Unit-4

Component-Based Software Design

Overview

4.1 Introduction to component-based development
4.2 Components, interfaces, and contracts
4.3 Component composition, integration, and communication
4.4 Designing reusable and maintainable components
4.5 Component lifecycle management and versioning
4.6 Component-based design in large-scale systems

Component based software Design(CBD) or CBSE(Component Based Software Engineering)

3

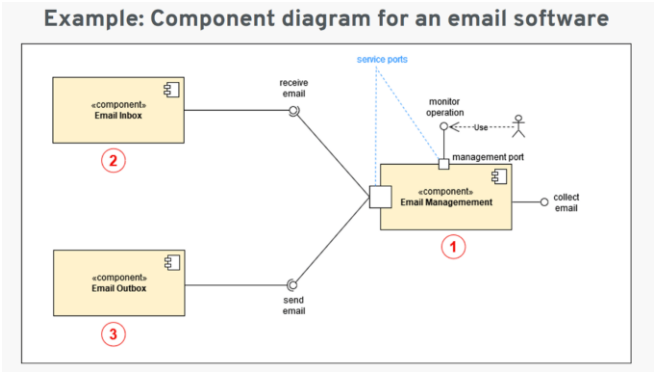
Component

- Is a modular, portable, replaceable and reusable set of well-defined functionality that encapsulates it's implementation and exporting it as a higher-level interface.
- Is a s/w object, intended to interact with other components, encapsulating certain functionality certain functionality or a set of functionalities. It has an obviously defined interface and confirms to a recommended behavior common to all components within an architecture.

4

Component-Based Development (CBD)

- CBD is a software engineering approach that emphasizes building software systems by assembling pre-existing, reusable components.

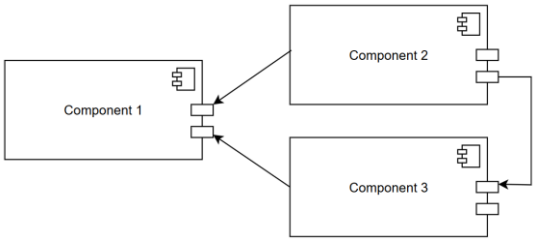


5

Component-Based Development (CBD)..

Key Characteristics:

- Modularity
- Reusability
- Extensibility
- Separation of concerns



6

Component-Based Development (CBD)..

Benefits:

- Faster development cycles
- Improved maintainability
- Enhanced scalability
- Reduced development costs

7

Component-Based Development (CBD)..

- **Objectives of CBD**
  - **Encapsulation**:
    - Hide internal workings to expose only essential functionalities.
  - **Reuse**:
    - Leverage existing components to minimize redundancy.
  - **Interoperability**:
    - Ensure seamless interaction between components.
  - **Composability**:
    - Enable developers to combine components into larger systems easily.

8

Component-Based Development (CBD)..

Core Concepts of CBD

- **Component**:
  - A modular and deployable unit of software.
- **Interface**:
  - Defines how components communicate with each other.
- **Contract**:
  - Specifies the rules and obligations for component interactions.

9

Component-Based Development (CBD)..

Fig: Component

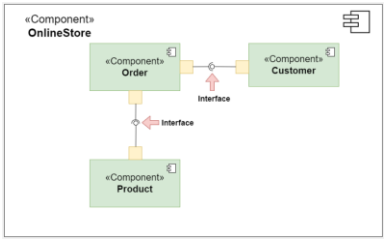


Fig: Interface(Provided and Required Interface)

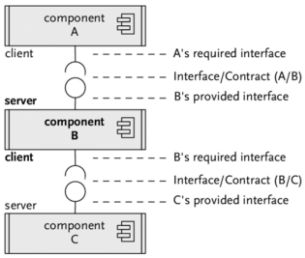


Fig: Provided and Required Interface

10

Component-Based Development (CBD)..

Relationship

- Depict the connections and dependencies between components and interfaces.
- Symbol: Lines and arrows.
  - **Dependency (dashed arrow):** Indicates that one component relies on another.
  - **Association (solid line):** Shows a more permanent relationship between components.
  - **Assembly connector:** Connects a required interface of one component to a provided interface of another.
- **Function**: Visualize how components interact and depend on each other, highlighting communication paths and potential points of failure.

11

Component-Based Development (CBD)..

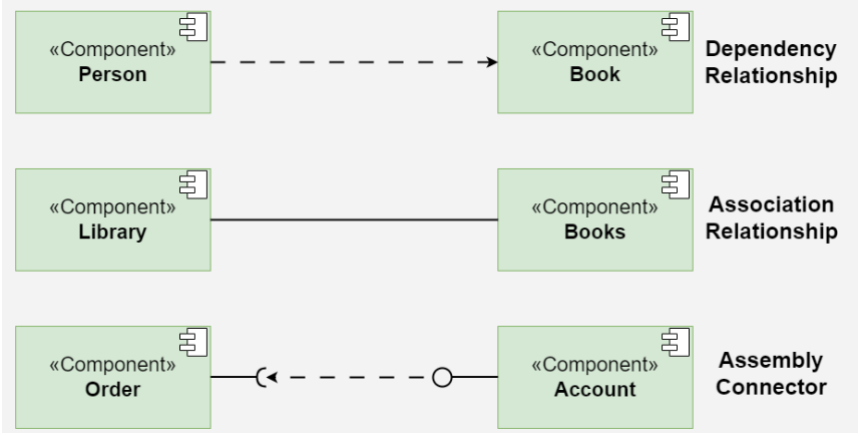


Fig: Relationship in CBD

12

Component-Based Development (CBD)..

