**Day 1:15/07/2024**

**Name: Brindha Selvaraj**

**Software**

Software is a set of instructions or programs that tell a computer what to do. It is a collection of data, instructions, and algorithms that are used to operate computers and manage data. Software is an essential part of a computer system, and it is used to perform a wide range of tasks, from simple calculations to complex simulations.  
  
**Types of Software:**

There are two main categories of software:

* Application Software
* System Software.

**Application Software:**

Application software is designed to perform specific tasks or functions. It is used to solve real-world problems, automate tasks, and provide services to users. Examples of application software include:

* Microsoft Office (word processing, spreadsheet, presentation)
* Google Chrome (web browser)
* Adobe Photoshop (image editing)
* Skype (video conferencing)
* Games (e.g., Minecraft, Fortnite)

Application software is designed to help users accomplish specific tasks or goals. It provides a user interface and functionality to perform tasks such as:

* Creating documents and spreadsheets
* Browsing the internet
* Editing images
* Communicating with others
* Playing games

**System Software:**

System software is designed to manage and control computer hardware components and provide a platform for running application software. It is responsible for managing the computer's resources, such as memory, storage, and processing power.

Examples of system software include:

* Operating Systems (e.g., Windows, macOS, Linux)
* Device drivers (e.g., printer drivers, graphics drivers)
* Utilities (e.g., disk formatting, disk defragmentation)
* Firmware (e.g., BIOS, UEFI)

System software is designed to manage and control the computer's hardware components, providing a platform for running application software. It is responsible for:

* Managing memory and storage
* Controlling input/output operations
* Providing a interface for hardware devices
* Managing system resources
* Booting the computer

Here's an analogy to help illustrate the difference:

Application software is like a car's GPS system, which provides directions and helps you navigate to your destination.

* System software is like the car's engine, transmission, and brakes, which work together to power and control the vehicle.
* While the GPS system (application software) helps you get where you want to go, the car's engine and other systems (system software) make it possible for the car to move and function in the first place

**Day2:16/07/2024**

**Software Architecture styles:**

* Layer (osi model)
* Client-server (Request-response)
* Micro service (solving problems)
* Event driven
* SOA(Service Oriented Architecture)

Case study on SOA [- https://www.infopulse.com/blog/integration-approaches-service-oriented-systems](-%20https:/www.infopulse.com/blog/integration-approaches-service-oriented-systems)

**Summary:**

Service-Oriented Architecture (SOA) has become a widely adopted approach in software design, enabling organizations to integrate autonomously deployed and maintained software components. This case study explores the common integration approaches in SOA-based systems, including direct connection and message-driven design patterns.

**Background**

In today's fast-paced business environment, organizations need to regularly implement new services to stay competitive. SOA provides the necessary flexibility and adaptability to achieve this goal. With the increasing number of applications adopted by companies, efficient governance approaches are essential.

**Problem Statement**

The main challenge in SOA-based systems is integrating autonomously deployed and maintained software components. This requires an efficient communication mechanism between services, which can be achieved through various integration approaches.

**Analysis**

The two most common integration approaches in SOA-based systems are:

**Direct Connection**

In this approach, services communicate with each other using HTTP requests. While it is the easiest and fastest way to implement connections between services, it has drawbacks such as tight coupling and synchronous communication.

**Message-Driven Design Pattern**

This approach allows applications, services, and systems to send messages to a delivery service, which then brings messages to consumers. This approach provides loose coupling, asynchronous communication, and high scalability and resilience.

**Solution**

The message-driven design pattern is a more suitable approach for SOA-based systems, as it provides loose coupling and asynchronous communication. This approach can be implemented using message brokers or enterprise service buses.

**Implementation**

The implementation of the message-driven design pattern involves setting up a message broker or enterprise service bus, which acts as a delivery service between producers and consumers.

**Results**

The message-driven design pattern provides a more efficient and scalable integration approach in SOA-based systems, enabling organizations to achieve greater flexibility and adaptability.

