelli. The architect's departure was not because of the reorganization, but merely coincident with it. The project manager had orchestrated it. The architect had, the manager felt, become too autocratic and dictatorial, and the manager wanted the junior design staff to be given the opportunity to mature and contribute. The architect's mid-exercise departure was exactly what the project manager had wanted. And the design team's emergence under fire had been the primary purpose of the evaluation exercise all along. Although we found several important issues related to the architecture, the project manager knew about every one of them before we ever arrived. In fact, he made sure we uncovered some of them by a few discreet remarks during breaks or after a day's session.

Was this exercise a success? The client could not have been more pleased. His instincts about the architecture's strengths and weaknesses were confirmed. We were instrumental in helping his design team, which would guide the system through the stormy seas of the company's reorganization, come together as an effective and cohesive unit at exactly the right time. And the client was so pleased with our final report that he made sure the company's board of directors saw it.

These cliffhangers certainly stand out in our memory. There was no architecture documented. But it was OK. It wasn't the right architecture. But it was OK. There was no architect. But it was OK. The client really only wanted to effect a team reorganization. In every instance we reacted as reasonably as we could, and each time it was OK.

Why? Why, time after time, does it turn out OK? I think there are three reasons.

First, the people who have commissioned the architecture evaluation really want it to succeed. The architect, developers, and stakeholders

assembled at the client's behest also want it to succeed. As a group, they help to keep the exercise marching toward the goal of architectural insight. Second, we are always honest. If we feel that the exercise is derailing, we call a timeout and confer among ourselves, and usually confer with the client. While a small amount of bravado can come in handy during an exercise, we never, ever try to bluff our way through an evaluation. Participants can detect that instinctively, and the evaluation team must never lose the respect of the other participants. Third, the methods are constructed to establish and maintain a steady consensus throughout the exercise. There are no surprises at the end. The participants lay down the ground rules for what constitutes a suitable architecture, and they contribute to the risks uncovered at every step of the way.

So: Do the best job you can. Be honest. Trust the methods. Trust in the goodwill and good intentions of the people you have assembled. And it will be OK. (Adapted from [Clements 01b])

—PCC

21.3 Lightweight Architecture Evaluation

Although we attempt to use time in an ATAM exercise as efficiently as possible, it remains a substantial undertaking. It requires some 20 to 30 person-days of effort from an evaluation team, plus even more for the architect and stakeholders. Investing this amount of time only makes sense on a large and costly project, where the risks of making a major mistake in the architecture are unacceptable.

For this reason, we have developed a Lightweight Architecture Evaluation method, based on the ATAM, for smaller, less risky projects. A Lightweight Architecture Evaluation exercise may take place in a single day, or even a half-day meeting. It may be carried out entirely by members internal to the organization. Of course this lower level of scrutiny and objectivity may not probe the architecture as deeply, but this is a cost/benefit tradeoff that is entirely appropriate for many projects.

Because the participants are all internal to the organization and fewer in number than for the ATAM, giving everyone their say and achieving a shared understanding takes much less time. Hence the steps and phases of a Lightweight Architecture Evaluation can be carried out more quickly. A suggested schedule for phases 1 and 2 is shown in Table 21.4.

TABLE 21.4 A Typical Agenda for Lightweight Architecture Evaluation

Step	Time Allotted	Notes
1: Present the ATAM	0 hrs	The participants are familiar with the process. This step may be omitted.
2: Present Business Drivers	0.25 hrs	The participants are expected to understand the system and its business goals and their priorities. Fifteen minutes is allocated for a brief review to ensure that these are fresh in everyone's mind and that there are no surprises.
3: Present Architecture	0.5 hrs	Again, all participants are expected to be familiar with the system and so a brief overview of the architecture, using at least module and C&C views, is presented and 1 to 2 scenarios are traced through these views.
4: Identify Architectural Approaches	0.25 hrs	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5: Generate Quality Attribute Utility Tree	Variable 0.5 hrs – 1.5 hrs	Scenarios might exist: part of previous evals, part of design, part of requirements elicitation. If you've got 'em, use 'em and make them into a tree. Half hour. Otherwise, it will take longer. A utility tree should already exist; the team reviews the existing tree and updates it, if needed, with new scenarios, new response goals, or new scenario priorities and risk assessments.
6: Analyze Architectural Approaches	2–3 hrs	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.
7: Brainstorm and Prioritize Scenarios	0 hrs	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8: Analyze Architectural Approaches	0 hrs	This step is also omitted, since all analysis is done in step 6.
9: Present Results	0.5 hrs	At the end of an evaluation, the team reviews the existing and newly discovered risks, nonrisks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.
TOTAL	**4–6 hrs**	

There is no final report, but (as in the regular ATAM) a scribe is responsible for capturing results, which can then be distributed and serve as the basis for risk remediation.

An entire Lightweight Architecture Evaluation can be prosecuted in less than a day—perhaps an afternoon. The results will depend on how well the assembled team understands the goals of the method, the techniques of the method, and the system itself. The evaluation team, being internal, is typically *not* objective, and this may compromise the value of its results—one tends to hear fewer new ideas

and fewer dissenting opinions. But this version of evaluation is inexpensive, easy to convene, and relatively low ceremony, so it can be quickly deployed whenever a project wants an architecture quality assurance sanity check.

21.4 Summary

If a system is important enough for you to explicitly design its architecture, then that architecture should be evaluated.

The number of evaluations and the extent of each evaluation may vary from project to project. A designer should perform an evaluation during the process of making an important decision. Lightweight evaluations can be performed several times during a project as a peer review exercise.

The ATAM is a comprehensive method for evaluating software architectures. It works by having project decision makers and stakeholders articulate a precise list of quality attribute requirements (in the form of scenarios) and by illuminating the architectural decisions relevant to carrying out each high-priority scenario. The decisions can then be understood in terms of risks or nonrisks to find any trouble spots in the architecture.

Lightweight Architecture Evaluation, based on the ATAM, provides an inexpensive, low-ceremony architecture evaluation that can be carried out in an afternoon.

21.5 For Further Reading

For a more comprehensive treatment of the ATAM, see [Clements 01b].

Multiple case studies of applying the ATAM are available. They can be found by going to www.sei.cmu.edu/library and searching for "ATAM case study."

To understand the historical roots of the ATAM, and to see a second (simpler) architecture evaluation method, you can read about the software architecture analysis method (SAAM) in [Kazman 94].

Several lighter weight architecture evaluation methods have been developed. They can be found in [Bouwers 10], [Kanwal 10], and [Bachmann 11].

Maranzano et al. have published a paper dealing with a long tradition of architecture evaluation at AT&T and its successor companies [Maranzano 05].

21.6 Discussion Questions

1. Think of a software system that you're working on. Prepare a 30-minute presentation on the business drivers for this system.
2. If you were going to evaluate the architecture for this system, who would you want to participate? What would be the stakeholder roles and who could you get to represent those roles?
3. Use the utility tree that you wrote for the ATM in Chapter 16 and the design that you sketched for the ATM in Chapter 17 to perform the scenario analysis step of the ATAM. Capture any risks and nonrisks that you discover. Better yet, perform the analysis on the design carried out by a colleague.
4. It is not uncommon for an organization to evaluate two competing architectures. How would you modify the ATAM to produce a quantitative output that facilitates this comparison?
5. Suppose you've been asked to evaluate the architecture for a system in confidence. The architect isn't available. You aren't allowed to discuss the evaluation with any of the system's stakeholders. How would you proceed?
6. Under what circumstances would you want to employ a full-strength ATAM and under what circumstances would you want to employ a Lightweight Architecture Evaluation?

Using the Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study

Mario Barbacci
Paul Clements
Anthony Lattanze
Linda Northrop
William Wood

July 2003

Architecture Tradeoff Analysis Initiative

Technical Note
CMU/SEI-2003-TN-012

Contents

List of Tables

About the Technical Note Series on Business and Acquisition Guidelines

The Product Line Systems Program at the Software Engineering Institute (SEI[SM]) is publishing a series of technical notes designed to condense knowledge about architecture tradeoff analysis practices into a concise and usable form for the Department of Defense (DoD) acquisition manager and practitioner. This series is a companion to the SEI series on product line acquisition and business practices.

Each technical note in the series will focus on applying architecture tradeoff analysis in the DoD. Our objective is to provide practical guidance to early adopters on ways to integrate sound architecture tradeoff analysis practices into their acquisitions. By investigating best commercial and government practices, the SEI is helping the DoD to overcome challenges and increase its understanding, maturation, and transition of this technology.

Together, these two series of technical notes will lay down a conceptual foundation for DoD architecture tradeoff analysis and product line business and acquisition practices. Further information is available on the SEI's Product Line Systems Program Web page at <http://www.sei.cmu.edu/activities/plp/plp_init.html>.

[SM] Software Engineering Institute is a service mark of Carnegie Mellon University.

Abstract

The quality of a software-intensive system depends heavily on the system's software architecture. When used appropriately, software architecture evaluations can have a favorable effect on a delivered or modified government system. This technical note describes the application of the Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) to an Army avionics system acquisition. A government–contractor team is developing the Common Avionics Architecture System (CAAS) for a family of U.S. Army Special Operations helicopters. This technical note presents the contextual background about the software architecture, the organization, and the system being evaluated. It also provides a general overview of the ATAM process, describes the application of the ATAM to the CAAS, and presents important results and benefits.

1 Introduction

Because software architecture is a major determinant of software quality, it follows that software architecture is critical to the quality of any software-intensive system. For a Department of Defense (DoD) acquisition organization, the ability to evaluate software architectures before they are realized in finished systems can substantially reduce the risk that the delivered systems will not meet their quality goals.

Over the past several years, the Software Engineering Institute (SEI[SM]) has developed the Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) and validated its usefulness in practice [Clements 02b, Kazman 00]. This method not only permits evaluation of specific architectural quality attributes (e.g., modifiability, performance, security, and reliability) but also engineering tradeoffs to be made among possibly conflicting quality goals.

This technical note describes an ATAM evaluation of the software architecture for an avionics system developed for the Technology Applications Program Office (TAPO) of the U.S. Army Special Operations Command Office. The system, called the Common Avionics Architecture System (CAAS), is being developed by Rockwell Collins in Cedar Rapids, Iowa.

Following this introduction, Section 2 provides background on software architecture, a description of TAPO and its motivations for commissioning the evaluation, and an overview of the CAAS. Section 3 contains an overview of the ATAM including its purpose and primary steps. Section 4 describes how the ATAM was applied specifically to the CAAS, and Section 5 presents some results of the ATAM evaluation.

[SM] SEI, Architecture Tradeoff Analysis Method, and ATAM are service marks of Carnegie Mellon University.

2 Context for the Architecture Evaluation

2.1 Software Architecture

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 03].

The software architecture for a system represents the earliest software design decisions. As such, they are the most critical things to get right and the most difficult things to change downstream in the development life cycle. Even more important, the software architecture is the key to software quality; it permits or precludes the system's ability to meet functional and quality attribute goals and requirements such as reliability, modifiability, security, real-time performance, and interoperability. The right software architecture can pave the way for successful system development, while the wrong architecture results in a system that fails to meet critical requirements and incurs high maintenance costs.

Modern treatment of software architecture takes advantage of the fact that there are many relevant views of a software architecture. A *view* is a representation of some of the system's elements and the relationships associated with them. Views help us to separate concerns and achieve intellectual control over an abstract concept. Different views also speak to different stakeholders—those who have a vested interest in the architecture. Which ones are relevant depends on the stakeholders and the system properties that interest them. If we consider the analogy of a building's architecture, various stakeholders (such as the construction engineer, plumber, and electrician) all have an interest in how the building is to be constructed. Although they are each interested in different elements and relationships, each of their views is valid—each one represents a structure that maps to one of the building's construction goals. A suite of views is necessary to engineer the architecture of the building fully and to represent that architecture to stakeholders.

Similarly, a software architecture has a variety of stakeholders, including possibly the development organization, end user, system maintainer, operator, and acquisition organization. Each stakeholder has a vested interest in different system properties and goals that are represented by different structural views of the system. These different properties and goals, and their corresponding architectural views are important to engineer, understand, and analyze. They all provide the basis for reasoning about the appropriateness and quality of the architecture.

Some experts prescribe using a fixed set of views. Rational's Unified Process (RUP), for example, relies on Kruchten's "4+1 view" approach to software architecture. A current and

more healthy trend, however, is to recognize that architects should choose a set of views based on the needed engineering leverage that each view provides and the stakeholder interests that each one serves. This trend is exemplified by the recent American National Standards Institute/Institute of Electrical and Electronics Engineers (ANSI/IEEE) recommended practice for architectural documentation on software-intensive systems [IEEE 00] and the "views and beyond" approach to architecture documentation from the SEI [Clements 03].

Some common architectural views include [Clements 03]

- the module decomposition view, which comprises a set of implementation units related by the "is part of" relation, showing how the system's functionality is assigned to software. A system's software modifiability flows from carefully assigning responsibilities to modules.
- the layered view, which shows how a system's modules are related using the "allowed to use" relation. Layers group elements into sets that provide key abstractions and impose a usage restriction so that upper layers can use only layers below them, imparting modifiability and portability to the architecture.
- the communicating-processes view, which comprises a set of processes or threads, and shows how they interact with each other at runtime. This view provides engineering leverage on performance, schedulability, and absence of deadlock, among other qualities.
- the deployment view, which shows how software elements are allocated to hardware such as processors, storage devices, and external devices or sensors, along with the communication paths that connect them. The deployment view provides insight into real-time performance and reliability, among other qualities.

Other software architectural views are described in *Documenting Software Architectures: Views and Beyond* [Clements 03] and *Software Architecture in Practice*, 2nd edition [Bass 03].

2.2 The TAPO Organization

TAPO's mission is to use streamlined acquisition procedures to rapidly procure and integrate non-developmental item (NDI) equipment and systems for Special Operations Aviation (SOA), manage SOA's modifications and configuration control, provide logistics sustainment of SOA-specific equipment and systems, and, where applicable, transition SOA-specific equipment and systems to conventional Army aircraft. TAPO's role is that of an NDI systems integrator.

TAPO hopes to reduce integration costs, logistical support, training resources, and technical risk, and shorten the schedule by exploiting the commonality among the three classes of aircraft that it supports: A/MH-6, MH-47, and MH-60. Best commercial practice has demonstrated significant advantages through a product line approach to software. A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements 02a]. A software

product line is most effectively built from a common architecture that is used to structure a common set of components and other assets from which the set of related products is built. TAPO's goal is to develop a single cockpit architecture for all MH-47s, MH-60s, andA/MH-6s that will form the basis for a software product line approach for all the systems that TAPO supports.

The CAAS was already well into development when the SEI was commissioned to perform the architecture evaluation. While an argument could be made that an evaluation earlier in the life cycle would have been a better risk-mitigation approach, this evaluation served three important purposes:

1. It gave the government a concise idea of where its architecture was at risk and identified immediate remedial actions.
2. It gave the government confidence in many areas where the architecture was shown to be sound and well engineered.
3. It produced a cohesive community of stakeholders in the architecture. These stakeholders articulated several possible new evolutionary directions for the system that the program office may not have considered previously.

2.3 The CAAS

Over the next several years, U.S. Army Special Operations helicopters will be upgraded with modern avionics to improve their capabilities, while making them easier to maintain, allowing third-party upgrades, and providing a common "look and feel" across all platforms. The CAAS will be deployed in all the upgraded helicopters, replacing two different closed and proprietary avionics systems previously in use across the Special Operations fleet. The CAAS provides capabilities to "aviate, navigate, and communicate"—in the parlance of the domain—and helps manage the cockpit workload and enhance situational awareness.

If the engine is an aircraft's heart, then the avionics system is its brain. The avionics system is responsible for getting critical information to the aircrew in a timely manner, managing aircraft systems, and helping the crew do its job. As described in *National Defense* magazine:

> The aircraft of the Army's 160th Special Operations Aviation Regiment (SOAR) are uniquely equipped to fly long missions in hostile airspace at night and in adverse weather. In one typical mission early in Operation Enduring Freedom last year, an MH-47E carried a Special Forces A-team from a forward support base in Uzbekistan to a rendezvous with anti-Taliban forces in Afghanistan. The Chinook crew used multi-mode radar to find a way through mountains in zero visibility, refueled at low altitude from an Air Force MC-130P tanker and exercised aircraft survivability equipment to counter an apparent air defense threat. The MH-47E then penetrated a dust storm before climbing over mountains into solid cloud. From 16,500 feet, the Chinook crew again used terrain following radar to let down into the target

area and deliver the special operators. On the return, they descended from altitudes near 20,000 feet in zero visibility, refueled once more from an MC-130, and again used radar to negotiate the mountains. Overall, the mission lasted 8.3 hours, including 6.3 hours in adverse weather over hostile territory [Colucci 03].

3 The ATAM

The ATAM relies on the principle that an architecture is suitable (or not suitable) only in the context of specific quality attributes that it must impart to the system. The ATAM uses stakeholders' perspectives to produce a collection of scenarios that define the qualities of interest for the particular system under consideration. Scenarios give specific instances of usage, performance and growth requirements, various types of failures, and various possible threats and modifications. Once the important quality attributes are identified in detail, the architectural decisions relevant to each one can be illuminated and analyzed with respect to their appropriateness.

The steps of the ATAM leading up to analysis are carried out in two phases.[1] In Phase 1, the evaluation team interacts with the system's primary decision makers: the architect(s), manager(s), and perhaps a marketing or customer representative. During Phase 2, a larger group of stakeholders is assembled, including developers, testers, maintainers, administrators, and users. The two-phase approach ensures that the analysis is based on a broad and appropriate range of perspectives.

The steps of the ATAM are as follows.

Phase 1:

1. **Present the ATAM.** The evaluators explain the method so that those who will be involved in the evaluation understand it.
2. **Present the business drivers.** Appropriate system representative(s) present an overview of the system, its requirements, business goals, and context, and the architectural quality attribute drivers.
3. **Present the architecture.** The system or software architect (or another lead technical person) presents the architecture.
4. **Catalog the architectural approaches.** The system or software architect presents general architectural approaches to achieving specific qualities. The evaluation team captures a list and adds to it any approaches observed during Step 3 or learned during the pre-exercise review of the architecture documentation; for example, "A cyclic executive is used to ensure real-time performance." Known architectural approaches have known quality attribute properties, and those approaches will help carry out the analysis steps.

[1] These two phases, visible to the stakeholders, are bracketed by a preparation phrase up front and a follow-up phase at the end; both are carried out behind the scenes.

5. **Generate a quality attribute utility tree.** Participants build a utility tree—a prioritized set of detailed statements about which quality attributes are most important for the architecture to carry out (such as performance, modifiability, reliability, or security) and specific scenarios that express those attributes.
6. **Analyze the architectural approaches.** The evaluators and the architect(s) map the utility tree scenarios to the architecture to see how it responds to each important scenario.

