UNIT-III

SOA & WEB SERVICE FOR INTEGRATION (P1)

SOA is a way of designing s/w where,

Different parts of a system like applications (or)

Services, can communicate and work together.

Here each services does a specific job, you can combine them to create something bigger & more powerful.

In Business Context,

It means breaking down tasks into smaller services.

EX: ECOMMERCE WEBSITE

You might have services for

Product Information, Payment Processing, Order Tracking

Each of these services can work independently & talk to each other when needed.

Web Services:

Like Messengers that allow different s/w applications to communicate with each other over the internet.

(OR)

A web service is a software system designed to support machine-to-machine interaction over a network. It allows different applications from various sources to communicate with each other using standardized protocols and data formats.

Standard Protocols: They often use protocols like:

- **SOAP** (Simple Object Access Protocol): A protocol for exchanging structured information.
- **REST** (Representational State Transfer): An architectural style that uses standard HTTP methods.

Data Formats: Commonly use XML or JSON to format the data exchanged between services.

EX:

INTEROPERABILITY

1. You have 2 computer programs that need to talk to each other (but they're written in different languages)
   Here "WS" acts as Translators allowing them to understand each other's requests & responses.

INTEGRATION

2. You have weather app on your phone. It needs to get weather data from server. The app sends a request to a web service on that server asking for the current weather.

   The web Service processes the request & sends back the weather information in a format that the app can understand.

HOW INTEGRATION & INTEROPERABILITY

WORK TOGETHER..?

ONLINE STORE: Needs to get product information from an inventory system.

The online store sends request to the inventory system using a web service (XML) so both can understand it.

INTEGRATION USING SOA & WEB SERVICES:

Now let's join together.

EX:

You have a product catalog service (SOA) that keeps track, all your products & their details.

&

You also have a payment service (SOA) that handles payments.

When a customer makes a purchase, your website (front-end) can send a request to the product catalog service asking for product information.

(This request is sent using a web service)

The product catalog service responds with product details in a format.

(Website understands)

Once the customer chooses a product, your website can use another web service to communicate with the payment services. It sends a request to process the payment.

(The Payment service process the request & sends back responses confirming whether the payment is successful (or) not.)

OVERVIEW OF INTEGRATION

Early days, Information Management (IT) didn't have any integration problems.

→ Because of lesser diverse software ecosystems

No Huge, Fewer 3rd party applications, Limited number of integration points.

MAINLY,

Limited Interconnectedness:

- Networks are not as extensive (or) Interconnected as they are today.
- Many organizations operated in Standalone Environments.

Limited Technology Stack:

- Early IT systems have limited set of technologies they were relatively isolated (separate) from each other.
- Simpler with few components & standardized protocols.

Later somebody wrote the "2nd Business Application"

Integration has been (IT's dirty little secret ever since)

They didn't properly think about the challenges (or) may not receive sufficient attention on integration.

After that based on International Data Corporation (IDC) estimates in 2005 companies will spend more than $15 billion for **ENTERPRISE INTEGRATION SOFTWARE.**

(2ND BUSINESS APPLICATION)

ENTERPRISE INTEGRATION S/W:

Enterprise Integration Software (EIS) refers to tools and solutions designed to facilitate the integration of different systems, applications, and data sources within an organization. These solutions help ensure that various software applications can communicate and share data efficiently, enabling streamlined business processes and improved data consistency.

Key Features:

1. **Data Transformation**: Convert data formats between different systems to ensure compatibility.
2. **Message Routing**: Direct messages between systems based on defined rules and conditions.
3. **Workflow Automation**: Automate business processes by connecting disparate systems.
4. **Real-time Integration**: Enable real-time data exchange between applications, enhancing responsiveness.
5. **Scalability**: Support integration across multiple systems as an organization grows.
6. **Monitoring and Management**: Provide tools to monitor integration flows, track performance, and manage errors.

Common Types of Enterprise Integration Software:

1. **Enterprise Service Bus (ESB)**: A middleware tool that facilitates communication between different services and applications.
2. **API Management Platforms**: Tools that help create, manage, and secure APIs for application integration.
3. **Data Integration Tools**: Solutions that focus on integrating data from different sources, such as ETL (Extract, Transform, Load) tools.
4. **Business Process Management (BPM) Tools**: Platforms that help model, execute, and monitor business processes across integrated systems.

Popular Examples:

- **MuleSoft**: Provides an integration platform for connecting applications, data, and devices.
- **Apache Camel**: An open-source integration framework that helps integrate various systems using routing and mediation rules.
- **IBM App Connect**: Offers tools for integrating applications and automating workflows.

How & why web services will change integration

We need to review some of the Business Drivers & Technical Factors that make integration such a hard problem in the 1st place.

COMMON BUSINESS DRIVERS FOR INTEGRATION

Shape & Guide the strategic decisions & activity of a business

- **Mergers & Acquistions (Target/Acquire):**

  M: Occurs when 2 companies agree to combine their operations & form a new single entity

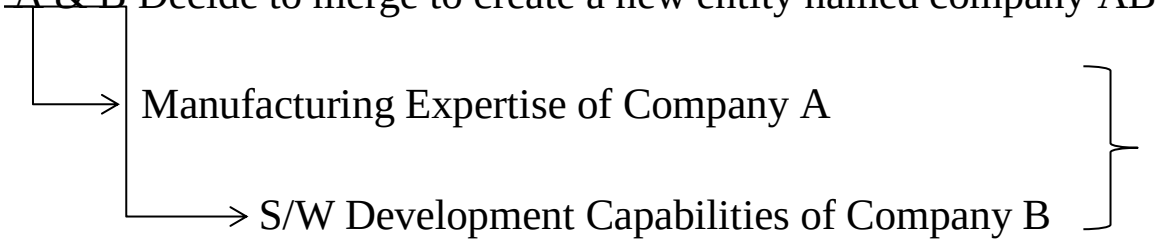
  A: Purchases another company (Includes Assets, Shares)

Merger Scenario:

COMPANY A: A leading manufacturer of smartphones.

COMPANY B: A well established s/w development company specializing in mobile applications.

A & B Decide to merge to create a new entity named company AB.



Acquisition Scenario:

COMPANY X: A Global E-Commerce giant

COMPANY Y: A Successful logistics (Know Customer Demands) & delivery services company.

X Aims to strengthen its logistics & delivery network decides to acquire Y.

- **Internal Reorganization:** Reconstructing (or) Rearranging the internal structure, operations (or) functions.
- **Application/System Consolidation:** Process of combining (or) Centralizing multiple independent systems (or) components into a single unified system.

  EX: Handling similar transactions by multiple IT Systems.

- **Inconsistent/Duplicated/Fragmented data**: Important business data is spread across many systems & must be consolidated (Merged) & cleansed (Remove duplicates & invalid data).
- **New Business Strategies:** Need to adapt to evolving circumstances, stay competitive & capitalize on emerging opportunities. (i.e, Possibiities (or) situations that present the potential for growth & positive developments.)

COMMON TECHNICAL CHALLENGES FACED DURING INTEGRATION:

1. Reconcile/Recheck Incompatible (inefficient) business process implemented by different systems.
2. Reconcile/Match the different between the data used by different systems.
3. Reconcile the inefficient technologies used to implement the different systems.

REQUIREMENTS THAT THE "IDEAL" INTEGRATION SOLUTION MUST SATISFY:

1. Inexpensive
2. Easy to learn & easy to administer/supervise/manage
3. Non Invasive: Existing system should remain untouched i.e, Decision to maintain the current state of a system without making changes (or) alterations.
4. Scalable, Reliable, Highly Available, Fault tolerance, Secure.
5. Flexible & easily customized (so it can be adapted to each project requirements).

INTEGRATION & INTEROPERABILITY USING

XML & WEB SERVICE

Introduction

Integration and interoperability are vital for modern applications, particularly in distributed environments. XML (eXtensible Markup Language) and web services play crucial roles in facilitating seamless communication between heterogeneous systems.

XML Overview

- **Definition**: XML is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
- **Structure**: XML uses a hierarchical structure with tags, allowing it to represent complex data structures clearly. This structure supports various data types and relationships, making it versatile for different applications.
- **Interoperability**: Due to its platform-independent nature, XML enables different systems, regardless of their underlying technology, to share data effectively.

Web Services Overview

- **Definition**: Web services are standardized protocols that allow different applications to communicate over the internet. They utilize XML (in the case of SOAP) and other formats (like JSON in REST) to exchange data.
- **Types of Web Services**:
  - **SOAP (Simple Object Access Protocol)**: A protocol that defines a standard communication format using XML. It includes specifications for message format, encoding rules, and conventions for procedure calls.
  - **REST (Representational State Transfer)**: An architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) and can handle multiple formats, including XML and JSON. It is often simpler and more flexible than SOAP.

Example 1: SOAP Web Service Integration

Scenario: Hotel Booking System

Objective: Integrate a hotel booking system with a payment processing system using a SOAP web service.

Components

1. **Hotel Booking System (HBS)**: Manages hotel reservations.

2. **Payment Processing System (PPS)**: Handles payment transactions.

1. XML Structure for Booking Submission

When a customer makes a hotel reservation, the HBS constructs a SOAP message to send to the PPS.