**Discussion**

The choice of integration approach depends on business demands, platforms, and SOA best practices. The message-driven design pattern is a more suitable approach for SOA-based systems, as it provides loose coupling and asynchronous communication.

**Conclusion**

In conclusion, the message-driven design pattern is a more efficient and scalable integration approach in SOA-based systems, enabling organizations to achieve greater flexibility and adaptability.

**Recommendations**

Organizations should consider implementing the message-driven design pattern in their SOA-based systems to achieve greater flexibility and adaptability.

UML DIAGRAM:

SEND()

RECEIVE()

SEND()

RECEIVE()

SEND()

RECEIVE()

PROCESS()

SEND MESSAGE

RECEIVE MESSAGE

SEND MESSAGE

DIRECT CONNECTION

Message Broker

Message Queue

Management Service

ESB

Processing Service

DIRECT ACTION

**Case Study: Online Shopping Platform**

**Background:**

E-commerce has become a significant part of our daily lives, and online shopping platforms have made it convenient for customers to purchase products from anywhere in the world. Our company, "Easy-Shop," wants to develop an online shopping platform that provides a seamless shopping experience to its customers.

**Requirements:**

* The online shopping platform should have the following features:
* Product Catalog: A service that provides a list of products available for sale, along with their descriptions, prices, and images.
* Order Management: A service that handles customer orders, including payment processing, order tracking, and order fulfillment.
* Inventory Management: A service that manages the inventory of products, including tracking stock levels, updating product availability, and alerting administrators when stock levels are low.
* Payment Gateway: A service that integrates with various payment gateways, such as PayPal, Visa, and Master card, to process payments.
* Customer Management: A service that manages customer information, including login, registration, and order history.

**Why We Use SAO In This Case Study**

We are using Service-Oriented Architecture (SOA) in the case study of Service-Oriented Architecture because it provides a scalable, flexible, and reusable solution for the online shopping platform.

In this case study, we have multiple services that need to communicate with each other, such as the Product Service, Order Service, Inventory Service, Payment Service, and Customer Service. By using SOA, we can break down the system into independent services that can be developed, deployed, and scaled independently without affecting other services.

**Benefits of Using SOA In This Case Study**

* Loose Coupling (Aync)
* Reusability
* Flexibility
* Scalability

**Implementation of SOA In This Case Study**

* Product Service: Responsible for managing the product catalog, including adding, updating, and deleting products.
* Order Service: Responsible for managing customer orders, including payment processing, order tracking, and order fulfillment.
* Inventory Service: Responsible for managing the inventory of products, including tracking stock levels, updating product availability, and alerting administrators when stock levels are low.
* Payment Service: Responsible for integrating with various payment gateways to process payments.
* Customer Service: Responsible for managing customer information, including login, registration, and order history.

**Web Services:**

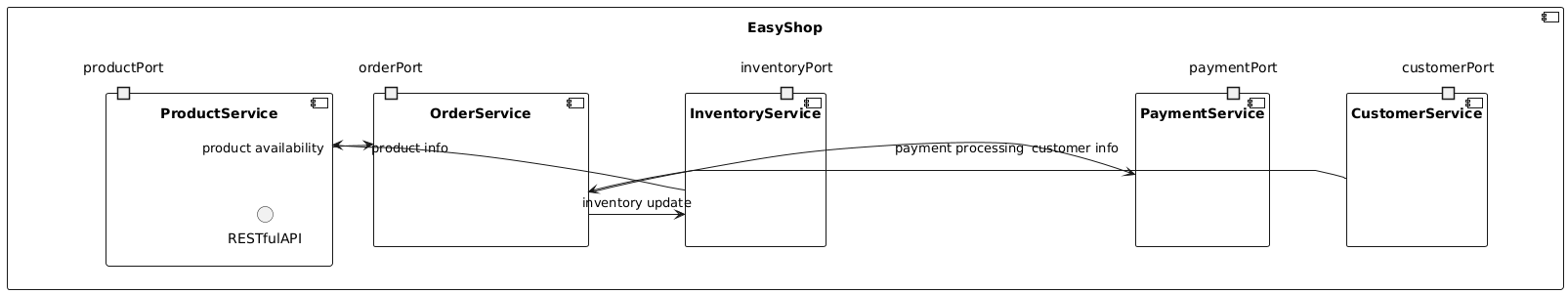
* Each service will expose a web service interface using RESTful APIs, allowing other services to communicate with each other. For example:
* The Product Service will expose a RESTful API to retrieve product information, which can be consumed by the Order Service to display product details to customers.
* The Order Service will expose a RESTful API to process payments, which can be consumed by the Payment Service to complete the payment transaction.
* The Inventory Service will expose a RESTful API to update product availability, which can be consumed by the Product Service to reflect the updated availability in the product catalog.

