DESIGN OF SOFTWARE ARCHITECTURE

Instructor: Dr. Hany H. Ammar
Dept. of Computer Science and Electrical Engineering, WVU
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Defining Subsystems
  - Defining Subsystem Interfaces
- Design Using Architectural Styles
  - Software Architecture Styles
  - The Attribute Driven Design (ADD)
UML Development - Overview

- **Requirements Elicitation**
  - Actors
  - Scenarios

- **Analysis**
  - Specify Domain
  - Objects
  - Analysis Class Diagrams

- **Detailed Design**
  - Include Design Objects
  - Subsystem Class or Component Diagrams

- **Implementation Choices**

- **Design**
  - Deployment Diagram
  - Design Diagrams
  - Design Sequence/Communication Diagram

- **Implementation**
  - Program
The Requirements, Analysis, and Design Models

Requirements Elicitation Process

The Analysis Process

The Design Process

Functional/Nonfunctional Requirements

Static Analysis Dynamic Analysis

Use Case Diagrams/Sequence Diagrams (the system level)

- Analysis Class Diagrams
- State Diagrams/Refined Sequence Diagrams (the object level)

- Design Class Diagrams and Components Diagrams
- Design Sequence/
- Collaboration Diagrams
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Defining Subsystems
  - Defining Subsystem Interfaces
- Design Using Architectural Styles
What is Software Architecture?

A simplified Definition

A **software architecture** is defined by a configuration of architectural elements—**components, connectors, and data**—constrained in their relationships in order to achieve a desired set of architectural properties.
Software Architecture Elements

- A *component* is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.

- A *connector* is an abstract mechanism that mediates communication, coordination, or cooperation among components.
A *datum* is an element of information that is transferred from a component, or received by a component, via a connector.

A *configuration* is the structure of architectural relationships among components, connectors, and data during a period of system run-time.

*Software Architecture views:* Architectures are described using multiple views such as the *static view*, the *dynamic view*, and *deployment view*.

An *architectural style* is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.
The static view

Figure 21.8. Banking System: major subsystems
The dynamic view, a high level diagram

Figure 12.2. Dynamic view of client/server software architecture: high-level communication diagram for Banking System
The dynamic view of the ATMClient for a certain Use Case Scenario
Figure 21.12. Sequence diagram: ATM client Validate PIN use case
The deployment view

Figure 21.36. Deployment diagram for Banking System
Introducing Architecture Styles
More details on architecture styles to be discussed later

- The Layered Architecture
  e.g Network Services
  Architecture
Network Services Architecture

Deployment view

Figure 12.5. Layers of Abstraction architectural pattern: Internet communication with TCP/IP
Layered Software Architectural styles
Example of Web Applications Architecture Style
Service Oriented Architecture (SOA):

Makes use of an Enterprise Service Bus (ESB)

Used in web-based systems and distributed computing environments. Nodes on a network make resources available to other participants in the network as independent services that the participants access in a standardized way using the ESB.
Examples of Architecture Styles

- **Embedded Systems architecture style**

  ![Diagram showing the relationship between Monitors, Controllers, Schedulers, Input devices or actors, and Output devices or actors.]

  - Monitors
  - Controllers
  - Schedulers
  - Input devices or actors
  - Output devices or actors
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Defining Subsystems
  - Defining Subsystem Interfaces
- Design Using Architectural Styles
Example: Interactive Electronic Technical Manual (IETM) System

Web Services 3-tier architecture

- Data Services
- Business Services
- User Services

IETM Data  

Data Access  

IETM Electronic Display System (EDS)  

User Interface
Recall Analysis diagram for EMS, Context Diag.
Figure 23.8. Integrated communication diagram for Emergency Monitoring System
EMS Deployment Architecture view

Figure 13.5. Example of geographical distribution: Emergency Monitoring System
Example of Hierarchical Architecture: Cruise Control and Monitoring System
Example: Consolidated Collaboration Diagram of the Elevator Control System

Figure 12.4  Example of distributed software architecture: Elevator Control System
Figure 22.25. Service-oriented software architecture for the Online Shopping System
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Step 1: Defining Subsystems
  - Step 2: Defining Subsystem Interfaces
- Design Using Architectural Styles
Information Available At Architectural Design

- The Requirements model
  - Use cases, Use case Diagram, system sequence diagrams

- The Analysis model
  - Analysis class diagram,
  - stateCharts for multi-modal classes, and
  - Domain Object sequence diagrams
Artifacts Developed at Architectural Design

- Subsystems + their public interfaces (APIs)
- Subsystems class diagrams. A class diagram for each subsystem
- Subsystem dependencies (interaction diagrams)

Requirements and Analysis models → Architecture design → Design Class/ and Interaction Diagrams
The Process of Designing Software Architectures

- Software Architecture
  - **Step 1:** Define overall structure of the system into components or subsystems, or classes
  - **Step 2:** Define Component interfaces and interconnections separately from component internals (defined during details design)

- Each subsystem performs major service
  - Contains highly coupled objects
  - Relatively independent of other subsystems
  - May be decomposed further into smaller subsystems
  - Subsystem can be an aggregate or a composite object
Step 1 - Subsystem/Components Structuring Criteria

Decompose the system into subsystems or classes such that each performs a specific function or task to maximize cohesion and minimize coupling, the following are typical examples of subsystems or classes

