Software Architecture

Sneheet Mishra

April 26, 2012

Executive Summary

- This presentation is an introduction to the topic of software architecture.
- Significance of software architecture, key architectural principles, major design considerations, responsibilities of an architect and factors affecting choice of a particular architecture are described.
- Various common architectural styles – their key features, benefits and applications are presented.

Software Architecture – Definition (1)

- Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- It consists of all the important design decisions about the software structures and the interactions between those structures that comprise the system.

Software Architecture – Definition (2)

- Software architecture is described as the organization or structure of a system.
- The system represents a collection of components that accomplish a specific function or set of functions.
- Architecture is focused on organizing components to support specific functionality.

Why is architecture important?

- It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application.
- Failing to consider key scenarios, failing to design for common problems, or failing to appreciate the long term consequences of key decisions can put your application at risk.
- The risks exposed by poor architecture include software that is unstable, is unable to support existing or future business requirements, or is difficult to deploy or manage in a production environment.

Some high level concerns related to architecture

- How will the users be using the application?
- How will the application be deployed into production and managed?
- What are the quality attribute requirements for the application, such as security, performance, concurrency, and configuration?
- How can the application be designed to be flexible and maintainable over time?
- What are the architectural trends that might impact your application now or after it has been deployed?

Key Architecture Principles (1)

- Build to change instead of building to last
  - Consider how the application may change over time to address new requirements and challenges, and build in the flexibility to support this.
- Model to analyze and reduce risks
  - Use design tools, modeling systems such as Unified Modeling Language (UML), and visualizations to help capture requirements and architectural and design decisions, and to analyze their impact.
- Identify key engineering decisions

Key Architecture Principles (2)

- Use an incremental and iterative approach to refining the architecture.
- Start with a baseline architecture to get the big picture right.
- Iteratively add details to the design over multiple passes
  - Common pitfall is to dive into the details too quickly and get the big decisions wrong by making incorrect assumptions, or by failing to evaluate the architecture effectively.

Key Architecture Principles (3)

- Separation of concerns
  - Divide your application into distinct features with as little overlap in functionality as possible.
  - The important factor is minimization of interaction points to achieve high cohesion and low coupling.
- Single Responsibility Principle
  - Each component or module should be responsible for only a specific feature or functionality.
- Principle of Least Knowledge
  - A component or object should not know about internal details of other components or objects.

Key Architecture Principles (4)

- Specify intent in only one place
  - Specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.
- The goal of a software architect is to minimize the complexity by separating the design into different areas of concern.
  - For example, the user interface (UI), business processing, and data access all represent different areas of concern.
  - Within each area, the components should focus on that specific area and should not mix code from other areas of concern.
  - For example, UI processing components should not include code that directly accesses a data source, but instead should use either business components or data access components to retrieve data.

Important Design Pointers

- Keep design patterns consistent within each layer of the architecture
- Prefer composition to inheritance when reusing functionality
- Establish a coding style and naming convention for development
- Maintain system quality using automated QA techniques during development
- Define a clear contract for components

Key Architecture Decisions (1)

- Determine the Application Type
  - Choice is governed by specific requirements and infrastructure limitations
- Some common types:
  - Applications designed for mobile devices.
  - Rich client applications designed to run primarily on a client PC.
  - Rich Internet applications designed to be deployed from the Internet, which support rich UI and media scenarios.
  - Service applications designed to support communication between loosely coupled components.
  - Web applications designed to run primarily on the server in fully connected scenarios.

Key Architecture Decisions (2)

- Determine the Deployment Strategy
  - Application may be deployed in a variety of environments, each with its own specific set of constraints such as physical separation of components across different servers, a limitation on networking protocols, firewall and router configurations, and more.
- Several common deployment patterns exist, which describe the benefits and considerations for a range of distributed and non-distributed scenarios.
- Requirements of the application should be balanced with the appropriate patterns that the hardware can support, and the constraints that the environment exerts on the deployment options.