**Benefits:**

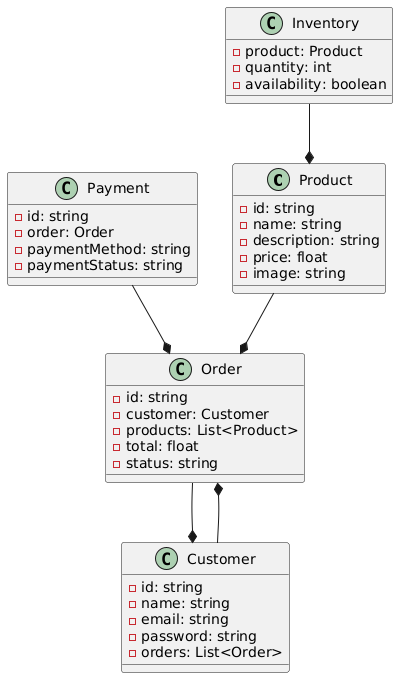
The service-oriented architecture and web services approach provides several benefits, including:

* Loose Coupling: Each service is independent and can be developed, deployed, and scaled independently without affecting other services.
* Reusability: Services can be reused across multiple applications and platforms.
* Flexibility: Services can be easily replaced or updated without affecting other services.
* Scalability: Services can be scaled independently to meet increasing demand.

**Sequence uml diagram**:



Class uml diagram:



**DAY 3:17/07/2024**

PRESENTATION ON SOA:

[Service-Oriented Architecture (SOA).pptx](https://docs.google.com/presentation/d/1kvLxvHvasJlYkxR5n-iNLhAdu-A9xmQc/edit?usp=drive_link&ouid=112441306205811376802&rtpof=true&sd=true)

**DAY4:18/07/2024**

**MVC** (Model-View-Controller)

MVC is a software architectural pattern that separates an application into three interconnected components. This pattern helps to organize code in a way that is easy to maintain, modify, and extend.

**Model:**

The Model represents the data and business logic of the application. It manages the data, performs calculations, and enforces business rules. The Model is responsible for:

* Data storage and retrieval
* Business logic and calculations
* Data validation

**View:**

The View is responsible for rendering the user interface (UI) of the application. It receives input from the user and displays the output. The View is responsible for:

* Rendering the UI
* Displaying data to the user
* Handling user input

**Controller:**

The Controller acts as an intermediary between the Model and View. It receives input from the user, communicates with the Model to perform business logic, and updates the View accordingly. The Controller is responsible for:

* Handling user input
* Communicating with the Model
* Updating the View

**MVC Variants:**

**MVP (Model-View-Presenter)**

In MVP, the Presenter acts as an intermediary between the Model and View. The Presenter receives input from the View, communicates with the Model, and updates the View accordingly.

* **Presenter**: Replaces the Controller, responsible for handling user input, communicating with the Model, and updating the View.
* **Model**: Represents the data and business logic.
* **View:** Responsible for rendering the UI and displaying data to the user.

**MVVM (Model-View-ViewModel)**

In MVVM, the ViewModel acts as an intermediary between the Model and View. The ViewModel exposes the data and functionality of the Model in a form that is easily consumable by the View.

* **ViewModel**: Exposes the data and functionality of the Model, provides data-binding and commanding capabilities.
* **Model**: Represents the data and business logic.
* **View:** Responsible for rendering the UI and displaying data to the user.