Ports:

- Represent specific interaction points on the boundary of a component where interfaces are provided or required.

- **Symbol**: Small squares on the component boundary.
- **Function**: Allow for more precise specification of interaction points, facilitating detailed design and implementation.

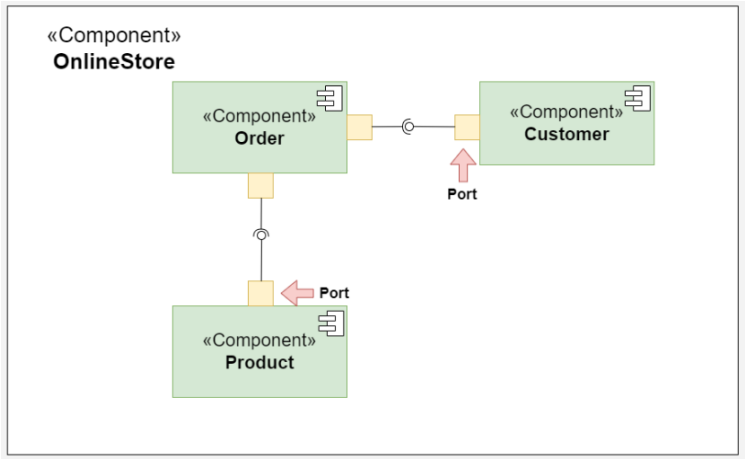


Fig: Ports in CBD

13

Component-Based Development (CBD)..

Artifacts:

- Represent physical files or data that are deployed on nodes.

- **Symbol**: Rectangles with the artifact stereotype («artifact»).
- **Function**: Show how software artifacts, like executables or data files, relate to the components.

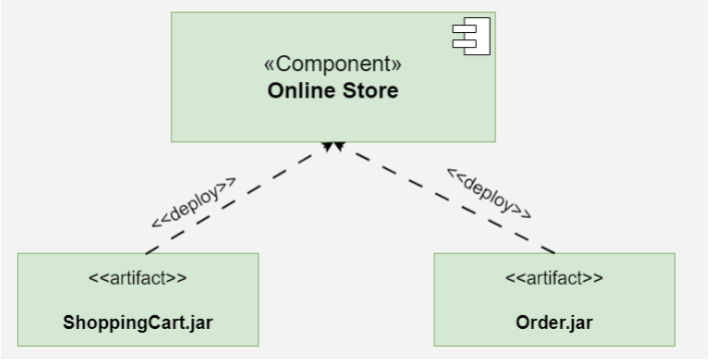


Fig: Artifacts

14

Component-Based Development (CBD)..

Node:

- Represent physical or virtual execution environments where components are deployed.

- **Symbol**: 3D boxes.
- **Function**: Provide context for deployment, showing where components reside and execute within the system's infrastructure.

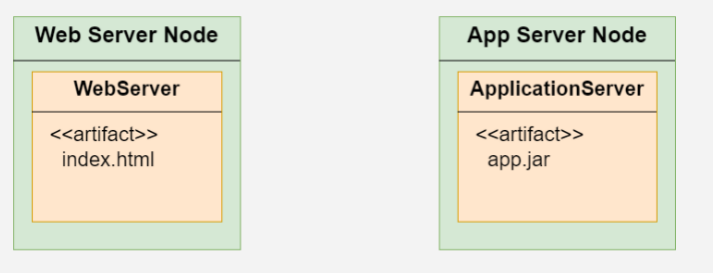


Fig: Nodes

15

Element	Symbol/Notation	Explanation
Component	«component» Name	Symbol for modules in a system (interaction and communication occur via interfaces).
Package	package	A package combines multiple elements in a system (e.g. classes, components, interfaces) into a group.
Artifact	<<artifact>> manual.jar	Artifacts are physical units of information (e.g. source code, *.exe* files, scripts, documents) that are created or required during a system's development process or at runtime.
Provided interface	)	Symbol for one or more clearly defined interfaces that provide functions, services or data to the outside (the open semi-circle is also called a socket).
Required interface	—○	Symbol for a required interface that receives functions, services or data from the outside (the circle-with-stick notation is also referred to as a lollipop).
Port	□	This symbol indicates a separate point of interaction between a component and its environment.
Relationship	———	Lines act as connectors and show relationships between components.
Dependency	------>	This special connector indicates a dependency between two parts of a system (not always explicitly shown).

16

Steps to create a component-based diagram:

- **Step 1: Identify the System Scope and Requirements**
  - **Understand the system**: Gather all necessary information about the system, including its purpose, functionality, constraints, and requirements.
  - **Define the boundaries**: Clearly outline what will be included in the diagram and what will be excluded. This helps in focusing on the relevant components.

