Mapping Models to Code

What is Mapping Models to Code?

- **Definition:** The process of converting conceptual models (e.g., UML diagrams) into executable code.
- **Why is it required?**
  - **Bridges the gap** between abstract design and concrete implementation.
  - **Ensures consistency** between design intentions and the final software product.
  - **Facilitates communication** between designers and developers.
- Improves maintainability and adaptability of the software system.

What is Transformation

- **Definition:**
- The act of systematically **converting one representation (model) into another** (code) while preserving essential information.
- In the context of software engineering, transformation is the systematic process of converting one representation or form of something into another.
- Think of it like translating a book from one language to another - you're taking the essential meaning and ideas and expressing them in a different way.
- In software, this often means turning abstract design models (like diagrams) into actual code that a computer can execute.

Why is transformation needed?

- **Bridging the Gap**:
  - Software design often starts with high-level concepts and models that are easy for humans to understand but can't be directly run by a computer.
  - Transformation fills the gap, taking these models and making them executable.
- **Automation and Efficiency**:
  - Manually translating designs into code is time-consuming and error-prone.
  - Transformation automates much of this process, speeding up development and reducing mistakes.

- **Maintaining Consistency**:
  - As designs evolve, it's crucial that the code stays in sync.
  - Transformation helps ensure the implemented software reflects the latest design decisions.
- **Enabling Change**:
  - Software needs to adapt over time.
  - Transformation makes it easier to update code when the underlying models change, supporting agile development practices.

Transformation Example

- Example:
- Imagine you're designing a basic calculator app. You might start with a simple diagram showing buttons for numbers and operations, and how they connect to the underlying calculation logic. Transformation would take this diagram and generate the actual code that creates the buttons, handles user input, performs calculations, and displays results on the screen.
- In essence: Transformation is like a bridge between the world of ideas and the world of working software, making it possible to turn designs into reality efficiently and accurately.

Transformation Activities

- **Optimization:** Improves the efficiency of generated code by eliminating redundancies or applying performance-enhancing techniques.
  - *Example:* Combining multiple database queries into a single optimized query.

- **Realizing associations:** Implements relationships between classes (e.g., associations, aggregations, compositions) in code.
  - *Example:* Creating foreign key relationships in a database schema based on class associations.

- **Mapping class models to a storage schema:** Converts class structures and attributes into database tables and columns.
  - *Example:* Mapping a Customer class with name and address attributes to a Customers table with corresponding columns.

- **Mapping contracts to exceptions:**
- Translates preconditions and postconditions from models into exception handling mechanisms in code.
- A **contract** defines how a method/class/component should behave.
- It usually includes:
  - Preconditions → What must be true before execution.
  - Postconditions → What must be true after execution.
  - Invariants → Conditions that should always hold true.

  When a contract is violated, the program cannot continue safely.In such cases, the violation is reported using an exception.
- *Example:* Generating code that throws an exception if a method's input parameters violate a specified precondition.

Types of Transformation:

- Transformation in software development encompasses a range of techniques that help us modify and adapt our software throughout its creation, maintenance, and evolution.

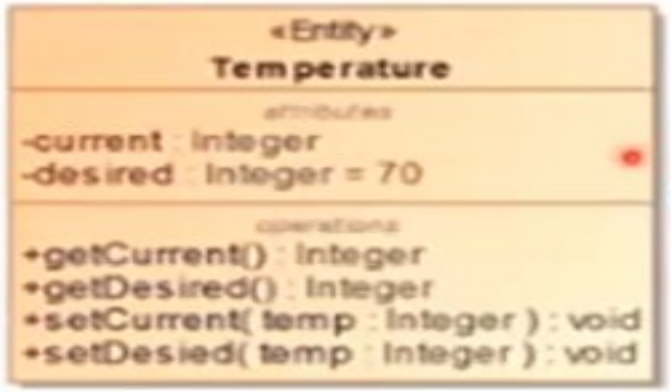
1. Model Transformations: Changing Perspectives

Explanation	Sample Java Code Scenario: Transforming a UML class diagram into Java code

Convert one type of model into another, often to suit different stages of development or target different platforms.
• Example 1: You start with a high-level UML class diagram outlining the structure of your software. A model transformation could convert this into a more detailed Entity-Relationship diagram, ready for database design.
• Example 2: You have a platform-independent model of your application. Model transformations help tailor this to specific platforms (like web or mobile) by generating platform-specific models or code snippets.
	Code Before Transformation (UML Diagram):
	---------- 	
	| Person |
	---------- 	
	| - name: String |
	| - age: int |
	---------- 	
	| + getName(): String |
	| + getAge(): int |
	---------- 	
	Code After Transformation (Java Code):
	Java
	public class Person {
	private String name;
	private int age;
	
	public String getName() {
	return name;
	}
	
	public int getAge() {
	return age;
	}
	}

DCD(Design Class Diagrams)

- Example: Basic class translation




```
public class Temperature {
    int current;
    int desired = 70;

    public int getCurrent(){
        return current;
    }

    public int getDesired(){
        return desired;
    }

    public void setCurrent(int temp){
        current = temp;
    }

    public void setDesired(int temp){
        desired = temp;
    }
}
```

2. Refactoring: Improving Code's Inner Workings

Explanation	Sample Java Code Scenario: Extracting a method to improve code readability and reusability.

Restructure existing code to make it cleaner, more readable, and easier to maintain, without altering its external behavior. Think of it like reorganizing a messy room - everything is still there, just better arranged.
 	
Example 1: You have a long, complex method. Refactoring might involve extracting smaller, more focused methods from it, improving clarity and reusability.
 	
Example 2: You notice duplicate code in several places. Refactoring can help consolidate this into a single, shared function or class, reducing redundancy and making future changes easier.
	Code Before Transformation:
	Java
	public class Calculator {
	public int calculateArea(int length, int width) {
	// ... (complex calculations involving length and width)
	return area;
	}
	}
	Code After Transformation:
	Java
	public class Calculator {
	public int calculateArea(int length, int width) {
	int area = performComplexCalculations(length, width);
	return area;
	}
	
	private int performComplexCalculations(int length, int width) {
	// ... (complex calculations)
	return area;
	}}

3. Forward Engineering: From Design to Reality

Explanation	Sample Java Code Scenario: Generating Java code from a database schema.