SOAP Request:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header/>
  <soap:Body>
    <ProcessPayment xmlns="http://www.example.com/pps">
      <Booking>
        <BookingID>78910</BookingID>
        <Customer>
          <Name>Emily Johnson</Name>
          <Email>emily.johnson@example.com</Email>
        </Customer>
        <Amount>200.00</Amount>
      </Booking>
    </ProcessPayment>
  </soap:Body>
```

</soap:Envelope>

2. Sending the SOAP Request

Endpoint: https://api.example.com/pps/payment

HTTP Request:

POST /pps/payment HTTP/1.1
Host: api.example.com
Content-Type: text/xml; charset=utf-8
Content-Length: [length]

3. PPS Processing the SOAP Request

Upon receiving the SOAP message, the PPS:

- Parses the XML to extract booking and payment details.
- Validates the payment information.
- Processes the payment and updates its records.

4. Sending the SOAP Response

After processing the payment, the PPS sends a response back to the HBS.

SOAP Response:

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ProcessPaymentResponse xmlns="http://www.example.com/pps">
      <Response>
        <Status>Success</Status>
        <TransactionID>TRANS123456</TransactionID>
        <Message>Payment processed successfully.</Message>
      </Response>
    </ProcessPaymentResponse>
  </soap:Body>
</soap:Envelope>
```

Example 2: REST Web Service Integration

Scenario: Hotel Booking System

Objective: Integrate the same hotel booking system with an inventory management system for room availability using a RESTful web service.

Components

1. **Hotel Booking System (HBS)**: Manages hotel reservations.
2. **Inventory Management System (IMS)**: Tracks room availability.

1. JSON Structure for Room Availability Update

After successfully processing a booking, the HBS sends a request to update room availability.

JSON Example:

```
{
  "roomID": "101",
  "status": "Booked"
}
```

2. Sending the RESTful Request

Endpoint: https://api.example.com/ims/update

HTTP Request:

```
POST /ims/update HTTP/1.1
Host: api.example.com
Content-Type: application/json
Content-Length: [length]
```

Request Body: (The JSON message defined above)

3. IMS Processing the REST Request

Upon receiving the REST request, the IMS:

- Parses the JSON to extract room status information.
- Validates the provided data.
- Updates the room availability records accordingly.

4. Sending the RESTful Response

Once the IMS has updated the room status, it sends a response back to the HBS.

HTTP Response:

HTTP/1.1 200 OK
Content-Type: application/json

Response Body:


```
{
  "Status": "Success",
  "roomID": "101",
  "message": "Room status updated to booked."
}
```

Conclusion

This example illustrates two distinct integrations in a hotel booking context:

1. **SOAP Example**: The hotel booking system integrates with the payment processing system using a structured SOAP message with XML for secure and formal communication.
2. **REST Example**: The hotel booking system integrates with the inventory management system using a lightweight JSON format for room status updates, leveraging REST's simplicity and efficiency.

Web Services Description Language (WSDL)

Overview: WSDL (Web Services Description Language) is an XML-based language used for describing the functionality of web services. It provides a standard way to describe the services available, how to access them, and the data types used in requests and responses.

Key Components of WSDL

1. **Definitions:**
   - The root element of a WSDL document, containing all the other elements.
2. **Types:**
   - Defines the data types used by the web service. This often includes XML Schema definitions for complex types.
3. **Messages:**
   - Describes the data elements of the operations performed by the web service. Each message consists of one or more parts.
4. **Port Types:**
   - Defines a set of operations (functions) that the web service offers. Each operation includes the input and output messages.

5. **Bindings:**
    - Specifies the protocol used for communication (e.g., SOAP) and the format of the messages.
6. **Service:**
    - Describes the actual web service, including the endpoint (URL) where the service can be accessed.

Example of a Simple WSDL Document

Here's a basic example of a WSDL document for a web service that provides weather information:

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://example.com/weather"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        name="WeatherService"
        targetNamespace="http://example.com/weather">

  <types>
    <xsd:schema>
      <xsd:element name="GetWeatherRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="City" type="xsd:string"/>
          </xsd:sequence>
```

```
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="GetWeatherResponse">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="Temperature" type="xsd:float"/>
                    <xsd:element name="Condition" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
</types>

<message name="GetWeatherRequestMessage">
    <part name="parameters" element="tns:GetWeatherRequest"/>
</message>
<message name="GetWeatherResponseMessage">
    <part name="parameters" element="tns:GetWeatherResponse"/>
</message>
```

```
    <portType name="WeatherPortType">
        <operation name="GetWeather">
            <input message="tns:GetWeatherRequestMessage"/>
            <output message="tns:GetWeatherResponseMessage"/>
        </operation>
    </portType>

    <binding name="WeatherBinding" type="tns:WeatherPortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetWeather">
            <soap:operation
soapAction="http://example.com/weather/GetWeather"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
```