- **Controllers**
  - Subsystem controls a given aspect of the system (e.g., Cruise cont. Fig. 20.45)

- **Coordinators/Schedulers**
  - Coordinates several control subsystems (e.g., Cruise cont Fig 20.45, 20.46)

- **Data Collectors/Monitors**
  - Collects data from external environment (e.g., Cruise cont Fig. 20.45)

- **Data analyzers**
  - Provides reports and/or displays (e.g., Cruise cont Fig. 20.26)

- **Servers**
  - Provides service for client subsystems (e.g., MyTrip example)

- **User/Device Interface**
  - Collection of objects supporting needs of user (e.g., Cruise cont Fig. 20.26)
Control, Coordinator, Data Collection Subsystems

Figure 12.4  Example of distributed software architecture: Elevator Control System
Figure 22.25. Service-oriented software architecture for the Online Shopping System
Service subsystems, Input & User Interface

Figure 13.9. Examples of service subsystems
Figure 13.10. Example of control and coordinator subsystems in Factory Automation System
Figure 13.11. Example of coordinator subsystem in service-oriented architectures
Another way of forming subsystems

- **Aggregate** into the same subsystem
  - Objects that participate in the same use case (functional cohesion)
  - Objects that have a large volume of interactions (e.g., Control object & objects it controls) or share common data or file structures (communicational cohesion)
  - Object that execute in the same time (temporal cohesion)
Figure 13.8. Examples of user interaction subsystem with multiple windows
Figure 23.8. Integrated communication diagram for Emergency Monitoring System
Aggregate Control, input, and output of each distributed controller
Example: MyTrip System, uses a Global Positioning System to locate and coordinate a trip for a driver in an automobile software system

The Analysis Class Diagram

- RouteAssistant
- Location
- Crossing
- Trip
- Direction
- Destination
- PlanningService
- Segment
Design Class Diagram
MyTrip Subsystems

RoutingSubsystem
  RouteAssistant
    Location

PlanningSubsystem
  Trip
    Direction
      Crossing
      Segment
  PlanningService
    Destination
MyTrip Deployment Diagram

Components must be associated with a processor node in the deployment diagram
New Classes and Subsystems

RoutingSubsystem
- RouteAssistant
- Location
- TripProxy
- SegmentProxy

PlanningSubsystem
- PlanningService
- Trip
- Destination
- Direction
- Crossing
- Segment

CommunicationSubsystem
- Message
- Connection
MyTrip Data Storage

- RoutingSubsystem
- PlanningSubsystem
- CommunicationSubsystem
- TripFileStoreSubsystem
- MapDBStoreSubsystem
Example: Cruise Control and Monitoring System

Figure 12.8 Cruise Control and Monitoring System: major subsystems
Example: Cruise Control And Monitoring System

Class Diagram of the Cruise Control Subsystem
Example: Cruise Control System;
The Monitoring Subsystem

Figure 20.26 Class diagram for Monitoring Subsystem
Example: Aggregating classes into a subsystem using temporal cohesion

Figure 20.45  Detailed software design of Auto Sensors task
Example: aggregating classes
Using functional cohesion

Figure 20.46  Detailed software design of Speed Adjustment task
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Step 1: Defining Subsystems
  - Step 2: Defining Subsystem Interfaces
- Design Using Architectural Styles
Step 2 - Define Subsystem Interfaces

- The set of public operations forms the *subsystem interface* or *Application Programming Interface* (API)
- Includes operations and also their parameters, types, and return values
- Operation *contracts* are also defined (pre- and post-conditions) and accounted for by client subsystems – they can be considered part of the API
Subsystem Interfaces

Interfaces can be methods such as Notify, update, or can be classes such as context.
Figure 4.5
Diagram of the SOA view for the Adventure Builder system. The OPC (Order Processing Center) component coordinates the interaction with internal and external service consumers and providers.
Client-Server Interfaces (Informal Notation)

Figure 4.3
Client-server architecture of an ATM banking system. The ATM main process sends requests to Bank transaction authorizer corresponding to user operations (such as deposit, withdrawal). It also sends messages to ATM monitoring server informing the overall status of the ATM (devices, sensors, and supplies). The Reconfigure and update process component sends requests to ATM reconfiguration server to find out if a reconfiguration command was issued for that particular ATM. Reconfiguration of an ATM (for example, enabling or disabling a menu option) and data updates are issued by bank personnel using the Monitoring station program. Monitoring station program also sends periodic requests to ATM monitoring server to retrieve the status of the...
Client-Server Interfaces (Informal Notation)

Figure 7.1
Graphical notations for interfaces typically show a symbol on the boundary of the icon for an element. Lines connecting interface symbols denote that the interface exists between the connected elements. Graphical notations like this can show only the existence of an interface, not its definition. (a) An element with multiple interfaces. For elements with a single interface, the interface symbol is often omitted. (b) Multiple actors at an interface. Internal client and External client both interact with Transaction Authorizer via the same interface. This interface is provided by Transaction Authorizer and required by both Internal client and External client.
Interfaces in UML Notation

(a) And (b) are equivalent

Figure 7.2
UML uses a lollipop to denote a provided interface, which can be appended to classes, components, and packages. Required interfaces are represented with the socket symbol, which is also appended to classes and other types of elements. UML also allows a class symbol to be stereotyped as an interface; a dashed line with a closed, hollow arrowhead shows that an element realizes an interface. The operations compartment of the class symbol can be annotated with the interface's signature information: methods.
Figure 7.3
An interface can be shown separately from any element that realizes it, thus emphasizing the interchangeability of element implementations. OrderDao (and other classes not shown) require an object that implements a database connection, which is represented by the Connection interface. Many elements realize this interface, representing the interchangeable alternatives of database connection implementations.