Phase 2 begins with an encore of Step 1 and a recap of the results of Steps 2 through 6 for the larger group of stakeholders. Then Phase 2 continues with these steps:

7. **Brainstorm and prioritize the scenarios.** The stakeholders brainstorm additional scenarios that express specific quality concerns. Afterwards, the group chooses the most important ones using a facilitated voting process.
8. **Analyze the architectural approaches.** As in Step 6, the evaluators and the architect(s) map the high-priority brainstormed scenarios to the architecture.
9. **Present the results.** A presentation and final report are produced that capture the results of the process and summarize the key findings.

Scenario analysis produces the following results:

- a collection of sensitivity and tradeoff points. A *sensitivity point* is an architectural decision that affects the achievement of a particular quality. A *tradeoff point* is an architectural decision that affects more than one quality attribute (possibly in opposite ways).
- a collection of risks and non-risks. A *risk* is an architectural decision that is problematic in light of the quality attributes that it affects. A *non-risk* is an architectural decision that is appropriate in the context of the quality attributes that it affects.
- a list of issues or decisions not yet made. Often during an evaluation, issues not directly related to the architecture arise. They may have to do with an organization's processes, personnel, or special circumstances. The ATAM process records them so that they can be addressed by other means. The list of decisions not yet made arises from the stage of the evaluation's life cycle. An architecture represents a collection of decisions. All relevant decisions might have been made at the time of the evaluation, even when designing the architecture. Some of those decisions are known to the development team as having not been made and are on a list for further consideration. Others are news to the development team and stakeholders.

Results of the overall exercise also include the summary of the business drivers, the architecture, the utility tree, and the analysis of each chosen scenario. All these results are recorded visibly so all stakeholders can verify that the results have been identified correctly.

The number of scenarios that are analyzed during the evaluation is controlled by the amount of time allowed for the evaluation, but the process ensures (via active prioritization) that the most important ones are addressed.

After the evaluation, the evaluators write a report documenting the evaluation and recording the information discovered. This report will also document the framework for ongoing analysis that was discovered by the evaluators.

4 The Evaluation of the CAAS Software Architecture

4.1 Background

Phase 1 of the evaluation took place at the Rockwell Collins facility in Cedar Rapids, Iowa on October 16, 2002. Twelve "decision maker" stakeholders were present, a number somewhat above average. They included five members of the contractor organization (architects and program managers), two members of the TAPO program office, and five members of the 160th Special Operations Aviation Regiment from Ft. Campbell, Kentucky, representing the user community. During Phase 1, six scenarios were analyzed.

Phase 2 took place at Fort Campbell on December 17 and 18, 2002. Fifteen stakeholders were present, representing various roles with a vested interest in the CAAS software architecture. Once again, the stakeholders came from the program office, the user community, and the contractor. Here, an additional 9 scenarios were analyzed, for a total of 15.

In both cases, the evaluation team consisted of four members of the technical staff from the SEI's Product Line Systems Program.

4.2 Business and Mission Drivers

Step 2 of the ATAM is a presentation of business drivers. A TAPO representative described the business objectives for the CAAS from the point of view of the government (in particular, the acquiring organization and the user organization), the driving business requirements for the CAAS products, and the architecture goals derived from those requirements.

Overall, the goal of the CAAS is to create a scalable system that meets the needs of multiple helicopter cockpits to address modernization issues. Its approach is to use a single, open, common avionics architecture system for all platforms to reduce the cost of ownership. This approach is based on Rockwell Collins' Cockpit Management System (CMS) in its Flight 2 family of avionics systems, augmented with IAS[2] functionality.

The four primary business drivers for the CAAS are

1. Minimize systems acquisition and sustainment costs.
2. Reduce the total ownership cost over the life of the avionics system.
3. Modernize the system with minimal impact to Army Special Operations readiness.
4. Ensure that new CAASs work well with other programs.

Strategies to achieve these drivers include

[2] IAS is a legacy avionics system developed by another contractor.

- Leverage existing code and documentation where practical.
- Aim for a “plug and play” architecture in which hardware is portable between platforms.
- Move away from single proprietary components and use standards where appropriate.
- Provide cross-platform commonality (hardware, software, and training).
- Reduce the logistics base.
- Use the Service Life Extension Program (SLEP) as a fleet modification/installation center as much as possible.
- Leverage other platform developments (both commercial and DoD).
- Employ a system infrastructure or framework that facilitates system maintenance and enhancements (including third-party participation in them).

Additional business drivers that play a role are the constraints on the system and its architecture, which, in the CAAS, include a large set of applicable standards.

The business drivers are elicited to establish the broad requirements and context for the system, but also to identify the driving quality attribute requirements for the architecture. For the CAAS, the important quality attributes (in order of customer importance) are

- availability
- performance (i.e., provide timely answers)
- modifiability

Modifiability, as is often the case, manifests itself in a number of ways. For the CAAS, modifiability refers to the following capabilities:

- growth
- testability
- openness
- portability of hardware and software across different aircraft platforms
- reconfigurability
- repeatability (That is, every system must look like every other system and provide the same answer on every platform.)
- supportability (i.e., the ability of a third party to maintain the system)
- reuse
- affordability

These quality attributes serve as the first approximation qualities of importance in the generation of the quality attribute utility tree, detailed in Section 4.4.

4.3 Architectural Approaches

Step 4 of the ATAM captures the list of architectural approaches gleaned from the architecture presentation, as well as from the evaluation team's pre-exercise review of the architecture documentation. Because architectural approaches exhibit known effects on quality attributes (e.g., redundancy increases availability, whereas layering increases portability), explicitly identifying the approaches provides input for the analysis steps that follow.

The overarching strategies used in the CAAS are strongly partitioning applications and structuring the software as a series of layers. POSIX provides the primary partitioning mechanism that establishes inviolable timing and memory walls between applications to prevent conflicts. For example, an important result of this design is that an application that for some reason overruns its processing time limit cannot cause another application to be late. Similarly, an application that overruns its memory allotment cannot impinge on another application's memory area.

Applications hence represent encapsulations. One of the strongest aspects of the CAAS architecture is that its design allows applications to be changed more or less independently of each other. That aspect, along with location transparency (an application's independence from its hardware location), makes adding new applications a straightforward process.

The CAAS software architectural approaches are listed below along with the quality attributes that each one nominally affects. The attributes are shown in parentheses.

1. consistent partitioning strategy: definition of a partition, "brick-wall partitioning"
   (availability, safety, modifiability, testability, maintainability)
2. encapsulation: used to isolate partitions. Between partitions, applications can share only their state via the network. The remote service interface (RSI) and remote service provider (RSP) are examples of encapsulation that isolate the network implementation details.
   (modifiability, availability)
3. interface strategy: Accessing components only via their interfaces is strictly followed. Besides controlling interactions and eliminating the back-door exploitation of changeable implementation details, this strategy reduces the number of inputs and outputs per partition.
   (modifiability, maintainability)
4. layers: used to partition and isolate high-level graphics services
   (portability, modifiability)
5. distributed processing: Predominantly, a client-server approach is used to decouple "parts" of the system. Also, the Broadcast feature is used to broadcast information periodically.
   (maintainability, modifiability)

6. Access to sockets, bandwidth, and data is guaranteed.
   (performance)
7. virtual machine: a flight-ready operating system that's consistent with POSIX and that has a standard POSIX application program interface (API) and Ada 95 support, which both provide Level-A design assurance
   (modifiability, availability)
8. health monitors: for checking the health of control display units (CDUs) and multi-function displays (MFDs)
   (availability)
9. use of commercial standards: including ARINC 661, POSIX, Common Object Request Broker Architecture (CORBA), IEEE P1386/P1386.1, OpenGL, and DO 178B
   (portability, maintainability, modifiability)
10. locational transparency: Applications do not know where other applications reside, and, hence, are unaffected when applications migrate to other hardware for scheduling or load-balancing reasons. The location is bound at configuration time.
    (portability, modifiability)
11. isolation of system services: a by-product of the layering strategy
    (portability, modifiability)
12. redundant software: For flight-critical functions, redundant software is introduced using a master/slave protocol to manage failover.
    (portability, availability)
13. Every application is resident on every box.
    (portability)
14. Some applications are active on multiple boxes.
    (availability)
15. memory and performance analysis: Partitions are cyclic; however, rate monotonic analysis (RMA) is used to assign priorities to threads within partitions. The result is assured schedulability.
    (performance)
16. application templates (the shell): A standard template for applications incorporates application, common software, and common reusable elements (CoRE), and ensures that complicated protocols (such as failover) are handled consistently across all applications.
    (reuse, modifiability, repeatability, affordability)

4.4 Utility Tree

Step 5 of the ATAM produces a quality attribute utility tree. That tree provides a vehicle for translating the quality attribute goals articulated in the business drivers presentation to "testable" quality attribute scenarios. The tree uses "Utility" as the root node—an expression

of the overall “goodness” of the system. Performance, modifiability, security, and availability are typical of the high-level nodes, placed immediately under “Utility.” For the CAAS evaluation, the second-level nodes were identified as availability, performance, modifiability, affordability, and reliability.

Under each quality factor are specific subfactors called “attribute concerns” that arise from considering the quality-attribute-specific stimuli and responses that the architecture must address. For example, for the CAAS, availability was defined by the stakeholders to mean “having a non-crashing operational flight program (OFP),” “graceful degradation in the presence of failures,” and “no degradation in the presence of failures for which there are redundant components/paths.” Finally, each attribute concern is elaborated by a small number of scenarios that are leaves of the utility tree; thus, the tree has four levels:

Utility/quality attributes/attribute concern/scenarios

A scenario represents a use or modification of the architecture, applied not only to determine if the architecture meets a functional requirement, but also (and more significantly) to predict system qualities such as performance, reliability, modifiability, and so forth.

The scenarios at the leaves of the utility tree are prioritized along two dimensions:

1. importance to the system
2. perceived risk in achieving this goal

These nodes are prioritized relative to each other, using relative rankings of high, medium, and low.

The portion of the utility tree covering the quality attribute of availability from the CAAS evaluation is reproduced in Table 1, without the rankings. The full utility tree contained 29 scenarios covering 5 quality attributes.

Table 1: Utility Tree for the Availability Quality Attribute

Phase 1: Quality Attribute Utility Tree	
Quality Attribute	availability
Attribute Concerns	The OFP doesn't crash.
Scenarios	1. Invalid data is entered by the pilot, and the system does not crash.
	2. Invalid data comes from an actor on any bus, and the system does not crash.
	3. When a 1.9-second power interruption occurs, the system will execute a warm boot and be fully operational in 2 seconds.
Attribute Concerns	graceful degradation in the presence of failures
Scenarios	1. A loss of Doppler occurs, the pilot is notified, and the Doppler timer begins a countdown (for multi-mode radar [MMR] validity).
	2. A partition fails, the rest of the processor continues working, and the system continues to function.
Attribute Concerns	no degradation in the presence of failures for which there are redundant components/paths
Scenarios	1. The data concentrator suffers battle damage, and all flight-critical information is still available.
	2. The mission processor in the outboard MFD fails, and that display and the rest of the system continue to operate normally.

For the CAAS evaluation, the evaluation team chose six of the highest priority scenarios and analyzed them during Phase 1. The scenarios that were not analyzed were distributed to Phase 2 participants as "seed scenarios" that they could place into the brainstorming pool, if desired.

4.5 Scenario Generation and Prioritization

In addition to the scenarios at the leaves of the utility tree, a scenario elicitation process in Step 7 allows stakeholders to contribute additional scenarios that reflect their concerns and understanding of how the architecture will accommodate their needs. A particular scenario may, in fact, have implications for many stakeholders. For a modification, one stakeholder may be concerned with the difficulty of a change and its performance impact, while another may be interested in how the change will affect the integrability of the architecture.

After the scenarios were generated, the stakeholders were given the opportunity to merge those that seemed to address closely related concerns. The purpose of scenario consolidation is to prevent votes from being split across two almost-alike scenarios.

After merging, the scenarios were prioritized using a voting process in which participants were given six votes[3] that they could allocate to any scenario or group of scenarios.

A few of the 21 scenarios brainstormed during Step 7 are shown in Table 2.

[3] The number of votes is 30% of the number of brainstormed scenarios, rounded up to the nearest integer. This is a common facilitated brainstorming and group-consensus technique.

Table 2: Brainstormed Scenarios from Step 7

Phase 2: Brainstormed Scenarios		
Scenario Number	**Scenario Text**	**Number of Votes**
2	Changes to the CAAS are reflected in the simulation and training system concurrently with the airframe changes, without coding it twice (simulation and training stakeholder).	5
3	No single point of failure in the system will affect the system's safety or performance (system architect stakeholder).	10
5	Multiple versions of the system must be fielded at the same time. Those versions should be distinguishable and should not have a negative impact on the rest of the system (system implementer stakeholder).	1
9	75% of the CAAS is built from reused components increasing new business opportunities (from Phase 1, program manager stakeholder).	9
13	Given maximum "knob twiddling" to the level that the system's performance is degraded, the system can prioritize its flight-critical functions, so they are NOT degraded (safety stakeholder).	6
15	Given the need for a second ARC231, the radio can be incorporated into the existing system by reusing existing software at minimal or no cost (requirements stakeholder).	2
20	An application doesn't crash, but starts producing bad data. The system can detect the errant data and when applications crash (reliability stakeholder).	3

Step 7 concludes with an examination of how the newly introduced high-priority scenarios compare with the high-priority scenarios identified in the utility tree. A low degree of overlap (in terms of the specific, detailed quality attributes addressed by the scenarios) could indicate that the project's decision makers and stakeholders had different expectations. That would constitute a risk to the project.

4.6 Overview of the Analysis Process

A total of 15 scenarios were analyzed during the course of the CAAS evaluation: 6 from the Phase 1 utility tree and 9 from the Phase 2 scenario generation/prioritization process.

The evaluation team concluded that the CAAS architecture seemed sound with respect to most of the behavioral and quality attribute requirements levied on it. The architecture was found to be well partitioned, and although the size of some of the partitions was a concern with respect to their modifiability, overall, the partitioning scheme provided a robust foundation for evolutionary flexibility. The evaluation found that the architecture was robust with respect to the addition of new functionality and hardware.

The evaluation identified 18 risks related to the software architecture's ability to satisfy its behavioral, quality attribute, and evolutionary goals. In addition, 12 non-risks (areas of

design strength) were identified.[4] An example risk was "OFP has no built-in hooks to aid in simulation/training capability," referring to the goal of making the same operational software drive both the simulators and the actual helicopters. An example non-risk was "the number of sockets used by the system is known and guaranteed," allowing reliable performance estimation to be carried out.

Five sensitivity points (e.g., "isolating operating system dependencies enhances portability") and two tradeoff points (e.g., "letting users set I/O parameters [such as turbine gas temperature limits] increases flexibility and usability, while decreasing safety") were cataloged.

In addition to the items described above, the evaluation team collected a series of *issues*—areas of programmatic concern not directly related to the technical aspects of the software architecture. For example, one issue involved the stakeholders' expressed need for a new functional capability that was out of the scope of the current requirements and that would be hard to provide. These issues were then brought to the attention of the program office, where they were handled appropriately.

The identified risks suggest a set of four "themes" in the architecture. These themes represent the key architectural issues posing potential problems for future success and possibly threatening the business drivers identified during Step 2. For example, several of the analyzed scenarios dealt with performance, and revealed risks about unknown performance requirements and certain performance goals not being met. These scenarios suggested a risk theme: More attention should be focused on performance. Such concise syntheses allowed the program office to focus on a few key areas that would improve its chances for success in both current development and future evolution.

4.7 Post-ATAM Activities

Although the details of post-ATAM activities are still being worked out with the program office, there are several possibilities. For example, the evaluation uncovered places where the software architecture documentation could be improved, and the SEI can help TAPO to do that.

[4] The ATAM process concentrates on identifying risks rather than non-risks, and so oftentimes, more risks are uncovered than non-risks. The relative numbers are not indicative of the architecture's quality.

5 Conclusions

5.1 Benefits

Every ATAM exercise is followed by a survey of stakeholders in attendance. The survey asks, "Do you feel you will be able to act on the results of the architecture evaluation? What follow-on actions do you anticipate? How much time do you anticipate spending on them?" Here is what those responding had to say verbatim:

- "Yes, greater review of latency"
- "Yes, I would expect the government to consider program changes to make improvements to the system. There will be detailed review and action plans delivered for risk items."
- "Difficult to ascertain since this review was conducted during code and test phase of program. We will act on those items that affect future business/enhancements in conjunction with our customer."
- "Minimal actions by a user are available."
- "Because we are so far down the road with the CAAS, we don't see any major changes, but we are better aware of risk areas."
- "I expect to spend at least hours in meetings/dialogue about the [risks and issues found]."
- "Immediate impact is limited due to the point we are at in the program. Little will be done now. May have impact as the system is evolved."

The survey asks the participants if they feel the exercise was useful. Here is what they said:

- "Yes, [it] caused a critical look at the CAAS. It validated some architectural decisions and raised questions about others."
- "In general, evaluation process seems worthwhile.
- "As a maintainer/trainer, it helped me to understand the system."
- "Unknown at this point"
- "Yes, very useful"
- "Yes, it brought issues and concerns to our attention."

Several stakeholders lamented that the evaluation was not performed earlier. Since the project was already in the development phase, the evaluation may have had a lesser impact than it would have otherwise:

- "Would have been more useful earlier on in the design/requirements definition phase"
- "[It was useful], but done too late to be of significant impact."

- "Do earlier! This type of exercise would have been more useful 12-14 months ago when the design decisions were being made. If ATAM was accomplished then, more issues/risks could have been addressed."
- "The whole process would have been more useful if it had taken place one to two years ago."

5.2 Summary

Overall, this evaluation succeeded in

- raising awareness of the importance of stakeholders in the architecture process
- establishing a community of vested stakeholders and opening channels of communication among them
- identifying a number of risk themes that can be made the subject of intense mitigation efforts that, even though the system is in development, can be effective in heading off disaster
- raising a number of issues with respect to previously unplanned capabilities for which some stakeholders expressed an acute need
- elevating the role of software architecture in system acquisition

It demonstrates the applicability and usefulness of software architecture evaluation in a DoD acquisition context. The evaluation comments underscore the SEI's advice that software architecture evaluations be performed *before* code is developed so that surfaced risks can be mitigated when it is least costly to do so. There never seems to be time in a development schedule to insert an architecture evaluation, but other ATAM evaluations have proven that the time spent saves considerable time and cost later [Clements 02b].

References

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, 2nd edition. Boston, MA: Addison-Wesley, 2003.

[Clements 02a] Clements, P. & Northrop, L. *Software Product Lines: Practices and Patterns.* Boston, MA: Addison-Wesley, 2002.

[Clements 02b] Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Boston, MA: Addison-Wesley, 2002.

[Clements 03] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2003.

[Colucci 03] Colucci, F. "Avionics Upgrade Underway for Special Ops Helicopters." *National Defense 87*, 591 (February 2003): 24-26. <http://www.nationaldefensemagazine.org/article.cfm?Id=1029>.