The classic transformation: generating code from models or designs. This is often the core activity in the initial development phase.
• Example 1: You have a UML class diagram defining your classes and their relationships. Forward engineering tools can automatically generate the corresponding code structure in your chosen programming language (e.g., Java classes from the UML diagram).
• Example 2: You've designed a user interface using a visual tool. Forward engineering can translate this design into the actual HTML, CSS, and JavaScript code that brings the interface to life in a web browser.
	Code Before Transformation (Database Schema):
	Table: Products
	Columns: id (int, primary key), name (varchar), price (decimal)
	Code After Transformation (Java Code):
	Java
	public class Product {
	private int id;
	private String name;
	private BigDecimal price;
	
	// ... (getters and setters)
	}

Example

Before transformation =>

```
class Library {
    public void issueBook(String bookName, String studentName) {
        System.out.println("Book " + bookName + " issued to " + studentName);
        System.out.println("Update database for issued book...");
        System.out.println("Send email notification...");
    }

    public void returnBook(String bookName, String studentName) {
        System.out.println("Book " + bookName + " returned by " + studentName);
        System.out.println("Update database for returned book...");
        System.out.println("Send email notification...");
    }
```

```
class Library {
    public void issueBook(String bookName, String studentName) {
        System.out.println("Book " + bookName + " issued to " + studentName);
        performCommonTasks("issued", bookName, studentName);
    }

    public void returnBook(String bookName, String studentName) {
        System.out.println("Book " + bookName + " returned by " + studentName);
        performCommonTasks("returned", bookName, studentName);
    }

    private void performCommonTasks(String action, String bookName, String studentName) {
        System.out.println("Update database for " + action + " book...");
        System.out.println("Send email notification to " + studentName);
    }
}
```

<= After transformation

4. Reverse Engineering: Unveiling the Hidden

Explanation	Sample Java Code Scenario: Generating a UML class diagram from existing Java code.

The opposite of forward engineering: extracting models or designs from existing code. This is invaluable when dealing with legacy systems or when documentation is lacking.
• Example 1: You're working with a large, old codebase with minimal documentation. Reverse engineering tools can analyze the code and generate UML diagrams, helping you understand its structure and dependencies.
• Example 2: You want to understand the sequence of operations in a complex piece of code. Reverse engineering can create a sequence diagram visualizing the interactions between different parts of the code, aiding in debugging or understanding the flow.
	Code Before Transformation (Java Code):
	Java
	public class Order {
	private int orderId;
	private Customer customer;
	private List<Product> products;
	
	// ... (getters and setters)
	}
	Code After Transformation (UML Diagram):
	---------- 	
	| Order |
	---------- 	
	| - orderId: int |
	| - customer: Customer |
	| - products: Product[]|
	---------- 	
	| + getOrderId(): int |
	| // ... other getters & setters
	---------- 	

Transformation Principles

A transformation aims at improving the design of the system with respect to some criterion.

- To avoid introducing new errors, all transformations should follow certain principles:

Mapping Models to Code

1. ***Each transformation must address a single criteria.***

- It means that one transformation should focus on one improvement/aspect at a time instead of mixing multiple goals.
- This keeps the change small and manageable, making it easier to understand, test, and validate.
- **Example:**
- Suppose you are adding a feature to allow users to reset their passwords in a web application. You would focus solely on adding the logic for sending a reset link to the user's email, without making unrelated changes to the authentication process or other parts of the system. This keeps the transformation focused on one criterion: password reset functionality.

2. Each transformation must be local.

- A transformation should change only a few methods or a few classes at once.
- The change should be isolated to a specific part of the system, affecting only the module or component related to the transformation.
- This minimizes the risk of introducing bugs in unrelated areas.
- **Example:**

If you are refactoring the database connection logic, you would limit your changes to the database access module and avoid modifying other parts of the code, such as the user interface or business logic. This ensures the transformation remains local, making it easier to debug if issues arise.

3. ***Each transformation must be applied in isolation to other changes*.**

- Each transformation should be applied independently from other ongoing changes.
- This allows developers to easily trace the impact of each transformation and ensures that unrelated changes do not interfere with each other.
- **Example:**
  Let's say you're refactoring the code to improve performance, while another team member is fixing a bug in the same system. By applying your changes (performance improvement) separately from the bug fix, each transformation can be tested individually, and any issues that arise can be clearly attributed to the respective change.

- ***4. Each transformation must be followed by a validation step*.**
- **Principle:**
- After making any change, the system must be validated to ensure the transformation was successful and the system remains functional.
- This can involve unit testing, integration testing, or running automated tests.
- **Example:**
  After adding the password reset functionality, you would write and run test cases to verify that the password reset email is sent, the link works correctly, and the user can successfully reset their password. This validation ensures the transformation meets its intended purpose without introducing defects.

Important Takeaway

- Each type of transformation plays a vital role in the software lifecycle, allowing us to:
- • Bridge the gap between design and implementation
- • Improve code quality and maintainability
- • Adapt to changing requirements and platforms
- • Understand and evolve existing systems
- By mastering these transformations, we empower ourselves to create and manage software that's not only functional but also elegant, adaptable, and sustainable.

Important Notes:

- • These are simplified examples to illustrate the concepts. Real-world transformations can involve much more complex code and transformations.
- • Automated tools are often used to perform these transformations, especially for larger projects.
- • The specific code generated or extracted will depend on the tools and configurations used.
- Remember, the goal of these transformations is to improve the software development process by automating tasks, enhancing code quality, and facilitating communication between different representations of the software system.

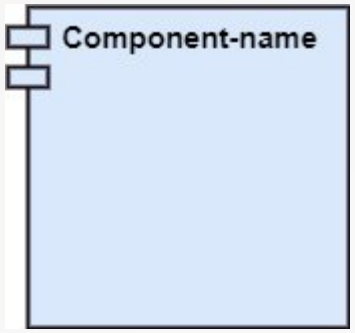
Conclusion

- Mapping models to code is a critical step in software development.
- Transformation automates and streamlines the process, improving efficiency and quality.
- Different types of transformation support various stages of the software lifecycle.
- By effectively bridging design and implementation, mapping models to code contributes to successful software projects.

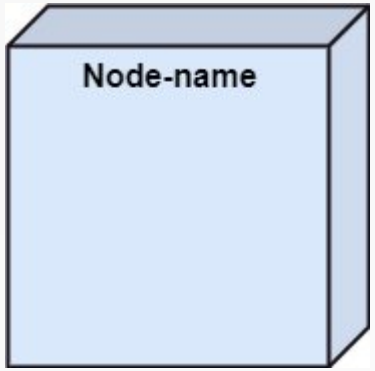
Component diagram

- Providing a visual representation of a system's structure by showcasing its various **components and their interactions**.
- Component diagram *shows components*, provided and required interfaces, ports, and **relationships between them.**
- It does not describe the functionality of the system but it **describes the components used to make those functionalities.**
- Component diagrams can also be described as a static implementation view of a system.
- It models the physical view of a system such as executables, files, libraries, etc. that resides within the node

- A component is a single unit of the system, which is replaceable and executable.
- The implementation details of a component are hidden
- **Notation of a Component Diagram**



Component



Node

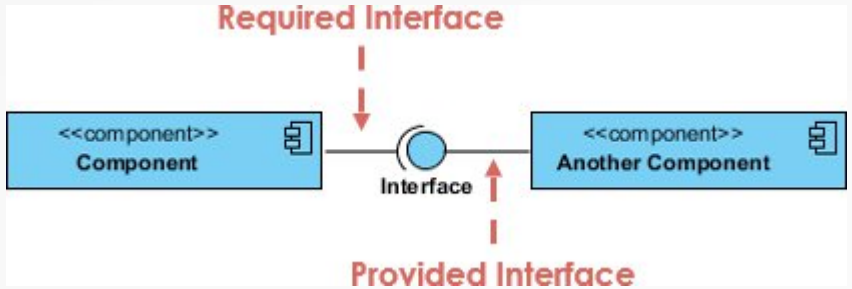
Basic Concepts of Component Diagram

Different ways to represent component:

- A rectangle with the component's name
- A rectangle with the component icon
- A rectangle with the stereotype text and/or icon

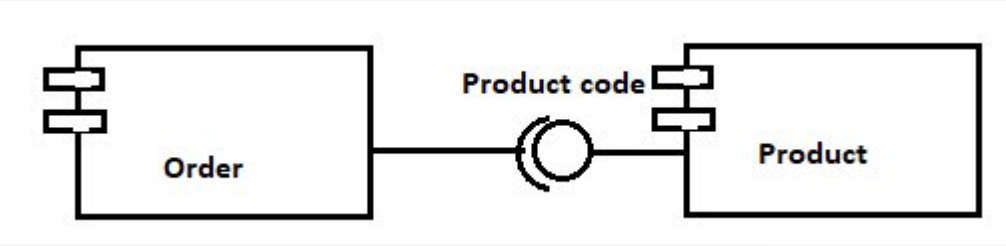
Interface

- In the example below shows two type of component interfaces:
- **Provided interface** symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.
- **Required Interface** symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).