(Implement the methods open(), etc.)
Figure 14-160. Interface suppliers and clients
implements the methods in both Interfaces
Example: A Digital Sound Recorder
From Requirements-to-Analysis-to-Design

- The main function of the DSR is to record and playback speech.
- The messages are recorded using a built-in microphone and they are stored in a digital memory.
- The DSR contains an alarm clock with a calendar. The user can set a daily alarm. The alarm beeps until the user presses a key, or after 60 seconds.
Digital Sound Recorder: A Complete Example From Requirements-to-Analysis-to-Design

Figure 2.1: External appearance

Figure 2.3: Use Case diagram
Digital Sound Recorder: A Complete Example

Figure 2.2: Context-Level diagram
Digital Sound Recorder: A Complete Example

System Sequence Diagram

Figure 2.4: Playing message scenario
Digital Sound Recorder: A Complete Example

Figure 2.5: Alarm while playing scenario
Digital Sound Recorder: A Complete Example

Figure 2.6: Entering and exiting stand-by mode scenario
Digital Sound Recorder: A Complete Example

Analysis Class Diagram

Figure 3.2: Sound Recorder class diagram
Analysis Sequence Diagram

Help find operations of classes during design

Figure 3.8: Deleting a message while playing it
Digital Sound Recorder: A Complete Example

Design Class Diagram: Designing The Subsystems,

The names of subsystems should be improved.

Figure 3.3: Subsystems in the sound recorder
Digital Sound Recorder: A Complete Example

Interactions between Objects are defined Using Design Sequence diagrams

Figure 3.4: Audio subsystem class diagram
Digital Sound Recorder: A Complete Example

Figure 3.7: Message memory class diagram
Digital Sound Recorder:
A Complete Example

Figure 3.9: Alarm clock class diagram
Digital Sound Recorder: A Complete Example

Figure 3.11: User interface subsystem class diagram
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Defining Subsystems
  - Defining Subsystem Interfaces
- Design Using Architectural Styles
  - Software Architecture Styles
  - The Attribute Driven Design (ADD)
OUTLINE of SW Architecture Styles

- Introduction
- Software Architecture Styles
  - Independent Components
  - Virtual Machines
  - Data Flow
  - Data-Centered
  - Call-and return
- Other Important Styles
  - Model-View-Controller
  - Broker Architecture Style
  - Service Oriented Architecture (SOA)
  - Peer-to-Peer Architecture
- SW Systems Mix of Architecture Styles
An architectural style is a class of architectures characterized by:

- Components types: are component classes characterized by either SW packaging properties or functional or computational roles within an application.
- Communication patterns between the components: kinds of communications between the component types.
There is a number of families of styles that has been defined and used in many software systems. Notable examples are:

1. Independent Components: Event-based Architectures
2. Virtual Machines
3. Data Flow: Pipes and Filters
4. Data-Centered Systems
5. Call-and Return Architectures
Architectural Styles

Grouped Into Five Families

1. Independent Components. SW system is viewed a set of independent processes or objects or components that communicate through messages.
   
   Two subfamilies:
   
   - Event based systems (implicit and direct invocation style), and
   - Communicating processes family (client-server style).
Architectural styles: Event-based Architecture
Some processes post events, others express an interest in events

The publish and subscribe event-based architectural style.
Event-based Architecture
Implicit Invocation: *The Observer Pattern* (to be discussed later)
Events at Different Levels of Abstraction

- **Event Target**
  - Pixel
  - Character
  - Window
  - Text field
  - Article window
  - Article moved
  - News article list
  - Article dropped
  - Article
  - News feed
  - Message created
  - News arrived

- **Events**
  - Key pressed on keyboard
  - Mouse moved
  - Menu item selected

- **Physical I/O Objects**
  - Actions
  - Events
  - User

- **Logical GUI Objects**
  - Actions
  - Events

- **Application Objects**
  - Actions
  - Events

- **Database Objects**
  - Actions
  - Events

- **Representation by System**
Example: GUI Event Processing

- **Event**: “Button” “double-clicked” “17:31:22”
- **EventSource**: Button managed by the GUI subsystem of the operating system
- **EventHandler**: Notification method in the application code
- **EventManager**: Operating system or GUI library code
OUTLINE of SW Architecture Styles

• Introduction

Software Architecture Styles
  • Independent Components
  • **Virtual Machines**
    • Data Flow
    • Data-Centered
    • Call-and return
  • Other Important Styles
    • Buffered Massage-Based
    • Model-View-Controller
    • Presentation-Abstraction-Control
    • Broker Architecture Style
    • Service Oriented Architecture (SOA)
    • Peer-to-Peer Architecture
  • SW Systems Mix of Architecture Styles
2. Virtual Machines. Originated from the concept that programs are treated as data by a virtual machine, which is an abstract machine implemented entirely in software, that runs on top of the actual hardware machine.
Java Virtual Machine. Java code translated to platform independent bytecodes. JVM is platform specific and interprets the bytecodes.
Virtual Machines: The primary benefits are the separation between instruction and implementation, (Used when inputs are defined by a script or Commands, and data)
OUTLINE of SW Architecture Styles

- Introduction

Software Architecture Styles
  - Independent Components
  - Virtual Machines
  - **Data Flow**
    - Data-Centered
    - Call-and return
  - Other Important Styles
    - Buffered Massage-Based
    - Model-View-Controller
    - Presentation-Abstraction-Control
    - Broker Architecture Style
    - Service Oriented Architecture (SOA)
    - Peer-to-Peer Architecture
  - SW Systems Mix of Architecture Styles
3. Data Flow. Include batch sequential systems (BSS) and pipes and filters (PF).