```
    <service name="WeatherService">
        <port name="WeatherPort" binding="tns:WeatherBinding">
            <soap:address location="http://example.com/weather"/>
        </port>
    </service>
</definitions>
```

Key Benefits of WSDL

- **Standardization:** Provides a standardized way to describe web services, promoting interoperability between different platforms and languages.
- **Automation:** Tools can generate client code based on the WSDL description, simplifying the development process.
- **Documentation:** Serves as documentation for developers, detailing how to interact with the web service.

Conclusion

WSDL is a crucial component in the world of web services, enabling developers to understand and utilize web services effectively. By defining the operations, messages, and protocols involved, it facilitates seamless integration between different systems and technologies.

Universal Description, Discovery, and Integration (UDDI)

Overview: UDDI (Universal Description, Discovery, and Integration) is a platform-independent framework that enables the publishing, discovery, and integration of web services. It serves as a directory for businesses to list their services, allowing clients and other services to find and interact with them.

Key Components of UDDI

1. **Registry:**
    - UDDI acts as a registry where service providers can publish their services, making them available for discovery by potential consumers.
2. **Service Descriptions:**
    - Each service listed in UDDI includes descriptions that specify how to interact with the service, often including details like WSDL (Web Services Description Language) documents.
3. **Business Entities:**
    - UDDI supports the registration of businesses and their services, allowing users to find services based on the organization providing them.

4. **Bindings:**
    - It describes the protocols and data formats used by the services, enabling consumers to understand how to connect and communicate with them.

Key Features of UDDI

- **Discovery:** UDDI allows users to search for services based on various criteria, such as service type, business name, or technical details.
- **Standardization:** UDDI is built on open standards, promoting interoperability among different systems and technologies.
- **Dynamic Integration:** It facilitates the dynamic discovery of services, enabling systems to adapt to changes in available services without hardcoding specific endpoints.

How UDDI Works

1. **Publishing:**
    - Service providers create a UDDI entry that includes information about their services, such as WSDL URLs, descriptions, and business details.

2. **Discovering:**
    - Clients can query the UDDI registry to find services that meet specific criteria, retrieving the relevant service details to initiate communication.
3. **Integrating:**
    - Once a service is found, clients can use the provided WSDL to understand how to interact with the service, including how to format requests and interpret responses.

Use Cases for UDDI

- **Enterprise Services:** Large organizations can use UDDI to manage and publish internal and external web services, facilitating easier integration across various departments.
- **B2B Interactions:** Businesses can discover and integrate with partner services, improving collaboration and efficiency.
- **Microservices Discovery:** In a microservices architecture, UDDI can help manage service registrations and facilitate communication between independently deployed services.

Current Status

While UDDI was widely discussed and used in the early 2000s, its adoption has declined with the rise of simpler service discovery mechanisms, especially in RESTful architectures. Modern approaches

often use alternatives like API gateways, service meshes, and more lightweight service discovery solutions.

Conclusion

UDDI provided a foundational framework for discovering and integrating web services. While its usage has diminished, it played a significant role in shaping how services can be published and found, contributing to the development of more modern service-oriented architectures.

Two approaches for using XML & Web Services for integration & Interoperability

They are:-

1. Web Services Integration (WSI)
2. Service Oriented Integration (SOI)

✓ Both Approaches Built on XML, SOAP, WSDL, but they use these technologies in different ways.

Web Services Interoperability (WSI)

1. **XML:**
    - WSI uses XML to encode messages, providing a standard format that can be easily understood by different systems.
    -

2. **SOAP:**
    - WSI often employs SOAP as the communication protocol. SOAP messages are XML-based and include features for security, reliability, and transaction support.
3. **WSDL:**
    - WSDL is used to describe the web services, specifying their operations, input/output messages, and protocols. This allows different systems to understand how to interact with the service.

Service-Oriented Integration (SOI)

1. **XML:**
    - SOI also uses XML for data representation, particularly when services need to exchange structured information.
2. **SOAP (and REST):**
    - While SOI can use SOAP, it often employs RESTful APIs, which may use XML or JSON as data formats. REST emphasizes simplicity and performance, making it popular in modern applications.
3. **WSDL (if using SOAP):**
    - If SOI uses SOAP-based services, WSDL will be used to describe those services. However, many modern RESTful

services do not use WSDL, as they often provide their API documentation in other formats (like OpenAPI or Swagger).

Summary

- **WSI** primarily focuses on interoperability through SOAP-based web services and is heavily reliant on XML and WSDL for formal service descriptions.
- **SOI** emphasizes the integration of services, which can be either SOAP-based or RESTful, and while it can use WSDL for SOAP services, it often relies on other forms of documentation for RESTful services.

✓ Both Approaches uses UDDI Registry, but their use of UDDI may differ in context.

Usage in WSI and SOI

1. Web Services Interoperability (WSI)

- **Service Discovery:** WSI can leverage UDDI to allow services to be easily discovered by clients or other services. This is particularly useful in enterprise environments where multiple services need to interact.

- **Standard Compliance:** WSI emphasizes standards, and using UDDI helps ensure that services are compliant and can be located and consumed by any client that adheres to WSI standards.

2. Service-Oriented Integration (SOI)

- **Service Registry:** In SOI, UDDI can be used as a registry for microservices or APIs, allowing developers to publish and discover services in a decentralized architecture.
- **Dynamic Integration:** SOI often involves integrating multiple services that may change frequently. UDDI helps facilitate this dynamic integration by providing a way to locate services at runtime.

Summary

While both WSI and SOI can use UDDI for service discovery and integration, WSI typically relies more heavily on it due to its focus on standardization and interoperability. In contrast, SOI may utilize UDDI in more flexible and decentralized ways, particularly in environments with numerous microservices.

Web Service Integration:-

“Best for Opportunistic”

Used in Tactical integration projects (Specific Objectives, Short term Focus, Immediate Impact)

- Have clear & specific objectives related to the integration of specific systems, applications (or) processes
- These Projects are generally designed to deliver results in the short term (Addresses the immediate needs (or) Challenges in the organization).
- Bring Immediate improvements without necessarily consider the broader organizational strategy.

WSI Projects involves a smaller number of systems (2 to 4) need to exchange data.

The project team defines soap messages based on the following:

- ✓ Data the systems need to exchange
- ✓ Legacy message formats that the systems already understand
- ✓ Legacy APIs/Methods that are available for accessing the systems.

Advantages:-

1. Faster time - to – market (especially number of systems is small)

2. Lower integration cost

Limitations:-

1. Minimal consideration is given to creating data, service, process models. That are broadly reusable in this service domain.
2. Applications send soap messages by using Transport Level (Involves in Physical/Network of data b/w s/w components) (or) Middleware (Intermediary layer) APIs directly.

➔ Difficult to migrate to alternative transports (or) Middleware when necessary.

Service Oriented Integration:-

Best for Organizations that are trying to maximize the long – term results of an integration architecture by heavily investing in one.

Dedicating significant resources both financial & Technological.

Unlike WSI,

Implementation starts from the start-up Phase (Before the project begins)

- SOA governance framework, processes, guidelines, mmodels & tools are defined.

- Formal service domain is used (These model do not have to be complete) but they identify key datatypes, service contracts, & processes that are used in the organization.
- A service taxanomy is defined consistently for multiple projects to promote future service reuse.

Advantages:-

1. Creates formal & reusable data, service, & process models that are applicable in service domain.
2. Creates abstraction layer reduces vendor lock-in (where a customer becomes dependent on a particular vendor for products or services, making it difficult or costly to switch to another provider) & simplifies application migration & consolidation in the future.

Limitations:-

1. Requires a skilled & dedicated cadre of enterprise architects (who supports the organization strategic goals and also align the business process, information flows)
2. Requires the commitment of business managers & technical managers.

➔ These 2 approaches are used to solve integration & interoperability problems.

Scenarios that illustrate how Web Services Integration (WSI) and Service-Oriented Integration (SOI) can be used to solve integration and interoperability problems:

1. Web Services Integration (WSI)

Scenario: Financial Transaction Processing

Context: A bank needs to integrate its online banking system with a third-party payment processor to facilitate online transactions securely.

Example:

- **Challenge:** Customers want to make payments directly from their bank accounts when shopping online. The bank's system needs to communicate with the payment processor to handle these transactions securely.
- **Implementation:**
    - The bank develops a SOAP-based web service that exposes operations like ProcessPayment and GetTransactionStatus.
    - The payment processor also uses WSDL to describe its service and the expected formats for messages.
- **Outcome:** When a customer initiates a payment, the bank’s online system sends a SOAP request to the payment processor’s web service, adhering to the WSI standards. The processor processes

the payment and sends back a confirmation. Because both systems follow the same standards, they can interoperate seamlessly.

2. Service-Oriented Integration (SOI)

Scenario: E-Commerce Order Processing

Context: An e-commerce platform needs to integrate various services for order management, inventory checking, and shipping.

Example:

- **Challenge:** When a customer places an order, the system must check inventory, process payment, and arrange shipping, all through different services.
- **Implementation:**
  - The e-commerce platform uses a microservices architecture:
    - **Order Service:** Handles order creation.
    - **Inventory Service:** Checks stock availability.
    - **Payment Service:** Processes transactions.
    - **Shipping Service:** Arranges delivery.
  - Each service has a RESTful API that can be called independently.
- **Outcome:** When an order is placed, the Order Service calls the Inventory Service to check stock, the Payment Service to process payment, and the Shipping Service to schedule delivery. Because

the services are loosely coupled, each can be updated or replaced independently, allowing for a flexible and scalable integration.

Summary

- **WSI** focuses on integrating systems using standardized protocols (e.g., SOAP) to ensure secure and reliable communication, exemplified by the bank and payment processor integration.
- **SOI** emphasizes a modular architecture where different services communicate through APIs, facilitating dynamic interactions, as seen in the e-commerce order processing example.

Applying SOA and Web Services for Integration-.NET and J2EE Interoperability

Web services are typically created for the purpose of exchanging data between applications or services, or for exposing an object method for access by another software program. A Web service contract defines how the Web services messages are mapped between various applications, technologies, and software systems.

➔To expose an object method for access by another software program, you can use a web service, such as a RESTful API. Here's a simple example in Python using Flask.

Step 1: Create a Flask Application

First, install Flask if you haven't already:

```
pip install Flask
```

Step 2: Define Your Object and Method

```
from flask import Flask, jsonify, request

app = Flask(_name_)

class Calculator:
    def add(self, a, b):
        return a + b

calculator = Calculator()

@app.route('/add', methods=['POST'])
def add():
    data = request.json
    result = calculator.add(data['a'], data['b'])
    return jsonify({'result': result})
```

```
if _name_ == '_main_':
    app.run(debug=True)
```

Step 3: Run the Application

Run the script:

```
python your_script_name.py
```

Step 4: Access the Method from Another Program

You can access the add method from another software program using an HTTP client. Here’s an example using Python's requests library.

```
import requests

url = 'http://127.0.0.1:5000/add'

data = {'a': 5, 'b': 3}

response = requests.post(url, json=data)

print(response.json())
```

Summary

- Create a web service to expose the object method.
- Define a route that corresponds to the method you want to expose.
- Call the service from another program using HTTP requests.

This approach allows external programs to access your object's methods easily.

- ➔Creating a web service contract involves defining the API endpoints, request and response formats, and any relevant data schemas. For the Flask calculator service you provided, here's an example of a simple web service contract.

Web Service Contract for the Calculator API

1. API Overview

Service Name: Calculator API

Base URL: http://127.0.0.1:5000

2. Endpoints

POST /add

Description: Adds two numbers provided in the request body.

Request

Headers:

Content-Type: application/json

Body (JSON):

json

```
{
  "a": <number>,
  "b": <number>
}
```

Parameters:

a: (required) The first number (integer or float).

b: (required) The second number (integer or float).

Response

Status Codes:

200 OK: Successful addition.

400 Bad Request: Invalid input data.

Body (JSON):

json

```
{
  "result": <number>
```

}

Example Response:

json

{

"result": 8

}

3. Error Handling

400 Bad Request: Returned when the input data is not valid (e.g., missing parameters or non-numeric values).

Response Example:

json

{

"error": "Invalid input data"

}

4. Data Types

Numbers: Can be integers or floats, depending on the requirement of the application.

5. Usage Example

To use the service, a client application should send a POST request to /add with a JSON body containing the two numbers. Here’s an example using Python's requests library:

python

```
import requests

url = 'http://127.0.0.1:5000/add'

data = {'a': 5, 'b': 3}

response = requests.post(url, json=data)