Key Architecture Decisions (3)

- Determine the Appropriate Technologies
  - Compare the capabilities of the technologies chosen against application requirements, taking into account following factors before making decisions:
    - Type of application being developed
    - Preferred options for application deployment technology
    - Organization policies, infrastructure limitations and resource skills

Key Architecture Decisions (4)

- Determine the quality attributes
  - Quality attributes are system properties that are separate from the functionality of the system
- Attributes like security, performance, and usability can be used to focus on the critical problems that the design should solve
- Understand the requirements and deployment scenarios first to determine which quality attributes are important.
- Quality attributes may conflict; for example, security often requires a trade-off against performance or usability.

Key Architecture Decisions (5)

- Determine the crosscutting concerns
  - Crosscutting concerns represent key areas of design that are not related to a specific layer in the application
- Some of the key concerns are:
  - **Instrumentation and logging -** Instrument all of the business-critical and system-critical events, and log sufficient details to recreate events in the system without including sensitive information
  - **Authorization -** Ensure proper authorization with appropriate granularity within each layer, and across trust boundaries
  - **Exception management -** Catch exceptions at functional, logical, and physical boundaries; and avoid revealing sensitive information to end users
  - **Caching -** Identify what should be cached, and where to cache, to improve application's performance and responsiveness

Architectural Styles - Definition

- An architectural style is a set of principles that shapes an application
- It is a coarse grained pattern that provides an abstract framework for a family of systems.
- An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems
- The architecture of a software system is almost never limited to a single architectural style, but is often a combination of architectural styles that make up the complete system
- Key factors that influence the choice of a particular style are the capacity of the organization for design and implementation, the capabilities and experience of the developers, and the infrastructure and organizational constraints.

Key Architectural Styles (1)

- Client / Server Architecture
  - Segregates the system into two applications, where the client makes requests to the server
- Service-Oriented Architecture (SOA)
  - Refers to applications that expose and consume functionality as a service using contracts and messages
- Object-Oriented Architecture
  - A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behaviour relevant to the object
- Layered architecture
  - Partitions the concerns of the application into stacked groups (layers)

Key Architectural Styles (2)

- Component-Based Architecture
  - Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces
- Domain Driven Design
  - An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain
- Message Bus Architecture
  - An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other
- N-Tier / 3-Tier Architecture
  - Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer

Client / Server Architecture

- It describes distributed systems that involve a separate client and server system, and a connecting network.
- The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style
- The client/server architectural style describes the relationship between a client and one or more servers as:
  - The client initiates one or more requests, waits for replies, and processes the replies on receipt
  - The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client
- Examples include Web browser-based programs running on the Internet; applications that access remote data stores (e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).

Benefits of Client / Server Architecture

- **Higher security -** All data is stored on the server, which generally offers a greater control of security than client machines.
- **Centralized data access -** Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- **Ease of maintenance -** Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Variations of client/server style

- **Client-Queue-Client systems**
  - Allows clients to communicate with other clients through a server-based queue
  - Clients can read data from and send data to a server that acts simply as a queue to store the data
  - This allows clients to distribute and synchronize files and information
- **Peer-to-Peer (P2P) applications**
  - Developed from the Client-Queue-Client style, the P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients
  - It extends the client/server style through multiple responses to requests, shared data, resource discovery, and resilience to removal of peers
- **Application servers**
  - A specialized architectural style where the server hosts and executes applications and services that a thin client accesses through a browser or specialized client installed software
  - An example is a client executing an application that runs on the server through a framework such as Terminal Services

Component-Based Architecture

- It focuses on the decomposition of the design into individual functional or logical components that expose well-defined communication interfaces containing methods, events, and properties.
- This provides a higher level of abstraction than object-oriented design principles, and does not focus on issues such as communication protocols and shared state.
- The key principle of the component-based style is the use of components that are:
  – **Reusable -** Components are usually designed to be reused in different scenarios in different applications.
  – **Replaceable -** Components may be readily substituted with other similar components.
  – **Extensible -** A component can be extended from existing components to provide new behaviour.
  – **Encapsulated -** Components expose interfaces that allow the caller to use its functionality, and do not reveal details of the internal processes or any internal variables or state.
  – **Independent -** Components are designed to have minimal dependencies on other components. Therefore components can be deployed into any appropriate environment without affecting other components or systems.