17

Steps to create a component-based diagram:

- **Step 2: Identify and Define Components**
  - **List components**: Identify all the major components that constitute the system. These could be modules, services, or subsystems.
  - **Detail functionality**: For each component, define its responsibilities and the specific functionalities it provides.
  - **Encapsulation**: Ensure that each component is self-contained and encapsulates a specific set of functionalities.

18

Steps to create a component-based diagram:

- **Step 3: Identify Provided and Required Interfaces**
  - **Provided Interfaces**: Determine the services or functionalities that each component offers to other components.
  - **Required Interfaces**: Identify the services or functionalities that each component needs from other components to operate.
  - **Define Interfaces**: Clearly specify the operations (methods, inputs, outputs) included in each interface.

19

Steps to create a component-based diagram:

- **Step 4: Identify Relationships and Dependencies**
  - **Determine connections**: Identify how components interact with each other. This includes communication protocols, data flow, and control flow.
  - **Specify dependencies**: Outline which components depend on others to function. This helps in understanding the system's structure and potential failure points.

20

Steps to create a component-based diagram:

- **Step 5: Identify Artifacts**
  - **List artifacts**: Identify the physical or logical artifacts (e.g., files, libraries, executables) associated with each component.
  - **Map artifacts**: Determine how these artifacts are deployed, used, or generated by the components.
- **Step 6: Identify Nodes**
  - **Execution environments**: Identify the physical or virtual nodes (e.g., servers, containers, devices) where the components will be deployed.
  - **Define nodes**: Specify the hardware, software, or infrastructure requirements for each node.

21

Steps to create a component-based diagram:

- **Step 7: Draw the Diagram**
  - **Use a UML tool**: Utilize a UML diagramming tool like Lucidchart, Microsoft Visio, Enterprise Architect, or any other preferred software.
  - **Draw components**: Represent each component as a rectangle with the «component» stereotype.
  - **Draw interfaces**: Use lollipop symbols (circles) for provided interfaces and socket symbols (half-circles) for required interfaces.
  - **Connect components**: Use assembly connectors (lines) to link provided interfaces to required interfaces.
  - **Add artifacts**: Represent artifacts as rectangles with the «artifact» stereotype and associate them with the appropriate components.
  - **Draw nodes**: Represent nodes as 3D boxes and place the components and artifacts within these nodes to show deployment.

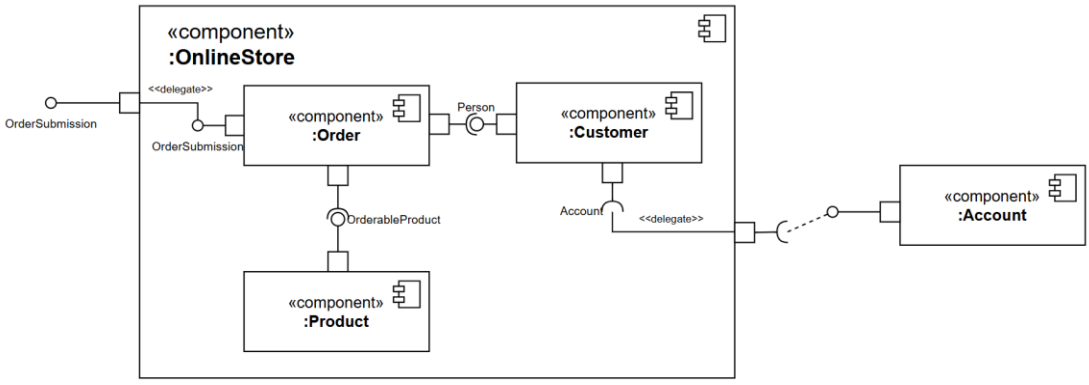
22

Steps to create a component-based diagram:

- **Step 8: Review and Refine the Diagram**
  - **Validate accuracy**: Ensure that all components, interfaces, relationships, and dependencies are accurately represented.
  - **Seek feedback**: Share the diagram with stakeholders, team members, or domain experts to validate its correctness and completeness.
  - **Refine as needed**: Make adjustments based on feedback to improve clarity, accuracy, and alignment with system requirements.

23

Example



24

What is a Component?

- A software unit that encapsulates functionality and data, conforming to a predefined interface.

Characteristics:

- Self-contained
- Replaceable
- Reusable
- Independent of deployment environment

Example:

Login module, Payment gateway, etc.

25

What is an Interface?

- A specification that defines the operations a component offers.
- **Role in CBD**:
  - Acts as a contract between components.
  - Abstracts implementation details.
- **Types of Interfaces**:
  - Provided Interface: Services the component offers.
  - Required Interface: Services the component depends on.
- **Example**:

  An API or method signatures.

26

What is a Contract?

- A formal agreement between components specifying the expectations and obligations.
- **Components of a Contract**:
  - **Preconditions**: What must be true before the operation executes.
  - **Postconditions**: What will be true after the operation executes.
  - **Invariants**: Conditions that must remain true throughout the component's lifecycle.
- **Importance**:
  - Ensures reliability and predictable behavior.
  - Facilitates component testing and validation.