print(response.json())  # Expected output: {'result': 8}
```

Conclusion

This web service contract provides a clear specification for how to interact with the Calculator API, detailing how requests and responses should be structured. This ensures that different applications can reliably communicate with the service.

In an ideal world, a Java bean could seamlessly invoke any .NET Framework object developed using Visual Basic, C#, or Visual C++, but because of platform and language differences, this is not possible. In a nutshell, the .NET platform is designed for close compatibility with the Windows operating system, and it takes full advantage of native Windows features such as multithreading, memory management, file system access, and other system-level APIs. On the other hand, the J2EE platform takes advantage of the Java virtual machine's portability layer to provide the same features and functionality across all operating systems on which it runs.

Web services can be used to provide interoperability across applications developed using .NET and J2EE,

Practical Example

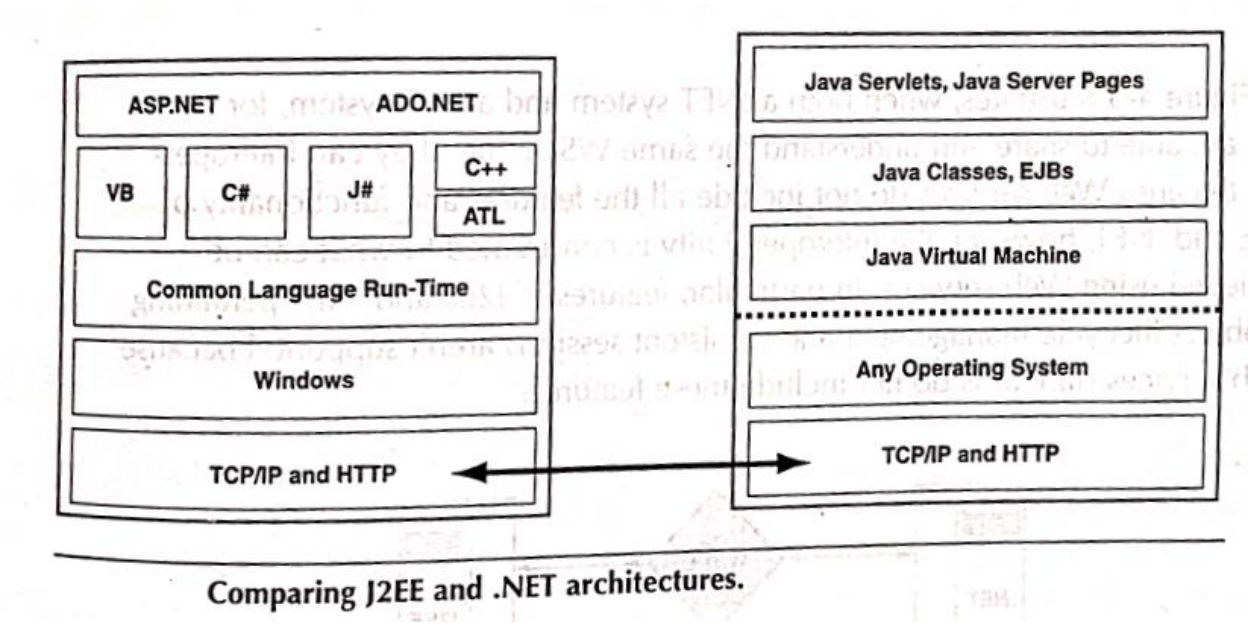
Imagine a scenario where a .NET-based application needs to access data from a J2EE-based inventory system:

1. The J2EE application exposes a web service that allows clients to query inventory data.
2. The .NET application consumes this web service, sending a request over HTTP.
3. The J2EE service processes the request, retrieves the necessary data, and responds with the information in a standardized format (like JSON).

4. The .NET application receives this data and can then display it or perform other operations.

but there are limitations because of the level of functionality currently available in Web services and because of significant differences between the .NET architecture and the J2EE architecture.

Below Figure places the .NET and J2EE environments side by side, highlighting the fundamental difference in their designs with respect to operating system integration. .NET is designed to integrate very closely with the Windows operating system, while J2EE is designed to work on any operating system, including Windows.



Comparing J2EE and .NET architectures.

Because of key differences, interoperability between the .NET platform and the J2EE platform is limited and can only be achieved at a fairly high level of abstraction.

The best approach is to define service contracts (i.e., WSDL interfaces) that either exchange coarse-grained data objects or encapsulate multiple method invocations into a single WSDL service.

For example, if both applications need to share customer data, then you should define an XML Schema for the customer record, use it to define the appropriate WSDL operations, and generate SOAP messages based upon it. The WSDL file and the associated XML Schema are crucial because they define the shared data model.

Example Scenario

Imagine two applications:

- **App A** is built using .NET and needs to access customer records.
- **App B** is a Java application (J2EE) that manages customer data.

Define XML Schema: You create an XML Schema for the customer record:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Customer">
    <xs:complexType>
      <xs:sequence>
```

```
            <xs:element name="CustomerID" type="xs:int"/>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Email" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Define WSDL Operations: You create a WSDL file that describes operations like GetCustomer and UpdateCustomer, specifying the request and response message formats based on the XML Schema.

```
<definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://example.com/customers"
        xmlns:wsoap="http://schemas.xmlsoap.org/wsdl/http/"
        name="CustomerService"
        targetNamespace="http://example.com/customers"
```

```
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">


<!-- Type Definitions -->

<types>

   <xs:schema targetNamespace="http://example.com/customers">

      <xs:element name="Customer">

         <xs:complexType>

            <xs:sequence>

               <xs:element name="CustomerID" type="xs:int"/>

               <xs:element name="Name" type="xs:string"/>

               <xs:element name="Email" type="xs:string"/>

            </xs:sequence>

         </xs:complexType>

      </xs:element>

      <xs:element name="GetCustomerRequest">

         <xs:complexType>
```

```
        <xs:sequence>
            <xs:element name="CustomerID" type="xs:int"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="GetCustomerResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Customer" type="tns:Customer"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="UpdateCustomerRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Customer" type="tns:Customer"/>
```

```
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="UpdateCustomerResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Success" type="xs:boolean"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
  </xs:schema>
</types>

<!-- Message Definitions -->
<message name="GetCustomerRequest">
  <part name="parameters" element="tns:GetCustomerRequest"/>
```

```
    </message>
    <message name="GetCustomerResponse">
        <part name="parameters" element="tns:GetCustomerResponse"/>
    </message>
    <message name="UpdateCustomerRequest">
        <part name="parameters"
element="tns:UpdateCustomerRequest"/>
    </message>
    <message name="UpdateCustomerResponse">
        <part name="parameters"
element="tns:UpdateCustomerResponse"/>
    </message>

    <!-- Port Type Definitions -->
    <portType name="CustomerServicePortType">
        <operation name="GetCustomer">
```

```
            <input message="tns:GetCustomerRequest"/>
            <output message="tns:GetCustomerResponse"/>
        </operation>
        <operation name="UpdateCustomer">
            <input message="tns:UpdateCustomerRequest"/>
            <output message="tns:UpdateCustomerResponse"/>
        </operation>
    </portType>

    <!-- Binding Definitions -->
    <binding name="CustomerServiceBinding"
type="tns:CustomerServicePortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetCustomer">
            <soap:operation
soapAction="http://example.com/customers/GetCustomer"/>
```

```
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="UpdateCustomer">
            <soap:operation
soapAction="http://example.com/customers/UpdateCustomer"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
```

```
        </operation>
    </binding>

    <!-- Service Definitions -->
    <service name="CustomerService">
        <port name="CustomerServicePort"
binding="tns:CustomerServiceBinding">
            <soap:address
location="http://localhost:8080/customerservice"/>
        </port>
    </service>
</definitions>
```

Generate SOAP Messages: When App A wants to get customer details, it constructs a SOAP message that adheres to the format defined in the WSDL, ensuring that the message is understood by App B.

```
using System;
using System.Net.Http;
```

```
using System.Text;
using System.Threading.Tasks;


class Program
{
    static async Task Main(string[] args)
    {
        int customerId = 1; // Example customer ID
        string soapUrl = "http://localhost:8080/customerservice"; // WSDL service endpoint


        string soapEnvelope = $@"<?xml version=""1.0"" encoding=""utf-8""?>
            <soap:Envelope xmlns:soap=""http://schemas.xmlsoap.org/soap/envelope/""
                           xmlns:tns=""http://example.com/customers"">
                <soap:Body>
```

```
            <tns:GetCustomerRequest>
                <CustomerID>{customerId}</CustomerID>
            </tns:GetCustomerRequest>
        </soap:Body>
    </soap:Envelope>";

    using (HttpClient client = new HttpClient())
    {
        HttpRequestMessage request = new
HttpRequestMessage(HttpMethod.Post, soapUrl);
        request.Content = new StringContent(soapEnvelope,
Encoding.UTF8, "text/xml");
        request.Headers.Add("SOAPAction",
"http://example.com/customers/GetCustomer");

        try
        {
```

```
            HttpResponseMessage response = await
client.SendAsync(request);

            response.EnsureSuccessStatusCode();



            string responseContent = await
response.Content.ReadAsStringAsync();

            Console.WriteLine("Response from service:");

            Console.WriteLine(responseContent);

        }

        catch (Exception ex)

        {

            Console.WriteLine("Error occurred: " + ex.Message);