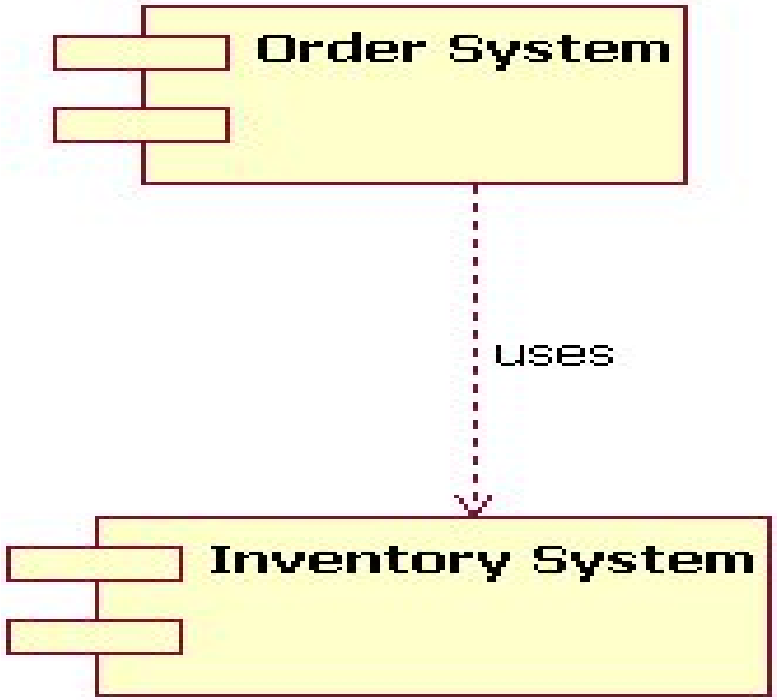
Component diagram

- Example of simple order system



Dependancy

Figure 1: This simple component diagram shows the Order System's



- Additional compartment shows the interface

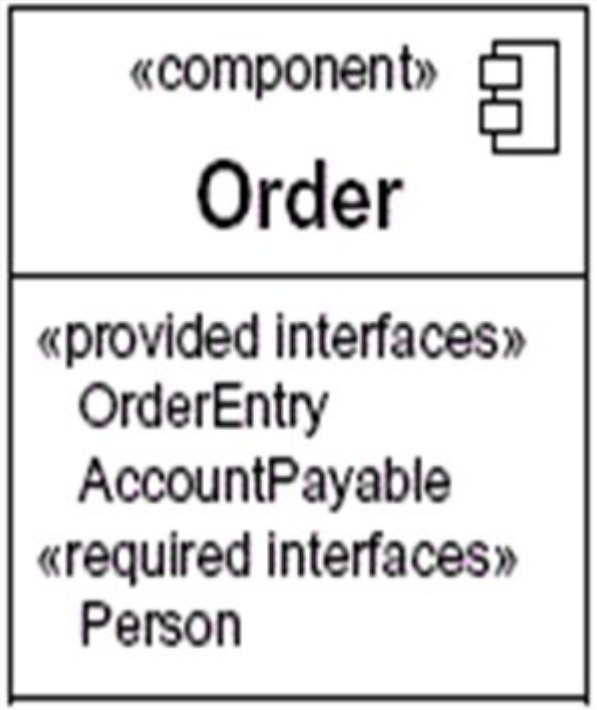
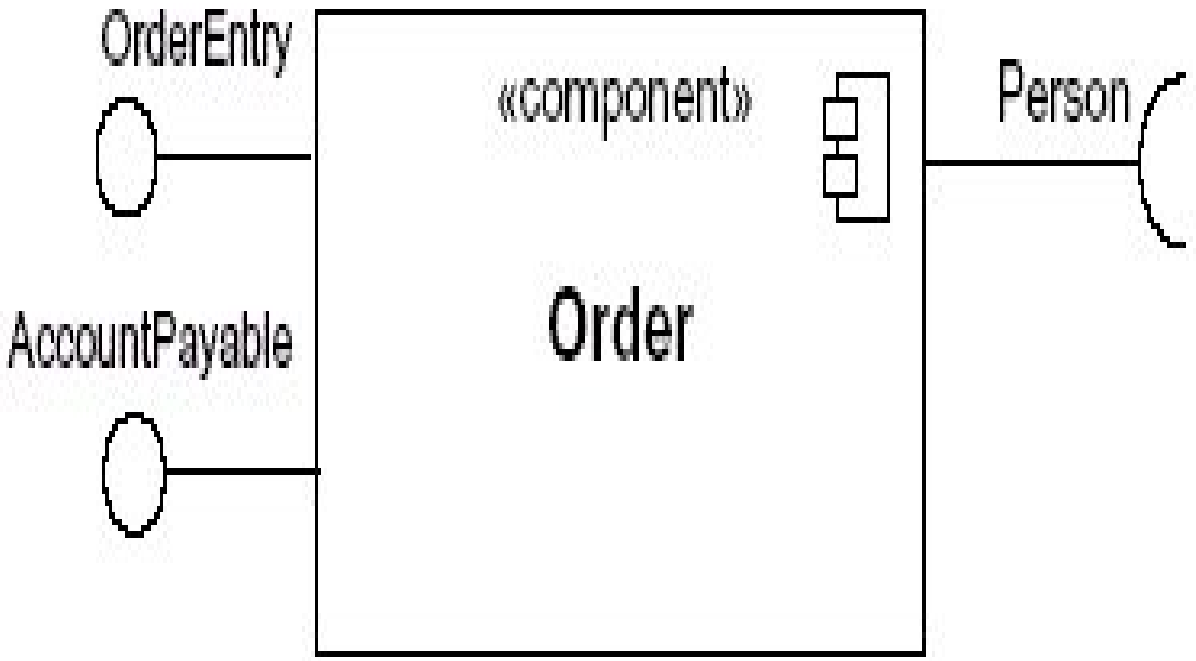


Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols



Component diagram

- The assembly connector bridges component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires.