27

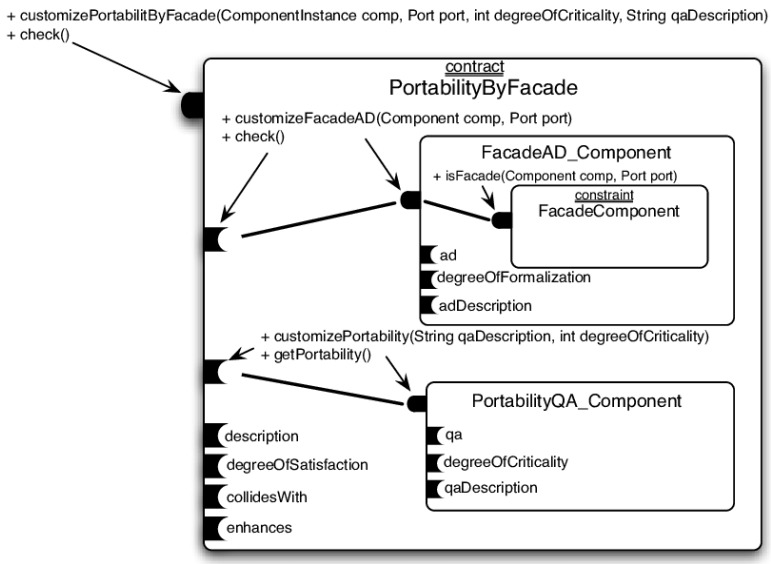


Fig: Contract

28

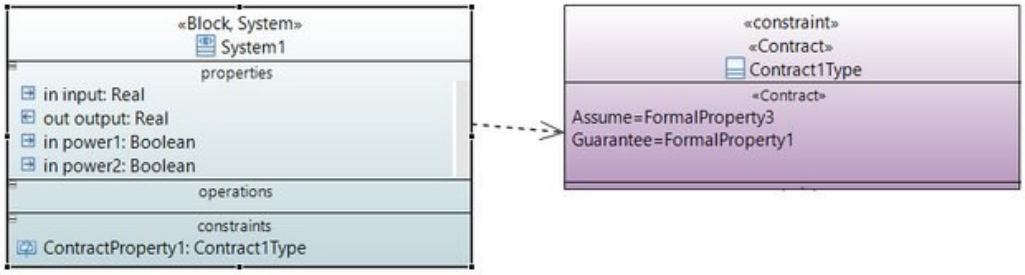


Fig: Contract

29

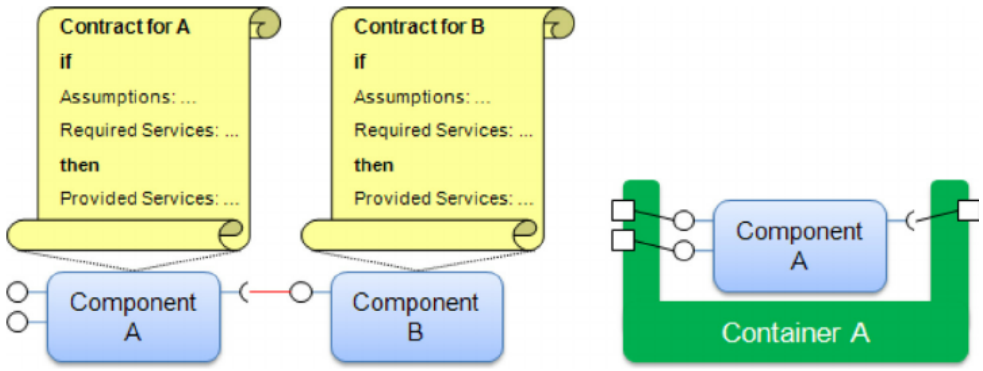
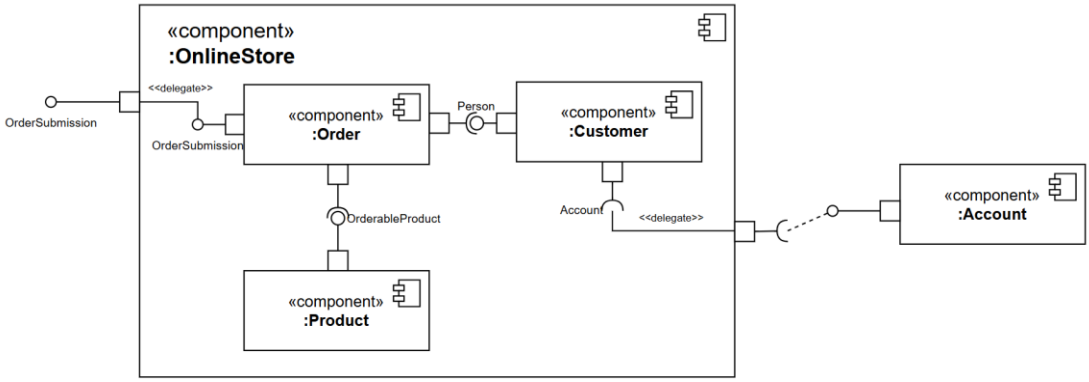


Fig: Contract

31

Example



- A delegate is a mechanism used to connect a provided or required interface of a component to another interface or port.
- It acts as a link to delegate requests or services from one interface to another.

32

Advantages of CBD Using Components, Interfaces, and Contracts

- **Improved Reusability**:
  Shared functionality reduces development efforts.
- **Enhanced Flexibility**:
  Plug-and-play architecture supports system changes.
- **Greater Reliability**:
  Contracts enforce predictable interactions.
- **Cost-Effectiveness**:
  Reusable components lower the development costs.