More on Components

- Common types of components used in applications include user interface components such as grids and buttons, and helper and utility components that expose a specific subset of functions used in other components.
- Components depend upon a mechanism within the platform that provides an environment in which they can execute, often referred to as *component architecture*.
- Examples are the Component Object Model (COM), Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA) and Enterprise JavaBeans (EJB).
- Component architectures manage the mechanics of locating components and their interfaces, passing messages or commands between components, and in some cases maintaining state.

Benefits of Component-Based style

- **Ease of deployment**
  - As new compatible versions become available, existing versions can be replaced with no impact on the other components or the system as a whole
- **Reduced cost**
  - The use of third-party components allows to spread the cost of development and maintenance
- **Ease of development**
  - Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system
- **Reusable**
  - The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems
- **Mitigation of technical complexity**
  - Components mitigate complexity through the use of a component container and its services. Example component services include component activation, lifetime management, method queuing, and transactions

Domain Driven Design Architectural Style

- An object-oriented approach to designing software based on the business domain, its elements and behaviours, and the relationships between them.
- It aims to enable software systems that are a realization of the underlying business domain by defining a domain model expressed in the language of business domain experts.
- The domain model can be viewed as a framework from which solutions can then be rationalized.
- To apply Domain Driven Design, a good understanding of the business domain to be modelled is essential.
- Used in complex domains to improve communication and understanding within the development team, or to express the design of an application in a common language that all stakeholders can understand.
- An ideal approach for large and complex enterprise data scenarios that are difficult to manage using other techniques.

Key Benefits of Domain Driven Design

- **Communication**
  - All parties within a development team can use the domain model and the entities it defines to communicate business knowledge and requirements using a common business domain language, without requiring technical jargon.
- **Extensible**
  - The domain model is often modular and flexible, making it easy to update and extend as conditions and requirements change.
- **Testable**
  - The domain model objects are loosely coupled and cohesive, allowing them to be more easily tested.

Layered Architectural Style

- It focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other.
- Functionality within each layer is related by a common role or responsibility.
- Communication between layers is explicit and loosely coupled.
- Layering an application appropriately helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.
- It has been described as an *inverted pyramid of reuse* where each layer aggregates the responsibilities and abstractions of the layer directly beneath it.
- Examples include line-of-business (LOB) applications such as accounting and customer-management systems; enterprise Web-based applications and Web sites, and enterprise desktop or smart clients with centralized application servers for business logic.

Common design principles of Layered Architecture

- **Abstraction**
  - Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
- **Encapsulation**
  - No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.
- **Clearly defined functional layers**
  - The separation between functionality in each layer is clear.
- **High cohesion**.
  - Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
- **Reusable**
  - Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.
- **Loose coupling**
  - Communication between layers is based on abstraction and events to provide loose coupling between layers.

Key Benefits of Layered Architectural Style

- **Abstraction**
  - Layers allow changes to be made at the abstract level.
- **Isolation**
  - Allows isolation of technology upgrades to individual layers in order to reduce risk and minimize impact on the overall system.
- **Manageability**
  - Separation of core concerns helps to identify dependencies, and organizes the code into more manageable sections.
- **Performance**
  - Distributing the layers over multiple physical tiers can improve scalability, fault tolerance, and performance.
- **Reusability**
  - Roles promote reusability. For example, in MVC, the Controller can often be reused with other compatible Views in order to provide a role specific or a user-customized view on to the same data and functionality.
- **Testability**
  - Increased testability arises from having well-defined layer interfaces, as well as the ability to switch between different implementations of the layer interfaces.

Message Bus Architectural Style

- It describes the principle of using a software system that can send and receive messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
- It is a style for designing applications where interaction between applications is accomplished by passing messages over a common bus.
- The most common implementations of message bus architecture use either a messaging router or a Publish/Subscribe pattern, and are often implemented using a messaging system such as Message Queuing.
- Many implementations consist of individual applications that communicate using common schemas and a shared infrastructure for sending and receiving messages.