**MVA (Model-View-Adapter)**

In MVA, the Adapter acts as an intermediary between the Model and View. The Adapter is responsible for adapting the Model's data and functionality to the View's requirements.

* Adapter: Adapts the Model's data and functionality to the View's requirements.
* Model: Represents the data and business logic.
* View: Responsible for rendering the UI and displaying data to the user.

**MVI (Model-View-Intent)**

In MVI, the Intent acts as an intermediary between the Model and View. The Intent represents the user's intention, such as clicking a button or submitting a form.

* Intent: Represents the user's intention, such as clicking a button or submitting a form.
* Model: Represents the data and business logic.
* View: Responsible for rendering the UI and displaying data to the user.

**Software design patterns**

Software design patterns are reusable solutions to common problems that arise during the design and development of software systems. They provide a proven, standardized approach to solving a specific design problem, making it easier to develop maintainable, flexible, and scalable software systems.

**Types of Design Patterns:**

* Creational Patterns: Deal with object creation mechanisms, such as singleton, factory, and abstract factory.
* Structural Patterns: Concerned with the composition of objects and classes, such as adapter, bridge, and composite.
* Behavioral Patterns: Focus on the interactions between objects, such as observer, strategy, and template method.

**Benefits of Design Patterns**:

* **Improved Code Quality**: Design patterns promote modular, flexible, and reusable code.
* **Faster Development**: By using established patterns, developers can save time and effort.
* **Easier Maintenance**: Design patterns make it easier to understand and modify existing code.
* **Improved Communication:** Design patterns provide a common language and understanding among developers.

**Common Design Patterns:**

1. **Singleton Pattern**: Ensures a single instance of a class is created.
2. **Factory Pattern**: Provides a way to create objects without specifying the exact class of object.
3. **Observer Pattern**: Allows objects to notify other objects of changes to their state.
4. **Strategy Pattern:** Enables algorithms to be selected at runtime.
5. **Template Method Pattern**: Defines a skeleton of an algorithm, allowing subclasses to customize it.
6. **MVC Pattern**: Separates an application into three interconnected components: Model, View, and Controller.
7. **Decorator Pattern**: Allows objects to add additional responsibilities to an existing object.
8. **Adapter Pattern**: Converts the interface of one class to match the interface of another class.

**How to Apply Design Patterns**:

* Identify the Problem: Recognize the design problem you're trying to solve.
* Choose the Right Pattern: Select the most suitable design pattern for the problem.
* Understand the Pattern: Study the pattern, its components, and its relationships.
* Implement the Pattern: Apply the pattern to your code, following best practices and principles.
* Refactor and Test: Refactor your code to ensure it's maintainable and test it thoroughly.

**Cloud Computing**:

Cloud computing is a model of delivering computing services over the internet, where resources such as servers, storage, databases, software, and applications are provided as a service to users on-demand. This allows users to access and use computing resources on a pay-as-you-go basis, without the need to manage and maintain the underlying infrastructure.

**Key Characteristics of Cloud Computing:**

1. **On-demand self-service**: Users can provision and de-provision resources as needed, without requiring human intervention.
2. **Broad network access**: Resources are accessible over the internet, from any device, anywhere in the world.
3. **Resource pooling**: Resources are pooled together to provide a multi-tenant environment, where resources can be dynamically allocated and re-allocated based on demand.
4. **Rapid elasticity**: Resources can be quickly scaled up or down to match changing business needs.
5. **Measured service**: Users only pay for the resources they use, rather than provisioning for peak capacity.