33

CBD in Practice..

- **Industry Examples**:
  - **E-Commerce Systems**: Payment gateways, Inventory management.
  - **Web Applications**: Authentication modules, Analytics services.
  - **Enterprise Software**: ERP, CRM.
- **Tools and Frameworks**:
  - JavaBeans, COM+, .NET Framework, Microservices architectures.

34

Challenges in CBD

- **Integration Complexity**: Ensuring compatibility between components.
- **Dependency Management**: Handling interdependencies without tight coupling.
- **Versioning Issues**: Managing updates and backward compatibility.
- **Testing and Debugging**: Difficulties in isolating component-level issues.

35

Component | Interface | Contract

Aspect	Software Component	Interface	Contract
Definition	A modular, reusable, and deployable part of a software system that encapsulates functionality.	A set of defined methods, properties, or behaviors that a component exposes.	A formal agreement or specification that defines how components interact.
Purpose	To encapsulate and provide specific functionality in a reusable manner.	To define how a component can be interacted with or used.	To ensure consistent and predictable interactions between components.
Focus	Implementation of functionality.	Communication and interaction points.	Rules, obligations, and expectations for interaction.
Example	A login module, payment gateway, or database connector.	Methods like login(), processPayment(), or fetchData().	Rules like "login() must return a token" or "fetchData() must handle errors gracefully."
Relationship	A component may implement one or more interfaces.	An interface defines the interaction points for a component.	A contract governs the interaction between components via interfaces.
Flexibility	Can be replaced or updated as long as it adheres to the same interface and contract.	Can be extended or modified, but changes may affect dependent components.	Must remain stable to ensure compatibility between components.
Level of Abstraction	Concrete implementation.	Abstraction of interaction points.	Abstraction of rules and expectations for interaction.

39

Component Composition, Integration, and Communication

- **Component Composition**
  - Combining multiple components to create a larger, more complex system or application.
- **Types of Composition**:
  - **Aggregation**: Components coexist but retain independence.
  - **Inheritance**: Extending existing components with new behavior.
  - **Dependency Injection**: Components provided as dependencies.

42

Principles of Component Composition

- **Loose Coupling**:
  - Minimize dependencies between components.
- **High Cohesion**:
  - Ensure components perform specific, focused tasks.
- **Interface-Based Design**:
  - Define interactions through standardized interfaces.
- **Encapsulation**:
  - Hide internal workings to expose only necessary functionality.

43

Component Composition, Integration, and Communication

- **Component Integration**
  - The process of connecting components to form a functional system.
- **Integration Approaches**:
  - **Static Integration**: Predefined at compile time (e.g., libraries).
  - **Dynamic Integration**: Performed at runtime (e.g., plugins, microservices).

44

Challenges in Integration

- **Compatibility Issues**:
  - Mismatched data formats or protocols.
- **Versioning Conflicts**:
  - Different component versions causing incompatibilities.
- **Dependency Management**:
  - Complex interdependencies leading to errors.
- **Testing Difficulties**:
  - Validating integrated components in various environments.

45

Component Composition, Integration, and Communication

- **Communication Between Components**
  - The process by which components interact and exchange information.
- **Communication Mechanisms**:
  1. **Direct Method Calls**: Tight coupling, typically within a monolith.
  2. **Message Passing**: Asynchronous, decoupled communication (e.g., queues, events).
  3. **Shared Data**: Accessing common resources (e.g., databases).
  4. **Middleware**: Using brokers like REST APIs, SOAP, or RPC.

46

1. Direct Method Calls (Tight Coupling)..

```
class PaymentService {
   public void processPayment(double amount) {
      System.out.println("Processing payment of $" +
amount);
   }
}

class UserService {
   private PaymentService paymentService = new
PaymentService();

   public void makePayment(double amount) {
      paymentService.processPayment(amount);
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      UserService userService = new UserService();
      userService.makePayment(100.0); // Direct
method call
   }
}
```

48

2. Message Passing (Asynchronous, Decoupled Communication)..

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

class PaymentService {
   private BlockingQueue<String> queue;

   public PaymentService(BlockingQueue<String> queue) {
      this.queue = queue;
   }

   public void processPayment(double amount) {
      System.out.println("Processing payment of $" + amount);
      queue.offer("Payment processed for $" + amount); // Send message to queue
   }
}

class NotificationService implements Runnable {
   private BlockingQueue<String> queue;

   public NotificationService(BlockingQueue<String> queue) {
      this.queue = queue;
   }

   @Override
   public void run() {
      while (true) {
         try {
            String message = queue.take(); // Listen to queue
            System.out.println("Sending email: " + message);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}

public class Main {
   public static void main(String[] args) {
      BlockingQueue<String> queue = new LinkedBlockingQueue<>();
      PaymentService paymentService = new PaymentService(queue);
      NotificationService notificationService = new NotificationService(queue);

      // Start NotificationService in a separate thread
      new Thread(notificationService).start();

      // Simulate payment processing
      paymentService.processPayment(100.0);
   }
}
```

50

3. Shared Data (Accessing Common Resources)..