[IEEE 00] IEEE Computer Society, Software Engineering Standards Committee, Institute of Electrical and Electronics Engineers, IEEE-SA Standards Board, and IEEE Xplore. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* New York, NY: Institute of Electrical and Electronic Engineers, 2000.

[Kazman 00] Kazman, R.; Klein, M.; & Clements, P. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, ADA382629). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2003	Final

4. TITLE AND SUBTITLE
Using the Architecture Tradeoff Analysis Method℠ (ATAM℠) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Mario Barbacci, Paul Clements, Anthony Lattanze, Linda Northrop, William Wood

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2003-TN-012

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

The quality of a software-intensive system depends heavily on the system's software architecture. When used appropriately, software architecture evaluations can have a favorable effect on a delivered or modified government system. This technical note describes the application of the Architecture Tradeoff Analysis Method℠ (ATAM℠) to an Army avionics system acquisition. A government–contractor team is developing the Common Avionics Architecture System (CAAS) for a family of U.S. Army Special Operations helicopters. This technical note presents the contextual background about the software architecture, the organization, and the system being evaluated. It also provides a general overview of the ATAM process, describes the application of the ATAM to the CAAS, and presents important results and benefits.

14. SUBJECT TERMS
TAPO, ATAM, Architecture Tradeoff Analysis Method, acquisition organizations, experience report

15. NUMBER OF PAGES
30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

ATAM: Method for Architecture Evaluation

Rick Kazman
Mark Klein
Paul Clements

August 2000

TECHNICAL REPORT
CMU/SEI-2000-TR-004
ESC-TR-2000-004

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

ATAM:[SM] Method for Architecture Evaluation

CMU/SEI-2000-TR-004
ESC-TR-2000-004

Rick Kazman
Mark Klein
Paul Clements

August 2000

Product Line Systems

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

If a software architecture is a key business asset for an organization, then architectural analysis must also be a key practice for that organization. Why? Because architectures are complex and involve many design tradeoffs. Without undertaking a formal analysis process, the organization cannot ensure that the architectural decisions made—particularly those which affect the achievement of quality attribute such as performance, availability, security, and modifiability—are advisable ones that appropriately mitigate risks. In this report, we will discuss some of the technical and organizational foundations for performing architectural analysis, and will present the Architecture Tradeoff Analysis Method[SM] (ATAM)—a technique for analyzing software architectures that we have developed and refined in practice over the past three years.

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

1 Introduction

The purpose of this report is to describe the theory behind the Architecture Tradeoff Analysis Method[SM](ATAM) and to discuss how it works in practice. The ATAM gets its name because it not only reveals how well an architecture satisfies particular quality goals (such as performance or modifiability), but it also provides insight into how those quality goals interact with each other—how they *trade off* against each other. Such design decisions are critical; they have the most far-reaching consequences and are the most difficult to change after a system has been implemented.

When evaluating an architecture using the ATAM, the goal is to understand the consequences of architectural decisions with respect to the quality attribute requirements of the system. Why do we bother? Quite simply, an architecture is the key ingredient in a business or an organization's technological success. A system is motivated by a set of functional and quality goals. For example, if a telephone switch manufacturer is creating a new switch, that system must be able to route calls, generate tones, generate billing information and so forth. But if it is to be successful, it must do so within strict performance, availability, modifiability, and cost parameters. The architecture is the key to achieving—or failing to achieve—these goals. The ATAM is a means of determining whether these goals are achievable by the architecture as it has been conceived, *before* enormous organizational resources have been committed to it.

We have developed an architecture analysis *method* so that the analysis is repeatable. Having a structured method helps ensure that the right questions regarding an architecture will be asked *early*, during the requirements and design stages when discovered problems can be solved relatively cheaply. It guides users of the method—the stakeholders—to look for conflicts and for resolutions to these conflicts in the software architecture.

This method has also been used to analyze legacy systems. This frequently occurs when the legacy system needs to support major modifications, integration with other systems, porting, or other significant upgrades. Assuming that an accurate architecture of the legacy system is available (which frequently must be acquired and verified using architecture extraction and conformance testing methods [Kazman 99]), applying the ATAM results in increased understanding of the quality attributes of the system.

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

The ATAM draws its inspiration and techniques from three areas: the notion of architectural styles; the quality attribute analysis communities; and the Software Architecture Analysis Method (SAAM) [Kazman 94], which was the predecessor to the ATAM. The ATAM is intended for analysis of an architecture with respect to its quality attributes. Although this is the ATAM's focus, there is a problem in operationalizing this focus. We (and the software engineering community in general) do not understand quality attributes well: what it means to be "open" or "interoperable" or "secure" or "high performance" changes from system to system, from stakeholder to stakeholder, and from community to community.

Efforts on cataloguing the implications of using design patterns and architectural styles contribute, frequently in an informal way, to ensuring the quality of a design [Buschmann 96]. More formal efforts also exist to ensure that quality attributes are addressed. These consist of formal analyses in areas such as performance evaluation [Klein 93], Markov modeling for availability [Iannino 94], and inspection and review methods for modifiability [Kazman 94].

But these techniques, if they are applied at all, are typically applied in isolation and their implications are considered in isolation. This is dangerous. It is dangerous because all design involves tradeoffs and if we simply optimize for a single quality attribute, we stand the chance of ignoring other attributes of importance. Even more significantly, if we do not analyze for multiple attributes, we have no way of understanding the tradeoffs made in the architecture—places where improving one attribute causes another one to be compromised.

1.1 What is the Purpose of the ATAM?

It is important to clearly state what the ATAM is and is not:

> **The purpose of the ATAM is to assess the consequences of architectural decisions in light of quality attribute requirements.**

The ATAM is meant to be a risk identification method, a means of detecting areas of potential risk within the architecture of a complex software intensive system. This has several implications:

- The ATAM can be done early in the software development life cycle.
- It can be done relatively inexpensively and quickly (because it is assessing architectural design artifacts).
- The ATAM will produce analyses commensurate with the level of detail of the architectural specification. Furthermore it need not produce detailed analyses of any measurable quality attribute of a system (such as latency or mean time to failure) to be successful. Instead, success is achieved by identifying *trends*.

This final point is crucial in understanding the goals of the ATAM; we are not attempting to precisely predict quality attribute behavior. That would be impossible at an early stage of design; one doesn't have enough information to make such a prediction. What we are interested in doing—in the spirit of a risk identification activity—is learning where an attribute of interest is affected by architectural design decisions, so that we can reason carefully about those decisions, model them more completely in subsequent analyses, and devote more of our design, analysis, and prototyping energies on such decisions.

Thus, what we aim to do in the ATAM, in addition to raising architectural awareness and improving the level of architectural documentation, is to record any *risks*, *sensitivity points*, and *tradeoff points* that we find when analyzing the architecture. Risks are architecturally important decisions that have not been made (e.g., the architecture team has not decided what scheduling discipline they will use, or has not decided whether they will use a relational or object oriented database), or decisions that have been made but whose consequences are not fully understood (e.g., the architecture team has decided to include an operating system portability layer, but are not sure what functions need to go into this layer). Sensitivity points are parameters in the architecture to which some measurable quality attribute response is highly correlated. For example, it might be determined that overall throughput in the system is highly correlated to the throughput of one particular communication channel, and availability in the system is highly correlated to the reliability of that same communication channel. A tradeoff point is found in the architecture when a parameter of an architectural construct is host to more than one sensitivity point where the measurable quality attributes are affected differently by changing that parameter. For example, if increasing the speed of the communication channel mentioned above improves throughput but reduces its reliability, then the speed of that channel is a tradeoff point.

Risks, sensitivity points, and tradeoff points are areas of potential future concern with the architecture. These areas can be made the focus of future effort in terms of prototyping, design, and analysis.

A prerequisite of an evaluation is to have a statement of quality attribute requirements and a specification of the architecture with a clear articulation of the architectural design decisions. However, it is not uncommon for quality attribute requirement specifications and architecture renderings to be vague and ambiguous. Therefore, two of the major goals of ATAM are to

- elicit and refine a precise statement of the architecture's driving quality attribute requirements
- elicit and refine a precise statement of the architectural design decisions

Given the attribute requirements and the design decisions, the third major goal of ATAM is to

- evaluate the architectural design decisions to determine if they satisfactorily address the quality requirements

2 The Underlying Concepts

The ATAM focuses on quality attribute requirements. Therefore, it is critical to have precise characterizations for each quality attribute. Quality attribute characterizations answer the following questions about each attribute:

- What are the stimuli to which the architecture must respond?
- What is the measurable or observable manifestation of the quality attribute by which its achievement is judged?
- What are the key architectural decisions that impact achieving the attribute requirement?

The notion of a *quality attribute characterization* is a key concept upon which ATAM is founded.

One of the positive consequences of using the ATAM that we have observed is a clarification and concretization of quality attribute requirements. This is achieved in part by eliciting scenarios from the stakeholders that clearly state the quality attribute requirements in terms of stimuli and responses. The process of brainstorming scenarios also fosters stakeholder communication and consensus regarding quality attribute requirements. *Scenarios* are the second key concept upon which ATAM is built.

To elicit design decisions we start by asking what architectural approaches are being used to achieve quality attribute requirements. Our goal in asking this question is to elicit the architectural approaches, styles, or patterns used that contribute to achieving a quality attribute requirement. You can think of an architectural style as a template for a coordinated set of architectural decisions aimed at satisfying some quality attribute requirements. For example, we might determine that a client/server style is used to ensure that the system can easily scale its performance so that its average-case latency is minimally impacted by an anticipated doubling of the number of users of the enterprise software system.

Once we have identified a set of architectural styles or approaches we ask a set of attribute-specific questions (for example, a set of performance questions or a set of availability questions) to further refine our knowledge of the architecture. The questions we use are suggested by the attribute characterizations. Armed with knowledge of the attribute requirements and the architectural approaches, we are able to analyze the architectural decisions.

Attribute-based architectural styles (ABASs) [Klein 99a] help with this analysis. Attribute-based architecture styles offer attribute-specific reasoning frameworks that illustrate how each architectural decision embodied by an architectural style affects the achievement of a quality attribute. For example, a modifiability ABAS would help in assessing whether a publisher/subscriber architectural style would be well-suited for a set of anticipated modifications. The third concept upon which ATAM is found is, thus, the notion of *attribute-based architectural styles.*

In the remainder of this report we will briefly introduce the ATAM, explain its foundations, discuss the steps of the ATAM in detail, and concludes with an extended example of applying the ATAM to a real system.

3 A Brief Introduction to the ATAM

The ATAM is an analysis method organized around the idea that architectural styles are the main determiners of architectural quality attributes. The method focuses on the identification of business goals which lead to quality attribute goals. Based upon the quality attribute goals, we use the ATAM to analyze how architectural styles aid in the achievement of these goals. The steps of the method are as follows:

Presentation

1. **Present the ATAM**. The method is described to the assembled stakeholders (typically customer representatives, the architect or architecture team, user representatives, maintainers, administrators, managers, testers, integrators, etc.).
2. **Present business drivers**. The project manager describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability or time to market or high security).
3. **Present architecture**. The architect will describe the proposed architecture, focussing on how it addresses the business drivers.

Investigation and Analysis

4. **Identify architectural approaches**. Architectural approaches are identified by the architect, but are not analyzed.
5. **Generate quality attribute utility tree**. The quality factors that comprise system "utility" (performance, availability, security, modifiability, etc.) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized.
6. **Analyze architectural approaches**. Based upon the high-priority factors identified in Step 5, the architectural approaches that address those factors are elicited and analyzed (for example, an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). During this step architectural risks, sensitivity points, and tradeoff points are identified.

Testing

7. **Brainstorm and prioritize scenarios**. Based upon the exemplar scenarios generated in the utility tree step, a larger set of scenarios is elicited from the entire group of stakeholders. This set of scenarios is prioritized via a voting process involving the entire stakeholder group.

8. **Analyze architectural approaches**. This step reiterates step 6, but here the highly ranked scenarios from Step 7 are considered to be test cases for the analysis of the architectural approaches determined thus far. These test case scenarios may uncover additional architectural approaches, risks, sensitivity points, and tradeoff points which are then documented.

Reporting

9. **Present results**. Based upon the information collected in the ATAM (styles, scenarios, attribute-specific questions, the utility tree, risks, sensitivity points, tradeoffs) the ATAM team presents the findings to the assembled stakeholders and potentially writes a report detailing this information along with any proposed mitigation strategies.

4 Quality Attribute Characterizations

Evaluating an architectural design against quality attribute requirements necessitates a precise characterization of the quality attributes of concern. For example, understanding an architecture from the point of view of modifiability requires an understanding of how to measure or observe modifiability and an understanding of how various types of architectural decisions impact this measure. To use the wealth of knowledge that *already* exists in the various quality attribute communities, we have created characterizations for the quality attributes of performance, modifiability, and availability, and are working on characterizations for usability and security. These characterizations serve as starting points, which can be fleshed out further in preparation for or while conducting an ATAM.

Each quality attribute characterization is divided into three categories: *external stimuli, architectural decisions*, and *responses*. *External stimuli* (or just *stimuli* for short) are the events that cause the architecture to respond or change. To analyze an architecture for adherence to quality requirements, those requirements need to be expressed in terms that are concrete and measurable or observable. These measurable/observable quantities are described in the *responses* section of the attribute characterization. *architectural decisions* are those aspects of an architecture—components, connectors, and their properties—that have a direct impact on achieving attribute responses.

For example, the external stimuli for performance are events such as messages, interrupts, or user keystrokes that result in computation being initiated. Performance architectural decisions include processor and network arbitration mechanisms; concurrency structures including processes, threads, and processors; and properties including process priorities and execution times. Responses are characterized by measurable quantities such as latency and throughput.

For modifiability, the external stimuli are change requests to the system's software. architectural decisions include encapsulation and indirection mechanisms, and the response is measured in terms of the number of affected components, connectors, and interfaces and the amount of effort involved in changing these affected elements. Characterizations for performance, availability, and modifiability are given in Appendix A.

Our goal in presenting these attribute characterizations is not to claim that we have created an *exhaustive* taxonomy for each of the attributes, but rather to suggest a framework for thinking about quality attributes; a framework that we have found facilitates a reasoned and efficient inquiry to elicit the appropriate attribute-related information.

The attribute characterizations help to ensure attribute coverage as well as offering a rationale for asking elicitation questions. For example, irrespective of the style being analyzed we know that latency (a measure of response) is at least a function of

- resources such as CPUs and LANs
- resource arbitration such as scheduling policy
- resource consumption such as CPU execution time
- and external events such message arrivals

We know that these architectural resources must be designed so that they can ensure the appropriate response to a stimulus. Therefore, given a scenario such as "*Unlock all of the car doors within one second of pressing the correct key sequence*," the performance characterization inspires questions such as

- Is the one-second deadline a hard deadline (response)?
- What are the consequences of not meeting the one-second requirement (response)?
- What components are involved in responding to the event that initiates unlocking the door (architectural decisions)?
- What are the execution times of those components (architectural decisions)?
- Do the components reside on the same or different processors (architectural decisions)?
- What happens if several "unlock the door" events occur quickly in succession (stimuli)?

Are the servers single- or multi-threaded?	What is the location of firewalls and their impact on performance?
How are priorities assigned to processes?	What information is cached versus regenerated? Based upon what principles?
How are processes allocated to hardware?	What is the performance impact of a thin versus a thick client?
What is the physical location of the hardware and its connectivity?	How are resources allocated to service requests?
What are the bandwidth characteristics of the network?	How do we characterize client loading, (e.g., how many concurrent sessions, how many users)?
How is queuing and prioritization done in the network?	What are the performance characteristics of the middleware: load balancing, monitoring, reconfiguring services to resources?
Do you use a synchronous or an asynchronous protocol?	
What is the impact of uni-cast or multi-cast broadcast protocols?	

Figure 1: Examples of Performance Related Questions

These questions are inspired by the attribute characterizations and result from applying the characterization to architecture being evaluated. For example, see the performance questions in Figure 1 and consider how these questions might have been inspired by the performance characterization in Figure 16 of Appendix A.

Other examples of attribute-specific questions are shown in Figure 2.

Modifiability:

If this architecture includes layers/facades, are there any places there where the layers/facades are circumvented?

If this architecture includes a data repository, how many distinct locations in the architecture have direct knowledge of its data types and layout?

If a shared data type changes, how many parts of the architecture are affected?

Performance:

If there are multiple processes competing for a shared resource, how are priorities assigned to these processes and the process controlling the resource?

If there are multiple pipelines of processes/threads, what is the lowest priority for each process/thread in each pipeline?

If multiple message streams arrive at a shared message queue, what are the rates and distributions of each stream?

Are there any relatively slow communication channels along an important communication path (e.g., a modem)?

Availability:

If redundancy is used in the architecture, what type of redundancy (analytic, exact, functional) and how is the choice made between redundant components?

How are failures identified?

Can active as well as passive failures be identified?

If redundancy is used in the architecture, how long does it take to switch between instances of a redundant component?

Figure 2: Examples of Attribute-Specific Questions

5 Scenarios

In a perfect world, the quality requirements for a system would be completely and unambiguously specified in a requirements document that is evolving ahead of or in concert with the architecture specification. In reality, requirements documents are not written, or are written poorly, or do not properly address quality attributes. In particular, we have found that quality attribute requirements for both existing and planned systems are missing, vague, or incomplete. Typically the first job of an architecture analysis is to precisely elicit the specific quality goals against which the architecture will be judged. The mechanism that we use for this elicitation is the *scenario*.

A scenario is a short statement describing an interaction of one of the stakeholders with the system. A *user* would describe using the system to perform some task; his scenarios would very much resemble *use cases* in object-oriented parlance. A *maintainer* would describe making a change to the system, such as upgrading the operating system in a particular way or adding a specific new function. A *developer*'s scenario might talk about using the architecture to build the system or predict its performance. A *customer*'s scenario might describe how the architecture is to be re-used for a second product in a product line.

Scenarios provide a vehicle for concretizing vague development-time qualities such as modifiability; they represent specific examples of current and future uses of a system. Scenarios are also useful in understanding run-time qualities such as performance or availability. This is because scenarios specify the kinds of operations over which performance needs to be measured, or the kinds of failures the system will have to withstand.

5.1 Types of Scenarios

In ATAM we use three types of scenarios: *use case scenarios* (these involve typical uses of the existing system and are used for information elicitation); *growth scenarios* (these cover anticipated changes to the system), and *exploratory scenarios* (these cover extreme changes that are expected to "stress" the system). These different types of scenarios are used to probe a system from different angles, optimizing the chances of surfacing architectural decisions at risk. Examples of each type of scenario follow.