- BSS: different components take turns at processing a batch of data, each saving the result of their processing in a shared repository that the next component can access. Ex. Dynamic control of physical processes based on a feedback loop.

- PF: A stream of data processed by a complex structure of processes (filters). Ex, UNIX.
Architectural Styles: Data Flow

Control Loop
BSS

The pipes-and-filters architectural style.
Example: P＆F Compiler Architecture (1)

- Sources & Sinks, Input & Output Streams
- Flexible composability
- Aggregation / Decomposition of Filters

Diagram:
- Unicode Character Stream
- Token Stream
- Abstract Syntax Tree Nodes
- Decorated Abstract Syntax Tree Nodes
- Machine Code Generator
- Machine Code Stream
- Bytecode Optimizer
- Bytecode Stream
- Error Message Stream
PF Another Architecture Example:
Watch for the Two Views
OUTLINE of SW Architecture Styles

• Introduction

Software Architecture Styles
  • Independent Components
  • Virtual Machines
  • Data Flow
  • **Data-Centered**
    • Call-and return
  • Other Important Styles
    • Buffered Massage-Based
    • Model-View-Controller
    • Presentation-Abstraction-Control
    • Broker Architecture Style
    • Service Oriented Architecture (SOA)
    • Peer-to-Peer Architecture
  • SW Systems Mix of Architecture Styles
4. Data-Centered Systems. Consist of having different components communicate through shared data repositories. When data repository is an active repository that notifies registered components of changes in it then-blackboard style.
Data-Centered Architectural Styles
Repository Architecture Style

Figure 6.5
Database system
Data-Centered Architectural Styles

Repository Architecture Example: CASE Tools Example
Data-Centered Architectural Styles