```
import java.util.HashMap;
import java.util.Map;

class Database {
    private Map<Integer, String> userData = new HashMap<>();

    public void saveUser(int userId, String userDetails) {
        userData.put(userId, userDetails);
    }

    public String getUserDetails(int userId) {
        return userData.get(userId);
    }
}

class UserService {
    private Database database;

    public UserService(Database database) {
        this.database = database;
    }

    public void addUser(int userId, String userDetails) {
        database.saveUser(userId, userDetails);
    }
}
```

```
class ReportingService {
    private Database database;

    public ReportingService(Database database) {
        this.database = database;
    }

    public void generateReport(int userId) {
        String userDetails = database.getUserDetails(userId);
        System.out.println("Report for user: " + userDetails);
    }
}

public class Main {
    public static void main(String[] args) {
        Database database = new Database();
        UserService userService = new UserService(database);
        ReportingService reportingService = new ReportingService(database);

        userService.addUser(1, "John Doe");
        reportingService.generateReport(1);
    }
}
```

52

4. Middleware (Using Brokers like REST APIs, SOAP, or RPC)..

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

@SpringBootApplication
@RestController
public class UserService {
    @GetMapping("/getUserDetails/{userId}")
    public String getUserDetails(@PathVariable int userId) {
        return "User Details for ID " + userId;
    }

    public static void main(String[] args) {
        SpringApplication.run(UserService.class, args);
    }
}
```

54

4. Middleware (Using Brokers like REST APIs, SOAP, or RPC)..

```
// BankService.proto (gRPC definition)
// Define a service with a method ProcessPayment
service BankService {
  rpc ProcessPayment(PaymentRequest) returns (PaymentResponse);
}

// PaymentService (Client)
public class PaymentService {
  public void processPayment(double amount) {
    // Make a gRPC call to BankService
    BankServiceGrpc.BankServiceBlockingStub stub = BankServiceGrpc.newBlockingStub(channel);
    PaymentResponse response = stub.processPayment(PaymentRequest.newBuilder().setAmount(amount).build());
    System.out.println("Payment response: " + response.getMessage());
  }
}
```

55

Overview

Approach	Example	Characteristics
Direct Method	UserService calls PaymentService, processPayment(amount)	Tightly coupled, synchronous, monolithic.
Message Passing	PaymentService sends a message to orderProcessQueue	Decoupled, asynchronous, scalable.
Shared Data	UserService writes to DB; ReportingService reads from DB	Coupled through shared resources, synchronous or asynchronous.
Middleware	REST API: /getUserDetails; RPC: PaymentService calls BankService	Decoupled, language-agnostic, scalable, synchronous or asynchronous.

56

Comparison

Aspect	Direct Method Calls	Message Passing	Shared Data	Middleware
Coupling	Tightly coupled	Loosely coupled	Coupled through shared resources	Loosely coupled
Communication Style	Synchronous	Asynchronous	Synchronous or asynchronous	Synchronous or asynchronous
Scalability	Limited (monolithic)	Highly scalable	Limited by shared resource	Highly scalable
Complexity	Low	Moderate	Low to moderate	Moderate to high
Use Case	Monolithic applications	Event-driven systems, microservices	Systems with shared databases or files	Distributed systems, microservices
Example	Direct function calls within a monolith	Message queues (e.g., RabbitMQ, Kafka)	Shared database or file system	REST APIs, SOAP, gRPC

57

Component Communication Models

- **Synchronous Communication**: Immediate response required (e.g. function call, RPC)
  - E.g.: HTTP Requests/Responses.
  - Pros: Predictable flow.
  - Cons: Blocks resources during interaction.
- **Asynchronous Communication**: Non-Blocking communication (e.g. Message Request)
  - E.g.: Event-driven systems (e.g., Kafka, RabbitMQ).
  - Pros: Non-blocking, scalable.
  - Cons: Requires handling of eventual consistency.

58

Best Practices for Integration and Communication

- Use **Standardized Protocols**: REST, GraphQL, gRPC, etc.
- Implement **Error Handling Mechanisms**: Graceful failure and retries.
- Adopt **Service Contracts**: Formalize interaction rules.
- Ensure **Security**: Authentication and encryption during communication.

59

Why Reusable components?

- Promotes efficiency, scalability and long term maintainability
- Reduces development cost and redundancy.

62

Characteristics of Reusable Components

1. **Modularity**
   - Self-contained units that perform specific tasks.
   - Easy to plug into different systems.
2. **Loose Coupling**
   - Minimal dependency on other components.
   - Ensures flexibility for future changes.
3. **High Cohesion**
   - A component performs a single, well-defined task.
   - Increases clarity and reusability.
4. **Configurable**
   - Parameterized or adaptable to fit multiple use cases.
5. **Standardized Interfaces**
   - Well-documented inputs, outputs, and behaviors.
   - Use industry standards (e.g., REST APIs, message queues).

63

Principles for Designing Maintainable Components

1. Separation of Concerns (SoC)
- Divide functionality into distinct sections with specific purposes.
- Example: Separate business logic, UI, and data access.

2. Encapsulation
- Hide internal implementation details.
- Expose only what is necessary through interfaces.

3. Abstraction
- Define general-purpose interfaces or abstract classes.
- Allow multiple implementations for flexibility.