        }

    }

  }

}
```

Conclusion

This example demonstrates how App A can construct and send a SOAP message to request customer details from a J2EE web service. The SOAP message adheres to the format defined in the WSDL, ensuring that it is understood by App B.

Enterprise Service Bus Pattern

The **Enterprise Service Bus (ESB) Pattern** is an architectural design pattern used in service-oriented architecture (SOA) to facilitate communication between different services, applications, or systems within an enterprise. It acts as a middleware layer that enables various services to communicate with one another in a flexible and scalable manner.

Key Features of the ESB Pattern

1. **Decoupling**:
    - Services are decoupled from each other, meaning that they do not need to know the specifics of one another's implementation. This allows for easier integration and modification of services.

2. **Message Routing**:
    - The ESB can route messages between services based on various criteria, such as content-based routing or predefined rules. This ensures that messages reach the correct service.
3. **Protocol Transformation**:
    - An ESB can handle different communication protocols (e.g., HTTP, JMS, FTP) and transform messages between them, allowing heterogeneous systems to interact seamlessly.
4. **Message Transformation**:
    - It can also transform the format of messages (e.g., from XML to JSON) as needed for different services.
5. **Service Orchestration**:
    - The ESB can orchestrate complex business processes by coordinating multiple services, managing their execution order, and handling dependencies.
6. **Error Handling**:
    - Centralized error handling and logging capabilities help manage exceptions and failures across the system.
7. **Scalability**:
    - The ESB pattern allows for the addition of new services without requiring changes to existing services, facilitating scalability.
    -

Components of an ESB

1. **Service Registry**:
    - A directory where services are registered and can be discovered by other services.
2. **Message Broker**:
    - The core component that routes and manages messages between services.
3. **Adapters/Connectors**:
    - Components that allow integration with various communication protocols and message formats.
4. **Transformation Engine**:
    - Responsible for converting messages between different formats and structures.
5. **Orchestration Engine**:
    - Manages the workflow and coordination of multiple services in a business process.

Example Use Case

Imagine an e-commerce application where different services handle orders, payments, and inventory. An ESB could facilitate communication among these services:

- When a customer places an order, the order service sends a message to the ESB.
- The ESB routes the message to the payment service to process the payment.
- Once the payment is successful, the ESB informs the inventory service to update stock levels.
- If any service fails, the ESB can handle the error gracefully and provide feedback to the user.

Summary

In summary, the **Enterprise Service Bus (ESB) Pattern** is a robust architecture that enhances communication between disparate services in an enterprise, promoting decoupling, flexibility, and scalability. It provides a centralized way to manage message routing, transformation, and orchestration, making it easier to build and maintain complex service-oriented systems.

UNIT- III

SOA & Multi Channel Access (P2)

Multi-Channel Access

Definition: Multi-Channel Access refers to providing users with various ways to interact with a system or service. This can include web applications, mobile apps, social media, and more.

OR

Capability of accessing & interacting with a system, service, platform through multiple communication channels,

Goal is to offer a seamless & integrated user experience across different communication channels.

EX:

1. Web Channels – Web Browser
2. Mobile Channels – Smart Phones
3. Social Media Channels – FB, IG, Twitter
4. Messaging Channels – WhatsApp, Telegram
5. Voice Channels – Google Assistant, SIRI, ALEXA
6. Call Center Channels – To seek assistance
7. Email Channels

The primary purpose of most organizations (commercial, government, non- profit, and so on) is to deliver services to clients, customers, partners, citizens, and other agencies. Table 5-1 illustrates this for four key industries by listing the services they deliver, the channels they use to deliver these services, and some of the end-user devices and technologies used to deliver these services.

Table 5-1 Some Examples of Service-Oriented Businesses

	Government	Telecom, Communication	Financial Services	Health Care
Service Requesters	citizens and other agencies	customers, business partners	customers, business partners	patients, doctors, insurance carriers, hospitals, government
Services	law enforcement health services disaster management community and social services	local phone service long-distance service mobile/wireless DSL/ADSL/Internet	mortgage/loans credit/debit cards investment management insurance	preventative care emergency care out-patient care nursing care prenatal care

continues

Table 5-1 Some Examples of Service-Oriented Businesses (continued)

	Government	Telecom, Communication	Financial Services	Health Care
Delivery Channels	government office call center mail/fax self-service (eGov) agency to agency	call center self-service (Web, IVR) business-to-business field service technician	retail branch office self-service (Web, IVR) home banking Automated Teller Machine (ATM)	doctor's office emergency ward telephone mail/fax
End-User Devices	office PC home PC telephone web browser mobile/handheld devices	office PC home PC telephone web browser mobile/handheld devices	ATM web browser telephone office PC home PC mobile/handheld devices	office PC home PC telephone medical equipment mobile/handheld devices

In the past, organizations often developed new monolithic applications with a single delivery channel in mind. This can be seen in systems ranging from 3270 applications for money transfer to browser-based applications specifically designed for e-commerce.

The proliferation of delivery channels and end-user devices has given service oriented businesses the opportunity to better serve their customers anytime and anywhere, but it has also placed an enormous strain on IT departments, as they struggle to convert monolithic applications to make them multi-channel-ready.