5.1.1 Use Case Scenarios

Use case scenarios describe a user's intended interaction with the completed, running system. For example,

1. There is a radical course adjustment during weapon release (e.g., loft) that the software computes in 100 ms. (performance)
2. The user wants to examine budgetary and actual data under different fiscal years without re-entering project data. (usability)
3. A data exception occurs and the system notifies a defined list of recipients by e-mail and displays the offending conditions in red on data screens. (reliability)
4. User changes graph layout from horizontal to vertical and graph is redrawn in one second. (performance)
5. Remote user requests a database report via the Web during peak period and receives it within five seconds. (performance)
6. The caching system will be switched to another processor when its processor fails, and will do so within one second. (reliability)

Notice that each of the above use case scenarios expresses a specific stakeholder's desires. Also, the stimulus and the response associated with the attribute are easily identifiable. For example, in the first scenario, "radical course adjustment during weapon release," the stimulus and latency goal of 100 ms. is called out as being the important response measure. For scenarios to be well-formed it must be clear what the stimulus is, what the environmental conditions are, and what the measurable or observable manifestation of the response is.

Notice also that the attribute characterizations suggest questions that can be helpful in refining scenarios. Again consider Scenario 1:

- Are data sampling rates increased during radical course adjustments?
- Are real-time deadlines shortened during radical course adjustments?
- Is more execution time required to calculate a weapon solution during radical course adjustments?

5.1.2 Growth Scenarios

Growth scenarios represent typical anticipated future changes to a system. Each scenario also has attribute-related ramifications, many of which are for attributes other than modifiability. For example, Scenarios 1 and 4 will have performance consequences and Scenario 5 might have performance, security and reliability implications.

1. Change the heads-up display to track several targets simultaneously without affecting latency.
2. Add a new message type to the system's repertoire in less that a person-week of work.
3. Add a collaborative planning capability where two planners at different sites collaborate on a plan in less than a person-year of work.
4. The maximum number of tracks to be handled by the system doubles and the maximum latency of track data to the screen is kept to 200 ms.
5. Migrate to a new operating system, or a new release of the existing operating system in less than a person-year of work.
6. Add a new data server to reduce latency in use case Scenario 5 to 2.5 seconds within one person-week.
7. Double the size of existing database tables while maintaining 1 second average retrieval time.

5.1.3 Exploratory Scenarios

Exploratory scenarios push the envelope and stress the system. The goal of these scenarios is to expose the limits or boundary conditions of the current design, exposing possibly implicit assumptions. Systems are never conceived to handle these kinds of modifications, but at some point in the future these might be realistic requirements for change. And so the stakeholders might like to understand the ramifications of such changes. For example,

1. Add a new 3-D map feature, and a virtual reality interface for viewing the maps in less than five person-months of effort.
2. Change the underlying Unix platform to a Macintosh.
3. Re-use the 25-year-old software on a new generation of the aircraft.
4. The time budget for displaying changed track data is reduced by a factor of 10.
5. Improve the system's availability from 98% to 99.999%.
6. Half of the servers go down during normal operation without affecting overall system availability.
7. Tenfold increase in the number of bids processed hourly while keeping worst-case response time below 10 seconds.

5.2 Eliciting and Prioritizing Scenarios

Scenarios are elicited and prioritized in ATAM using different mechanisms at different times with different stakeholder participation. The two mechanisms employed are utility trees and structured brainstorming. Table 1 highlights their common differences.

	Utility Trees	Facilitated Brainstorming
Stakeholders	Architects, project leader	All stakeholders
Typical Group size	2 evaluators; 2-3 project personnel	4-5 evaluators; 5-10 project-related personnel
Primary Goals	Elicit, concretize and prioritize the driving quality attribute requirements. Provide a focus for the remainder of the evaluation.	Foster stakeholder communication to validate quality attribute goals elicited via the utility tree.
Approach	Top-down (general to specific)	Bottom-up (specific to general)

Table 1: Utility Trees vs. Scenario Brainstorming

5.3 Utility Trees

Utility trees provide a top-down mechanism for directly and efficiently translating the business drivers of a system into concrete quality attribute scenarios. For example, in an e-commerce system two of the business drivers might be stated as: “security is central to the success of the system since ensuring the privacy of our customers’ data is of utmost importance”; and “modifiability is central to the success of system since we need to be able to respond quickly to a rapidly evolving and very competitive marketplace.” Before we can assess the architecture, these system goals must be made more specific and more concrete. Moreover, we need to understand the relative importance of these goals versus other quality attribute goals, such as performance, to determine where we should focus our attention during the architecture evaluation. Utility trees help to concretize and prioritize quality goals. An example of a utility tree is shown in Figure 3.

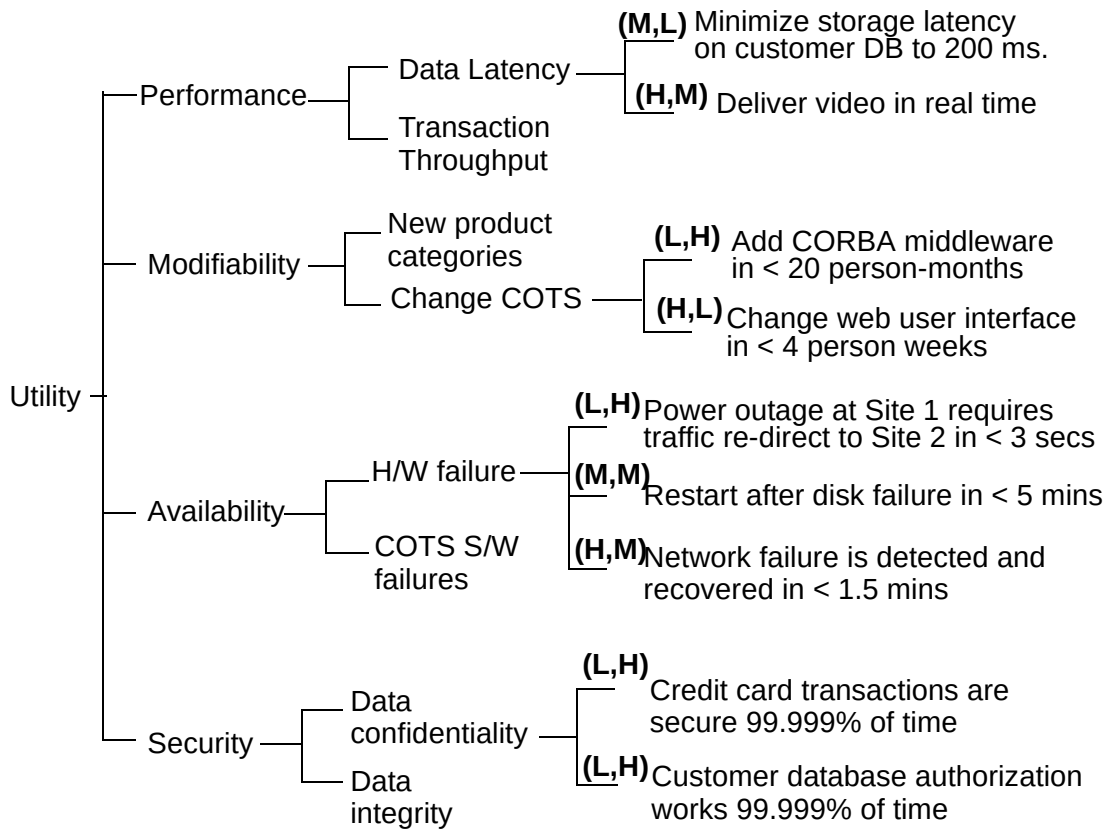


Figure 3: A Sample Utility Tree

The utility tree shown in Figure 3 contains *utility* as the root node. This is an expression of the overall "goodness" of the system. Typically the quality attributes of performance, modifiability, security, and availability are the high-level nodes immediately under utility, although different stakeholder groups may add their own quality attributes or may use different names for the same ideas (for example, some stakeholders prefer to speak of maintainability). Under each of these quality factors are specific sub-factors. For example, performance is broken down into "data latency" and "transaction throughput". This is a step toward refining the attribute goals to be concrete enough for prioritization. Notice how these sub-factors are related to the attribute characterizations. Latency and throughput are two of the types of response measures noted in the attribute characterization. Data latency is then further refined into "Minimize storage latency on customer database" and "Deliver video in real-time." Throughput might be refined into "Maximize average throughput to the authentication server." Further work on these scenarios would, in fact, make these architectural response goals even more specific, e.g., "Storage latency on customer database no more than 10 ms. on average."

These leaf nodes are now specific enough to be prioritized relative to each other. This prioritization may be on a 0-1 scale, or using relative rankings such as High, Medium, and Low; we typically prefer the latter approach as we find that the stakeholders cannot reliably and repeatably make finer distinctions than High, Medium, and Low. The prioritization of the utility tree is done along two dimensions: by the importance of each node to the success of the system and the degree of perceived risk posed by the achievement of this node (i.e., how easy the architecture teams feel this level of performance, modifiability, or other attribute will be to achieve). For example, "Minimize storage latency on customer database" has priorities of (M,L), meaning that it is of medium importance to the success of the system and low risk to achieve, while "Deliver video in real time" has priorities of (H,M), meaning that it is highly important to the success of the system and the achievement of this scenario is perceived to be of medium risk.

Refining the utility tree often leads to interesting and unexpected results. For example, in the example given in Figure 3, security and modifiability were initially designated by the stakeholders as the key attributes driving quality requirements. The subsequent elicitation and refinement of the quality attribute requirements via the utility tree resulted in determining that performance and availability were also important. Simply stated, creating the utility tree guides the key stakeholders in considering, explicitly stating, and prioritizing all of the current and future driving forces on the architecture.

The output of utility tree generation provides a prioritized list of scenarios that serves as a plan for the remainder of the ATAM. It tells the ATAM team where to spend its (relatively limited) time, and in particular where to probe for architectural approaches and risks. The utility tree guides the evaluators to look at the architectural approaches involved with satisfying the high priority scenarios at the leaves of the utility tree. Additionally, the utility tree serves to concretize the quality attribute requirements, forcing the evaluation team and the customer to define their "XYZ-ility" requirements very precisely. Statements, commonly found in requirements documents, such as "The architecture shall be modifiable and robust" are untenable here, because they have no operational meaning: they are not refutable.

5.4 Scenario Brainstorming

While utility tree generation is primarily used to understand how the architect perceived and handled quality attribute architectural drivers, the purpose of scenario brainstorming is to take the "pulse" of the larger stakeholder community. Scenario brainstorming works well in larger groups, creating an atmosphere in which the ideas and thoughts of one person stimulate others to think. The process fosters communication, creativity, and serves to express the collective mind of the participants. The prioritized list of brainstormed scenarios is compared with those generated via the utility tree exercise. If they agree, great. If additional driving scenarios are discovered, this is also an important outcome.

6 Attribute-Based Architectural Styles

An architectural style (as defined by Shaw and Garlan [Shaw 96] and elaborated on by others [Buschmann 96]) includes a description of component types and their topology, a description of the pattern of data and control interaction among the components, and an informal description of the benefits and drawbacks of using that style. Architectural styles are important since they differentiate classes of designs by offering experiential evidence of how each class has been used along with qualitative reasoning to explain why each class has certain properties.“Use pipes and filters when reuse is desired and performance is not a top priority” is an example of the type of description that is a portion of the definition of the pipe and filter style.

A style can be thought of as a set of constraints on an architecture—constraints on component types and their interactions—and these constraints define the set or family of architectures that satisfy them. By locating architectural styles in an architecture, we see what strategies the architect has used to respond to the system’s driving performance goals, modifiability goals, availability goals, and so forth. The ATAM uses a particular specialization of this called attribute-based architectural styles, or ABASs [Klein 99a, 99b].

An ABAS is an architectural style in which the constraints focus on component types and patterns of interaction that are particularly relevant to quality attributes such as performance, modifiability, security, or availability. ABASs aid architecture evaluation by focusing the stakeholders’ attention on the patterns that dominate the architecture. This focus is accomplished by suggesting attribute-specific questions associated with the style. The attribute-specific questions are, in turn, inspired by the attribute characterizations that we discussed above. For example, a performance ABAS highlights architectural decisions that are relevant to performance—how processes are allocated to processors, where they share resources, how their priorities are assigned, and so forth.

A particular performance ABAS will highlight a subset of those questions. For example, if an architecture uses a collection of interacting processes to achieve a processing goal within a specified time period, this would be recognized as a performance ABAS. The questions associated with this performance ABAS would probe important architectural decisions such as the priority of the processes, estimates of their execution time, places where they synchronize, queuing disciplines, etc.; information that is relevant to understanding the performance of this style. The answers to these questions then feed into an explicit analytic model such as RMA (rate monotonic analysis) [Klein 93] or queuing analysis for performance. By associating ana-

lytic models with architectural styles in ABASs, we can probe the strengths and weaknesses of the architecture with respect to each quality attribute.

Examples of ABASs include

- Modifiability Layering ABAS: analyzes the modifiability of the layers architectural style by examining potential effects of various modification scenarios.
- Performance Concurrent pipelines ABAS: applies rate monotonic analysis to multiples pipelines on a single processor, each of which have real-time deadlines.
- Reliability Tri-modular redundancy ABAS: applies Markov modeling to a classical style of redundancy used to enhance system reliability.

ABASs have been described elsewhere (e.g., Klein) and will not be discussed in depth here [Klein 99a, 99b]. The point in introducing them is that ABASs have proven to be useful in architecture evaluations even if none of the catalogued styles have been explicitly identified in the architecture. All architectures have embedded patterns, since no large system is a random collection of components and connection. But the patterns might not always be apparent; they might not always be documented; they might not be "pure." And the patterns might not always be desirable. But nevertheless there are patterns. Calling out these patterns during an architecture evaluation and explicitly reasoning about their behavior by asking attribute-specific questions and applying attribute-specific reasoning models is central to the evaluation. This is how we understand the "forest" of an architecture, without getting mired in understanding individual "trees."

7 Outputs of the ATAM

While the clarification of requirements and better architecture specifications are a positive consequence of performing an architecture evaluation, the central goal of an architecture evaluation is to uncover key architectural decisions. In particular, we want to find key decisions that pose risks for meeting quality requirements and key decisions that have not yet been made. Finally, the ATAM helps an organization to develop a set of analyses, rationale, and guidelines for ongoing decision making about the architecture. This framework should live and evolve as the system evolves.

7.1 Risks and Non-Risks

Risks are potentially problematic architectural decisions. Non-risks are good decisions that rely on assumptions that are frequently *implicit* in the architecture. Both should be understood and explicitly recorded.[1]

The documenting of risks and non-risks consist of

- an architectural decision (or a decision that has not been made)
- a specific quality attribute response that is being addressed by that decision along with the consequences of the predicted level of the response
- a rationale for the positive or negative effect that decision has on meeting the quality attribute requirement

An example of a risk is

> The rules for writing business logic modules in the second tier of your three-tier client-server style are not clearly articulated (*a decision that has not been made*). This could result in replication of functionality thereby compromising modifiability of the third tier (*a quality attribute response and its consequences*). Unarticulated rules for writing the business logic can result in unintended and undesired coupling of components (*rationale for the negative effect*).

1. Risks can also emerge from other sources. For example, having a management structure that is misaligned with the architectural structure might present an organizational risk. Insufficient communication between the stakeholder groups and the architect is a common kind of management risk.

An example of a non-risk is

> Assuming message arrival rates of once per second, a processing time of less than 30 ms, and the existence of one higher priority process (*the architectural decisions*), a one-second soft deadline seems reasonable (*the quality attribute response and its consequences*) since the arrival rate is bounded and the preemptive effects of higher priority processes is known and can be accommodated (*the rationale*).

Note that for a non-risk to remain a non-risk the assumptions must not change (or at least if they change, the designation of non-risk will need to be re-justified). For example, if the message arrival rate, the processing time or the number of higher priority processes changes, the designation of non-risk could change.

7.2 Sensitivity and Tradeoff Points

We term key architectural decisions *sensitivity points* and *tradeoff points*. A sensitivity point is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response. For example:

- The level of confidentiality in a virtual private network might be sensitive to the number of bits of encryption.
- The latency for processing an important message might be sensitive to the priority of the lowest priority process involved in handling the message.
- The average number of person days of effort it takes to maintain a system might be sensitive to the degree of encapsulation of its communication protocols and file formats.

Sensitivity points tell a designer or analyst where to focus attention when trying to understand the achievement of a quality goal. They serve as yellow flags: "Use caution when changing this property of the architecture." Particular values of sensitivity points may become risks when realized in an architecture. Consider the examples above. A particular value in the encryption level—say, 32 bit encryption—may present a risk in the architecture. Or having a very low priority process in a pipeline that processes an important message may become a risk in the architecture.

Sensitivity points use the language of the attribute characterizations. So, when performing an ATAM, we use the attribute characterizations as a vehicle for suggesting questions and analyses that guide us to potential sensitivity points. For example, the priority of a specific component (an architectural parameter) might be a sensitivity point if it is a key property for achieving an important latency goal (a response) of the system. A method interface (an archi-

tectural parameter) might be a sensitivity point if the amount of work required to change the interface (a response) is key in achieving a certain class of important system extensions.

A *tradeoff point* is a property that affects more than one attribute and is a sensitivity point for more than one attribute. For example, changing the level of encryption could have a significant impact on both security and performance. Increasing the level of encryption improves the predicted security but requires more processing time. If the processing of a confidential message has a hard real-time latency requirement then the level of encryption could be a tradeoff point. Tradeoff points are the most critical decisions that one can make in an architecture, which is why we focus on them so carefully.

Finally, it is not uncommon for an architect to answer an elicitation question by saying: "we haven't made that decision yet". In this case you cannot point to a component or property in the architecture and call it out as a sensitivity point because the component or property might not exist yet. However, it is important to flag key decisions that have been made as well as key decisions that have not yet been made.

7.3 A Structure for Reasoning

We do not do extensive formal modeling in an ATAM; this is not the intention of the method. The ATAM's primary goal is in determining where sensitivity points and tradeoffs exist. These areas of the architecture can then be made the objects of further in-depth analysis. The identification of sensitivity points and tradeoff points is frequently the result of *implicit* or *qualitative* analysis. That is, either from prior experience or from a basic understanding of the quality attribute in question, the evaluator is able to conclude that a quality goal is sensitive to certain properties of the architecture. A goal of any architecture evaluation is to make this reasoning explicit and to record it for posterity.

The reasoning is not always highly formal and mathematical, but it must be predictive and repeatable. The reasoning might manifest itself as a discussion that follows from the exploration the architectural approaches that address a scenario; it may be a qualitative model of attribute-specific behavior of the architecture; or it may be a quantitative model that represents how to calculate a value of a particular quality attribute. ABASs and quality attribute characterizations provide the technical foundations for creating these reasoning models.

7.4 Producing ATAM's Outputs

The power of the ATAM derives in part from the foundations provided by the technical concepts discussed here. These concepts work together synergistically.

The stimulus/response branches of the quality attribute characterizations provide a vocabulary for describing the problem the architecture is intended to solve. In Step 2 of the ATAM the customer presents the business context for the problem being solved. The utility tree is used to translate the business context first into quality attribute drivers and then into concrete scenarios that represent each business driver. Each scenario is described in terms of a stimulus and the desired response. These scenarios are prioritized in terms of how important they are to the overall mission of the system and the perceived risk in realizing them in the system. The highest priority scenarios will be used in the analysis of the architecture.