4. Dependency Injection (DI)
- Use DI to reduce tight coupling and improve testability.

5. Consistency and Naming Conventions
- Follow coding standards for easier understanding and maintenance.

6. Error Handling
- Design robust error handling mechanisms.
- Prevent unexpected failures during integration.

64

Strategies for Reusability

1.Design for Generalization
- Avoid hardcoding specific details.
- Use parameters, configuration files, or dependency injection.

2.Use of Interfaces and Contracts
- Define clear contracts (e.g., preconditions, postconditions).
- Ensure components adhere to these contracts for reliable integration.

3.Create Libraries or Frameworks
- Combine reusable components into libraries.
- Example: UI component libraries, utility libraries.

4.Document and Test
- Provide clear documentation for usage and integration.
- Write unit tests to validate component behavior.

5.Versioning and Backward Compatibility
- Use version control for components.
- Ensure updates do not break existing systems.

65

Best Practices for Designing Components

1.Keep Components Small and Focused
- Follow the Single Responsibility Principle (SRP).

2.Minimize Dependencies
- Reduce reliance on external modules.

3.Write Clean and Readable Code
- Use consistent naming and coding standards.

4.Make Components Testable
- Ensure components can be unit-tested in isolation.

5.Ensure Scalability
- Design components to handle increased workload without major changes.

66

Benefits of Reusable and Maintainable Components

1.Improved Development Speed
- Faster delivery by reusing existing components.

2.Reduced Cost
- Saves time and effort in building functionality from scratch.

3.Enhanced Quality
- Reusable components are tested and proven.

4.Easier Maintenance
- Modular design simplifies debugging, testing, and updates.

5.Greater Scalability
- Components can adapt to new requirements or systems.

67

Real-world Example

- **UI Component Libraries**
  - E.g.: React, Angular, or Bootstrap components for buttons, modals, etc.
  - Benefit: Easy to reuse in different projects while maintaining consistency.
- **Microservices Architecture**
  - Independent, reusable services for tasks like authentication, payment, or notifications.
  - Benefit: Loose coupling and easy scalability.

68

Component Extension..

Base Component

```
public interface Notification {
    void send(String message);
}
```

Extended Component

```
public class SMSNotification implements Notification {
    public void send(String message) {
        System.out.println("SMS sent: " + message);
    }
}
```

Usage

```
Notification email = new EmailNotification();
email.send("Order Confirmed");

Notification sms = new SMSNotification();
sms.send("Order Shipped");
```

69

Managing Components in Software Architecture

- **Components:** Reusable building blocks of software systems.
- **Lifecycle Management:** Tracking components from creation to retirement.
- **Versioning:** Managing changes and updates while maintaining compatibility.
- **Objective:** Explore techniques for managing the lifecycle and versions of components in software architecture.

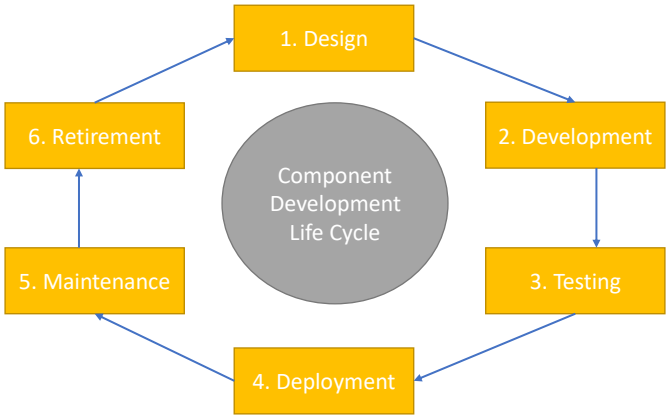
70

Component Lifecycle Overview

- The sequence of stages a component goes through during its existence.
- **Stages:**
  - **1.Design:** Define component requirements and interfaces.
  - **2.Development:** Implement the component.
  - **3.Testing:** Validate functionality and quality attributes.
  - **4.Deployment:** Integrate into the system.
  - **5.Maintenance:** Apply updates, fixes, and enhancements.
  - **6.Retirement:** Decommission and replace if necessary.

71

Component Life Cycle Management



72

Key Practices in Component Lifecycle Management

- **Documentation:**
  - Maintain clear records of functionality, dependencies, and usage.
- **Dependency Management:**
  - Track and manage dependencies to ensure compatibility.
- **Monitoring and Analytics:**
  - Monitor performance and usage.
- **Version Control:**
  - Employ tools to manage changes and track versions.
- **Automation:**
  - Use CI/CD pipelines for consistent updates and testing.

73

Challenges in Component Lifecycle Management

- **Compatibility:**
  - Ensuring backward and forward compatibility.
- **Integration Issues:**
  - Handling updates without breaking the system.
- **Documentation Gaps:**
  - Incomplete records complicate updates and replacements.
- **Dependency Conflicts:**
  - Mismatched versions of dependencies can cause failures.