It is now necessary for these organizations to deliver these same services and new ones in a consistent manner across all channels. This poses real problems because it is difficult to multi-channel-enable monolithic applications originally built for a single channel. The solution is to use SOA with Web services.

In general, business services change much more slowly than delivery channels (see Figure 5-1). This is because the business services represent long-standing business functions such as account management, order management, and billing, whereas client devices and delivery channels are often based on new devices or new market niches, which tend to change more frequently. In some cases, the rate of change at the presentation layer is 100 times faster than the rate of change at the business services layer.

Therefore, it only makes sense to reuse existing business services when possible. Many of the core business services of large organizations are mission- critical systems running on TP monitors such as CICS, IMS, or Tuxedo. Many core systems also run on SAP, PeopleSoft, Oracle applications, Siebel, CORBA, J2EE, and COM. SOA has proven to provide the right balance between abstraction for dealing with diverse technology and loose coupling necessary for reusing business services for multi-channel applications.

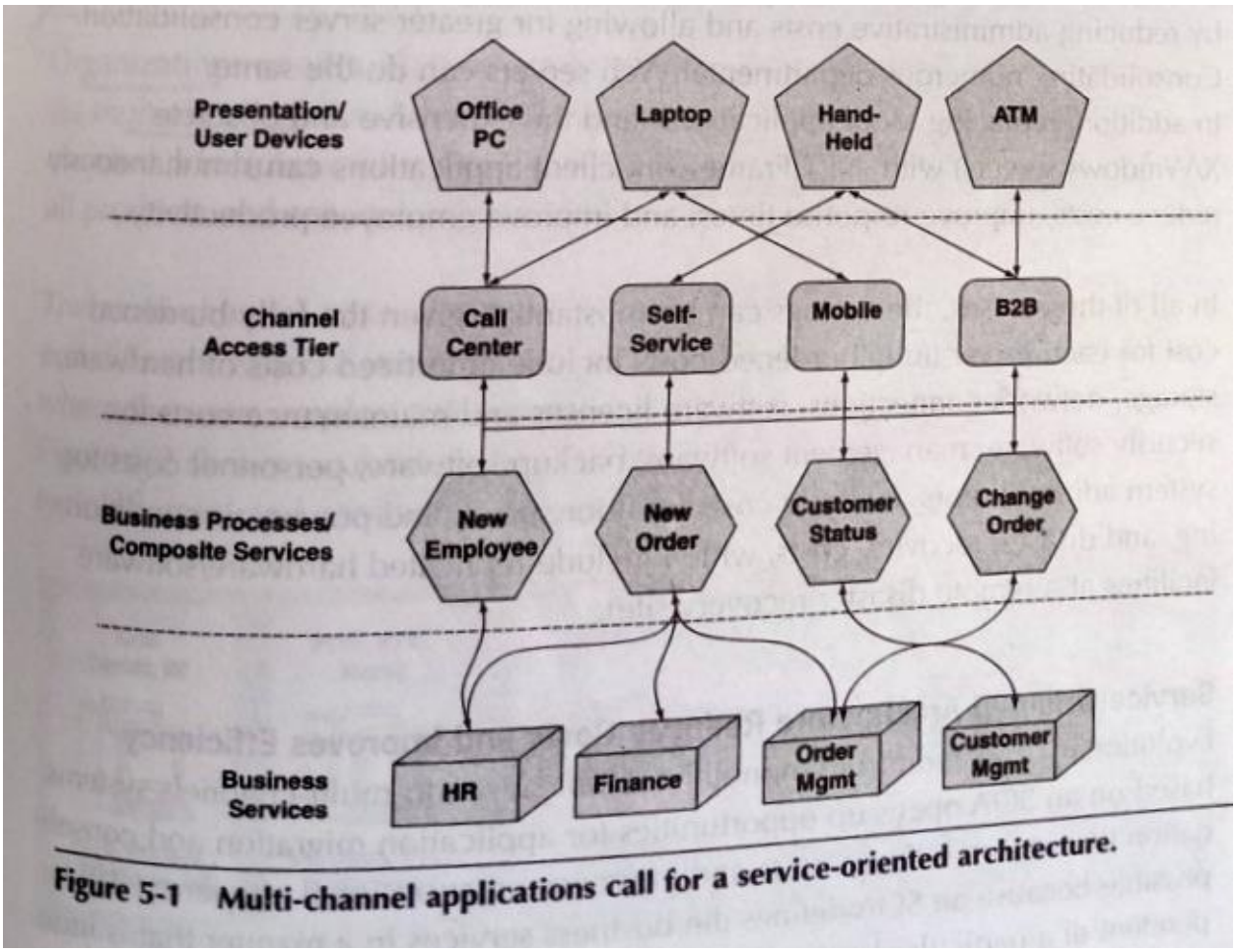


Figure 5-1 Multi-channel applications call for a service-oriented architecture.

Business Benefits of SOA and Multi-Channel Access

Multi-Channel Access Reduces Staffing Costs

Evolutionary migration from monolithic applications to multi-channel systems based on an SOA provides the opportunity to reduce staffing costs by moving some service delivery activities from human-intensive processes to less expensive self-service processes.

Multi-Channel Access Eliminates Obsolete and Expensive Infrastructure

Evolutionary migration from monolithic applications to multi-channel systems based on an SOA provides the opportunity to re-engineer existing processes and eliminate obsolete and expensive infrastructure.

For example, eliminating brittle departmental client/server applications and replacing them with more robust, scalable enterprise services can save money by reducing administrative costs and allowing for greater server consolidation. Consolidating numerous departmental Web servers can do the same. In addition, replacing Motif applications (and the expensive and obsolete X/Windows servers) with .NET Framework client applications can simultaneously reduce costs, improve response times, and improve employee productivity.

In all of these cases, the savings can be substantial, given the fully burdened cost for each server (fully burdened costs include amortized costs of hardware, storage, network connections, software licenses and maintenance costs for security software, management software, backup software, personnel costs for system administrators, facilities costs for floor space, and power, air conditioning, and disaster recovery costs, which include replicated hardware/software facilities at a remote disaster recovery site).

Service-Oriented Architecture Reduces Costs and Improves Efficiency

Evolutionary migration from monolithic applications to multi-channels systems based on an SOA opens up opportunities for application migration and consolidation that in turn reduces costs and improves organizational efficiency. This is possible because an SOA defines the business services in a manner that is independent of a particular legacy application or packaged application.

Here is a typical application migration scenario: Consider an existing application that is expensive to maintain and that no longer delivers the functionality needed to meet new business requirements. An SOA infrastructure allows the application to be wrapped as a set of business services and then incrementally replaced with a less expensive, easier-to-maintain application that delivers better overall capabilities.

Here is a typical application consolidation example: Consider an organization that has numerous customer care systems. Each one requires its own separate hardware infrastructure, administrators, user training, and so on. An SOA infrastructure allows all of these customer care systems to be seamlessly consolidated to one or two systems along with the savings in hardware, administrative costs, reduced user-training costs, and reduced software maintenance fees.

Service-Oriented Architecture for Multi-Channel Access

Organizations need to deliver products and services to customers and partners via multiple channels. A multi-channel access architecture based on service oriented principles makes the organization more agile by allowing it to deliver all products and services in a consistent manner across all distribution channels.

The multi-channel access pattern is characterized by the need to provide several different types of users with access to a common set of business services where the users employ a diverse set of end-user devices and technologies. Figure 5-2 shows a subset of the delivery channels that might be used in a typical system and some of the related client technology.

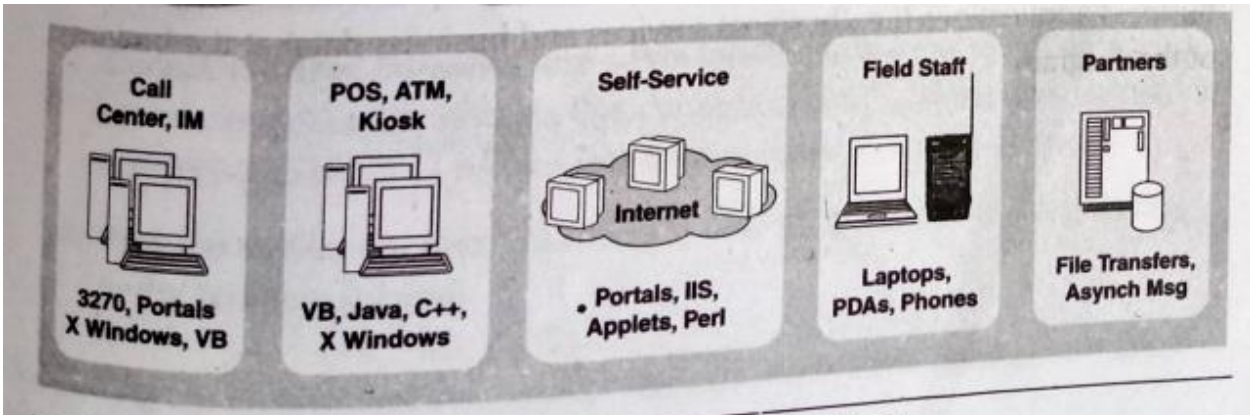


Figure 5-2 Some end-user devices used to access applications.

Architectural Challenges

The main architectural challenge of multi-channel access is mediating between the characteristics of a diverse set of end-user devices and the characteristics of the equally diverse set of internal systems and technologies. Here are some of the characteristics of these systems that need to be considered (more on these later):

Connectivity-For some access channels, we can assume that the user is sitting in front of a PC with a fast and reliable connection, while other access channels are constrained by slow and unreliable connections (dial-up users, mobile users, and field technicians).

Security-For some access channels, we can assume that the user is working inside of a corporate firewall, while other access channels are for requesters that are accessing these services over less secure wireless networks and the Internet.

Communication technology-Different user devices use different communication technology, including standard protocols (e.g., HTTP, Sockets, email, file transfer, Java RMI, CORBA) and proprietary protocols (e.g., MS DCOM, WebSphere MQ, Tibco Rendezvous).

Architecture for Multi-Channel Access