Role of Message Bus

- **Message-oriented communications**
  - All communication between applications is based on messages that use known schemas
- **Complex processing logic**
  - Complex operations can be executed by combining a set of smaller operations, each of which supports specific tasks
- **Modifications to processing logic**
  - Because interaction with the bus is based on common schemas and commands, applications can be inserted or removed on the bus to change the logic that is used to process messages
- **Integration with different environments**
  - By using a message-based communication model based on common standards, you can interact with applications developed for different environments

Key Benefits of Message Bus Architecture

- **Extensibility**
  - Applications can be added to or removed from the bus without having an impact on the existing applications.
- **Low complexity**
  - Application complexity is reduced because each application only needs to know how to communicate with the bus.
- **Flexibility**
  - The set of applications that make up a complex process can be changed easily to match changes in business or user requirements.
- **Loose coupling**
  - As long as applications expose a suitable interface for communication with the message bus, there is no dependency on the application itself, allowing changes, updates, and replacements that expose the same interface.
- **Scalability**
  - Multiple instances of the same application can be attached to the bus in order to handle multiple requests at the same time.

N-Tier Architectural Style

- It describes the separation of functionality into segments in much the same way as the layered style, but with each segment being a tier that can be located on a physically separate computer.
- It evolved through the component-oriented approach, generally using platform specific methods for communication instead of a message-based approach.
- N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization.
- The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th, and how to handle the results of the request.
- Examples include:
  - a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network.
  - a typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.

Key Benefits of N-Tier Architecture

- **Maintainability**
  - Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability**
  - Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**
  - Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**
  - Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Object-Oriented Architectural Style

- It is a design paradigm based on the division of responsibilities for an application into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
- An object-oriented design views a system as a series of cooperating objects, instead of a set of routines or procedural instructions.
- Objects are discrete, independent, and loosely coupled; they communicate through interfaces, by calling methods or accessing properties in other objects, and by sending and receiving messages.
- Common uses of the object-oriented style include defining an object model that supports complex scientific or financial operations, and defining objects that represent real world artifacts within a business domain.

Key principles of Object-Oriented Style

- **Abstraction**
  - This allows to reduce a complex operation into a generalization that retains the base characteristics of the operation.
- **Composition**
  - Objects can be assembled from other objects, and can choose to hide these internal objects from other classes or expose them as simple interfaces.
- **Inheritance**
  - Objects can inherit from other objects, and use functionality in the base object or override it to implement new behavior.
- **Encapsulation**
  - Objects expose functionality only through methods, properties, and events, and hide the internal details such as state and variables from other objects.
- **Polymorphism**
  - This allows to override the behavior of a base type that supports operations in an application by implementing new types that are interchangeable with the existing object.
- **Decoupling**
  - Objects can be decoupled from the consumer by defining an abstract interface that the object implements and the consumer can understand. This allows to provide alternative implementations without affecting consumers of the interface.

Key Benefits of Object Oriented Style

- **Understandable**
  - maps the application more closely to the real world objects, making it more understandable.
- **Reusable**
  - provides for reusability through polymorphism and abstraction.
- **Testable**
  - provides for improved testability through encapsulation.
- **Extensible**
  - encapsulation, polymorphism, and abstraction ensure that a change in the representation of data does not affect the interfaces that the object exposes, which would limit the capability to communicate and interact with other objects.
- **Highly Cohesive**
  - by locating only related methods and features in an object, and using different objects for different sets of features a high level of cohesion can be achieved.

Service-Oriented Architectural Style

- It enables application functionality to be provided as a set of services, and the creation of applications that make use of software services.
- Services are loosely coupled because they use standards-based interfaces that can be invoked, published, and discovered.
- Services in SOA are focused on providing a schema and message-based interaction with an application through interfaces that are application scoped, and not component or object-based.
- The SOA style can package business processes into interoperable services, using a range of protocols and data formats to communicate information.
- Clients and other services can access local services running on the same tier, or access remote services over a connecting network.
- Common examples include sharing information, handling multistep processes such as reservation systems and online stores, exposing industry specific data or services over an extranet, and creating mash-ups that combine information from multiple sources.

Key Principles of SOA Style

- **Services are autonomous**
  - Each service is maintained, developed, deployed, and versioned independently.
- **Services are distributable**
  - Services can be located anywhere on a network, locally or remotely, as long as the network supports the required communication protocols.
- **Services are loosely coupled**
  - Each service is independent of others, and can be replaced or updated without breaking applications that use it as long as the interface is still compatible.
- **Services share schema and contract, not class**
  - Services share contracts and schemas when they communicate, not internal classes.
- **Compatibility is based on policy**
  - Policy in this case means definition of features such as transport, protocol, and security.