In Step 3 the system architect presents the architecture to the evaluation team. The architect is encouraged to present the architecture in terms of the architectural approaches that are used to realize the most important quality attribute goals. The evaluation team also searches for architectural styles and approaches as the architecture is presented. The approaches that address the highest priority scenarios will be the subject of analysis during the remainder of the evaluation. Attribute-specific models (such as queuing models, modifiability models, and reliability models) as they apply to specific architectural styles, are codified in ABASs. These models provide attribute-specific questions that the evaluators employ to elicit the approaches used by the architect to achieve the quality attribute requirements.

The three components used to find risks, sensitivity points, and tradeoff points are shown in Figure 4: high-priority scenarios, attribute-specific questions (as guided and framed by the attribute characterizations), and architectural approaches. During an evaluation the architect traces the scenarios through the subset of the architecture represented by the approaches. To do this the architect must identify the components and connectors involved in realizing the scenario. As the architect identifies the relevant components the evaluators ask attribute-specific questions. Some of the answers to these questions lead to the identification of sensitivity points, some of which will turn out to be risks and/or tradeoff points.

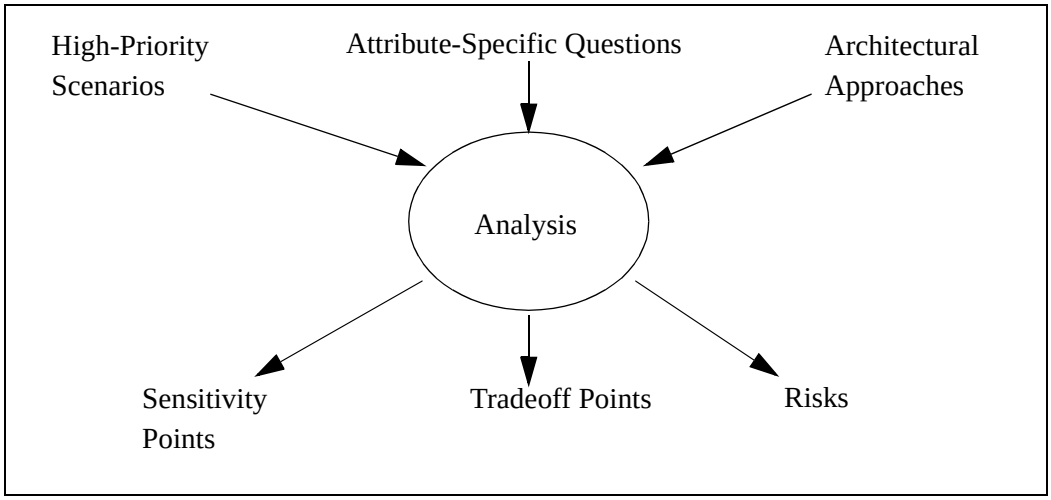


Figure 4: Concept Interactions

8 The Steps of the ATAM

The ATAM process consists of nine steps, as was briefly presented in Section 3. Sometimes there must be dynamic modifications to the order of steps to accommodate the availability of personnel or architectural information. Although the steps are numbered, suggesting linearity, this is not a strict waterfall process. There will be times when an analyst will return briefly to an earlier step, or will jump forward to a later step, or will iterate among steps, as the need dictates. The importance of the steps is to clearly delineate the activities involved in ATAM along with the outputs of these activities.

The amount of time it takes to carry out an ATAM varies depending on the size of the system and the maturity of the architecture and the state of its description. We have conducted ATAMs by executing the steps in three consecutive days. However we have found that the most effective ATAMs have been those in which the ATAM team and the customer establish a common understanding of the architecture during an initial phase of the evaluation and then later assemble a larger group of stakeholders for a more formal evaluation meeting.

8.1 Step 1 - Present the ATAM

In this step the evaluation team lead presents the ATAM to the assembled stakeholders. This time is used to explain the process that everyone will be following, allows time to answer questions, and sets the context and expectations for the remainder of the activities. It is important that everyone knows what information will be collected, how it will be scrutinized, and to whom it will be reported. In particular, the presentation will describe

- the ATAM steps in brief
- the techniques that will be used for elicitation and analysis: utility tree generation, architectural approach-based elicitation/analysis, and scenario brainstorming/mapping
- the outputs from the evaluation: the scenarios elicited and prioritized, the questions used to understand/evaluate the architecture, a “utility” tree, describing and prioritizing the driving architectural requirements, the set of identified architectural approaches and styles, the set of risks and non-risks discovered, the set of sensitivity points and tradeoffs discovered

8.2 Step 2 - Present Business Drivers

The system to be evaluated needs to be understood by all participants in the evaluation. In this step the project manager presents a system overview from a business perspective. A sample outline for such a presentation is given in Figure 5. The system itself must be presented, initially at a high level of abstraction, typically describing

- its most important functional requirements
- its technical, managerial, economic, or political constraints
- its business goals and context
- its major stakeholders
- the architectural drivers (major quality attribute goals that shape the architecture)

8.2.1 Business Case/Architecture Presentation

To ensure the quality, consistency, and volume of information presented in Steps 2 and 3 (the business driver and architecture presentations) we typically provide documentation templates to the presenters. An example of the documentation templates that we provide for the business case is shown in Figure 5.

Business Context/Drivers Presentation (~ 12 slides; 45 minutes)

- Description of the business environment, history, market differentiators, driving requirements, stakeholders, current need and how the proposed system will meet those needs/ requirements (3-4 slides)

- Description of business constraints (e.g., time to market, customer demands, standards, cost, etc.) (1-3 slides)

- Description of the technical constraints (e.g., COTS, interoperation with other systems, required hardware or software platform, reuse of legacy code, etc.) (1-3 slides)

- Quality attributes desired (e.g., performance, availability, security, modifiability, interoperability, integrability) and what business needs these are derived from (2-3 slides)

- Glossary (1 slide)

Figure 5: Example of Template for the Business Case Presentation

8.3 Step 3 - Present Architecture

The architecture will be presented by the lead architect (or architecture team) at an appropriate level of detail. What is an appropriate level? This depends on several factors: how much information has been decided upon and documented; how much time is available; how much risk the system faces. This is an important step as the amount of architectural information available and documented will directly affect the analysis that is possible and the quality of this analysis. Frequently the evaluation team will need to specify additional architectural information that is required to be collected and documented before a more substantial analysis is possible.

In this presentation the architecture should cover

- technical constraints such as an OS, hardware, or middleware prescribed for use
- other systems with which the system must interact
- architectural approaches used to meet quality attribute requirements

At this time the evaluation team begins its initial probing of architectural approaches.

An example of the template for the architecture presentation is shown in Figure 6. Providing these templates to the appropriate stakeholders (in this case, the architect) well in advance of the ATAM helps to reduce the variability of the information that they present and also helps to ensure that we stay on schedule.

Architecture Presentation (~20 slides; 60 minutes)

- Driving architectural requirements (e.g., performance, availability, security, modifiability, interoperability, integrability), the measurable quantities you associate with these requirements, and any existing standards/models/approaches for meeting these (2-3 slides)

- High Level Architectural Views (4-8 slides)

+ functional: functions, key system abstractions, domain elements along with their dependencies, data flow

+ module/layer/subsystem: the subsystems, layers, modules that describe the system's decomposition of functionality, along with the objects, procedures, functions that populate these and the relations among them (e.g., procedure call, method invocation, callback, containment).

+ process/thread: processes, threads along with the synchronization, data flow, and events that connect them

+ hardware: CPUs, storage, external devices/sensors along with the networks and communication devices that connect them

- architectural approaches or styles employed, including what quality attributes they address and a description of how the styles address those attributes (3-6 slides)

- Use of COTS and how this is chosen/integrated (1-2 slides)

- Trace of 1-3 of the most important use case scenarios. If possible, include the run-time resources consumed for each scenario (1-3 slides)

- Trace of 1-3 of the most important change scenarios. If possible, describe the change impact (estimated size/difficulty of the change) in terms of the changed components, connectors, or interfaces. (1-3 slides)

- Architectural issues/risks with respect to meeting the driving architectural requirements (2-3 slides)

- Glossary (1 slide)

Figure 6: Example of Template for the Architecture Presentation

8.4 Step 4 - Identify Architectural Approaches

The ATAM focuses on analyzing an architecture by understanding its architectural approaches. In this step they are identified by the architect, and captured by the analysis team, but are not analyzed.

We concentrate on identifying architectural approaches and architectural styles [1] because these represent the architecture's means of addressing the highest priority quality attributes; that is, the means of ensuring that the critical requirements are met in a predictable way [Buschmann 96, Shaw 96]. These architectural approaches define the important structures of the system and describe the ways in which the system can grow, respond to changes, withstand attacks, integrate with other systems, and so forth.

8.5 Step 5 - Generate Quality Attribute Utility Tree

In this step the evaluation team works with the architecture team, manager, and customer representatives to identify, prioritize, and refine the system's most important quality attribute goals, as described in Section 5.3. This is a crucial step in that it guides the remainder of the analysis. Analysis, even at the level of software architecture, is not inherently bound in scope. So we need a means of focussing the attention of all the stakeholders on the aspects of the architecture that are most critical to the system's success. We do this by building a utility tree.

The output of the utility tree generation process is a prioritization of specific quality attribute requirements, realized as scenarios. This prioritized list provides a guide for the remainder of the ATAM. It tells the ATAM team where to spend its limited time, and in particular where to probe for architectural approaches and their consequent risks, sensitivity points, and tradeoffs. Additionally, the utility tree serves to concretize the quality attribute requirements, forcing the evaluation team and the customer to define their "ility" requirements precisely.

8.6 Step 6 - Analyze Architectural Approaches

Once the scope of the evaluation has been set by the utility tree elicitation, the evaluation team can then probe for the architectural approaches that realize the important quality attributes. This is done with an eye to documenting these architectural decisions and identifying their risks, sensitivity points, and tradeoffs.

---

1. We look for approaches and styles because not all architects are familiar with the language of architectural styles, and so may not be able to enumerate a set of styles used in the architecture. But every architect makes architectural decisions, and the set of these we call "approaches." These can certainly be elicited from any conscientious architect.

What are we eliciting in this step? We are eliciting sufficient information about each architectural approach to conduct a rudimentary analysis about the attribute for which the approach is relevant. What are we looking for in this rudimentary analysis? We want to be convinced that the instantiation of the approach in the architecture being evaluated holds significant promise for meeting the attribute-specific requirements for which it is intended.

The major outputs of this phase are a list of architectural approaches or styles, the questions associated with them, and the architect's response to these questions. Frequently a list of risks, sensitivity points, and tradeoffs are generated. Each of these is associated with the achievement of one or more utility tree sub-factors (see Section 5.3), with respect to the quality-attribute questions that probed the risk. In effect, the utility sub-factor tells us where to probe the architecture (because this is a highly important factor for the success of the system), the architect (hopefully) responds with the architectural approach that answers this need, and we use the quality attribute-specific questions to probe the approach more deeply. The questions help us to

- understand the approach
- look for well-known weaknesses with the approach
- look for the approach's sensitivity points
- find interactions and tradeoffs with other approaches

In the end, each of these may provide the basic material for the description of a risk and this is recorded in an ever-growing list of risks.

The first action in this step is to associate the highest priority quality attribute requirements (as identified in the utility tree of Step 5) with the architectural approaches (from Step 4) used to realize them. For each highly ranked scenario generated by the utility-tree-creation step, the architect should identify the components, connectors, configuration, and constraints involved.

The evaluation team and the architecture team address each architectural approach presented by asking a set of approach-specific and quality-attribute-specific questions. These questions might come from documented experience with styles (as found in ABASs and their associated quality attribute characterizations), from books on software architecture (e.g., [Bass 98, Buchmann 96, Shaw 96, Soni 00]), or from the prior experiences of the assembled stakeholders. In practice we mine all three areas for questions.

These questions are not an end unto themselves. Each is a starting point for discussion, and for the determination of a potential risk, non-risk, sensitivity point, or tradeoff point. These, in turn, may catalyze a deeper analysis, depending on how the architect responds. For example, if the architect cannot characterize client loading and cannot say how priorities are allocated to processes and how processes are allocated to hardware, then there is little point doing a sophis-

ticated queuing or RMA performance analysis [Klein 93]. If such questions can be answered, then we perform at least a rudimentary, or back-of-the-envelope, analysis to determine if these architectural decisions are problematic or not vis-a-vis the quality attribute requirements they are meant to address. The level of analysis is not meant to be comprehensive and detailed but rather commensurate with the level of detail of the architectural specification. In other words, this will require some engineering judgement. However, the key thought to keep in mind is the need to establish some link between the architectural decisions that have been made and the quality attribute requirements that need to be satisfied. For example, it might be critical to understand how the priority of a server process responsible for managing access to shared data will impact the latency of clients who will access the shared data.

The template for capturing an architectural approach is shown in Figure 7.

Scenario: <a scenario from the utility tree of from scenario brainstorming>

Attribute: <performance, security, availability, etc.>

Environment: <relevant assumptions about the environment in which the system resides >

Stimulus: <a precise statement of the quality attribute stimulus (e.g., failure, threat, modification, …) embodied by the scenario>

Response: <a precise statement of the quality attribute response (e.g., response time, measure of difficulty of modification)>

Architectural Decisions	**Risk**	**Sensitivity**	**Tradeoff**
<list of architectural decisions affecting quality attribute response>	<risk #>	<sens. point #>	<tradeoff #>

Reasoning:

<qualitative and/or quantitative rationale for why the list of architectural decisions contribute to meeting the quality attribute response requirement>

Architecture diagram:

<diagram or diagrams of architectural views annotated with architectural information to support the above reasoning. These may be accompanied by explanatory text.>

Figure 7: Architectural Approach Documentation Template

So, for example, we might elicit the following information from an architect, in response to a utility tree scenario that required high availability from a system, as shown in Figure 8.

Scenario: S12 (Detect and recover from HW failure of main switch.)

Attribute: Availability

Environment: normal operations

Stimulus: CPU failure

Response: 0.999999 availability of switch

Architectural decisions	**Risk**	**Sensitivity**	**Tradeoff**
Backup CPU(s)	R8	S2	
No backup Data Channel	R9	S3	T3
Watchdog		S4	
Heartbeat		S5	
Failover routing		S6	

Reasoning:

- ensures no common mode failure by using different hardware and operating system (see Risk 8)
- worst-case rollover is accomplished in 4 seconds as computing state takes …
- guaranteed to detect failure with 2 seconds based on rates of heartbeat and watchdog …
- watchdog is simple and proven reliable
- availability requirement might be at risk due to lack of backup data channel … (see Risk 9)

Architecture diagram:

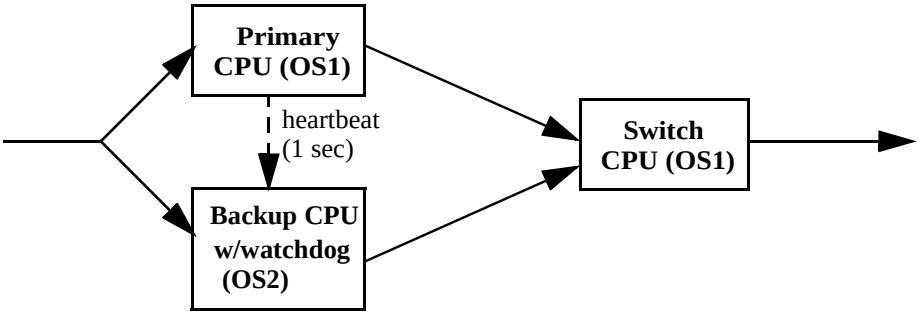


Figure 8: Example of Architectural Approach Description

As shown in Figure 8, based upon the results of this step we can identify and record a set of sensitivity points, tradeoff points, risks, and non-risks. (Recall that a *sensitivity point* is a property of one or more components and/or component relationships that is critical for achieving a

particular quality. Some of these property values will represent risks for the success of the architecture.)

All sensitivity points and tradeoff points are candidate risks. By the end of the ATAM, all sensitivity points and tradeoff points should be categorized as either a risk or a non-risk. The risks/non-risks, sensitivity points, and tradeoffs are gathered together in three separate lists. The numbers R8, T3, or S4 from Figure 8 are simply pointers into these lists.

At the end of this step, the team should now have a broad-brush picture of the most important aspects of the entire architecture, the rationale for design decisions that have been made, and a list of risks, sensitivity points, and tradeoff points. To recapitulate, let us point out where risks, sensitivity points and tradeoff points emerge. Recall that we ask attribute-specific questions of each architectural approach elicited; these questions probe the architectural approaches. The answers to these questions are potential sensitivity points. For some value of the sensitivity points we will have architectural risks and for some value we will have non-risks.

For example, the processor speed might control the number of transactions a transaction server can process in a second. Thus the processor speed is a sensitivity point with respect to the response of transactions per second. Some processor speeds will result in unacceptable values of this response—these are risks. Finally, when it turns out that an architectural decision is a sensitivity point for more than one attribute it is designated as a tradeoff point. At this point we can now test the utility and our understanding of the architectural representations that we have generated. This is the purpose of the next two steps.

8.7 Step 7 - Brainstorm and Prioritize Scenarios

Scenarios are the motor that drives the testing phase of the ATAM. Generating a set of scenarios has proven to be a great facilitator of discussion and brainstorming, when greater numbers of stakeholders are gathered to participate in the ATAM. Scenarios are examples of architectural *stimuli* used to both

- represent stakeholders' interests
- understand quality attribute requirements

The stakeholders now undertake two related activities: brainstorming *use case scenarios* (representing the ways in which the stakeholders expect the system to be used) and *change scenarios* (representing the ways in which the stakeholders expect the system to change in the future). Use case scenarios are a kind of scenario where the stakeholder is an end user, using the system to execute some function. Change scenarios represent changes to the system and are sub-divided into two categories: *growth scenarios* and *exploratory scenarios*.

Growth scenarios represent ways in which the architecture is expected to accommodate growth and change in the moderate near term: expected modifications, changes in performance or availability, porting to other platforms, integration with other software, and so forth. Exploratory scenarios, on the other hand, represent extreme forms of growth: ways in which the architecture might be stressed by changes: dramatic new performance or availability requirements (order of magnitude changes, for example), major changes in the infrastructure or mission of the system, and so forth. Growth scenarios are a way of showing the strengths and weaknesses of the architecture with respect to anticipated forces on the system. Exploratory scenarios are an attempt to find sensitivity points and tradeoff points. The identification of these points help us assess the limits of the system with respect to the models of quality attributes that we build.

Once the scenarios have been collected, they must be prioritized. We typically do this via a voting procedure where each stakeholder is allocated a number of votes equal to 30% of the number of scenarios, rounded up. So, for instance, if there were eighteen scenarios collected, each stakeholder would be given six votes. These votes can be allocated in any way that the stakeholder sees fit: all six votes allocated to one scenario, two votes to each of three scenarios, one vote to each of six scenarios, etc. In addition, at this point any stakeholder can suggest merging multiple scenarios if they are felt to represent the same stimulus/response behavior.