Figure 5-3 shows a layered architecture for providing multi-channel access to business services.

From a business perspective, the architecture is intended to connect client applications at the top of the diagram (i.e., the client/presentation tier) to the business applications (i.e., business services and business data) at the bottom of the diagram.

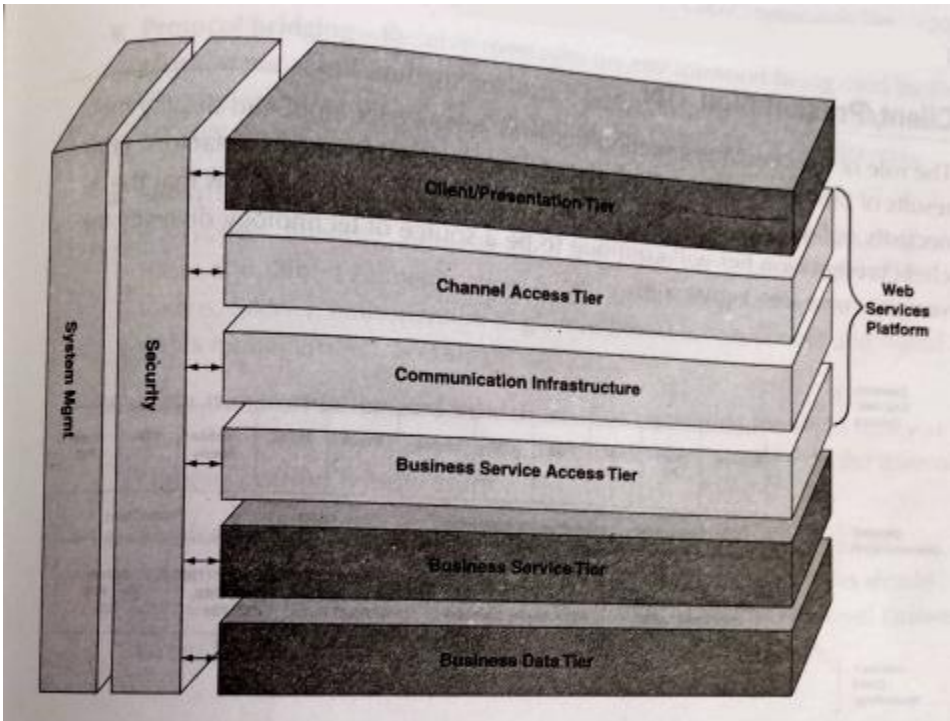


Figure 5-3 Layered architecture for providing multi-channel access.

In between these tiers are the following three layers necessary to support an SOA-based multi-channel access architecture:

1. **Channel access tier**

Mediates between the diverse client applications and user devices and the internal communication infrastructure and business services

2. **Communication infrastructure**

Provides the enterprise-wide middle- ware and messaging systems that connect internal systems and provide enterprise qualities of service that these internal systems rely upon

3**. Business services access tier**

Responsible for providing uniform access to the business services

Supporting the architecture are security services and system management facilities that span all layers of the architecture.

Client/Presentation Tier

The role of the client/presentation tier is to accept user input and display the results of user interactions. The myriad of end-user devices, form factors, connectivity options, user preferences, and client technologies ensures that the client/presentation tier will continue to be a source of technology diversity for years to come (see Figure 5-4).

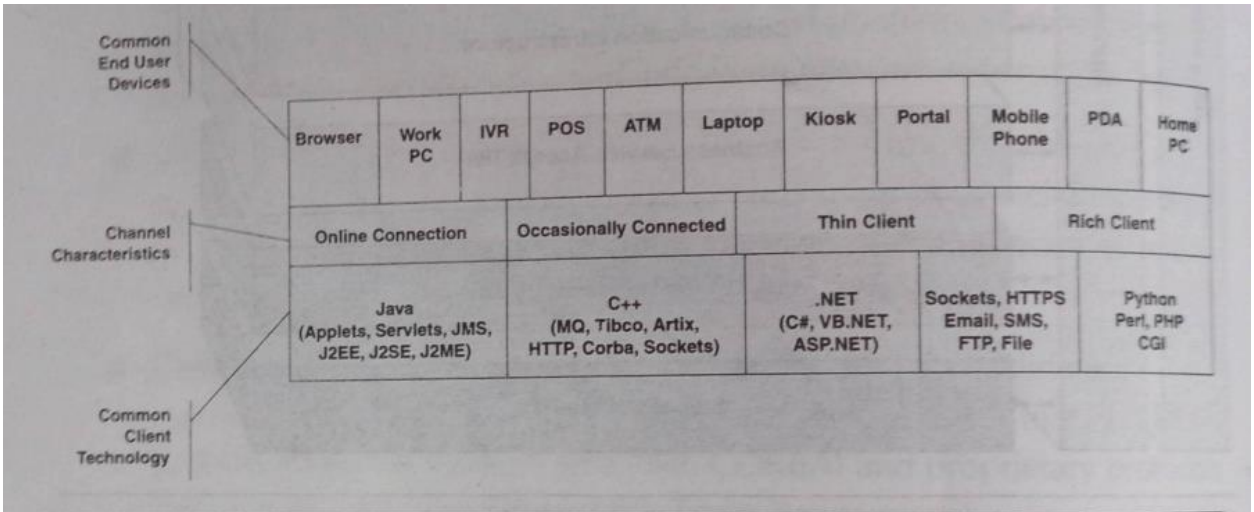


Figure 5-4 Client/presentation tier.

Channel Access Tier

The role of the channel access tier is to mediate between the client applications and the business services (see Figure 5-5). For example:

Support all common data formats and protocols

Including SOAP, XML name/value pairs, delimited format, fixed width format, Sockets, HTTP, FTP, SMTP, JMS, WebSphere MQ, Tibco Rendezvous, IIOP, and so on, so that a diverse set of clients can be easily supported.

Support all common communication interaction patterns

Including request/response, request/callback, asynchronous messaging, and publish/subscribe so that a diverse set of clients can be easily supported.

Payload mapping

Accept messages from clients in client-specific formats and automatically translate them into the enterprise message standard or the message format defined by the target business service.

Protocol bridging

Receive messages on any transport being used by the client applications and automatically route them to the enterprise's middleware standard including WebSphere MQ, IMS, Tibco Rendezvous, HTTP/S. or CORBA IIOP.

Security facilities

Support all major standards for security including encryption, integrity, authentication (e.g., user name/password, HTTP Basic and Digest Authentication, X.509 certificates, Kerberos security tokens, SAML), authorization (e.g., role-based access control and digital rights management), and single sign-on.

Data transformation and validation

For converting messages received from the client applications into the data formats required by the internal communication infrastructure.

Service lookup and service routing

So that client requests can be routed to the services they require. The service lookup facilities should support load balancing across service instances and service-level failover when a service fails.

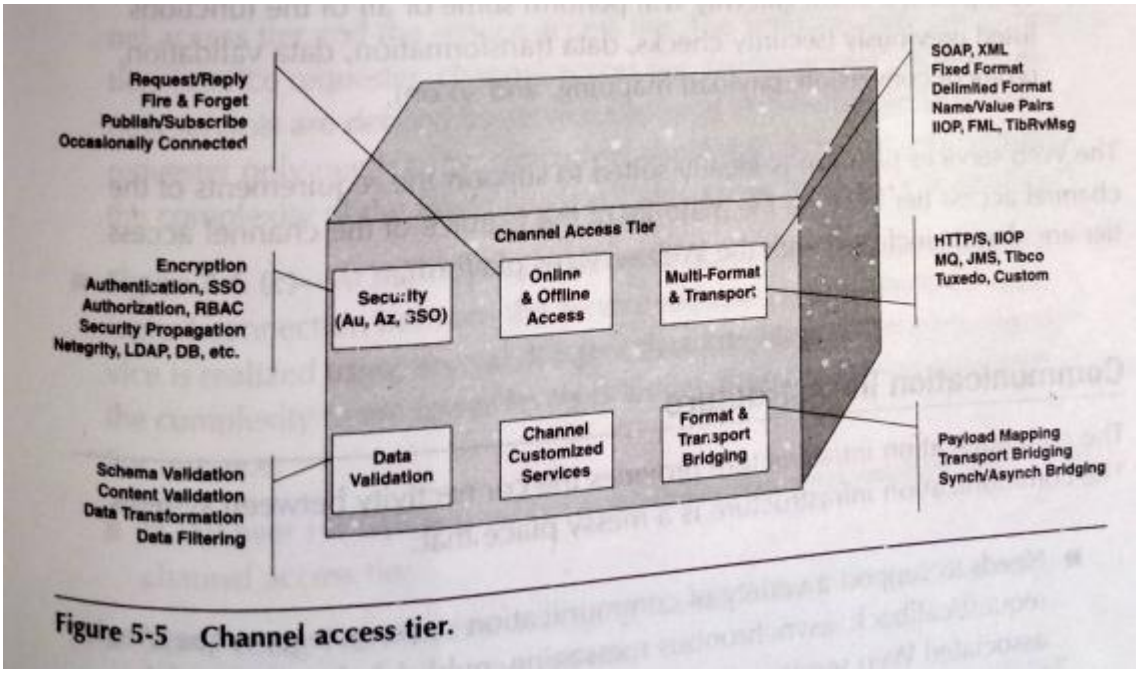


Figure 5-5 Channel access tier.

Typically, the channel access tier is composed of two types of components service proxies and client gateways:

Service proxies

A service proxy is a programming language object (e.g. Java, C++, or C#) for a service interface that is compiled into the client application. For example, in JAX-RPC, one Java class is created for each WSDL portType, which includes one member function for each operation defined in the WSDL portType.

This simplifies creating client applications because the client program can invoke the service-operation without having to create, manage, and manipulate the underlying XML documents. Depending on how it is configured, a service proxy (or associated handlers) can perform some or all of the functions listed previously (security checks, data transformation, data validation, protocol conversion, payload mapping, and so on).

Client gateways

A client gateway is typically a standalone message intermediary, which receives messages from clients on incoming ports and routes them to servers via outgoing ports. Depending on now it is configured, a client gateway will perform some or all of the functions listed previously (security checks, data transformation, data validation, protocol conversion, payload mapping, and so on).

The Web services platform is ideally suited to support the requirements of the channel access tier because the majority of the features of the channel access tier are already included with the Web services platform.

Communication Infrastructure

The communication infrastructure provides the connectivity between systems. The communication infrastructure is a messy place that:

- Needs to support a variety of communication patterns (e.g., request/reply request/callback, asynchronous messaging, publish/subscribe) and the associated Web services standards (e.g., WS-ReliableMessaging and WS-Eventing).
- Needs to support message routing.
- Needs to provide multi-level security (e.g., transport-level security, message-level security, authentication, authorization, role-based access control, or single sign-on).

The best way to do this is using an SOA where there is a clean separation between the logical service contracts and the physical contracts that define the bindings to particular data formats and protocols. Figure 5-6 illustrates this by showing three views of the same system:

Figure 5-6 (a) At the highest level of abstraction, the client application (Le, service requester) uses the business service based on the logical service contract, without any regard for the underlying communication infrastructure.

Figure 5-6 (b) At the next lower level of abstraction, we see that the logical connection between the service requester and the business service is realized by the service invocation being routed through the channel access tier and the service access tier. The interfaces between all four tiers (service requester, channel access tier, service access tier, and