Repository Architecture Example: Compiler Architecture

```
int x, y;
x = y + 1;
```

```
x  int  0
y  int  0

Var (Int) = exp (Int)

x (Int)
y (Int) + 1 (Int)

Symbol table

Parse tree

Compiler tool

Scanner

Parser

Code generator

mov ax, [y]
add ax, 1
mov [x], ax

[x] address of variable x
In symbol table

Type check (by semantic parser)
Data-Centered Systems: Central data repository
Components perusing shared data, and communicating through it.
Used in Database intensive systems
Data-Centered Architectural Styles

Blackboard Architecture Style Example

Compare with the PFs Style

Figure 4-16. Lunar Lander in blackboard style.

Figure 4-15. Lunar Lander in pipe-and-filter style.
Data-Centered Architectural Styles

Blackboard Architecture Style:

Intelligent Agent Systems Example

Figure 6.8
Blackboard architecture
Data-Centered Architectural Styles
Blackboard Architecture Style: Travel Counseling System Example

Figure 6.12
Blackboard architecture for a travel consulting system
OUTLINE of SW Architecture Styles

• Introduction

Software Architecture Styles
  • Independent Components
  • Virtual Machines
  • Data Flow
  • Data-Centered
  • **Call-and return**

• Other Important Styles
  • Model-View-Controller
  • Broker Architecture Style
  • Service Oriented Architecture (SOA)
  • Peer-to-Peer Architecture

• SW Systems Mix of Architecture Styles
Architectural styles

5. Call-and Return Architectures. Due to their simple control paradigm and component interaction mechanism, these architectures have dominated the SW landscape by the early decades of the SW Eng.

There are several styles within this family: examples are

1) Main program and subroutine,
2) Layered architectures.

- Main Program and Subroutine Style. Programs are modularized based on functional decomposition, single thread of control held by the main program, which is then passed to subprograms, along with some data on which the subprograms can operate.
Main Program and Subroutine Style

Course registration System example

Main component
Architectural styles

- Layered. Functionality is divided into layers of abstraction—each layer provides services to the layer(s) above it, and uses the services of layer(s) below it. In its purest form, each layer access only the layer below it, but does not depend on other lower layers.
Layered Architectural styles
Example of a Layered Application Architecture
OUTLINE

• Introduction

• Software Architecture Styles
  • Independent Components
  • Virtual Machines
  • Data Flow
  • Data-Centered
  • Call-and return

• Other Important Styles
  • Model-View-Controller
  • Broker Architecture Style
  • Service Oriented Architecture (SOA)
  • Peer-to-Peer Architecture
Model-View-Controller Architecture Style

- The Controller manipulates the data Model
- The View retrieves data from the model and displays needed information
Model-View-Controller Architecture Style
Dynamic Interactions

Figure 24: Sequence diagram for MVC architecture
Model-View-Controller Architecture Style
Web Applications Java-based Implementation Example

JavaServer Pages (JSP) lets you separate the dynamic part of your pages from the static HTML.
OUTLINE

• Introduction

• Software Architecture Styles
  • Independent Components
  • Virtual Machines
  • Data Flow
  • Data-Centered
  • Call-and return

• Other Important Styles
  • Model-View-Controller
  • Broker Architecture Style
  • Service Oriented Architecture (SOA)
  • Peer-to-Peer Architecture
Brokers gets requests from client proxies and manages them by forwarding to server Proxies or dispatches them to other connected brokers.
Broker Architecture Style

**Client-side proxy**
- +pack_data()
- +unpack_data()
- +send_request()
- +return()

Uses API

**Client**
- +call_server()
- +start_task()
- +use_Broker_API()

**Broker 1**
- +main_event_loop()
- +update_repository()
- +register_service()
- +acknowledgement()
- +find_server()
- +forward_request()
- +forward_response()

Calls

**Bridge 1**
- +pack_data()
- +unpack_data()
- +forward_message()
- +transmit_message()

**Bridge 2**

**Server**
- +initialize()
- +enter_main_loop()
- +run_service()
- +use_Broker_API()

**Client-side proxy**
- +pack_data()
- +unpack_data()
- +call_service()
- +send_response()
Broker Architecture Style

Diagram:
- **Client**
  - callServer()
- **Client-side proxy**
  - sendRequest()
  - packData()
- **Client-side**
  - forwardRequest()
- **Broker**
  - findServer()
  - callService()
  - unpackData()
  - forwardResponse()
  - findClient()
  - packData()
- **Server-side proxy**
  - register service()
  - update repository()
  - acknowledgment
- **Server**
  - runService()
Broker Architecture Style

Advantages:

- Server component implementation and location transparency
- Changeability and extensibility
- Simplicity for clients to access server and server portability
The Object Request Broker (ORB) protocol provides a software bus on the network for brokering the requests from clients and the responses from servers; the protocol also supports increased interoperability with other implementations.
CORBA also supports the Dynamic Invocation Interface (DII), which allows CORBA clients to use another CORBA object without knowing its interface information until runtime. Dynamic Skeleton Interface (DSI) is used by ORB to issue requests to objects that are implemented independently and for which the ORB has no compile-time knowledge of their implementation. Although the dynamic approach of DII and DSI is more flexible, they are always slower than their static IDL counterpart. The dynamic remote invocation mode was the only invocation mode available in the early version of CORBA. In some cases the IDL is not available at compilation time and the stub and skeleton cannot be generated at compilation time. For example, if a COM client wants to make a CORBA request or a DCOM object wants to provide its services on CORBA, a bridge interface is required to do the conversion. In the following...
OUTLINE

• Introduction

• Software Architecture Styles
  • Independent Components
  • Virtual Machines
  • Data Flow
  • Data-Centered
  • Call-and return

• Other Important Styles
  • Model-View-Controller
  • Broker Architecture Style
  • Service Oriented Architecture (SOA)
  • Peer-to-Peer Architecture
Service Oriented Architecture (SOA) Style

Makes use of an Enterprise Service Bus (ESB)

Used in web-based systems and distributed computing

Before SOA

The SOA Style

nodes make resources available to other participants in the system as independent services that the participants access in a standardized way using the ESB.
The SP publishes/updates services using the Web Service Description Language (WSDL) on the Universal Description Discovery and Integration (UDDI) registry.

**FIGURE 5.2**
A simple web service interaction among provider, user, and the UDDI registry.
The ESB Performs:
- data transformation
- Intelligent routing
- Real time monitoring