The prioritization and voting can be an open or a secret balloting procedure. Once the votes have been made, they are tallied and the scenarios are prioritized. A cutoff is typically made that separates the high-priority scenarios from the lower ones, and only the high-priority ones are considered in future evaluation steps. For example, a team might only consider the top five scenarios.

Figure 9 shows an example set of scenarios from a recent evaluation of a dispatching system (only the top five scenarios of the more than thirty collected are shown here), along with their votes.

4. Dynamically replan a dispatched mission within 10 minutes. [28]

27. Split the management of a set of vehicles across multiple control sites. [26]

10. Change vendor analysis tools after mission has commenced without restarting system. [23]

12. Retarget a collection of diverse vehicles to handle an emergency situation in less than 10 seconds after commands are issued. [13]

14. Change the data distribution mechanism from CORBA to a new emerging standard with less than six person-months' effort. [12]

Figure 9: Examples of Scenarios with Rankings

At this point we pause and compare the result of scenario prioritization with the results of the utility tree exercise and look for agreement or disagreement. Any difference between the high priority scenarios and the high priority nodes of the utility tree needs to be reconciled or at least explained. Reconciliation might entail clarifying the meaning of a scenario or changing one of the prioritizations. Explaining might entail understanding that the criteria used for prioritizing the utility tree were different than the criteria used in prioritizing scenarios. In any event, this is a good opportunity to make sure that all of the needs of the stakeholders are well understood and are not in irreconcilable conflict.

For example, consider Figure 10, where the highly ranked scenarios are shown along with an indication of the quality attribute or attributes that each scenario affects most heavily.

Scenario	#Votes	Quality Attributes
4	28	Performance
27	26	Performance, Modifiability, Availability
10	23	Integrability
12	13	Performance
14	12	Modifiability

Figure 10: Highly Ranked Scenarios with Quality Attribute Annotations

The utility tree generation and the scenario brainstorming activities reflect the quality attribute goals, but via different elicitation means and typically addressing different groups of stakeholders. The architects and key developers most often create the initial utility tree. The widest possible group of stakeholders is involved in the scenario generation and prioritization. Comparing the highly ranked outputs of both activities often reveals disconnects between what the architects believe to be important system qualities and what the stakeholders as a whole believe to be important. This, by itself, can reveal serious risks in the architecture, by highlighting entire areas of concern to which the architects have not attended. After this step the utility tree is the authoritative repository of the detailed high-priority quality attribute requirements from *all* sources.

8.8 Step 8 - Analyze Architectural Approaches

After the scenarios have been collected and so analyzed, the architect then begins the process of mapping the highest ranked scenarios onto whatever architectural descriptions have been presented. Ideally this activity will be dominated by the architect's mapping of scenarios onto previously discussed architectural approaches. In fact the whole point of the hiatus between the two phases is to *ensure* that this is the case. If this is not the case then either the architect has no approach- or style-based (and hence no architecture-wide) solution for the stimulus that the scenario represents, or the approach exists but was not revealed by any activities up until this point.

In this step we reiterate Step 6, mapping the highest ranked newly generated scenarios onto the architectural artifacts thus far uncovered. Assuming Step 7 didn't produce any high-priority scenarios that were not already covered by previous analysis, Step 8 is a testing activity: We hope and expect to be uncovering little new information. This is a testing activity: at this point we hope and expect to be uncovering little new information.

If we do uncover new information then this was a failing of our utility tree exercise and the architectural approaches that it led us to investigate. At this point we would need to go back to Step 4 and work through it, as well as Steps 5 and 6 until no new information is uncovered.

8.9 Step 9 - Present Results

Finally, the collected information from the ATAM needs to be summarized and presented back to the stakeholders. This presentation typically takes the form of a verbal report accompanied by slides but might, in addition, be accompanied by a more complete written report delivered subsequent to the ATAM. In this presentation we recapitulate the steps of the ATAM and all the information collected in the steps of the method including: the business context, driving requirements, constraints, and the architecture. Most important, however, is the set of ATAM outputs:

- the architectural approaches/styles documented
- the set of scenarios and their prioritization
- the set of attribute-based questions
- the utility tree
- the risks discovered
- the non-risks documented
- the sensitivity points and tradeoff points found

Each of these findings will be described and in some cases we might offer some mitigation strategies. Because we are systematically working through and trying to understand the architectural approaches it is inevitable that, at times, we make some recommendations on how the architecture might have been designed or analyzed differently. These mitigation strategies may be process related (e.g., a database administrator stakeholder should be consulted before completing the design of the system administration user interface), they may be managerial (e.g., three sub-groups within the development effort are pursuing highly similar goals and these should be merged), or they may be technical (e.g., given the estimated distribution of customer input requests, additional server threads need to be allocated to ensure that worst-case latency does not exceed five seconds). However, offering mitigation strategies is *not* an integral part of

the ATAM. The ATAM is about locating architectural risks. Addressing them may be done in any number of ways.

9 The Two Phases of ATAM

Up to this point in this report we have enumerated and described the steps of the ATAM. In this section we will describe how the steps of the ATAM are carried out over time. You will see that they are typically carried out in two phases. The first phase is architecture-centric and concentrates on eliciting architectural information and analyzing it. The second phase is stakeholder-centric and concentrates on eliciting stakeholder points of view and verifying the results of the first phase.

The reason for having two phases is that in Phase 1 we are gathering information: a subset of the evaluation team (typically one to three people), primarily interacting with the architect and one or two other key stakeholders (such as a project manager or customer/marketing representative), walks through all of the steps of the ATAM, gathering information. In almost every case this information is incomplete or inadequate to support the analysis. At this point the evaluation team interacts with the architect to elicit the necessary information. This interaction typically takes several weeks. When we feel that enough information has been collected and documented to support the goals of the evaluation, we proceed on to Phase 2.

9.1 Phase 1 Activities

The ATAM team (or more commonly a subset of the ATAM team) meets with a subset of the team being evaluated, perhaps for the first time. This meeting has two concerns: organization of the rest of the analysis activities, and information collection. Organizationally, the manager of the team being evaluated needs to make sure that the right people attend the subsequent meetings, that people are prepared, and that they come with the right attitude (i.e., a spirit of non-adversarial teamwork).

The first day is typically run as a miniature version of the entire ATAM process, concentrating on Steps 1-6. The information collection aspect of the first meeting is meant to ensure that the architecture is truly evaluable, meaning that it is represented in sufficient detail. Also, some initial “seed” scenario collection and analysis may be done on Day 1, as a way of understanding the architecture, understanding what information needs to be collected and represented, and understanding what it means to generate scenarios.

So, for example, on Day 1 the architect might present a portion of the architecture, identify some of the architectural styles or approaches, create a preliminary utility tree, and walk through a selected set of the scenarios, showing how each affects the architecture (e.g., for

modifiability) and how the architecture responds to it (e.g., for quality attributes such as performance, security and availability). Other stakeholders who are present, such as key developers, the customer, or the project manager, can contribute to describing the business context, building the utility tree, and the scenario generation processes.

Phase 1 is a small meeting, typically between a small subset of both the evaluation team and the customer team. The reason for this is that the evaluation team needs to gather as much information as possible to determine

- if the remainder of the evaluation is feasible and should proceed
- if more architectural documentation is required and, if so, precisely what kinds of documentation and how it should be represented
- what stakeholders should be present for Phase 2

At the end of this day, we will ideally have a good picture of the state and context of the project, its driving architectural requirements, and the state of its architectural documentation.

Depending upon how much is accomplished during the first phase, there is a hiatus between the first meeting and the start of the second meeting in which ongoing discovery and analysis are performed by the architecture team, in collaboration with the evaluation team. As we described earlier we do not foresee the building of detailed analytic models during this phase, but rather the building of rudimentary models that will give the evaluators and the architect sufficient insight into the architecture to make the Phase 2 meeting more productive. Also, during this hiatus the final composition of the evaluation team is determined, according to the needs of the evaluation, availability of human resources, and the schedule. For example, if the system being evaluated was safety critical, a safety expert might be recruited, or if it was database-centric then an expert in database design would be made part of the evaluation team.

9.2 Phase 2 Activities

At this point, it is assumed that the architecture has been documented in sufficient detail to support verification of the analysis already performed, and further analysis if needed. The appropriate stakeholders have been gathered and have been given advance reading materials such as a description of the ATAM, the seed scenarios, and system documentation including the architecture, business case, and key requirements. These reading materials aid in ensuring that the stakeholders know what to expect from the ATAM.

Since there will be a broader set of stakeholders attending the next meeting, and since a number of days or weeks may have transpired between the first and second meeting, it is useful to quickly recap the steps of the ATAM, so that all attendees have the same understanding and

expectations of the day's activities. In addition, it is useful to briefly recap each step immediately before it is executed.

9.3 ATAM Steps and Their Associated Stakeholders

Running an ATAM can involve as few as three to five stakeholders[1] or as many as 40 or 50. We have done evaluations at both ends of the spectrum and everywhere in between. As we have suggested, the process need not involve every stakeholder at every step. Depending on the size, criticality, and complexity of the system, the evaluation team might be smaller or larger. If the system has complex quality attribute requirements or is in a complex and unusual domain, specialists may be needed to augment the expertise of the core evaluation team. In addition, the characteristics of the system may determine which customer representatives may want to attend. Depending on whether the system is purely in-house development, shrink-wrapped software, part of a product line, or part of a industry-wide standards-based effort, different customer representatives will want to attend to ensure that their stake in the system's success is well represented. Table 2 below lists the typical categories of attendees for each ATAM step. [2]

Step	Activity	Stakeholder Groups
1	Present the ATAM	Evaluation team/Customer representatives/Architecture team
2	Present business drivers	"
3	Present architecture	"
4	Identify architectural approaches	"
5	Generate quality attribute utility tree	"
6	Analyze architectural approaches	"
7	Brainstorm and prioritize scenarios	All stakeholders
8	Analyze architectural approaches	Evaluation team/Customer representatives/Architecture team
9	Present results	All stakeholders

Table 2: ATAM Steps Associated with Stakeholder Groups

1. "All stakeholders" includes developers, maintainers, operators, reusers, end users, testers, system administrators, etc.

2. The full teams don't need to be represented for Steps 1-3, particularly in Phase 1.

9.4 A Typical ATAM Agenda

While every ATAM is slightly different than every other one, the broad brush activities do not change. In Figure 11 we show an example of a typical ATAM agenda. Each activity in this figure is followed by its step number, where appropriate, in parentheses. While we do not slavishly follow the times here, experience has shown that this represents a reasonable partitioning

of the available time in that we spend proportionately more time on those activities that experience has shown to produce more results (in terms of finding architectural risks).

Day 1

8:30	Introductions/ATAM Presentation (1)
10:00	Customer Presents Business Drivers (2)
10:45	Break
11:00	Customer Presents Architecture (3)
12:00	Identify Architecture Approaches (4)
12:30	Lunch
1:45	Quality Attribute Utility Tree Generation (5)
2:45	Analyze Architecture Approaches (6)
3:45	Break
4:00	Analyze Architecture Approaches (6)
5:00	Adjourn for the Day

Phase 1

Break of several weeks

Day 2

8:30	Introductions/ATAM Presentation (1)
9:15	Customer Presents Business Context/Drivers (2)
10:00	Break
10:15	Customer Presents Architecture (3)
11:15	Identify Architecture Approaches (4)
12:00	Lunch
1:00	Quality Attribute Utility Tree Generation (5)
2:00	Analyze Architecture Approaches (6)
3:30	Break
3:45	Analyze Architecture Approaches (6)
5:00	Adjourn for the Day

Day 3

8:30	Introductions/Recap ATAM
8:45	Analyze Architecture Approaches (6)
9:30	Scenario Brainstorming (7)
10:30	Break
10:45	Scenario Prioritization (7)
11:15	Analyze Architecture Approaches (8)
12:30	Lunch
1:30	Analyze Architecture Approaches (8)
2:45	Prepare Report of Results/Break
3:30	Present Results (9)
4:00	Further Analysis/Assignment of Action Items
5:00	Adjourn

Phase 2

Figure 11: A Sample ATAM Agenda

10 A Sample Evaluation: The BCS

We now present an example of the ATAM as it was realized in an evaluation. Although some of the details have been changed for pedagogical reasons and to protect the identity and intellectual property of the customer and contractor, the architectural risks that we uncovered are not materially affected by these changes.

We will call this system BCS (Battlefield Control System). This system is to be used by army battalions to control the movement, strategy, and operations of troops in real time in the battlefield. This system is currently being built by a contractor, based upon government furnished requirements.

In describing the BCS ATAM, we will not describe every step of the method. This is for two reasons. The first reason is that the steps involve a reasonable amount of repetition (of activities such as scenario elicitation); in describing the method we simply describe a single instance of each activity. The second reason is that no particular instance of the ATAM slavishly follows the steps as stated. Systems to be evaluated are in different stages of development, customers have different needs, and there will be different levels of architectural documentation in different development organizations. Each of these pressures may result in *slight* changes to the schedule in practice.

10.1 Phase 1

During the initial meeting we first presented the ATAM and the customer presented the business case. The customer's business case was dominated by information on the mission and its requirements. For example, the requirements state that the system supports a Commander who commands a set of Soldiers and their equipment, including many different kinds of weapons and sensors. The system needs to interface with numerous other systems (such as command-and-control systems) that feed it commands and intelligence information. These external systems also periodically collect the BMS system's status with respect to its current missions. The business case also presented requirements with respect to a standard set of hardware and software that was to be employed, the need for extreme levels of system robustness, and a number of performance goals.

Next, the contractor presented the architecture and the contractor and customer together described their initial quality attribute requirements and their initial set of scenarios. As a result of this meeting additional architectural documentation was requested. As is often the

case in evaluating architectures, the initial documentation that was produced was far too vague and limited in scope to support any analysis. The documentation consisted of high level data flows and divisions of functionality that had no clear mapping to software constructs. There was no discussion of architectural styles or approaches in the provided documentation.

Owing to the paucity of available documented architectural information, we did not investigate any architectural approaches or map any scenarios onto the architecture. Instead we created a plan to improve the architectural documentation for use in Phase 2: we requested specific additional architectural information to address the gaps in the documentation produced by the contractor. These requests were in the form of questions such as

- What is the structure of the message-handling software (i.e., how is the functionality is broken down in terms of modules, functions, APIs, layers, etc.)?
- What facilities exist in the software architecture (if any) for self-testing and monitoring of software components?
- What facilities exist in the software architecture (if any) for redundancy, liveness monitoring, failover, and how data consistency is maintained (so that one component can take over from another and be sure that it is in a consistent state with the failed component)?
- What is the process and/or task view of the system, including mapping of these processes/tasks to hardware and the communication mechanisms between them?
- What functional dependencies exist among the software components (often called a “uses” view)?
- What data is kept in the database (which was mentioned by one of your stakeholders), how big is it, how much does it change, and who reads/writes it?
- What is the anticipated frequency and volume of data being transmitted among the system components?

Between Phases 1 and 2 of the evaluation the contractor answered many of these questions and produced substantially more complete, more analyzable architectural documentation. This documentation formed the basis for the remainder of the ATAM.

10.2 Phase 2

After recapping the steps of the ATAM to the gathered stakeholders, Phase 2 consisted of building a utility tree (using the Phase 1 utility tree as a starting point), collecting architectural approach information, collecting and prioritizing scenarios, and mapping these onto the architecture.

10.2.1 Architectural Documentation

As mentioned above, the architectural documentation covered several different views of the system: a dynamic view, showing how subsystems communicated; a set of message sequence charts, showing run-time interactions; a system view, showing how software was allocated to hardware; and a source view, showing how components and subsystems were composed of objects. For the purpose of this example, we will just show the highest level hardware structure of the architecture, using the notation from [Bass 98].[1]

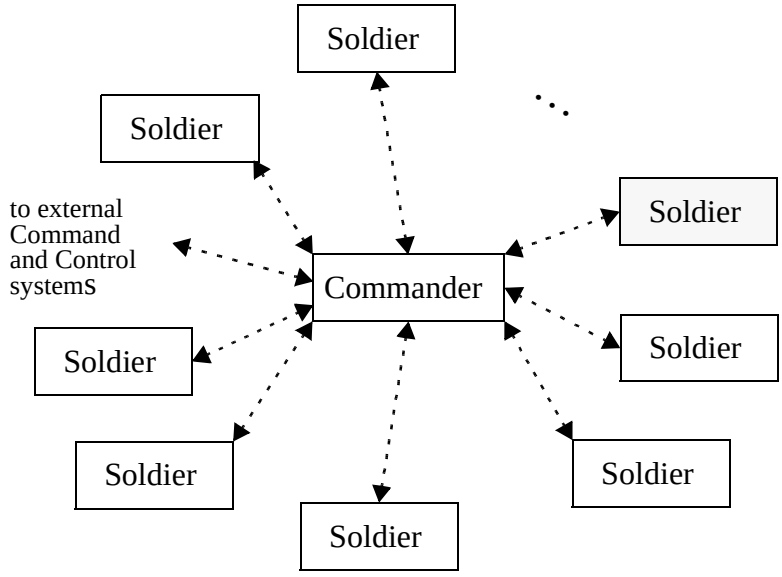


Figure 12: Hardware View of the BCS

This hardware architecture, shown in Figure 12, illustrates that the Commander is central to the system. In fact, the Commander node acts as a server and the Soldier nodes are its clients, making requests of it and updating the server's database with their status. Inter-node communication between the clients and the server is only through encrypted messages sent via a radio modem; neither subsystem controls the other. Note also that the radio modem is a shared communication channel: only one node can be broadcasting at any moment.

10.2.2 Identifying Architectural Approaches

We elicited information on the architectural approaches with respect to modifiability, availability, and performance scenarios. As stated above, the system was loosely organized around the notion of clients and servers. This dictated both the hardware architecture and the process

1. In this notation, rectangles represent processors and dashed lines represent data flow.

architecture and affected the system's performance characteristics, as we shall see. In addition to this style

- for availability, a backup commander approach was described
- for modifiability, standard subsystem organizational patterns were described
- for performance, an independent communicating components approach was described.

Each of these approaches was probed for risks, sensitivities, and tradeoffs via our style-specific questions, as we shall show below.

10.2.3 Eliciting the Utility Tree

The stakeholders in this ATAM were most interested in modifiability and performance. Upon probing, however, they admitted that availability was also of great importance to them. Based upon their stated concerns and our elicitation, a utility tree was created. A portion of this is shown in Figure 13. As part of our elicitation process we ensured that each of the scenarios in the utility tree had a specific stimulus and response associated with it.