business service) are defined by service contracts. (However, the service requester only needs to be concerned about the contract it uses because the complexity of the lower levels is hidden from the service requester.)

Figure 5-6 (c) At the next lower level of abstraction, we see that the logical connection between the service requester and the business service is realized using several different data formats and transports again, the complexity of the lower levels is hidden from the service requester), For instance:

- SOAP over HTTP/S is used between the service requester and the channel access tier.
- SOAP using WS-ReliableMessaging is used between the channel access tier and the service access tier.
- COBOL Copybooks over WebSphere MQ is used between the service access tier and the business service.
- In this case, the channel access tier and the service access tier are responsible for payload mapping, protocol conversion, and routing based on information in the physical portion of the logical contract and in a manner that conforms to the higher-level logical contract.

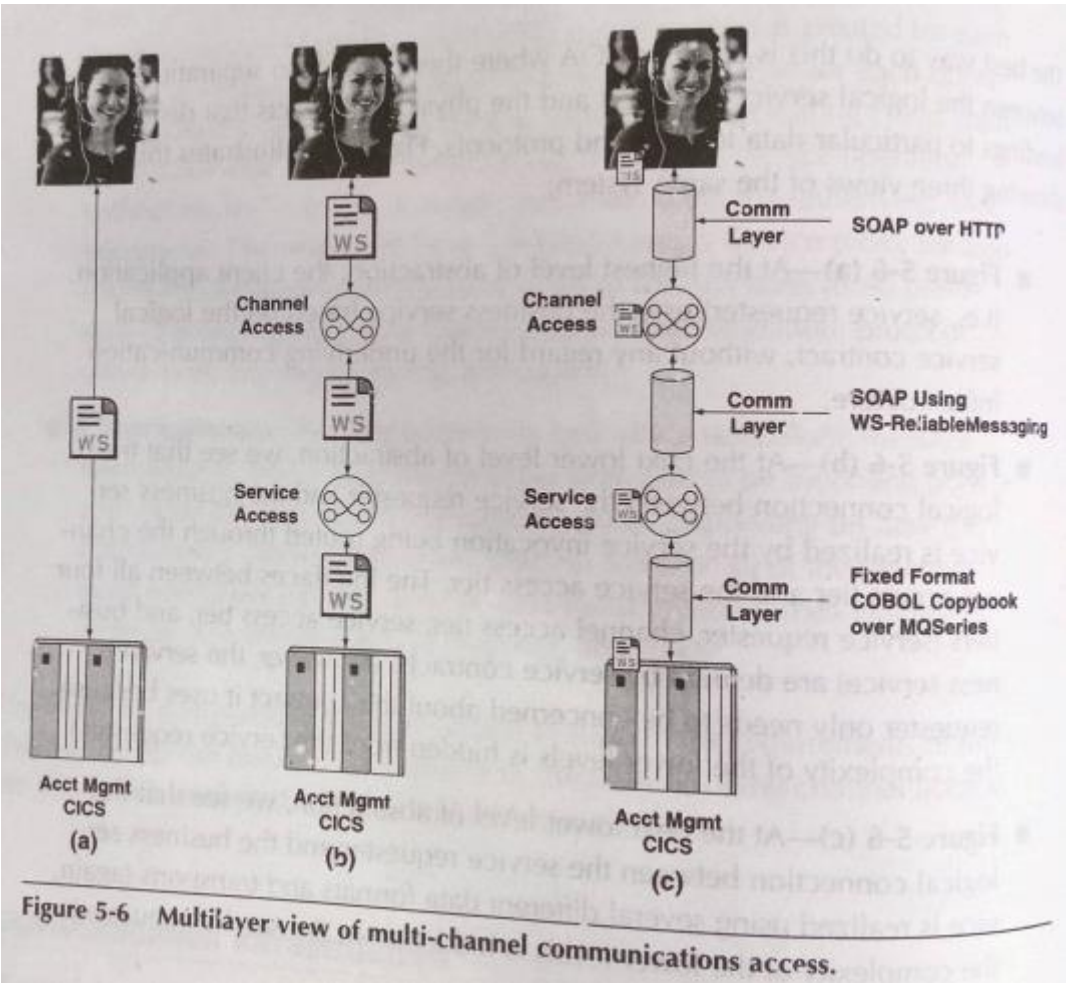


Figure 5-6 Multilayer view of multi-channel communications access.

Business Service Access Tier

The role of the business service access tier is to mediate between the communication infrastructure and the business services (see Figure 5-7).

Several of the key facilities provided by this layer of the architecture are similar to features provided by the channel access tier:

- **Service registration and service lookup**

  So that client applications can locate the services they require. The service lookup facilities should sup- port load balancing across service instances and service-level failover when a service fails.

- **Session management**

  For handling conversational interactions between client applications and stateful services. (Session management may be also required for stateless interactions especially when strong authentication is required, such as in WS-SecureConversation.)

- **Data transformation and validation**

  For converting messages received from the communication infrastructure into the data formats required by legacy systems.

- **Security services**

  Support all major standards for security, including encryption, integrity, authentication (e.g., WS-Security, user name/pass- word, HTTP Basic Authentication, X.509 certificates, Kerberos security tokens), authorization (e.g., role-based access control), and single sign-on.

- **Service enablement**

  Facilities for quickly and non-invasively exposing legacy systems as Web services.

- **Service orchestration and composition**

  Facilities for creating new services by composing existing services using WS-BPEL

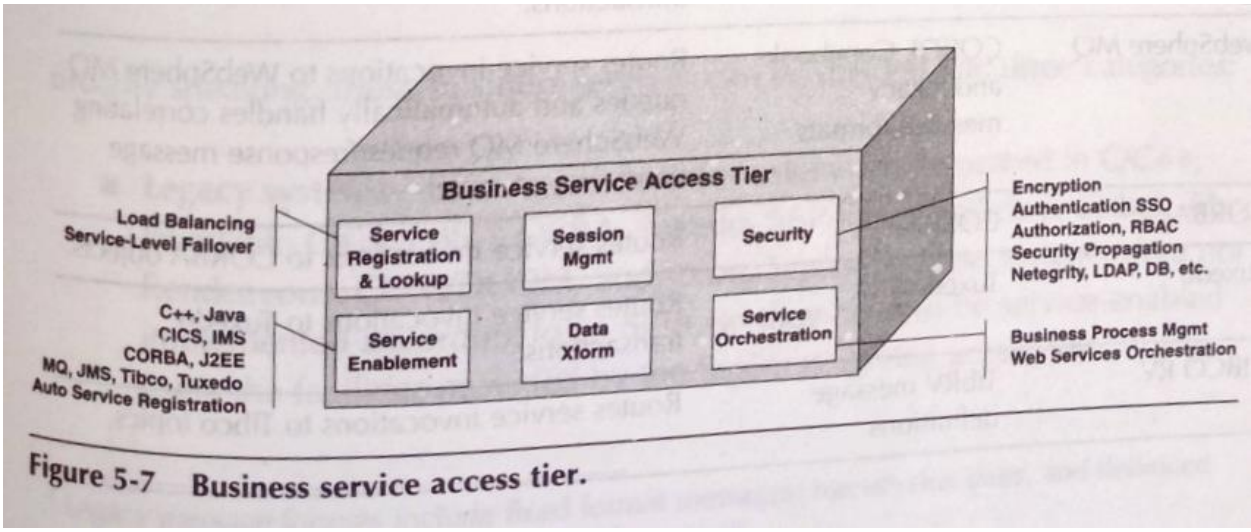


Figure 5-7 Business service access tier.

Many of these services are also included in the channel access tier. The main difference here is that the implementations of these services for the business service access tier must be faster, more scalable, more robust, and more reliable due to the transaction processing load that production-quality business services must handle.

The most important additional service that the business service access tier provides is legacy service gateways for quickly and non-invasively exposing legacy systems as Web services.

Typically, the legacy service gateways provide development-time tools and run- time facilities for turning legacy systems into services that can be invoked using any of the other supported data formats and transports and any of the major communication interaction paradigms. Usually the

development tools provide facilities for importing metadata describing the legacy systems and turning them into WSDL service contracts (e.g., WSDL, XML Schema, COBOL Copybooks. CORBA IDL, database schema, and delimited data formats such as comma separated files). Table 5-2 displays a sampling of legacy systems and possible metadata importers.

Table 5-2 Sampling of Legacy Systems and Possible Metadata Importers

Legacy Service Gateway	Potential Metadata Source	Notes
CICS and IMS	COBOL Copybooks	Routes service invocations to CICS/IMS transactions.
WebSphere MQ	COBOL Copybooks and legacy message formats	Routes service invocations to WebSphere MQ queues and automatically handles correlating WebSphere MQ request/response message pairs.
CORBA	CORBA IDL	Routes service invocations to CORBA objects.
Tuxedo	Tuxedo FML	Routes service invocations to Tuxedo transactions.
TIBCO RV	TibRV message definitions	Routes service invocations to Tibco topics.
C++	Legacy message formats[1]	Routes service invocations to C++ objects.
Java, EJBs, JMS	Java classes and remote interfaces of stateless session beans	Routes service invocations to JMS topics, Java classes, and stateless session beans.
RDBMS	Database schema	Reads/writes data to/from relational database tables.
Packaged apps	Various	Provides Web services interfaces to package applications such as SAP R/3, PeopleSoft, Siebel, and so on.

Business Service Tier

The role of the business service tier is to implement the business services (i.e., transactions, information updates, information retrievals, and so on) necessary for running the business (see Figure 5-8).

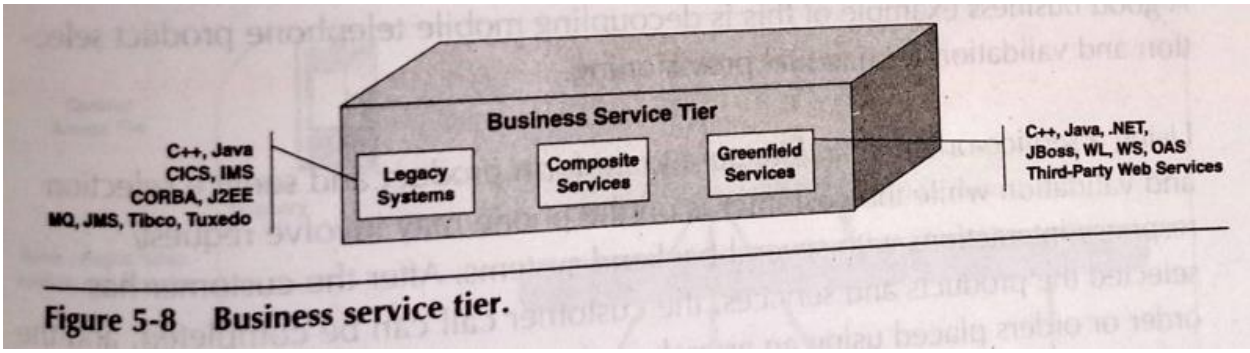


Figure 5-8 Business service tier.

Broadly speaking, these business services can be divided into three categories:

Legacy systems

Existing production systems implemented in C/C++, Java/J2EE, CICS, IMS, CORBA, Tuxedo, SAP R/3, PeopleSoft, Siebel, Tibco Rendezvous, COM/DCOM, and so on. Typically, these systems were not implemented according to an SOA, so they need to be service-enabled using the facilities provided by the business service access tier.

Greenfield Services

New services deployed to provide new business capabilities or to replace legacy systems that are being phased out. These might be implemented using Java/J2EE, .NET Framework, or C/C++, or by deploying a

packaged application. More and more, these tools include capabilities for exposing the new services as Web services.

Composite services

Business services that use one or more other business services. Composite business services should themselves be implemented according to SOA principles so that they can use multiple services implemented using different technologies. Web service orchestration tools based on WS-BPEL should be used to make building composite services easier.