**Cloud Service Models:**

**Infrastructure as a Service (IaaS):**

* Provides virtualized computing resources, such as servers, storage, and networking.
* Users have full control over the infrastructure, but are responsible for managing and maintaining it.
* Examples: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP)

**Platform as a Service (PaaS):**

* Provides a complete development and deployment environment for applications, including tools, libraries, and infrastructure.
* Users have control over the application and data, but not the underlying infrastructure.
* Examples: Heroku, Google App Engine, Microsoft Azure App Service

**Software as a Service (SaaS):**

* Provides software applications over the internet, eliminating the need for users to install, configure, and maintain software on their own devices.
* Users have limited control over the application and data, but can customize the user interface and experience.
* Examples: Salesforce, Microsoft Office 365, Google Workspace (formerly G Suite)

**Benefits of Cloud Computing:**

* Scalability: Quickly scale up or down to match changing business needs.
* Cost savings: Reduce capital and operational expenses by only paying for resources used.
* Increased flexibility: Access resources from anywhere, on any device, at any time.
* Reliability: Built-in redundancy and disaster recovery capabilities ensure high uptime and availability.
* Security: Cloud providers typically have advanced security measures in place to protect user data.

**Docker:**

Docker is a containerization platform that allows developers to package, ship, and run applications in containers. Containers are lightweight and portable, providing a consistent and reliable way to deploy applications across different environments.

**Key Concepts:**

* **Images**: A Docker image is a lightweight, standalone, and executable package that includes everything an application needs to run, such as code, libraries, and dependencies.
* **Containers:** A Docker container is a runtime instance of an image, providing a isolated environment for the application to run in.
* **Volumes**: A Docker volume is a directory that is shared between the host machine and a container, allowing data to be persisted even after the container is deleted.
* **Ports**: Docker containers can expose ports to the host machine, allowing incoming requests to be routed to the container.

**Benefits of Docker**:

* **Lightweight**: Containers are much lighter than virtual machines, making them faster to spin up and down.
* **Portable:** Docker containers are highly portable, allowing developers to deploy applications across different environments with minimal modifications.
* **Isolated**: Containers provide a high degree of isolation, ensuring that applications running in different containers do not interfere with each other.
* **Efficient**: Docker containers use fewer resources than virtual machines, making them more efficient in terms of CPU, memory, and storage.

**Docker Architecture**:

* **Docker Client**: The Docker client is the command-line interface used to interact with the Docker daemon.
* **Docker Daemon**: The Docker daemon is the background process that manages containers and provides the runtime environment for containers.
* **Docker Hub**: Docker Hub is a registry of Docker images, providing a central location for developers to share and discover images.

**Kubernetes:**

Kubernetes (also known as K8s) is an open-source container orchestration system for automating the deployment, scaling, and management of containerized applications. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation (CNCF).

**Key Concepts:**

* **Cluster**: A group of machines, called nodes, that run Kubernetes components and are used to deploy and manage applications.
* **Node:** A machine in the cluster that runs Kubernetes components and containers.
* **Pod**: The basic execution unit in Kubernetes, consisting of one or more containers.
* **ReplicaSet**: Ensures a specified number of replicas (identical pods) are running at any given time.
* **Deployment:** Manages the rollout of new versions of an application, including rolling updates and rollbacks.
* **Service**: Provides a network identity and load balancing for accessing applications.
* **Namespace**: A logical partitioning of resources, providing isolation and management of applications.

**Kubernetes Components:**

* API Server: The central management point for the cluster, providing a RESTful API for interacting with the cluster.
* Controller Manager: Runs and manages control plane components, such as the replication controller and deployment controller.
* Scheduler: Responsible for scheduling pods on nodes based on resource availability and constraints.
* etcd: A distributed key-value store used for storing cluster state and configuration.
* kubelet: An agent running on each node, responsible for managing pods and containers.

**Kubernetes Benefits:**

* Scalability: Automatically scales applications based on demand.
* High Availability: Ensures applications are always available, even in the event of node failures.
* Flexibility: Supports a wide range of container runtimes, frameworks, and languages.
* Automation: Automates deployment, scaling, and management of applications.
* Extensibility: Provides a rich ecosystem of extensions and plugins for customizing and integrating with other systems.

**Kubernetes Deployment Strategies**:

* Rolling Update: Gradually replaces old pods with new ones, ensuring minimal downtime.
* Recreate: Deletes old pods and creates new ones, suitable for stateless applications.
* Blue-Green: Deploys a new version of an application alongside the old one, allowing for quick rollbacks.