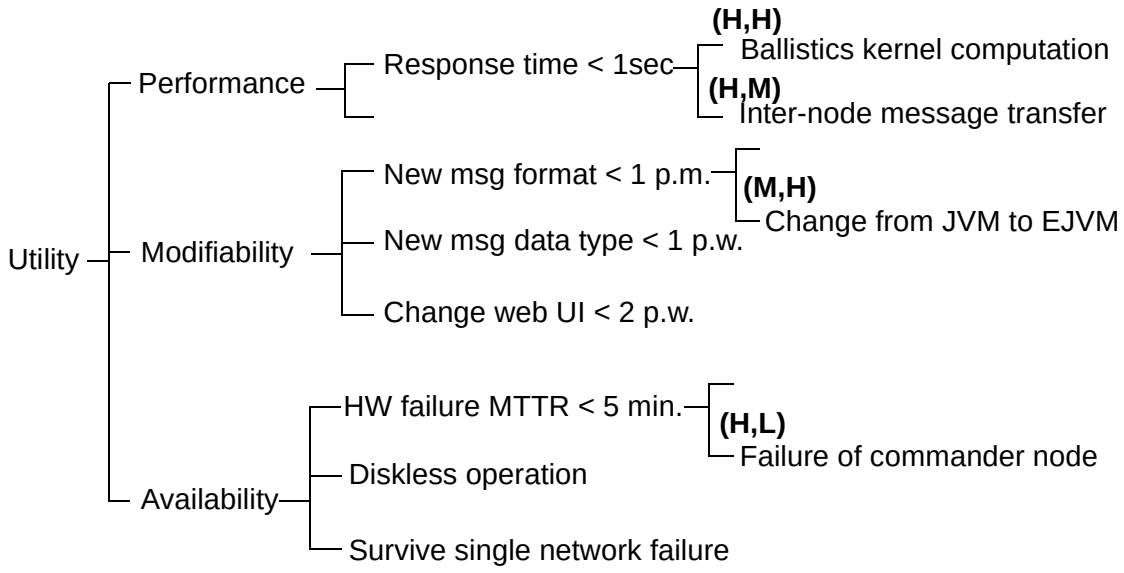


Figure 13: A Portion of the BMS Utility Tree

10.2.4 Analyze Architectural Approaches

At this stage in the analysis we had in our possession a set of architectural documentation including some architectural approaches that had been identified, but we had little insight into the way that the architectural approaches actually performed to accomplish the BCS's mission. So we used a set of screening questions and qualitative analysis heuristics to flesh out our

understanding of the approaches and to probe them for risks, sensitivity points, and tradeoffs. For example, for the backup commander approach (for availability) we generated a set of questions. The following is a subset of the questions applied to the style:

- How is the failure of a component detected?
- How is the failure of a communication channel detected?
- What is the latency for a spare component to be turned into a working replacement?
- By what means is a failed component marked for replacement?
- How are the system's working components notified that they should use the services of the spare?

These questions allowed us to probe the availability architectural approach and formed the basis for our analysis of this approach, as we shall show below.

It is important to note that by itself Figure 12 (or any similar representation of a software architecture view) tells us little about the system. However, when illuminated by a small number of scenarios and quality attribute-based analysis questions, a view can become the focus for an in-depth understanding and analysis.

Recall that a sensitivity point for a quality attribute is defined as a parameter in the architecture to which some measurable attribute is highly correlated; small changes in such parameters are likely to have significant effects on the measurable behavior of the system. For this reason we need to focus our attention on these points as they pose the highest risks to the system's success, particularly as the system evolves. We find sensitivity points by first using architectural approaches and their associated questions to guide us in our architectural elicitation. Based upon the elicited information we can begin to build models of some aspect of a quality attribute. For example, we might build a collection of formal analytic models such as rate monotonic analysis or queuing models to understand a performance goal such as predicting the worst-case inter-node latency [Klein 93]. We might use Markov models to understand an availability goal such as the expected availability of a system in the face of a server failure [Iannino 94]. We might also employ less formal qualitative analysis models such as that used in SAAM for modifiability [Kazman 94].

These models are frequently quite simple when we initially construct them; often we only have time for a back-of-the-envelope analysis during the 3 days of an ATAM. But when we build even these simple models we can experiment with their parameters until we determine which ones have substantial effects on a measurable attribute such as latency or throughput or mean time to failure. The point of the model building is thus twofold: to find sensitivity points (not to precisely characterize a measurable attribute, for it is typically too early in the development process to do this with any rigor); and to gain insight into the architecture, via the elicitation process that model building necessitates.

For the BCS system we realized, via the utility tree construction process, that three quality attributes were the major architectural drivers for overall system quality: availability, modifiability, and performance. Hence we can say that system quality (Q_S), is a function f of the quality of the modifiability (Q_M), the availability (Q_A), and the performance (Q_P):

$$Q_S = f(Q_M, Q_A, Q_P)$$

The next sections will describe the analyses that we created for performance and availability (the modifiability analysis was simply a SAAM analysis and will not be presented here [Kazman 94]). Each of these analyses was created by first eliciting an architectural approach and then applying quality attribute-specific analysis questions to elicit information about the attribute in question. Using this information we then built simple but adequate analytic models.

10.2.5 Availability

A key quality attribute for the BCS was determined to be its steady-state availability, i.e.,

$$Q_A = g(\text{the fraction of time that the system is working})$$

We determined that the system is considered to be working if there is a working Commander and any number of Soldier nodes. When the Commander fails the system has failed. Provisions had been made in the BCS architecture, however, to turn a Soldier into a Commander, i.e., converting a client into a server. The repair time for the system is the time to turn a Soldier node into the Commander and thus restore the system to operation. Failure of the Commander is detected via human-to-human communication.

As guided by the utility tree entry for availability, the key stimulus to model for the system is the failure of a node (specifically a Commander node) in the system due to an attack, hardware failure, or software failure. The architecture of the BCS, as presented to us, statically specified a Commander and a single backup selected from among the Soldier nodes, as indicated by the shaded Soldier node in Figure 12. In the existing design, *acknowledged* communication (state messages) takes place between the Commander and the backup Soldier node to allow the backup to mirror the Commander's state. This allows the backup to maintain a state of readiness in case of failure of the Commander. Upon failure of the Commander, the backup takes over as the new Commander, converting from a client to a server.

However, there is no provision to have one of the surviving Soldier nodes promoted to become the current backup. So we can refine our characterization of the system's availability as follows:

$$Q_A = g(\lambda_C, \mu_C, \mu_B)$$

That is, system availability is primarily affected by the failure rate of the Commander (λ_C), the repair rate of the Commander (μ_C, the time required to turn the backup into the Commander). The availability might also be affected by the repair rate of the backup, but in the BCS as it is currently designed, the repair rate of the backup (μ_B) is 0 since there is no provision for creating additional backups.

We determined that this is a potential risk. We can understand the nature of this risk by building a simple model of the system's availability. Using this model we can determine the effects of changing the repair rate of the backup on the system's overall availability. Specifically, we can determine the amount of time required for a new backup to enter a readiness state (i.e., where it could quickly become the Commander). This would, however, require a change to the architecture since the current architecture statically specifies only a single backup.

An alternative architecture could allow multiple (perhaps all) Soldier nodes to monitor the Commander-to-backup communication and thus maintain a higher level of readiness. And these additional backups could either acknowledge communication with the Commander (requesting resends of missed packets) or could be silent receivers of packets, or some mixture of these schemes (i.e., the top *n* backups acknowledge receipt of packets, and the remaining *m* backups are passive). In the case where packets are not acknowledged, the state of the backups database would increasingly drift from that of the Commander and if one of these backups is called upon to become the Commander, it would need to engage in some negotiation (with the external systems and/or the other Soldier nodes) to complete its database.

Thus, there are three considerations for changing the BCS architecture to improve the data distribution to the backups:

- a backup could be an "acknowledging backup", which is kept completely synchronized with the Commander
- a backup might be only a "passive" backup and not ask for re-sends when it misses a message; this implies that it has the means for determining that it has missed a message (such as a message numbering scheme)
- a backup, when it becomes the new Commander, or when it becomes an "acknowledging backup," could request any missed information from the upper level Command and Control systems and/or the other Soldier nodes.

The system does not need to choose a single option for its backups. It might have *n* acknowledging backups and *m* passive backups. Assuming that we have no control over the failure rate of the Commander, then the true sensitivities in this system with respect to availability are functions of the repair rates of the Commander and backups, which are themselves functions of the numbers of acknowledging and passive backups. Now we have a usable description of

the architectural sensitivities—a correlation between some architectural parameter and a measurable attribute:

$$Q_A = g(n, m)$$

What are the issues in the choice of the number of backups to maintain and whether they acknowledge communications or not? We consider this issue in terms of the failure and recovery rates of the system under the various options. Clearly, the availability of the system increases as the number of backups is increased, because the system can survive multiple failures of individual nodes without failing its mission. The availability of the system is also increased by increasing the number of acknowledging backups, for two reasons: 1) acknowledging backups can be ready to assume the responsibilities of an Commander much more quickly, because they do not need to negotiate with other nodes for missed information, and; 2) having more acknowledging backups means that there will not be an identifiable communication pattern between the Commander and the single backup, as there is currently, which means that the probability of two accurate incoming mortars disabling the system is reduced.

However, as the number of acknowledging backups is increased, the performance of the system is impacted, as each of these acknowledgments incurs a small communication overhead. Collectively, this overhead is significant, because as we will discuss next, communication latency is the major contributor to overall system latency for BCS.

10.2.6 Performance

Through the building of a simple performance model of the system and varying the input parameters to the model (the various processing times and communication latencies), it became clear that the slow speed of radio modem communication between the Commander and the Soldiers (9600 baud) was the single most important performance driver for the BCS, swamping all other considerations. The performance measure of interest—average latency of client-server communications—was found to be insensitive to all other architectural performance parameters (e.g., the time for the system to update its database, or to send a message internally to a process, or to do targeting calculations). But the choice of modem speed was given as a constraint, and so our performance model was then focused on capturing those architectural decisions that affected message sizes and distributions. Note here that we did not, and did not need to, build a complex RMA or queueing model to discover this correlation.

We thus began our investigation into message sizes by considering the performance implications and the communication requirements implied of the system's normal operations. We then

looked at the system's operations under the assumption of some growth in the performance requirements. To do this we considered three situations when building our performance model:

A) Regular, periodic data updates to the Commander) various message sizes and frequencies.

B) Turning a Soldier node into a backup: a switchover requires that the backup acquires information about all missions, updates to the environmental database, issued orders, current Soldier locations and status, and detailed inventories from the Soldiers.

C) Doubling number of weapons or the number of missions.

We created performance models of each of these situations. For the purposes of illustration in this example, we will only present the performance calculations for situation B), the conversion from Soldier backup to Commander.

After determining that a switchover is to take place the Soldier backup will need to download the current mission plans and environmental database from the external command and control systems. In addition, the backup needs the current locations and status of all of the remaining Soldiers, inventory status from the Soldiers, and the complete set of issued orders.

A typical calculation of the performance implications of this situation must take into account the various message sizes needed to realize the activities, the 9600 baud modem rate (which we equate to 9600 bits/second), and the fact that there are a maximum of 25 Soldiers per Commander (but since one is now being used as a Commander, the number of Soldier nodes in these calculations is 24):

Downloading mission plans:
280 Kbits / 9.6 Kbits/second ≅ 29.17 seconds

Updates to environmental database:
66 Kbits / 9.6 Kbits/second ≅ 6.88 seconds

Acquiring issued orders:
24 Soldiers * (18 Kbits/9.6 Kbits/second) = 45.0 seconds

Acquiring Soldier locations and status:
24 Soldiers * (12 Kbits/9.6 Kbits/second) = 30.0 seconds

Acquiring inventories:
24 Soldiers * (42 Kbits/9.6 Kbits/second) = 105.0 seconds

Total ≅ 216.05 seconds for Soldier to become backup

Note that since the radio modem is a shared communication channel, no other communication can take place while a Soldier/backup is being converted to a Commander.

There was no explicit requirement placed on the time to switch from a Commander to a backup. However, there was an initialization requirement of 300 seconds which we will use in lieu of an explicit switchover time budget. If we assume that the 280K bits in the mission plan file contains the three missions in the current configuration, then doubling the number of missions would imply doubling the mission message from 280K bits to 560K bits and the transmission time would increase by almost 30 seconds, still meeting the time budget. If, on the other hand, the number of Soldiers increases to 35, the total time will increase by about 90 seconds, which would not meet the time budget (when added to the 216.05 seconds for the soldier to become a backup).

Keeping each backup in a state of high readiness requires that they become acknowledging backups, or for a lower state of readiness they can be kept as passive backups. Both classes of backups require periodic updates from the Commander. From an analysis of situation B), we have calculated that these messages average 59,800 Kbits every 10 minutes. Thus, to keep each backup apprised of the state of the Commander requires 99.67 bits/second, or approximately 1% of the system's overall communication bandwidth. Acknowledgments and resends for lost packets would add to this overhead. Given this insight, we can characterize the system's performance sensitivities as follows:

$$Q_P = h(n, m, CO)$$

That is, the system is sensitive to the number of acknowledging backups (n), passive backups (m), and other communication overhead (CO). The main point of this simple analysis is to realize that the size and number of messages to be transmitted over the 9600 baud radio modem is important with respect to system performance and hence availability. Small changes in message sizes or frequencies can cause significant changes to the overall throughput of the system. These changes in message sizes may come from changes imposed upon the system from the outside.

10.2.7 Tradeoff Identification

As a result of these analyses we identified three sensitivities in the BCS system. Two of these are affected by the same architectural parameter: the amount of message traffic that passes over the shared communication channel employed by the radio modems, as described by some functions of n and m, the numbers of acknowledging and passive backups. Recall that availability and performance were characterized as

$$Q_A = g(n, m)$$

and

$$Q_P = h(n, m, CO)$$

These two parameters, *n* and *m*, control the tradeoff point between the overall performance of the system, in terms of the latency over its critical communication resource, and between the availability of the system in terms of the number of backups to the Commander, the way that the state of those backups is maintained, and the negotiation that a backup needs to do to convert to a Commander. To determine the criticality of the tradeoff more precisely, we can prototype or estimate the currently anticipated message traffic and the anticipated increase in message traffic due to acknowledgments of communications to the backups. In addition, we would need to estimate the lag for the switchover from Soldier to Commander introduced by not having acknowledged communication to the Solder backup nodes. Finally, all of this increased communication needs to be considered in light of the performance scalability of the system (since communication bandwidth is the limiting factor here).

One way to mitigate against the communication bandwidth limitation is to plan for new modem hardware with increased communication speeds. Presumably this means introducing some form of indirection into the modem communications software—such as an abstraction layer for the communications—if this does not already exist. This modifiability scenario was not probed during the evaluation.

While this tradeoff might seem obvious, given the presentation here, it was not so. The contractor was not aware of the performance and availability implications of the architectural decisions that had been made. In fact, in our initial pre-meeting, not a single performance or availability scenario was generated by the contractor; these simply were not among their concerns. The contractor was most worried about the modifiability of the system, in terms of the many changes in message formats that they expected to withstand over the BCS's lifetime. However, the identified tradeoff affected the very viability of the system. If this tradeoff was not carefully reasoned, it would affect the system's ability to meet its most fundamental requirements.

10.2.8 Scenario-Based Analysis

Scenarios represent uses of—stimuli to—the BCS architecture. These are applied not only to determine if the architecture meets a functional requirement, but also for further understanding of the system's architectural approaches and the ways in which these approaches meet the quality requirements such as performance, availability, modifiability, and so forth.

The scenarios for the BMS ATAM were collected by a round-robin brainstorming activity in which no criticism and little or no clarification was provided. Table 3 shows a few of the 40 growth-and-exploratory scenarios that were elicited in the course of the ATAM evaluation.[1]

Scenario	Scenario Description
1	Same information presented to user, but different presentation (location, fonts, sizes, colors, etc.).
2	Additional data requested to be presented to user.
3	User requests a change of dialog.
4	A new device is added to the network, e.g., a location device that returns accurate GPS data.
5	An existing device adds additional fields that are not currently handled to existing messages.
6	Map data format changes.
7	The time budget for initialization is reduced from 5 minutes to 90 seconds.
8	Modem baud rate is increased by a factor of 4.
9	Operating system changes to Solaris.
10	Operating schedule is unpredictable.
11	Can a new schedule be accommodated by the OS?
12	Change the number of Soldier nodes from 25 to 35.
13	Change the number of simultaneous missions from 3 to 6.
14	A node converts from being a Soldier/client to become a Commander/server.
15	Incoming message format changes.

Table 3: Examples of Scenarios for the BCS Evaluation

10.2.9 Scenario Prioritization

Prioritization of the scenarios allows the most important scenarios to be addressed within the limited amount of time (and energy) available for the evaluation. Here, “important” is defined entirely by the stakeholders. The prioritization was accomplished by giving each stakeholder a fixed number of votes; 30% of the total number of scenarios has been determined to be a useful heuristic. Thus, for the BCS, each stakeholder was given 12 votes that they use to vote for scenarios in which they were most interested. Typically, the resulting totals will provide an obvious cutoff point; 10-15 scenarios are the most that can be considered in a normal one-day session; for the BCS a natural cutoff occurred at 12 scenarios. Some negotiation is appropriate

1. These scenarios have been cleansed of proprietary specifics, but their spirit is true to the original.

in choosing which scenarios to consider; a stakeholder with a strong interest in a particular scenario can argue for its inclusion, even if it did not receive a large number of votes in the initial prioritization.

10.2.10 Analyze Architectural Approaches

In the ATAM process, once a set of scenarios have been chosen for consideration, these are "mapped" onto the architecture as test cases of the architectural approaches that have been documented. In the case of a scenario that implies a change to the architecture, the architect demonstrates how the scenario would affect the architecture in terms of the changed, added, or deleted components, connectors, and interfaces. For the use case scenarios, the architect traces the execution path through the relevant architectural views.

Stakeholder discussion is important here to elaborate the intended meaning of a scenario description and to discuss how the mapping is or is not suitable from their perspective. The mapping process also illustrates weaknesses in the architecture and its documentation, or even missing architectural approaches.

For the BCS, each of the high-priority scenarios were mapped onto the appropriate architectural approach. For example, when a scenario implied a modification to the architecture, the ramifications of the change were mapped onto the source view, and scenario interactions were identified as sensitivity points. For availability and performance, use case scenarios describing the execution and failure modes of the system were mapped onto run-time and system views of the architecture. During this mapping the models of latency and availability that we had built in the style-based analysis phase were further probed and refined.

11 Results Of The BCS ATAM

The ATAM that we performed on the architecture for the BCS revealed some potentially serious problems in the documentation of the architecture, the clarity of its requirements, its performance, its availability, and a potential architectural tradeoff. We will briefly summarize each of these problems in turn.

11.1 Documentation

The documentation provided at the inception of this project was minimal: two pages of diagrams that did not correspond to software artifacts in any rigorous way. This is, in our experience, typical, and is the single greatest impediment to having a productive architecture evaluation. Having a distinct Phase 1 and having the opportunity to request additional documentation from the contractor made the evaluation successful.