- Exception handling
- Service security
Cloud Services Architecture
SOA supports Cloud Computing Models

The Grid of Services and Resources

Clients request the Grid Services and Resources from the Service Directory

Grid = Service + Resource

Grid services scheduling

Virtualized resource

CPU resources

Database resources

Application resources

Physical resources

Disks

Printers
Cloud Services Architecture

Human as a service, Software as a service, Infrastructure as a service

Huaas  Saas  IaaS

- Cloud Services Architecture
- Human as a service, Software as a service, Infrastructure as a service
- Huaas  Saas  IaaS

1. **Cloud Services Architecture**
   - Human as a service
   - Software as a service
   - Infrastructure as a service

2. **Huaas**
   - Crowdsourcing
   - Information Aggregation Services
     - e.g., Mechanical Turk
     - Iowa Electronic Markets

3. **Saas**
   - Applications
     - e.g., Google Docs
   - Application Services
     - Composite Application Services
       - e.g., Opensocial
     - Basic Application Services
       - e.g., OpenID
   - Programming Environment
     - e.g., Django
   - Execution Environment
     - e.g., Google App Engine

4. **IaaS**
   - Infrastructure Services
     - Higher Infrastructure Services
       - e.g., Google Bigtable
     - Basic Infrastructure Services
       - Computational
         - e.g., Hadoop MapReduce
         - e.g., GoogleFS
       - Storage
       - Network
     - e.g., OpenFlow
   - Resource Set
     - Virtual Resource Set
       - e.g., Amazon EC2
     - Physical Resource Set
       - e.g., Emulab

5. **Business Support**
   - Administration
   - Deployment, Configuration, Monitoring, Life Cycle Management
   - Metering, Billing, Authentication, User Management

6. **Hardware**
The Internet of Things (IoT)

The architecture of an IoT consisting of sensing devices that are connected to various applications via mobile networks, the Internet, and processing clouds.
An example of how measured data can be transferred to doctors or medical professionals using a wireless sensor network.
Technology road map of the Internet of things.
• **Introduction**

• **Software Architecture Styles**
  • Independent Components
  • Virtual Machines
  • Data Flow
  • Data-Centered
  • Call-and return

• **Other Important Styles**
  • Model-View-Controller
  • Broker Architecture Style
  • Service Oriented Architecture (SOA)
  • Peer-to-Peer Architecture
Hybrid Client-Server/Peer-to-Peer: Napster

P2P systems became part of the popular technical parlance due in large measure to the popularity of the original Napster system that appeared in 1999. Napster was designed to facilitate the sharing of digital recordings in the form of MP3 files. Napster was not, however, a true P2P system. Its design choices, however, are instructive.

Figure 11-4. Notional view of the operation of Napster. In steps 1 and 2, Peers A and B log in with the server. In step 3, Peer A queries the server where it can find Rondo Veneziano’s “Masquerade.” The location of Peer B is returned to A (step 4). In step 5, A asks B for the song, which is then transferred to A (step 6).
Peer-to-Peer Architecture Style
The Gnutella Example

- Pure Peer-to-Peer Architecture
- A sends query for a data resource to neighbors B and H, they pass it on until the peer having the resource is found or until a certain threshold of hops is reached.
Recent Versions of Gnutella supports two types of peers: Ultra peers and Leaf peers. Ultra peers run in systems with fast internet connections and are responsible for request routing and responses. They are connected to a large number of other Ultra peers and Leaf peers, while the Leaf peers are connected to a small number of Ultra peers.
Peer-to-Peer Architecture Style

The Skype Example

- A mixed client-Server and Peer-to-Peer
- Skype Peers get promoted to a supernode status based on their network connectivity and machine performance
- Supernodes perform the communication and routing of messages to establish a call
- When a user logs in to the server he is connected to a supernode
- If a peer becomes a supernode he unknowingly bears the cost of routing a potentially large number of calls.
Several aspects of this architecture are noteworthy:

- A mixed client-server and peer-to-peer architecture addresses the discovery problem. The network is not flooded with requests in attempts to locate a buddy, such as would happen with the original Gnutella.
- Replication and distribution of the directories, in the form of supernodes, addresses the scalability and robustness problems encountered in Napster.
Conclusions

• An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements.
• Choosing a style to implement a particular system depends on several factors based on stakeholders concerns and quality attributes.
• Most SW systems use a mix of architecture styles.
Most SW systems use a mix of architecture styles. Ex, personnel management system with a scheduling component, implemented using the independent component style, and a payroll component, using the batch sequential style.

Choosing a style to implement a particular system depends on several factors. The technical factors concern the level of quality attributes that each style enables us to attain. Ex, event-based systems-achieve very high level of evolvability, at the expense of performance and complexity. Virtual-machine style-achieve very high level of portability, at expense of performance and perhaps even testability.
SW Systems - Mix of Architecture Styles
Components of each Layer use different architecture styles

Figure 11-3. Architecture of Globus Grid technology (recovered). (Mattmann, Medvidović et al., 2005).
Figure 11-12. The MIDAS wireless sensor network architecture. Diagram adapted from (Malek et al. 2007) © IEEE 2007.
Outline

- UML Development – Overview
- The Requirements, Analysis, and Design Models
- What is Software Architecture?
  - Software Architecture Elements
- Examples
- The Process of Designing Software Architectures
  - Defining Subsystems
  - Defining Subsystem Interfaces
- Design Using Architectural Styles
  - Software Architecture Styles
  - The Attribute Driven Design (ADD)
One method of designing an architecture to achieve quality and functional needs is called Attribute Driven Design (ADD).

- In ADD, architecture design is developed by taking sets of quality attribute scenario inputs and using knowledge of relationship between quality attributes and architecture styles.

  - [http://www.sei.cmu.edu/architecture/tools/define/add.cfm](http://www.sei.cmu.edu/architecture/tools/define/add.cfm)
  - [http://www.sei.cmu.edu/reports/07tr005.pdf](http://www.sei.cmu.edu/reports/07tr005.pdf)
Attribute-Driven Design (ADD)

A Method for producing software architecture based on process decomposition, stepwise refinement and fulfillment of attribute qualities.