74

Component Versioning

- The process of assigning unique identifiers to component versions to track changes over time.
- **Importance:**
  - Supports compatibility and integration.
  - Facilitates rollback to previous versions if necessary.
- **Versioning Standards:** Semantic Versioning (e.g., MAJOR.MINOR.PATCH):
  - **MAJOR:** Breaking changes that are incompatible with previous versions. Increment the major version when significant redesigns or overhauls occur, requiring users to adapt to the changes.
  - **MINOR:** Backward-compatible new features. Increment the minor version when new functionality is added without breaking existing features or interfaces.
  - **PATCH:** Backward-compatible fixes. Increment the patch version for bug fixes, security patches, or minor corrections that do not affect existing functionality.

75

Component Versioning

- **E.g.: Version 2.4.1**

  2: Major version—significant redesign.

  4: Minor version—new features added.

  1: Patch version—bug fixes or minor changes.

76

Versioning Strategies

- **Strict Versioning:**
  - Fixed versions; updates require explicit changes.
  - Suitable for systems with strict stability requirements.
- **Floating Versioning:**
  - Automatically uses the latest compatible version.
  - Suitable for agile and rapidly evolving systems.
- **Version Pinning:**
  - Lock dependencies to specific versions to ensure consistency.
- **Backward Compatibility Checks:**
  - Validate updates against older versions to prevent disruptions.

77

Component Evolution and Deprecation

- **Evolution:**
  - Adding features while maintaining compatibility.
  - Techniques: Interfaces with default implementations, feature flags.
- **Deprecation:**
  - Phasing out outdated components or features.
  - **Steps:**
    1. Announce deprecation.
    2. Provide alternatives.
    3. Set a timeline for removal.

78

Tools for Lifecycle Management and Versioning

- **Version Control Systems:** Git, Mercurial.
- **Dependency Management:**
  - Tools: Maven, npm, Gradle.
  - Ensure consistent builds and dependency resolution.
- **CI/CD Pipelines:** Jenkins, GitHub Actions.
  - Automate versioning, testing, and deployment.
- **Monitoring Tools:** Prometheus, New Relic.
  - Track component health and usage.

79

Best Practices for Lifecycle Management and Versioning

- **Lifecycle Management:**
  - Regularly update documentation.
  - Continuously test components in various environments.
  - Implement monitoring and alert systems.
- **Versioning:**
  - Follow semantic versioning.
  - Ensure compatibility through rigorous testing.
  - Communicate version changes to stakeholders.
- **Governance:** Establish policies for updates, deprecation, and retirement.

80

Why Component-Based Design in Large-Scale Systems?

- Benefits of CBD in Large-Scale Systems
  - **Modularity:** Simplifies system complexity.
  - **Scalability:** Components can scale independently.
  - **Maintainability:** Easier debugging and updates.
  - **Faster Development:** Parallel development of components.
  - **Technology Agnostic:** Different components can use different technologies.

82

Characteristics of a Large-Scale System

- Defining Features of Large-Scale Systems
  - **High Availability:** Minimal downtime, fault tolerance.
  - **Scalability:** Ability to handle increasing loads.
  - **Distributed Nature:** Multiple interacting components across networks.
  - **Heterogeneous Technologies:** Integration of various platforms and tools.
  - **Security and Compliance:** Robust authentication, data privacy measures.s

83

Component Composition in Large-Scale Systems

- Structuring Components in Large-Scale Systems
  - **Vertical Composition:**
    - Layers of components built on top of each other.
    - E.g.: ***Presentation Layer → Business Logic Layer → Data Layer.***
      - i.e. UI component calling backend API's.
  - **Horizontal Composition:**
    - Parallel, independent components working together.
    - E.g.: ***Microservices handling different functions.***
      - i.e. Microservices in a service layer interacting with each other.

84

Component Integration in Large-Scale Systems

- **<u>Challenges:</u>**
  - **Complexity in Integration:** Ensuring smooth component interaction.
  - **Version Control:** Managing updates across components.
  - **Performance Overhead:** Potential latency in communication.
  - **Security Risks:** Need for secure component interactions.
- **<u>Best Practices:</u>**
  - **Use Standardized APIs** (RESTful, gRPC, SOAP).
  - **Ensure Proper Documentation** for each component.
  - **Implement Robust Security Measures** (OAuth, JWT, TLS).
  - **Adopt CI/CD Pipelines** for seamless integration.
  - **Monitor Performance Metrics** to optimize scalability.

85

Patterns for Large-Scale Component Design

- Common Design Patterns for Large-Scale Systems
  - **Microservices Architecture:** Small, independent services communicating via APIs.
  - **Service-Oriented Architecture (SOA):** Modular, reusable services for enterprise applications.
  - **Event-Driven Architecture:** Components communicating through event messages.
  - **Layered Architecture:** Organizing components into layers for better manageability.
  - **Domain-Driven Design (DDD):** Structuring components based on business domains.

86

Case Study - Microservices Architecture in Component-Based Design

Case Study on Microservices in Large-Scale Systems

Netflix

- **Problem:** Monolithic architecture faced scalability and maintenance issues.
- **Solution:** Adopted a microservices architecture where each function (e.g., streaming, recommendations, user authentication) was built as an independent service.
- **Implementation:**
  - Used RESTful APIs for communication.
  - Deployed containerized services using Kubernetes.
  - Implemented CI/CD pipelines for continuous updates.
- **Results:**
  - Improved system scalability and resilience.
  - Enabled independent service upgrades and deployments.
  - Enhanced fault tolerance and system reliability.

87