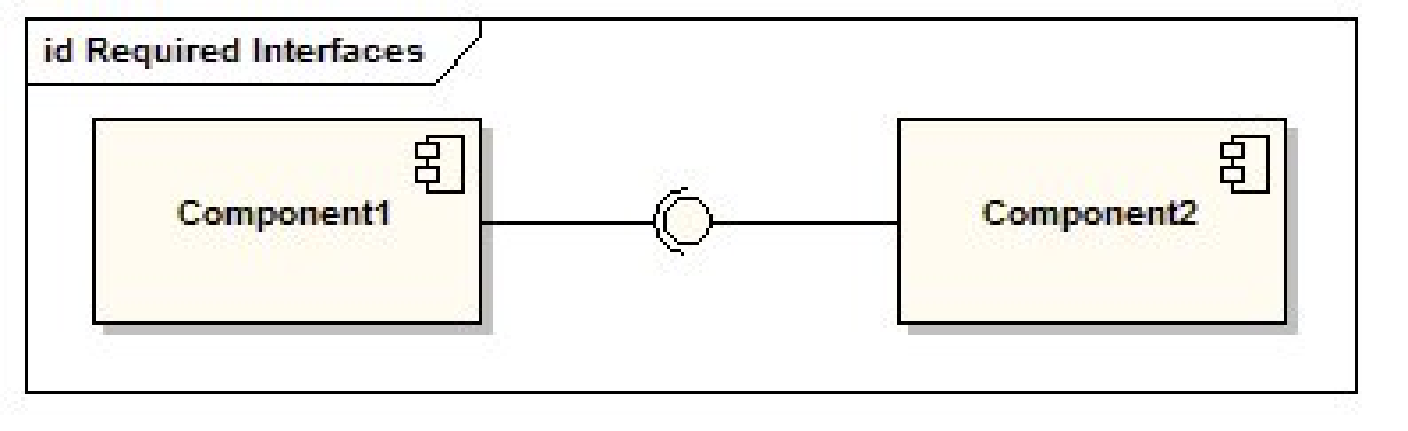


Figure 5: A component diagram that shows how the Order System component depends on other components

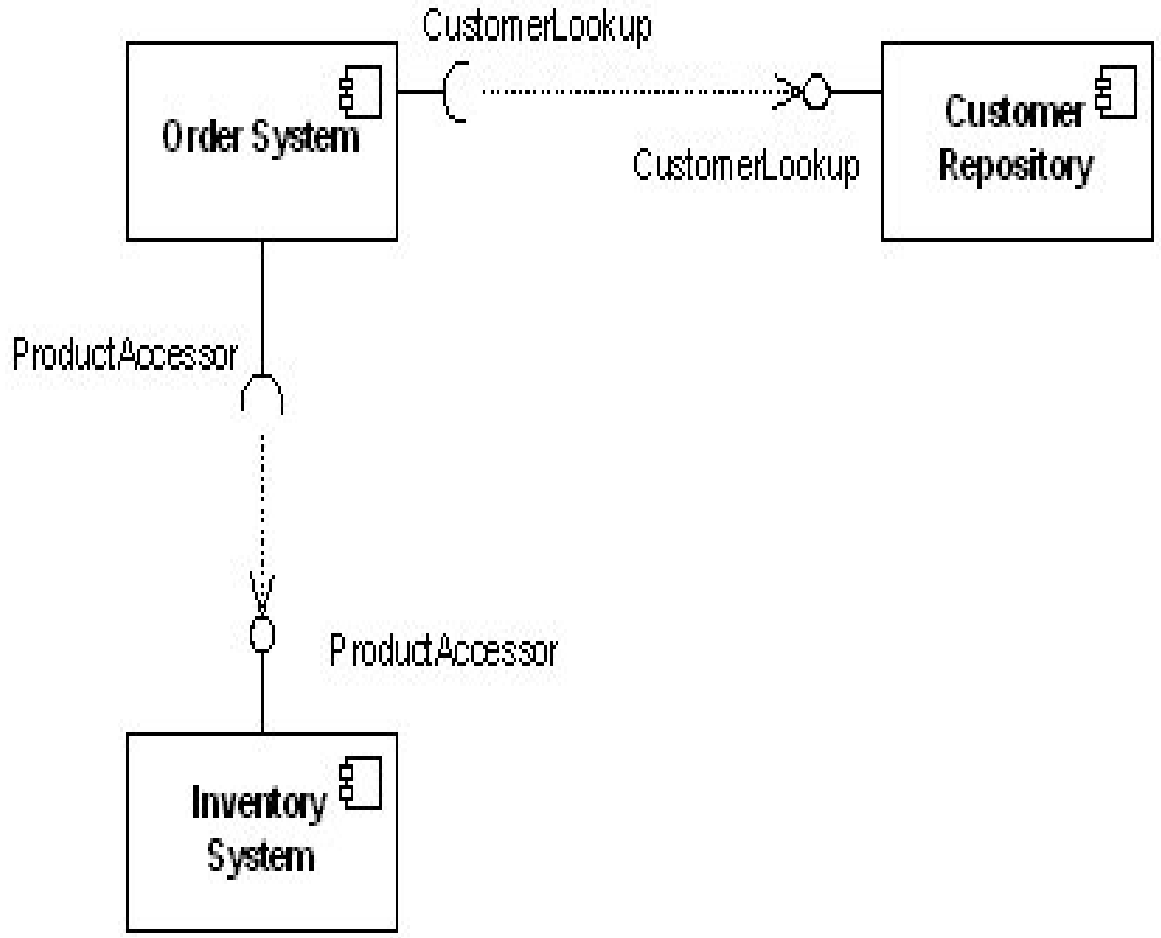
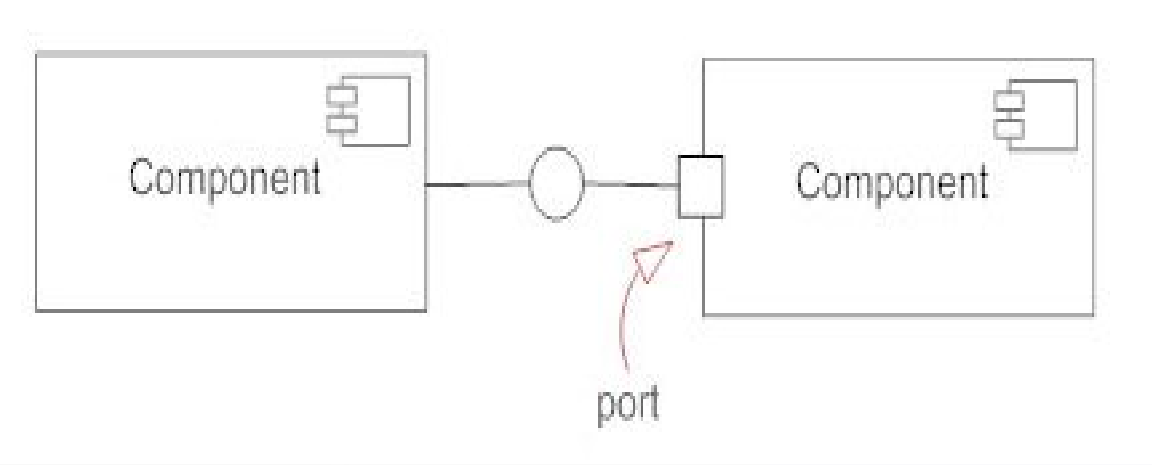