It is a recursive process where at each repetition, tactics and an architecture style or a pattern is chosen to fulfill quality attribute needs.
Attribute-Driven Design (ADD): Overview

Figure 1: The ADD Plan, Do, and Check Cycle
Design the Software Architecture Using the Attribute-Driven Design (ADD) Method

**Purpose:** The Attribute-Driven Design (ADD) Method is an approach to defining software architectures by basing the design process on the architecture’s quality attribute requirements. It follows a recursive decomposition process where, at each stage in the decomposition, architectural tactics and patterns are chosen to satisfy a set of quality attribute scenarios.

**Role:** Software architect [Software architect]

**Frequency:** This activity is optional in the Inception Phase. It should occur in the first iteration of the Elaboration Phase and can recur in later iterations if substantial changes or additions to the software architecture need to be explored.

**Steps:**
1. Choose the module to decompose.
2. Refine the module according to these steps:
   a. Choose the architectural drivers.
   b. Choose an architectural pattern that satisfies the architectural drivers.
   c. Instantiate modules and allocate functionality from the use cases. Represent the results using multiple views.
   d. Define interfaces of the child modules.
   e. Verify and refine the use cases and quality scenarios and make them constraints for the child modules.
3. Repeat the above steps for the next module.

**Input Artifacts:**
- vision [constraints]
- architectural proof-of-concept [constraints]
- use case model [functional requirements, quality requirements]
- supplementary specifications [quality requirements]

**Resulting Artifacts:**
- software architecture document [decomposition of the architecture expressed in module, concurrency, and deployment views]

**Tool Mentors:** None

**More Information:** Bass 03

**Workflow Details:**
- Analysis and Design
  - Define a Candidate Architecture
  - Perform Architectural Synthesis

---

*Figure 9: The ADD Method as a RUP Activity*
Updated ADD Steps

Step 1: Confirm There Is Sufficient Requirements Information

WHAT DOES STEP 1 INVOLVE?

1. Make sure that the system’s stakeholders have prioritized the requirements according to business and mission goals.
2. You should also confirm that there is sufficient information about the quality attribute requirements to proceed.
In this second step, you choose which element of the system will be the design focus in subsequent steps. You can arrive at this step in one of two ways:

1. You reach Step 2 for the first time. The only element you can decompose is the system itself. By default, all requirements are assigned to that system.

2. You are refining a partially designed system and have visited Step 2 before. In this case, the system has been partitioned into two or more elements, and requirements have been assigned to those elements. You must choose one of these elements as the focus of subsequent steps.
Step 3: Identify Candidate Architectural Drivers

WHAT DOES STEP 3 INVOLVE?
At this point, you have chosen an element of the system to decompose, and stakeholders have prioritized any requirements that affect that element.
During this step, you’ll rank these same requirements a second time based on their relative impact on the architecture.
This second ranking can be as simple as assigning “high impact,” “medium impact,” or “low impact” to each requirement.
Given that the stakeholders ranked the requirements initially, the second ranking based on architecture impact has the effect of partially ordering the requirements into a number of groups. If you use simple high/medium/low rankings, the groups would be (H,H) (H,M) (H,L) (M,H) (M,M) (M,L) (L,H) (L,M) (L,L)

The first letter in each group indicates the importance of requirements to stakeholders, the second letter in each group indicates the potential impact of requirements on the architecture.
From these pairs, you should choose several (five or six) high-priority requirements as the focus for subsequent steps in the design process.
Step 4: Choose a Design Concept that Satisfies the Architectural Drivers

In Step 4, you should choose the major types of elements that will appear in the architecture and the types of relationships among them. Design constraints and quality attribute requirements (which are candidate architectural drivers) are used to determine the types of elements, relationships, and their interactions. The process uses architecture patterns or styles.
Choose architecture patterns or styles that together come closest to satisfying the architectural drivers

Table 1: Structure of Matrix to Evaluate Candidate Patterns

<table>
<thead>
<tr>
<th>Architectural driver 1</th>
<th>Pattern 1</th>
<th>Pattern 2</th>
<th>...</th>
<th>Pattern n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pros</td>
<td>Cons</td>
<td></td>
<td>Pros</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Cons</td>
</tr>
<tr>
<td>Architectural driver 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architectural driver n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Step 4: Example

Mobile Robots example (to be discussed at the end)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Control Loop</th>
<th>Layers</th>
<th>Blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drivers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task coordination</td>
<td>+-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Dealing with uncertainty</td>
<td>-</td>
<td>+-</td>
<td>+</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>+-</td>
<td>+-</td>
<td>+-</td>
</tr>
<tr>
<td>Safety</td>
<td>+-</td>
<td>+-</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>+-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
Step 4: Major Design Decisions

- Decide on an overall design concept that includes the major types of elements that will appear in the architecture and the types of relationships among them.
- Identify some of the functionality associated with the different types of elements.
- Decide on the nature and type of communications (synchronous/asynchronous) among the various types of elements (both internal software elements and external entities).
Step 5: Instantiate Architectural Elements and Allocate Responsibilities

- In Step 5, you instantiate the various types of software elements you chose in the previous step. Instantiated elements are assigned responsibilities from the functional requirements (captured in use-cases) according to their types.

- At the end of Step 5, every functional requirement (in every use-case) associated with the parent element must be represented by a sequence of responsibilities within the child elements.

- This exercise might reveal new responsibilities (e.g., resource management). In addition, you might discover new element types and wish to create new instances of them.
EXAMPLE: SATELLITE CONTROL SYSTEM

Use-Case Diagram