Key Benefits of SOA style

- **Domain alignment**
  - Reuse of common services with standard interfaces increases business and technology opportunities and reduces cost.
- **Abstraction**
  - Services are autonomous and accessed through a formal contract, which provides loose coupling and abstraction.
- **Discoverability**
  - Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.
- **Interoperability**
  - Because the protocols and data formats are based on industry standards, the provider and consumer of the service can be built and deployed on different platforms.
- **Rationalization**
  - Services can be granular in order to provide specific functionality, rather than duplicating the functionality in number of applications, which removes duplication.

Conclusion

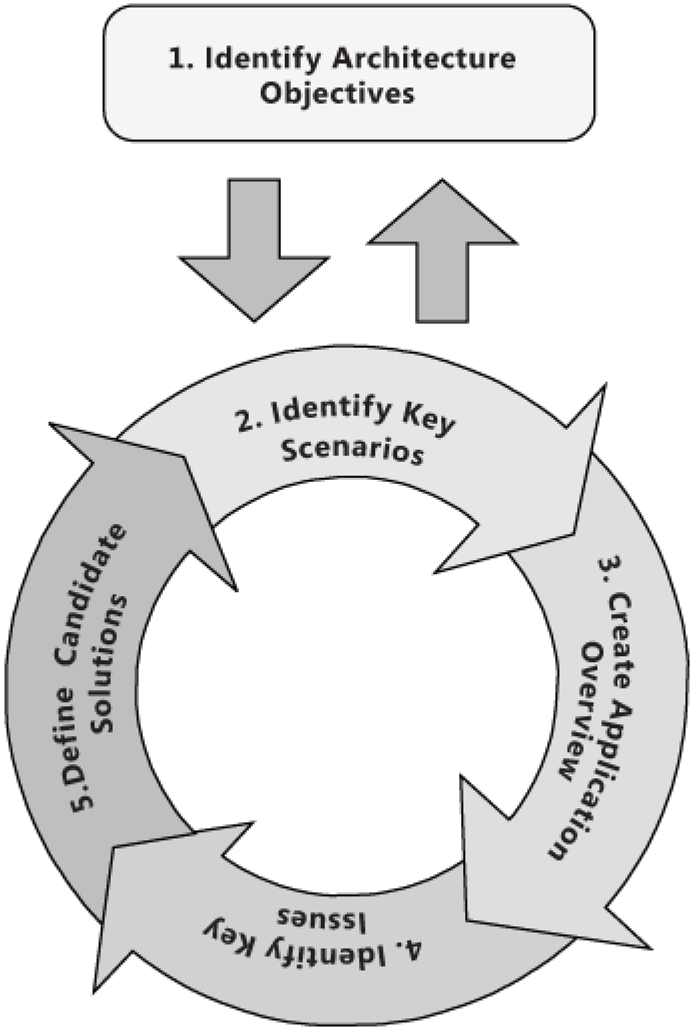
•**Identify Architecture Objective -** Clear objectives help you to focus on your architecture and on solving the right problems in your design. Precise objectives help you to determine when you have completed the current phase, and when you are ready to move to the next phase.

•**Key Scenarios -** Use key scenarios to focus your design on what matters most, and to evaluate your candidate architectures when they are ready.

•**Application Overview -** Identify your application type, deployment architecture, architecture styles, and technologies in order to connect your design to the real world in which the application will operate.

•**Key Issues -** Identify key issues based on quality attributes and crosscutting concerns..

•**Candidate Solutions -** Create an architecture spike or prototype that evolves and improves the solution and evaluate it against your key scenarios, issues, and deployment constraints before beginning the next iteration of your architecture.


1. Identify Architecture Objectives
2. Identify Key Scenarios
3. Create Application Overview
4. Identify Key Issues
5.Define Candidate Solutions

References

- Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice, Second Edition. Addison-Wesley, 2003.
- Fowler, Martin: Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
- Microsoft Patterns and Practices. Microsoft Press, 2009.
- David Garlan, Mary Shaw: An Introduction to Software Architecture. CMU, 1994