Figure 6: An example of a subsystem element

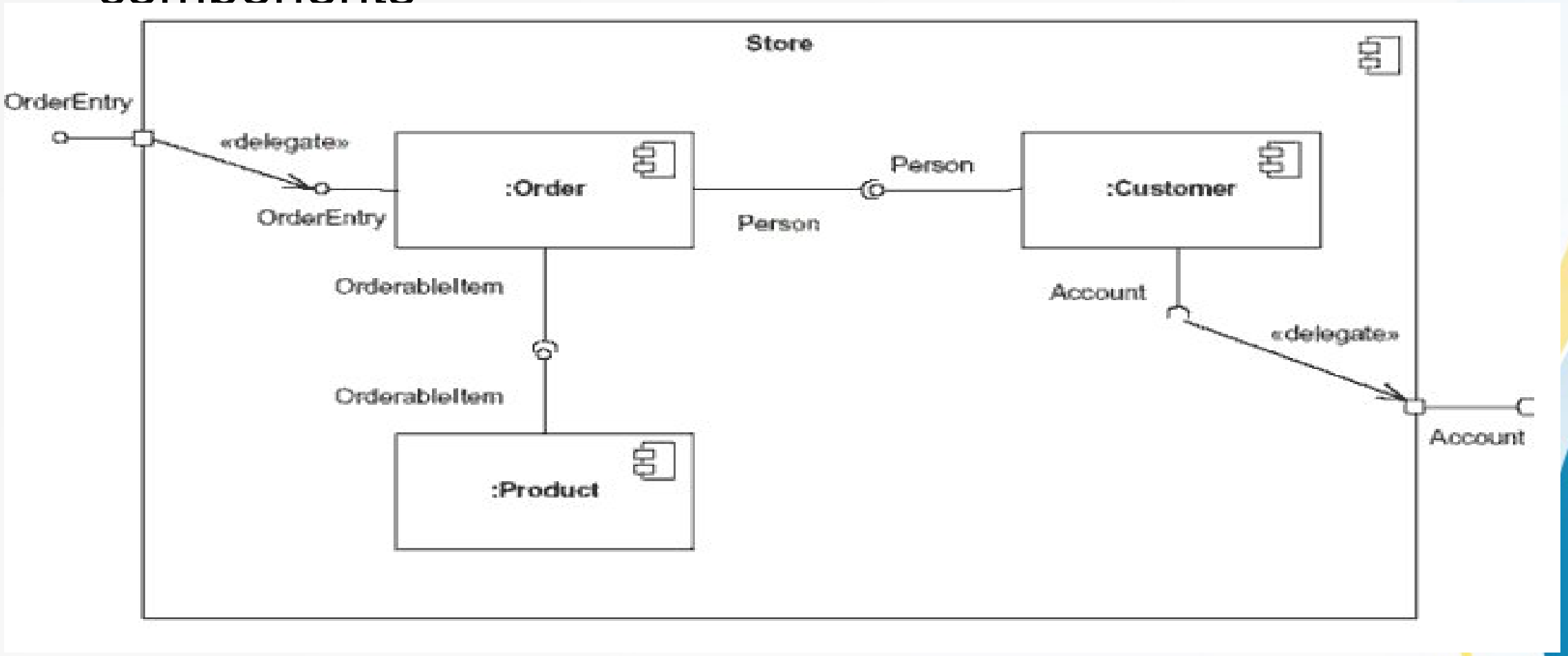
Component diagram

- **Port**
- Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



Component diagram

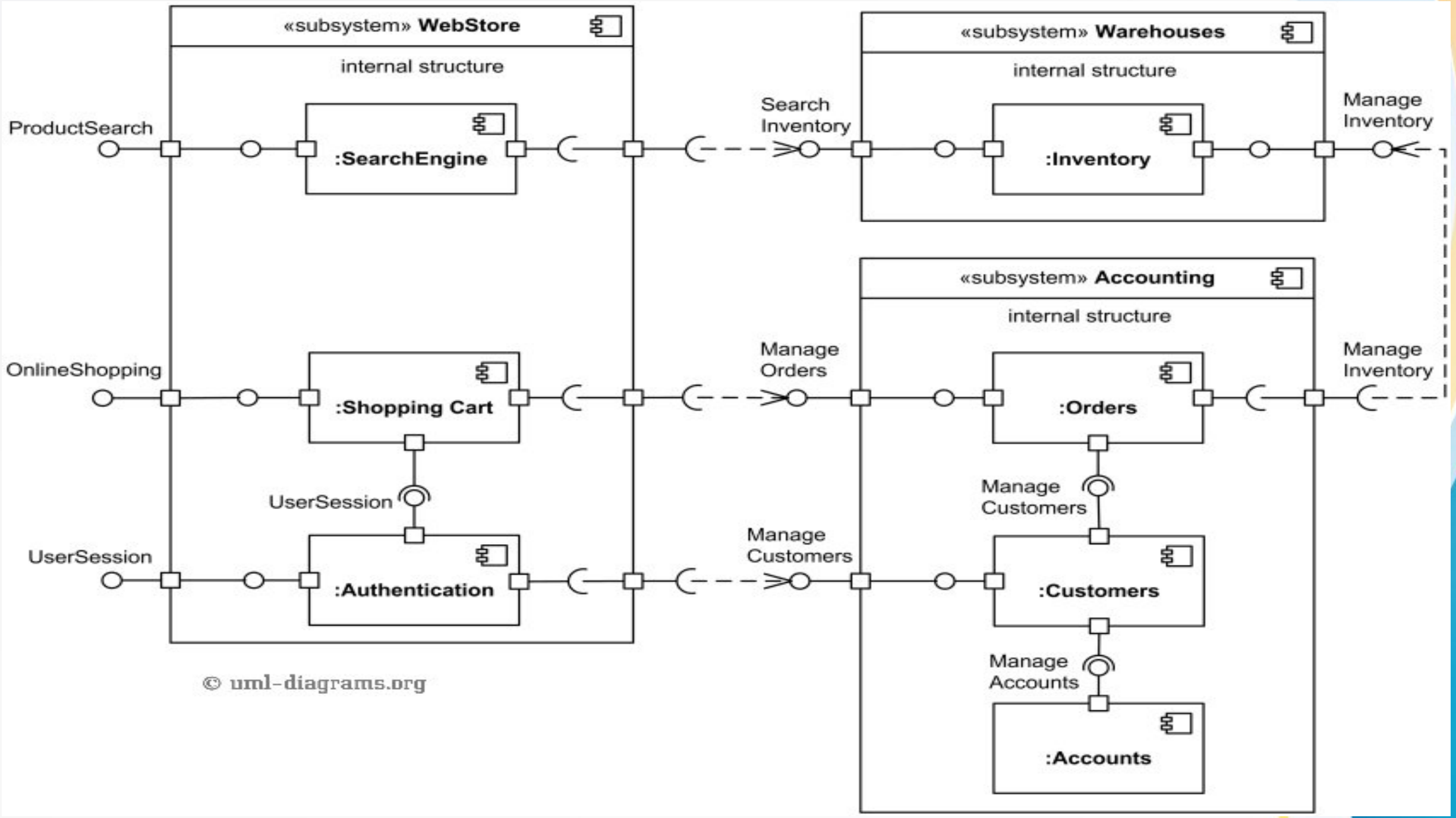
- This component's inner structure is composed of other components



Component diagram

- **Online Shopping *UML Component Diagram Example***

- Online shopping UML component diagram example with three related subsystems - **WebStore, Warehouses, and Accounting.**


«subsystem» WebStore
internal structure
ProductSearch
:SearchEngine
OnlineShopping
:Shopping Cart
UserSession
UserSession
:Authentication
Search Inventory
«subsystem» Warehouses
internal structure
:Inventory
Manage Inventory
Manage Orders
«subsystem» Accounting
internal structure
:Orders
Manage Inventory
Manage Customers
Manage Customers
:Customers
Manage Accounts
:Accounts
© uml-diagrams.org

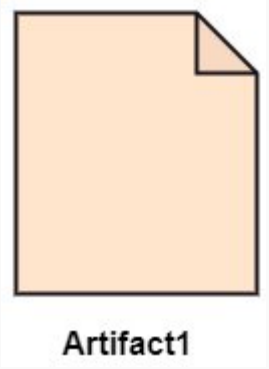
Deployment Diagram

- Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system.
- A deployment diagram is used to show the allocation of artifacts to nodes in the physical design of a system.
- Deployment Diagrams are made up of a graph of nodes connected by communication associations to show the physical configuration of the software and hardware.

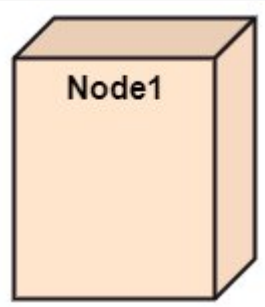
Essential elements of deployment diagram

1. Artifacts

- Artifact is a physical item that implements a portion of software design.
- It can be an source file , document, databases etc related to code.
- Notation of artifact consists of class rectangle name of artifact and label **<<artifact>>**
- Physical files deployed onto nodes, embodying the actual implementation of software components, such as executables, scripts, databases, etc.



Deployment Diagram



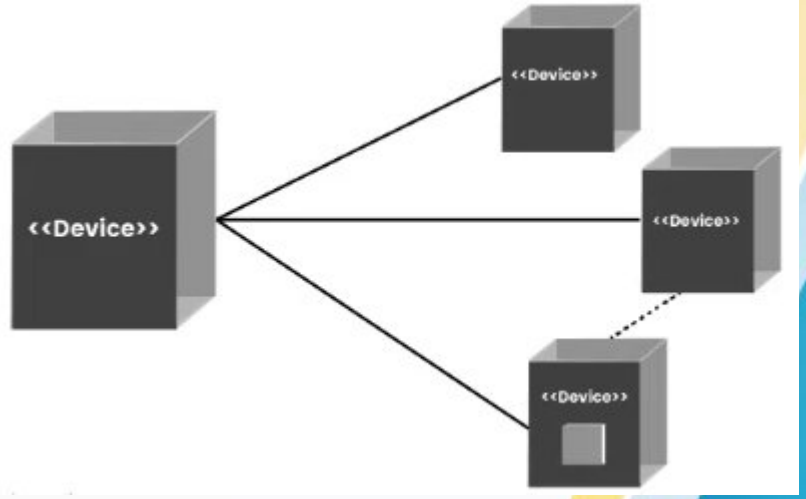
2. Nodes

- A node is computational resource containing memory and processing on which artifacts are deployed for execution.
- Notation of node three dimensional cube.
- Nodes denote hardware, not software entities.
- These represent the physical hardware entities where software components are deployed, such as servers, workstations, routers, etc.

Deployment Diagram

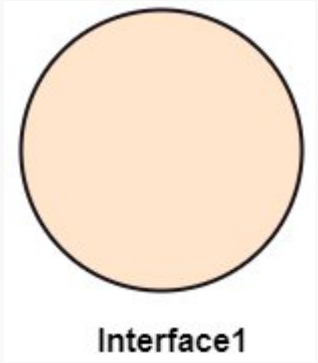
3. Connections/ dependencies

- Reflect relationships or connections between nodes and components
- Nodes communicate via messages and signals , through communication path indicated by solid line.
- Communication paths are usually considered to be bidirectional, for unidirectional, an arrow may be added to show the direction
- Each communication path may include an optional keyword label, such as «http» or «TCP/IP», that provides information about the connection.
- multiplicity for each of the nodes connected via a communication path.

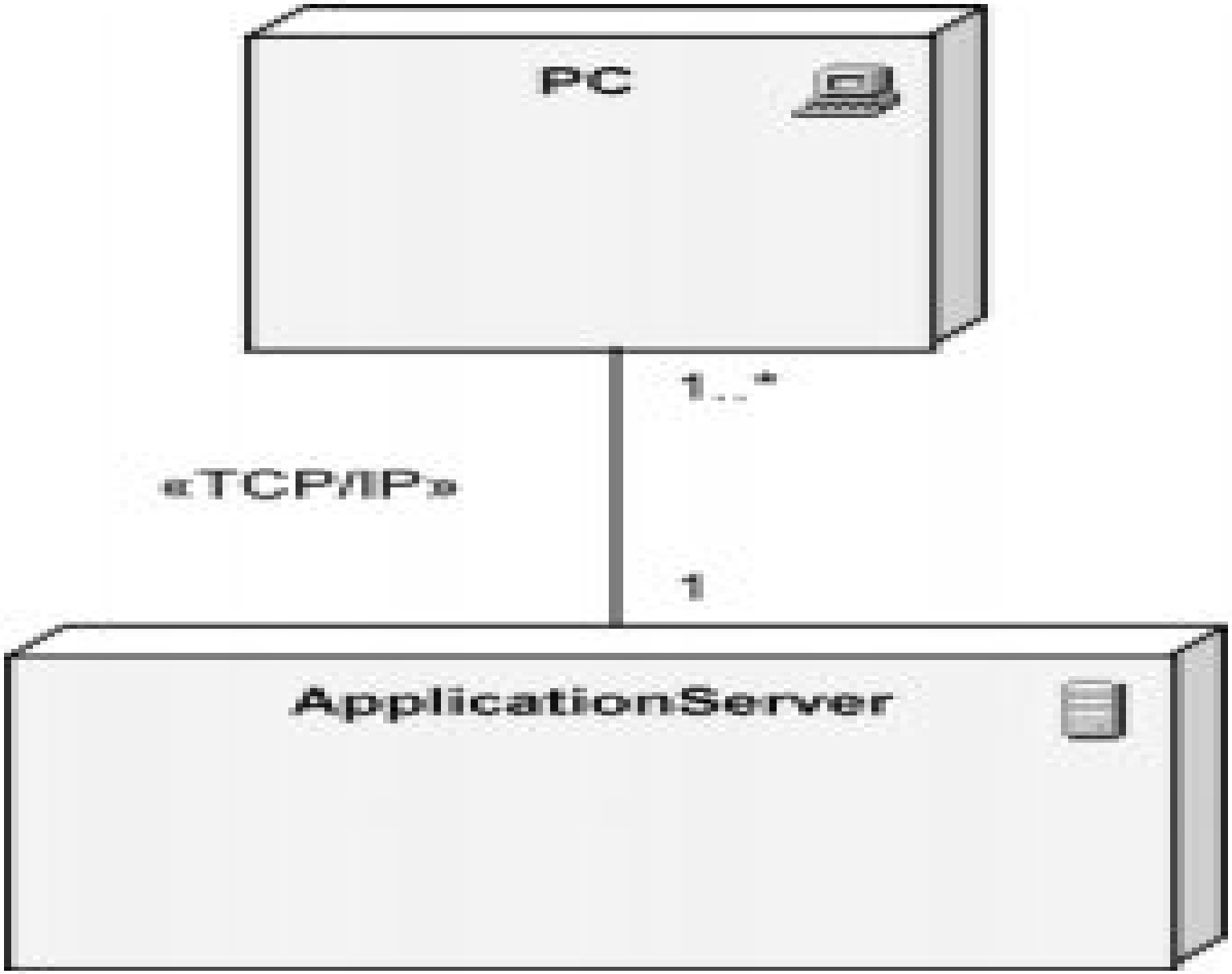


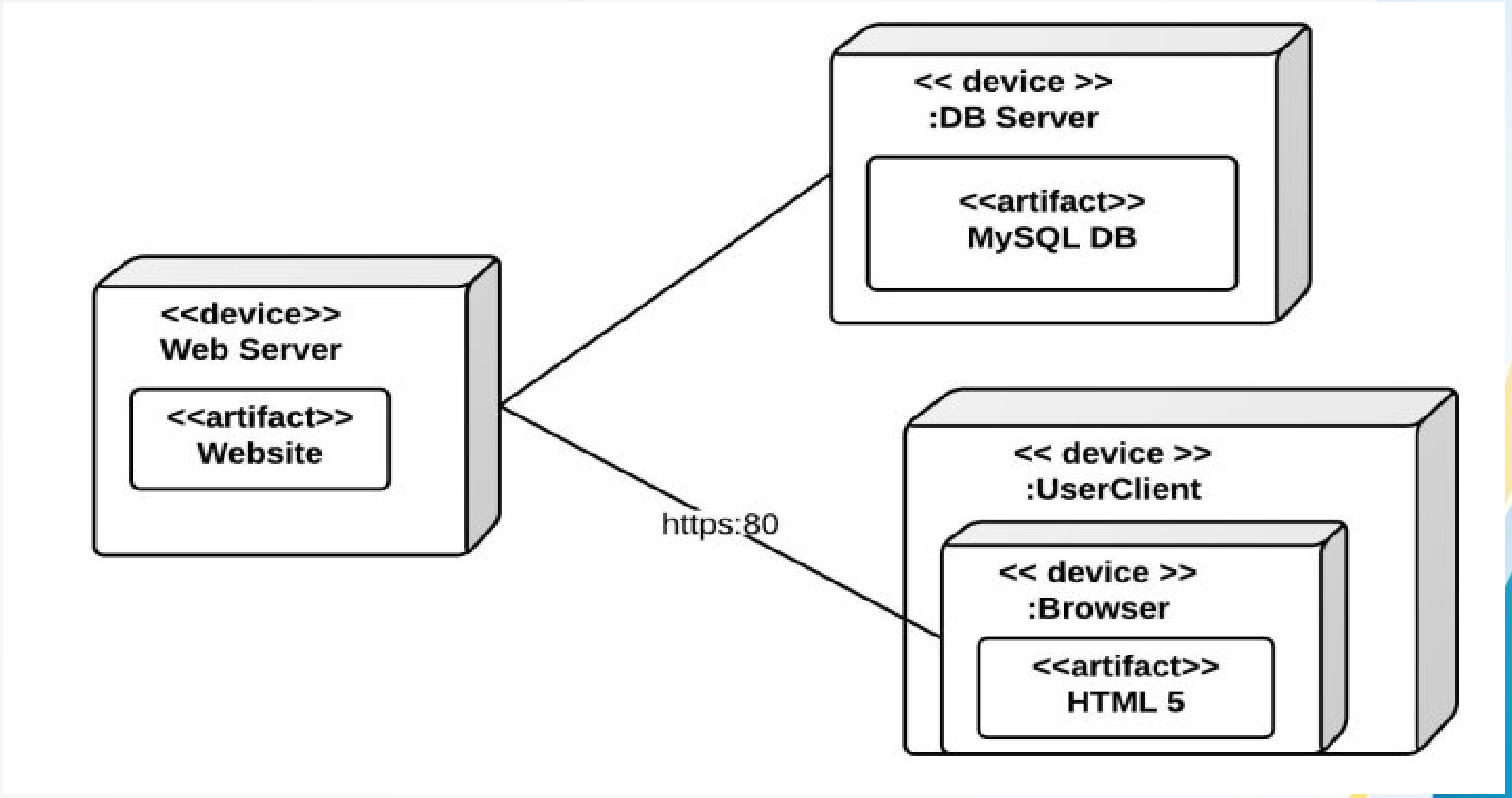
4. Interface

- An interface defines a contract specifying the methods or operations that a component must implement.
- It represents a point of interaction between different components or subsystems.
- *Represented as a **circle or ellipse** labeled with the interface's name. Interfaces can also include provided and required interfaces, denoted by "+" and "-" symbols, respectively.*



5.Component

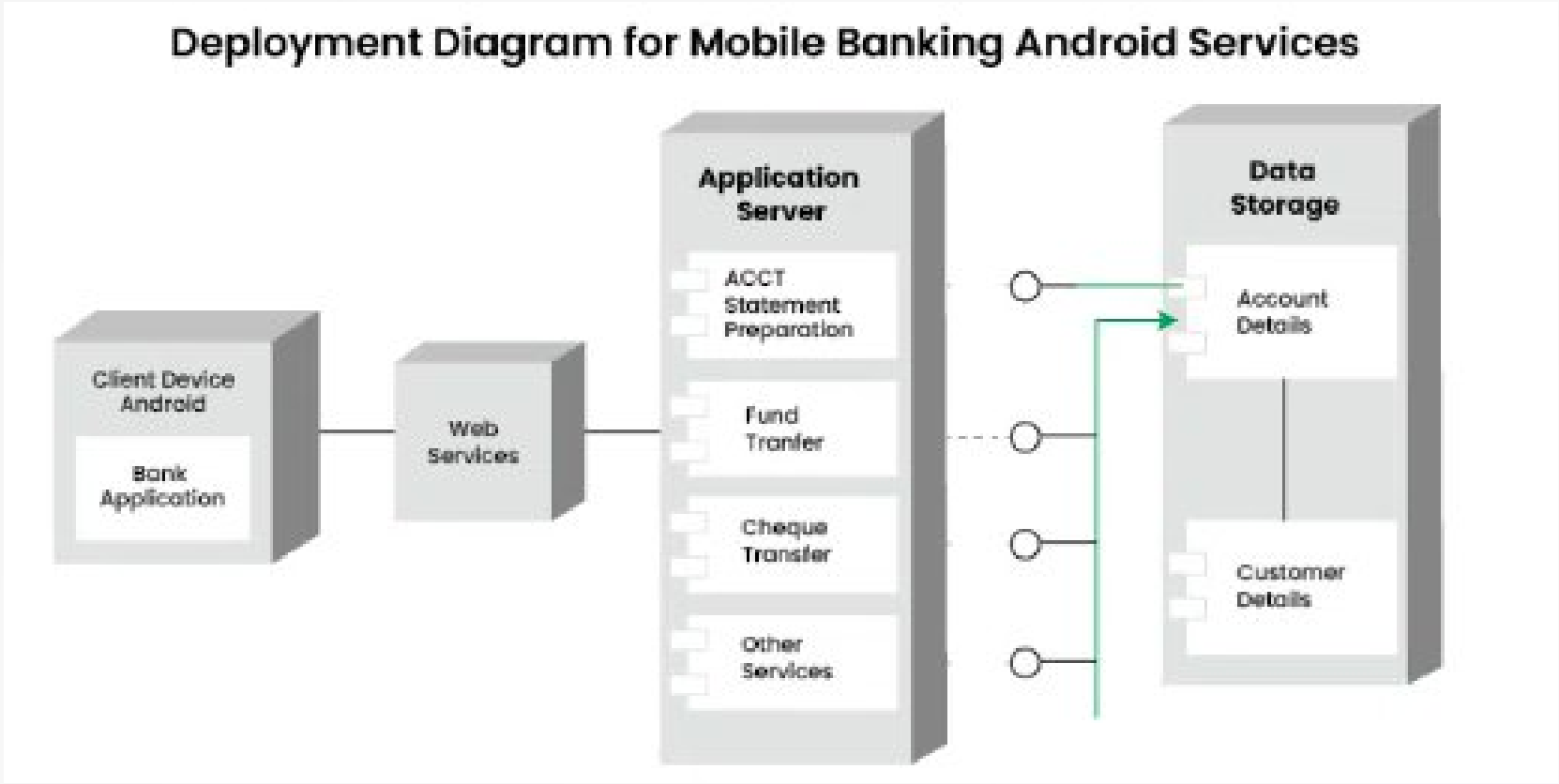

PC
1..*
«TCP/IP»
1
ApplicationServer


<< device >>
:DB Server
<<artifact>>
MySQL DB
<<device>>
Web Server
<<artifact>>
Website
https:80
<< device >>
:UserClient
<< device >>
:Browser
<<artifact>>
HTML 5

Use Cases of Deployment Diagrams

- **System Planning:** Deployment diagrams help plan how software systems will be set up on different devices. They show where each part of the system will go.
- **Infrastructure Design:** They help design the hardware needed to support the software. By showing which software parts go where, they help decide what devices and networks are needed.
- **Resource Allocation:** Deployment diagrams make sure each part of the software has enough resources, like memory or processing power, to run well.
- **Dependency Analysis:** They show how different parts of the software depend on each other and on the hardware. This helps understand how changes might affect the whole system.
- **Performance Optimization:** By seeing how everything is set up, teams can find ways to make the software run faster and smoother.
- **Security Planning:** Deployment diagrams help plan how to keep the system safe from hackers or other threats by showing where security measures are needed.
- **Documentation:** They provide a visual guide to how the system is set up, making it easier to understand and manage.

Example


Deployment Diagram for Mobile Banking Android Services
Client Device Android
Bank Application
Web Services
Application Server
ACCT Statement Preparation
Fund Tranfer
Cheque Transfer
Other Services
Data Storage
Account Details
Customer Details

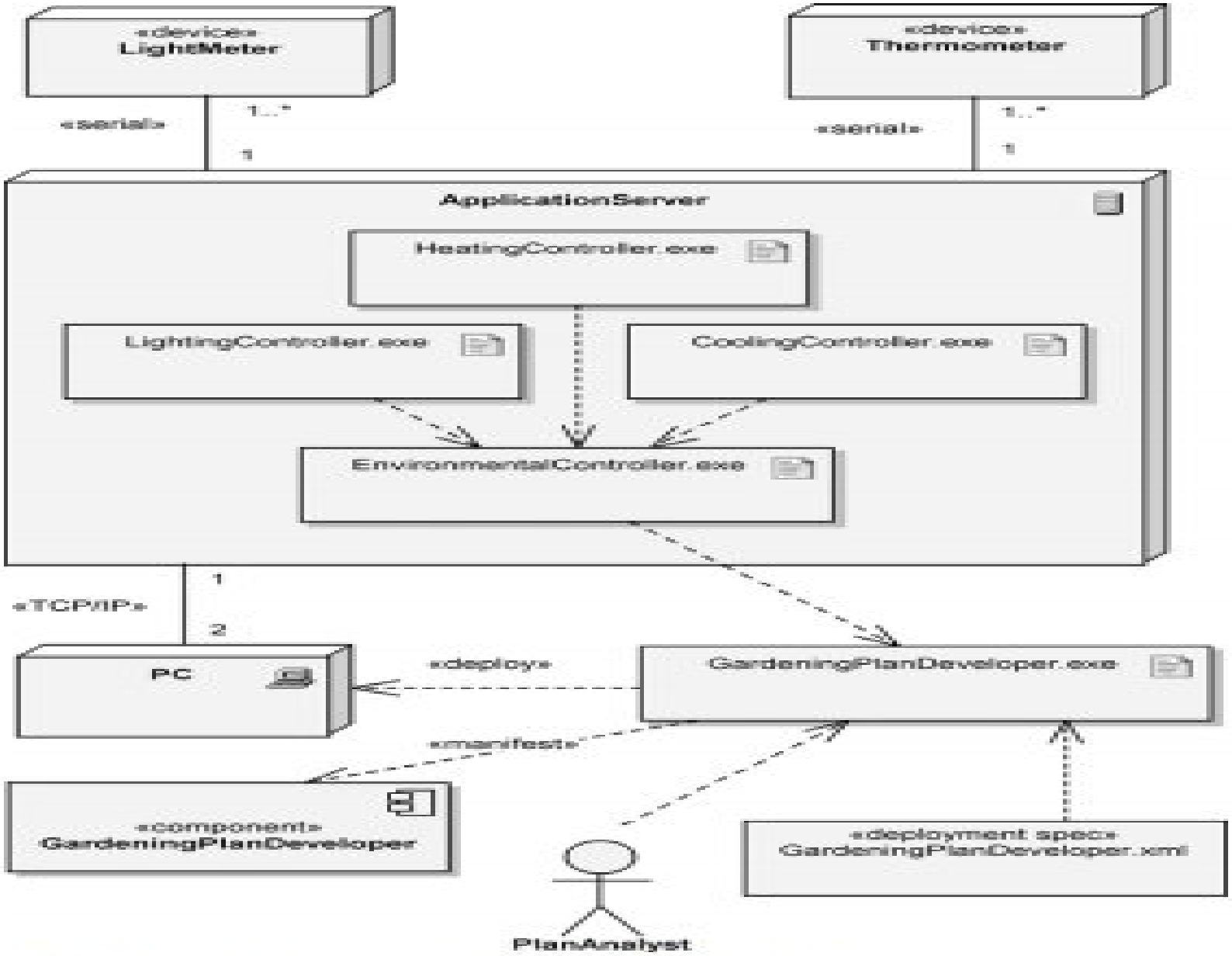


Figure 5–19 The Deployment Diagram for `EnvironmentalControlSystem`