A Simple Example of Software Architecture Using UML2

SATELLITE CONTROL SYSTEM Architecture composition

[Diagram of SATELLITE CONTROL SYSTEM architecture with classes and relationships labeled]
Step 6: Define Interfaces for Instantiated Elements

WHAT DOES STEP 6 INVOLVE?

- In step 6, you define the services and properties required and provided by the software elements in our design. In ADD, these services and properties are referred to as the element’s interface.

- Interfaces describe the *PROVIDES* and *REQUIRES* assumptions that software elements make about one another.
A Simple Example of Software Architecture Using UML2

SATELLITE CONTROL SYSTEM Architecture Structure
A Simple Example of Software Architecture Using UML2

SATELLITE CONTROL SYSTEM Architectural Behavior

interaction ProcessPositionMessage

<<actor>> 'Ground Control'

cc : CommunicationsController

db : DataBus

ac : AttitudeController

requiredPositionMessage(AA, DD, 10324, 10, 21, 18)

checkOk = messageVerified(10324, AA);

alt [ !checkOk ]

nak ()

[checkOk]

ack ()

requiredPositionToBus(10, 21, 18)

requiredPositionFromBus(10, 21, 18)

ac.storePosition(10, 21, 18);

Idle

Actuating

WaitForSensorData
Step 6: Major Design Decisions

Decisions will likely involve several of the following:

- The external interfaces to the system
- The interfaces between high-level system partitions, or subsystems
- The interfaces between applications within high-level system partitions
- The interfaces to the infrastructure (reusable components or elements, middleware, run-time environment, etc.)
Step 7: Verify and Refine Requirements and Make Them Constraints for Instantiated Elements

WHAT DOES STEP 7 INVOLVE?

- In Step 7, you verify that the element decomposition thus far meets functional requirements, quality attribute requirements, and design constraints. You also prepare child elements for further decomposition.

- Refine quality attribute requirements for individual child elements as necessary (e.g., child elements that must have fault-tolerance, high performance, high security, etc.)
Example 1 Mobile Robotics System

Overview

– controls manned, partially-manned, or unmanned vehicle--car, submarine, space vehicle, etc.
– System performs tasks that involve planning and dealing with obstacles and other external factors.
– System has sensors and actuators and real-time performance constraints.
Mobile Robotics System Requirements
(Candidate Architecture Drivers)

Req 1: System must provide both 
*deliberative* and *reactive* behavior.

Req 2: System must deal with *uncertainty*.

Req. 3 System must deal with *dangers* in robot’s operation and environment.

Req. 4: System must be *flexible* with respect to experimentation and reconfiguration of robot and modification of tasks.
Choose an architecture style
Evaluate Control Loop Architecture--Strengths and Weaknesses w.r.t candidate architecture drivers

• Req 1--deliberative and reactive behavior
  – advantage-simplicity
  – drawback-dealing with unpredictability
    • feedback loops assumes continuous changes in environment and continuous reaction
    • robots are often confronted with disparate, discrete events that require very different modes of reactive behavior.
  – drawback-architecture provides no leverage for decomposing complex tasks into cooperating components.
Control Loop Architecture

Control Loop Architecture--Continued

• Req 2--dealing with uncertainty
  – disadvantage-biased toward one way of dealing with uncertainty, namely iteration via closed loop feedback.

• Req 3--safety and fault-tolerance
  – advantage-simplicity
  – advantage-easy to implement duplication (redundancy).
  – disadvantage-reaction to sudden, discrete events.

• Req 4--flexibility
  – drawback--architecture does not exhibit a modular component structure

• Overall Assessment: architecture may be appropriate for
  – simple systems
  – small number of external events
  – tasks that do not require complex decomposition,
Choose another architecture style

Mobile Robots—Layered Architecture

- Supervisor
- Global planning
- Control
- Navigation
- Real-world modeling
- Sensor integration
- Sensor interpretation
- Robot control

Environment
Layered Architecture

Evaluate Layered Architecture--Strengths and Weaknesses

• Req 1--deliberative and reactive behavior
  – advantage-architecture defines clear abstraction levels to guide design
  – drawback-architectural structure does not reflect actual data and control-flow patterns
  – drawback-data hierarchy and control hierarchy are not separated.
Layered Architecture

Layered Architecture--Continued

• Req 2--dealing with uncertainty
  – advantage-layers of abstraction should provide a good basis for resolving uncertainties.

• Req 3--safety and fault-tolerance
  – advantage-layers of abstraction should also help (security and fault-tolerance elements in each layer)
  – drawback-emergency behavior may require short-circuiting of strict layering for faster recovery when failures occur.
Layered Architecture--Continued

• Req 4--flexibility
  – drawback-changes to configuration and/or behavior may involve several or all layers

• Overall Assessment
  – layered model is useful for understanding and organizing system functionality
  – strict layered architecture may break down with respect to implementation and flexibility.
Blackboard Architecture

Mobile Robotics--Blackboard Architecture

Lookout  Captain  Navigator  Pilot

Blackboard

Perception subsystem
Blackboard Architecture

Evaluate Blackboard Architecture--Strengths and Weaknesses

• Req1-- Deliberative and reactive behavior
  – advantage: Easy to integrate disparate, autonomous subsystems
  – drawback: blackboard may be cumbersome in circumstances where direct interaction among components would be more natural.

• Req 2--Dealing with uncertainty
  – advantage: blackboard is well-suited for resolving conflicts and uncertainties.
Blackboard Architecture

Blackboard Strengths and Weaknesses--Continued

- Req3--safety and fault-tolerance
  - advantage: subsystems can monitor blackboard for potential trouble conditions
  - disadvantage: blackboard is critical resource (this can be addressed using a back up)

- Req4--flexibility
  - advantage: blackboard is inherently flexible since subsystems retain autonomy.
## Architecture Comparison

### Mobile Robotics--Summary of Architectural Tradeoffs

<table>
<thead>
<tr>
<th></th>
<th>Control Loop</th>
<th>Layers</th>
<th>Blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task coordination</td>
<td>+-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Dealing with uncertainty</td>
<td>-</td>
<td>+-</td>
<td>+</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>+-</td>
<td>+-</td>
<td>+-</td>
</tr>
<tr>
<td>Safety</td>
<td>+-</td>
<td>+-</td>
<td>+</td>
</tr>
<tr>
<td>Performance</td>
<td>+-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Flexibility</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>