As a result of our interaction with the BCS contractor team, substantially augmented and higher quality architectural documentation was produced. This improved documentation became the basis for the evaluation. And the improvement in the documentation was identified by management as a major success of the ATAM process, even before we presented any findings.

11.2 Requirements

One benefit of doing any architectural evaluation is increased stakeholder communication, resulting in better understanding of requirements. Frequently, new requirements surface as a result of the evaluation. The BCS experience was typical, even though the requirements for this system were “frozen” and had been made public for over two years.

For example, in the BCS the only performance timing requirements were that the system be ready to operate in five minutes from power-on. In particular, there were no timing requirements for other specific operations of the system, such as responding to a particular order, or updating the environment database. These were identified as lacking by the questions we asked in building the performance model.

Furthermore, there was no explicit switchover requirement, i.e., the time that it takes for a Soldier to turn itself into a Commander is not identified as a requirement. This requirement surfaced as a result of building the availability model.

In addition, there was no stated availability requirement. Two well aimed hits, or two specific hardware failures and the system, as it is currently designed, is out of commission. By the end of the ATAM the stakeholders viewed this as a major oversight in the system's design.

11.2.1 Sensitivities and Tradeoffs

The most important tradeoff identified for the BCS was the communications load on the system, as it was affected by various information exchange requirements and availability schemes. The overall performance and availability of the system is highly sensitive to the latency of the (limited and shared) communications channel, as controlled by the parameters *n* and *m*. Not only should the current performance characteristics be modeled, but also the anticipated performance changes in the future as the system scales in its size and scope.

11.3 Architectural Risks

In addition to the sensitivities and tradeoffs, we discovered, in building the models of the BCS's availability and performance, a serious architectural weakness that had not been previously identified: there exists the possibility of an opposing force identifying the distinctive communication pattern between the Commander and the backup and thus targeting those nodes specifically. The Commander and backup exchange far more data than any other nodes in the system. This identification can be easily done by an attacker who could discern the distinctive pattern of communication between the Commander and (single) backup, even without being able to decrypt the actual contents of the messages. Thus, it must be assumed that the probability of failure for the Commander and its backup increases over the duration of a mission under the existing BCS architecture.

This was a major architectural flaw that was only revealed *because* we were examining the architecture from the perspective of multiple quality attributes simultaneously. This flaw is, however, easily mitigated by assigning multiple backups, which would eliminate the distinctive communication pattern.

11.4 Reporting Back and Mitigation Strategies

As described above, after the evaluation the contractors were sent a detailed report of the ATAM. In particular, they were alerted to the potential modifiability, performance, and availability problems in the BCS. As an aid to risk mitigation, the contractor was furnished with three architectural possibilities for adding multiple Soldier backups and keeping them more or

less synchronized with the Commander. Each of these architectural changes had their own ramifications that needed to be modeled and assessed. Each of these alternatives will respond differently depending on other architectural choices that affect the performance model, such as the radio modem speed.

The point of this example is not to show which alternative the contractor chose, for that is relatively unimportant and relied on their organizational and mission-specific constraints. The point here is to show that the process of performing an ATAM on the BCS raised the stakeholders' awareness of critical risk, sensitivity, and tradeoff issues. This, in turn, focused design activity in the areas of highest risk and caused a major iteration within the spiral process of design and analysis.

12 Conclusions

An architecture analysis method, any architecture analysis method, is a garbage-in-garbage-out process. The ATAM is no different. It crucially relies on the active and willing participation of the stakeholders (particularly the architecture team); some advance preparation by the key stakeholders; an understanding of architectural design issues and analytic models; and a clearly articulated set of quality attribute requirements and a set of business goals from which they are derived. Our purpose in creating a *method* (rather than, say, just putting some intelligent and experienced people together in a room and having them chat about the architecture or inspect it in an arbitrary way) is to increase the effectiveness and repeatability of the analysis.

To this end we have not only defined a set of steps and their associated stakeholders, but have also amassed a set of techniques, guidance, documentation templates, standard letters and agendas, and so forth. We have created quality attribute characterizations and sets of associated questions. We have documented attribute-based architectural styles. We have created sample templates for the evaluators to fill in with architectural information. We have developed guidelines and techniques for eliciting and voting on scenarios. We created the idea of a utility tree, along with ways of both eliciting and prioritizing it. The ATAM is an infrastructure that puts this set of techniques and artifacts together into a working repeatable whole.

The architectures of substantial software-intensive systems are large and complex. They involve many stakeholders, each of whom has their own agenda. These architectures are frequently incompletely thought out or only partially documented. A method for architecture evaluation has to consider and overcome all of these daunting challenges. Our techniques are aimed at taming this considerable complexity and ensuring the achievement of the main goals for an ATAM: to obtain a precise statement of the quality attribute requirements; to understand the architectural decisions that have been made; and to determine if the architectural decisions made adequately address the quality attribute requirements.

Bibliography

[Bass 98] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

[Buschmann 96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture*. Chichester, UK: Wiley, 1996.

[Iannino 94] Iannino, A. "Software Reliability Theory." *Encyclopedia of Software Engineering,* New York, NY: Wiley, May 1994, 1237-1253.

[Kazman 99] Kazman, R. & Carriere, S.J. "Playing Detective: Reconstructing Software Architecture from Available Evidence." *Automated Software Engineering, 6,* 2 (April 1999): 107-138.

[Kazman 94] Kazman, R.; Abowd, G.; Bass, L.; & Webb, M. "SAAM: A Method for Analyzing the Properties of Software Architectures," 81-90. *Proceedings of the 16th International Conference on Software Engineering*. Sorrento, Italy, May 1994.

[Klein 99] Klein, M. & Kazman, R. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022, ADA371802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW. URL: <http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022abstract.html>.

[Klein 93] Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; & Gonzales Harbour, M. *A Practitioner's Handbook for Real-Time Analysis*. New York, NY: Kluwer Academic, 1993.

[Leung 82] Leung, J.L.T. & Whitehead, J. "On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks." *Performance Evaluation, 2,* 4 (1982) 237-250.

[Rajkumar 91] Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Boston, MA: Kluwer Academic Publishers, 1991.

[Saaty 94] Saaty, T. *Fundamentals of Decision Making and Priority Theory With the Analytic Hierarchy Process.* Pittsburgh, PA: RWS Publications, 1994.

[Shaw 96] Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice-Hall, 1996.

Appendix A Attribute Characteristics

A.1 Performance

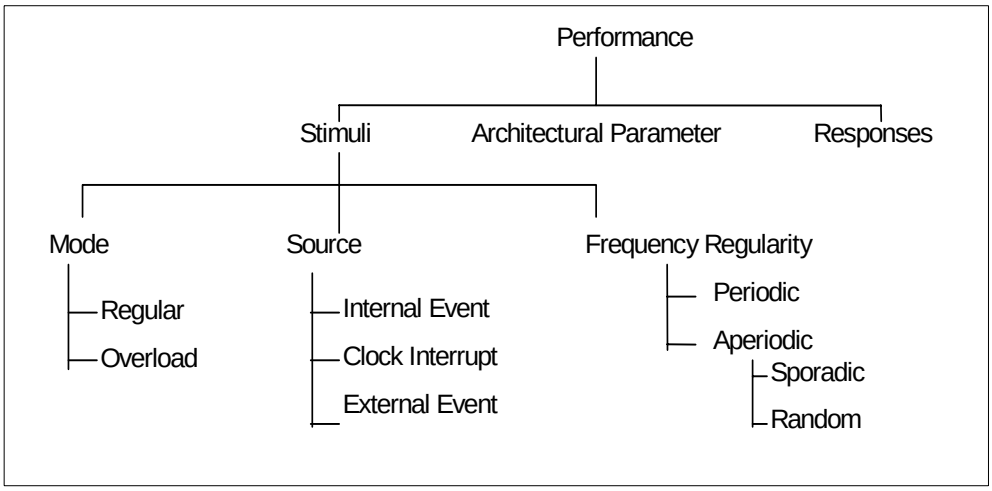


Figure 14: Performance Characterization—Stimuli

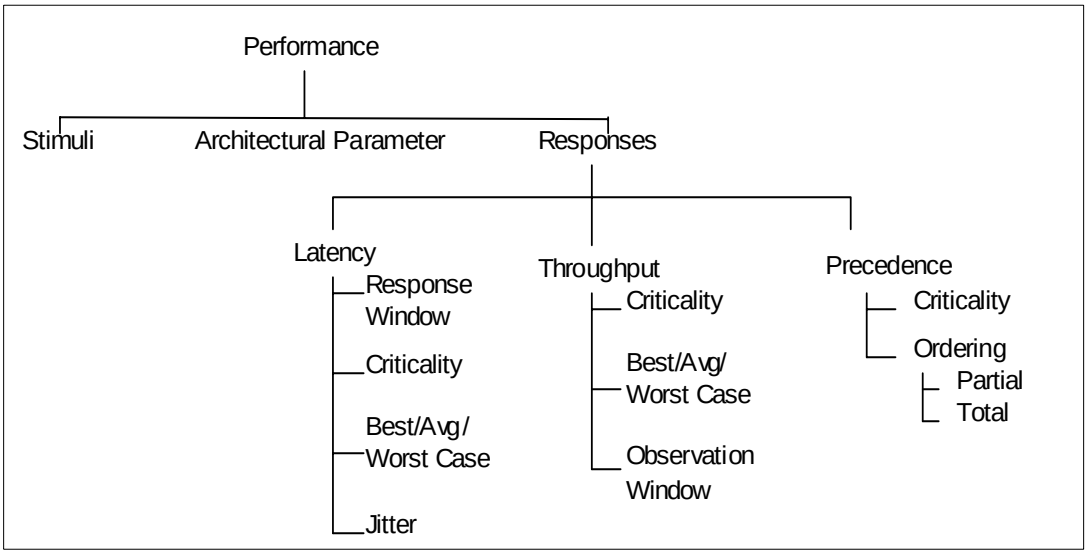


Figure 15: Performance Characterization—Responses

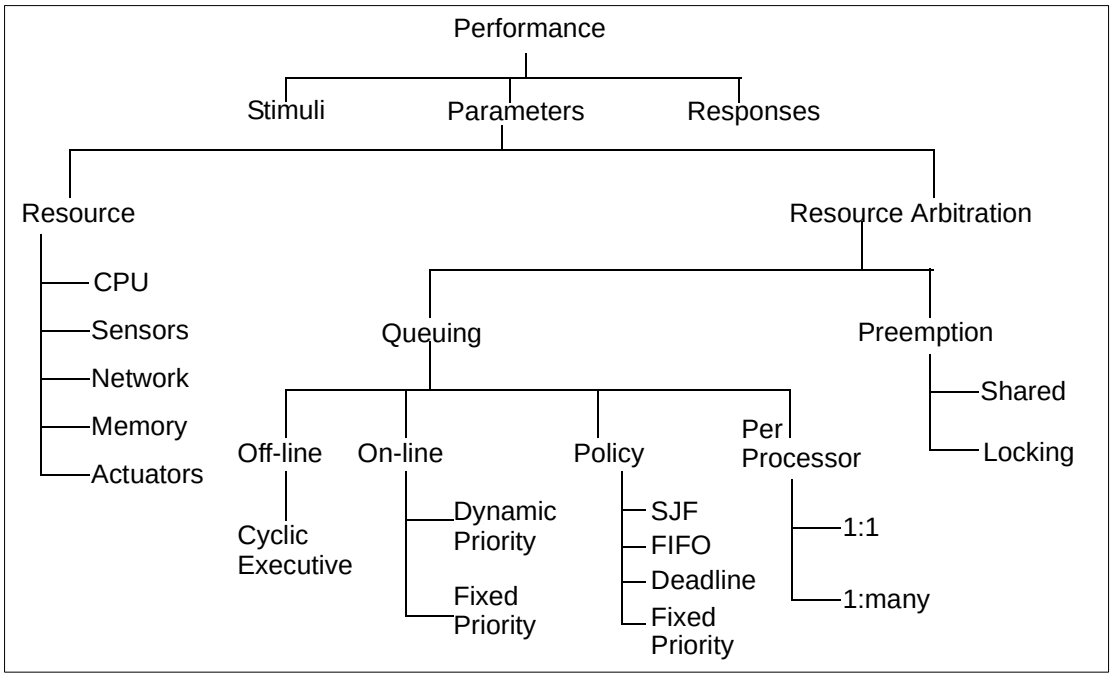


Figure 16: Performance Characterization—Architectural Decisions

A.2 Modifiability

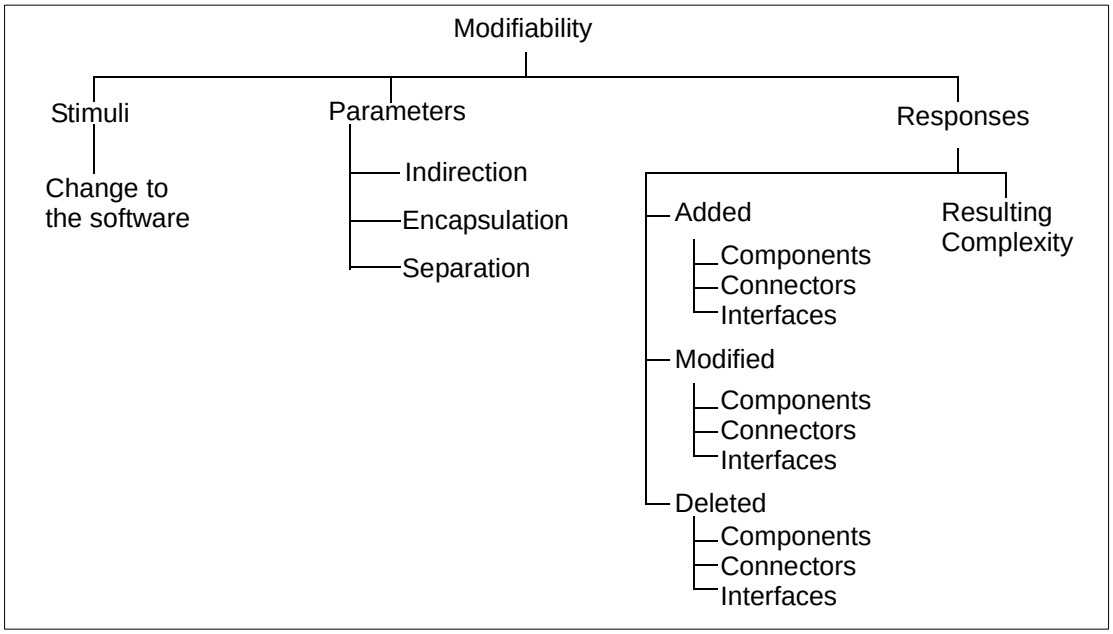


Figure 17: Modifiability Characterization

A.3 Availability

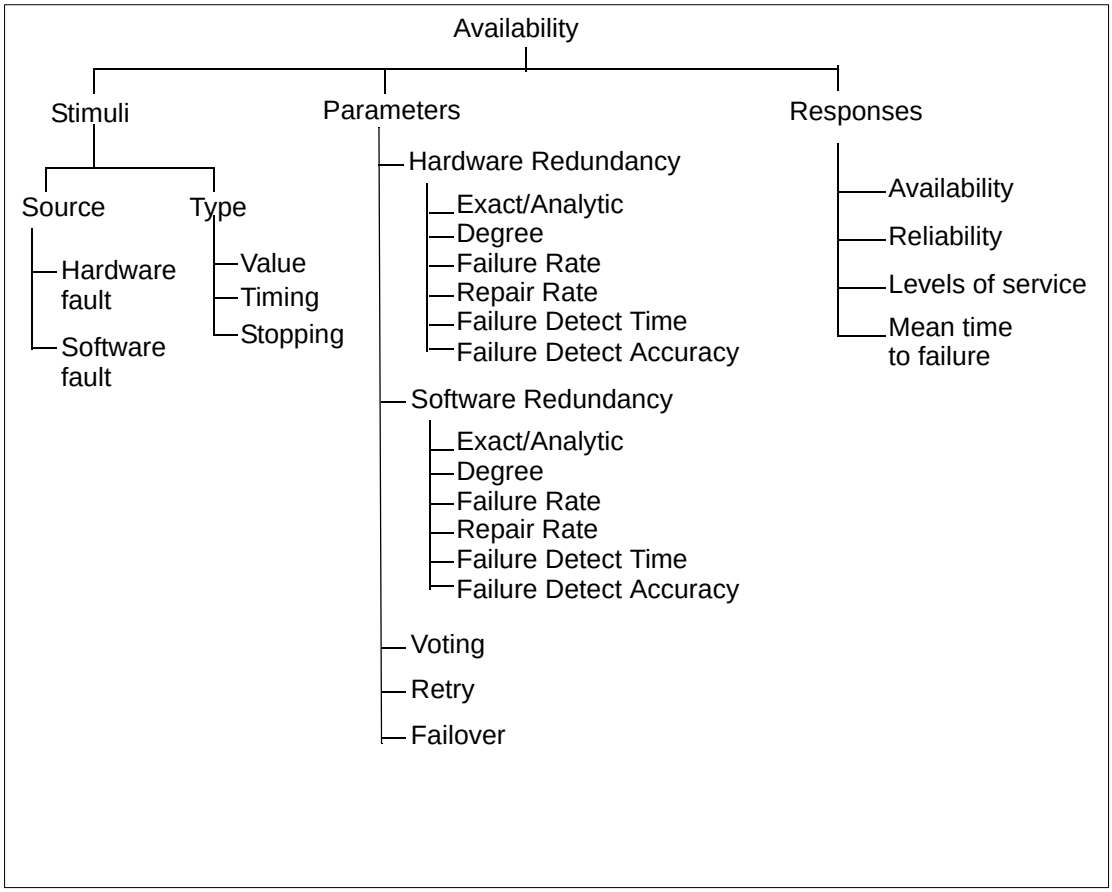


Figure 18: Availability Characterization

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

13. **AGENCY USE ONLY** (leave blank)	14. **REPORT DATE** August 2000	15. **REPORT TYPE AND DATES COVERED** Final
16. **TITLE AND SUBTITLE** *ATAM: Method for Architecture Evaluation*		17. **FUNDING NUMBERS** C — F19628-95-C-0003
18. **AUTHOR(S)** Rick Kazman, Mark Klein, Paul Clements		
19. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		20. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2000-TR-004
21. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		22. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2000-004
23. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

If a software architecture is a key business asset for an organization, then architectural analysis must also be a key practice for that organization. Why? Because architectures are complex and involve many design tradeoffs. Without undertaking a formal analysis process, the organization cannot ensure that the architectural decisions made—particularly those which affect the achievement of quality attribute such as performance, availability, security, and modifiability—are advisable ones that appropriately mitigate risks. In this report, we will discuss some of the technical and organizational foundations for performing architectural analysis, and will present the Architecture Tradeoff Analysis MethodSM (ATAM)—a technique for analyzing software architectures that we have developed and refined in practice over the past three years.

14. **SUBJECT TERMS** software architecture, architectural analysis, ATAM, quality attributes			15. **NUMBER OF PAGES** 84 16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL