CHAPTER 4

	System Implementation &Configuration Management
4.1	Packages and interfaces: Distinguishing between classes/interfaces, Exposing class and package interfaces
4.2	Mapping model to code , Mapping Object Model to Database Schema
4.3	Component and deployment diagrams: Describing dependencies
4.4	Managing and controlling Changes
4.5	Managing and controlling version

Version Control

- Version control **combines procedures and tools to manage different versions of configuration objects** that are created during the software process.

- A version control system implements four major capabilities:

(1) a **project database** that stores all relevant configuration objects

Version Control

(2) a **version management capability** that **stores all versions** of a configuration object.

(3) a make facility that enables **construct a specific version of the software.**

(4) version control and change control systems often implement an **issues tracking (also called bug tracking)** capability

Version Control

- A number of version control systems **establish a change set**—a collection of all changes that are required to create a specific version of the software.

- **named change sets can be identified** for an application or system.

- construct a version of the software by specifying the change sets **(by name)**

Version Control

- Modeling approach for building new versions contains:

1. Template for building version
2. Construction rules
3. Verification rules

Change Control

- Too much change control and we create problems.
- large software project, uncontrolled change rapidly leads to chaos.
- For **large projects** change control combines **human procedures and automated tools** to provide a mechanism for the control of change.

Change Control

- engineering change order (ECO).

- **elements of change management:**

1. **Access Control**
- Access control governs which software engineers have the authority to access and modify a particular configuration object.

Change Control

2. Synchronization Control.

- Synchronization control helps to ensure that parallel changes, performed by two different people, don’t overwrite one another.

- **Informal Change Control**
- Developer makes changes in project

Change Control

- **project level change control**
- developer must gain approval from the project manager to makes changes.

- **formal change control**
- is instituted when the software product is released to customers


Need for change is recognized
Change request from user
Developer evaluates
Change report is generated
Change control authority decides
Request is queued for action, ECO generated
Change request is denied
Assign individuals to configuration objects
User is informed
"Check out" configuration objects (items)
Make the change
Review (audit) the change
"Check in" the configuration items that have been changed
Establish a baseline for testing
Perform quality assurance and testing activities
"Promote" changes for inclusion in next release (revision)
Rebuild appropriate version of software
Review (audit) the change to all configuration items
Include changes in new version
Distribute the new version

Component diagram

- 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.**

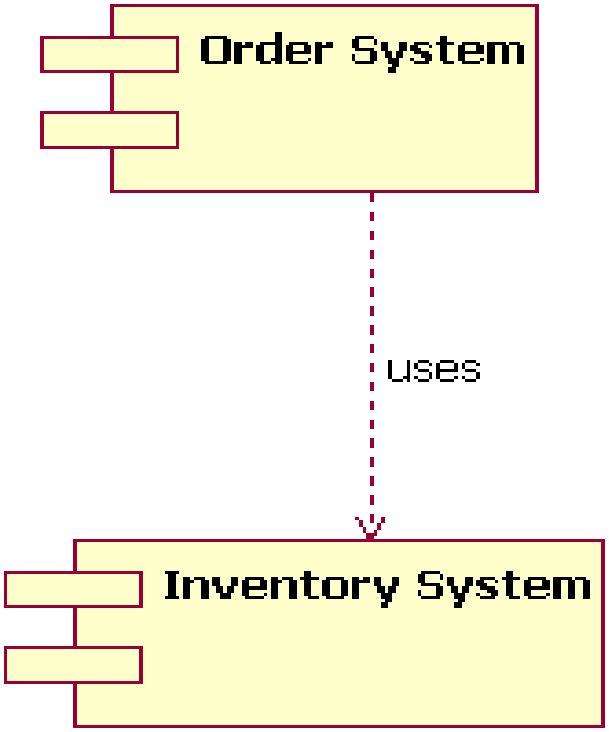
- Component diagrams can also be described as a static implementation view of a system.

Component diagram

- The component diagram's main purpose is to show the structural relationships between the components of a system.

Component diagram

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



Component diagram

Figure 2: The different ways to draw a component's name compartment

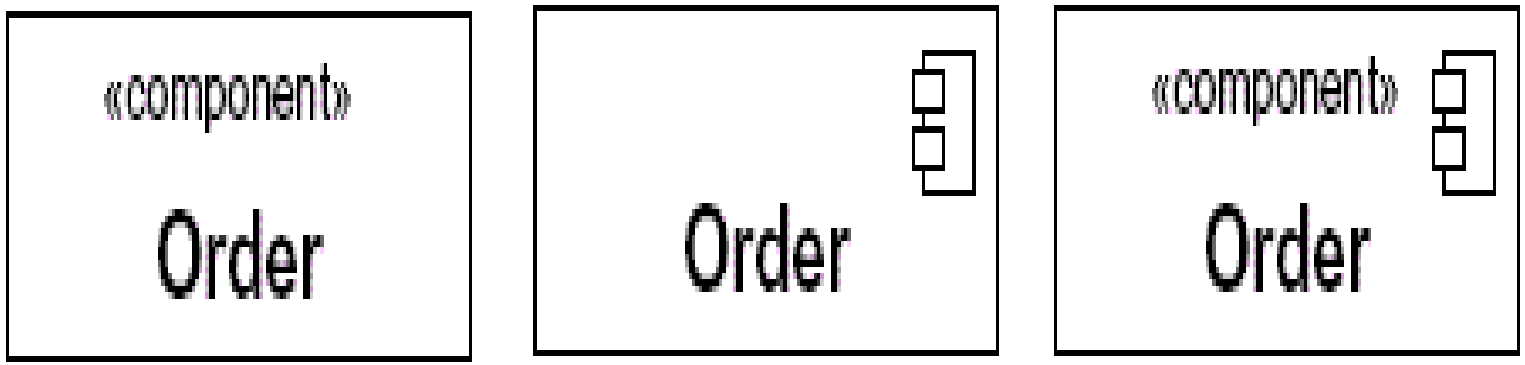


Figure 3: The additional compartment here shows the interfaces

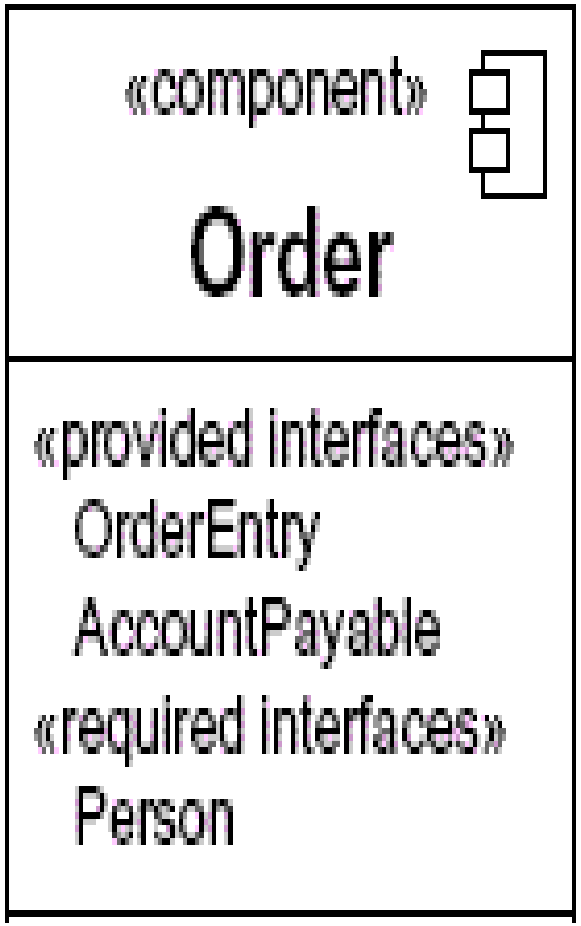
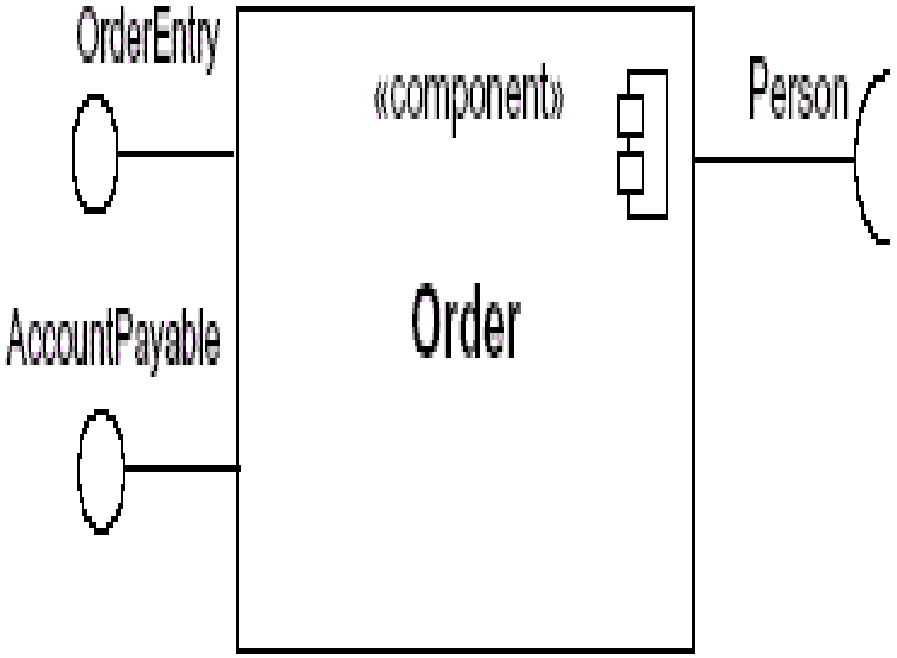


Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols



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.

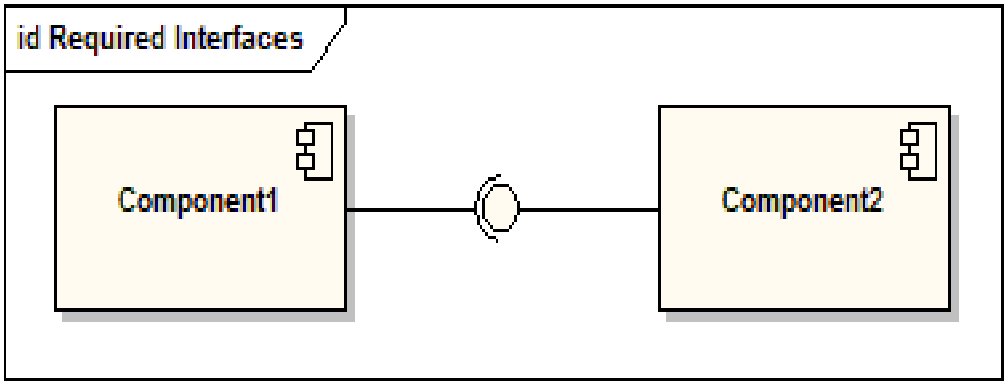


Figure 5: A component diagram that shows how the Order System component depends on other components

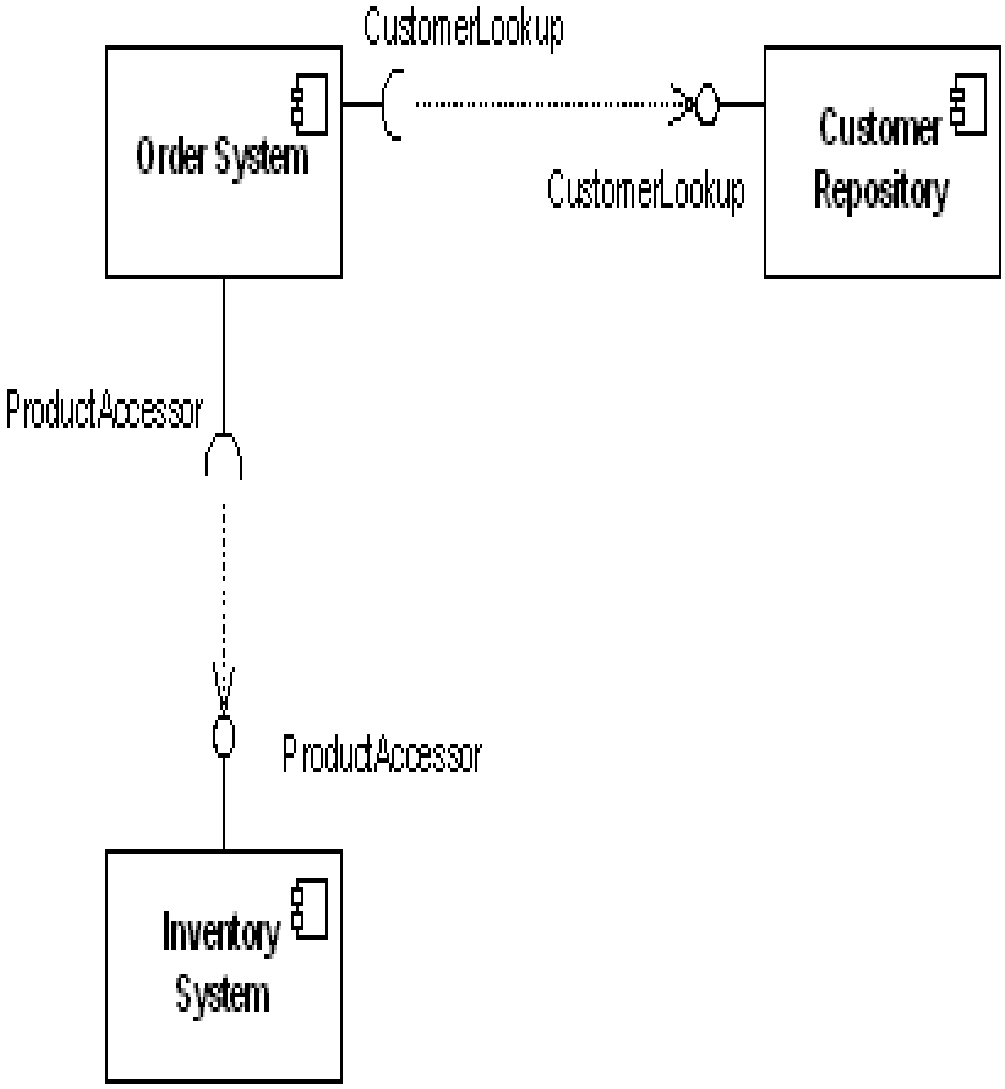
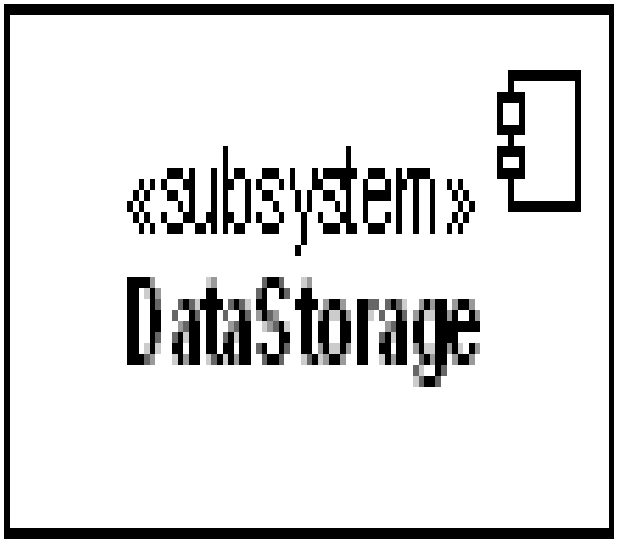
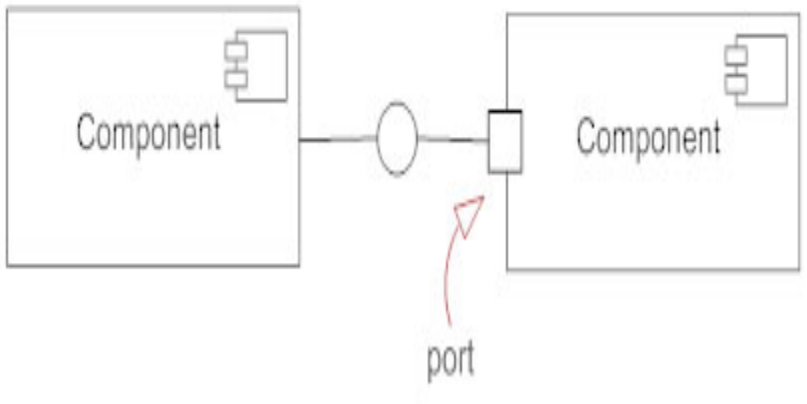


Figure 6: An example of a subsystem element



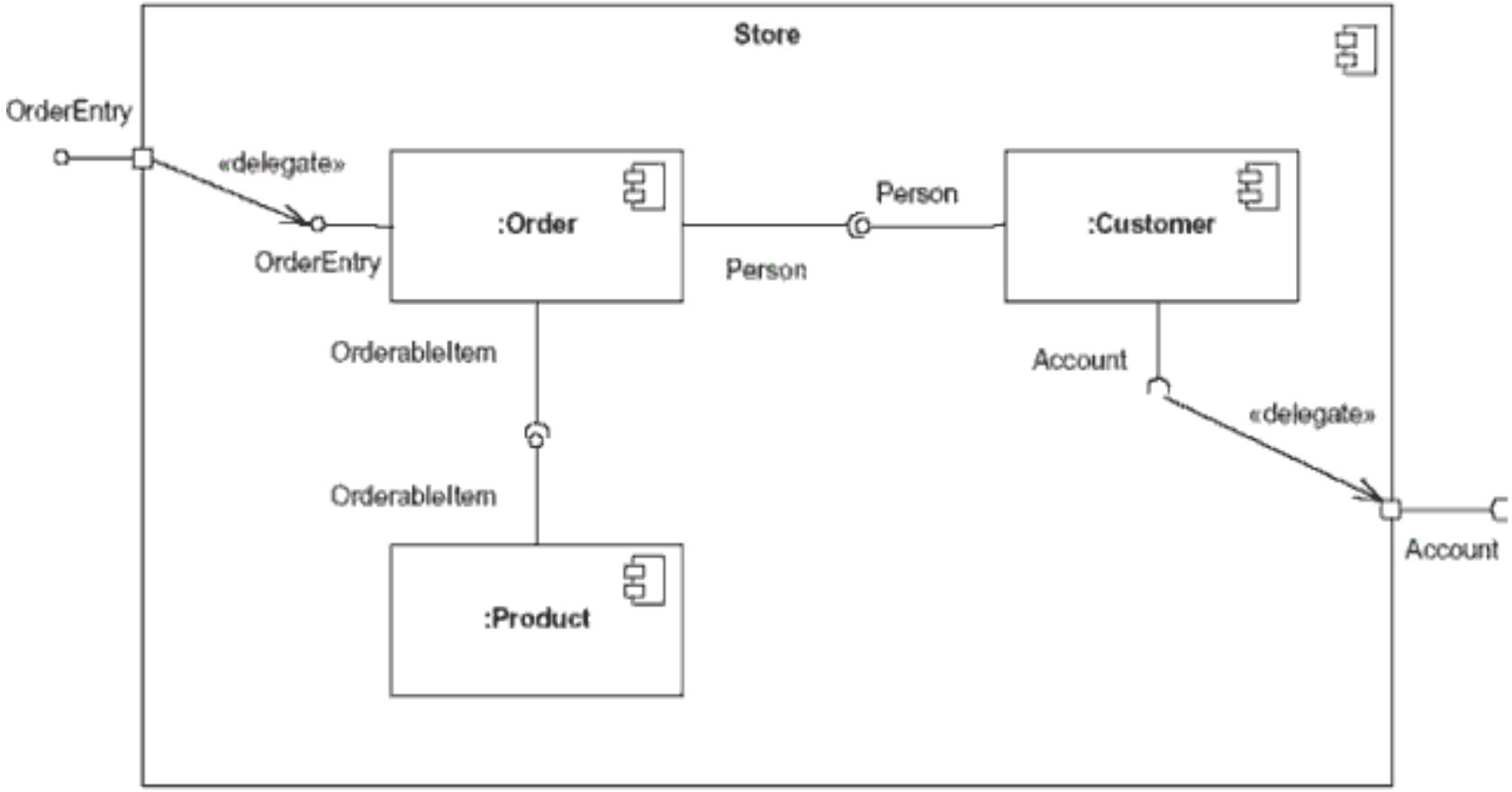
Component diagram

- **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

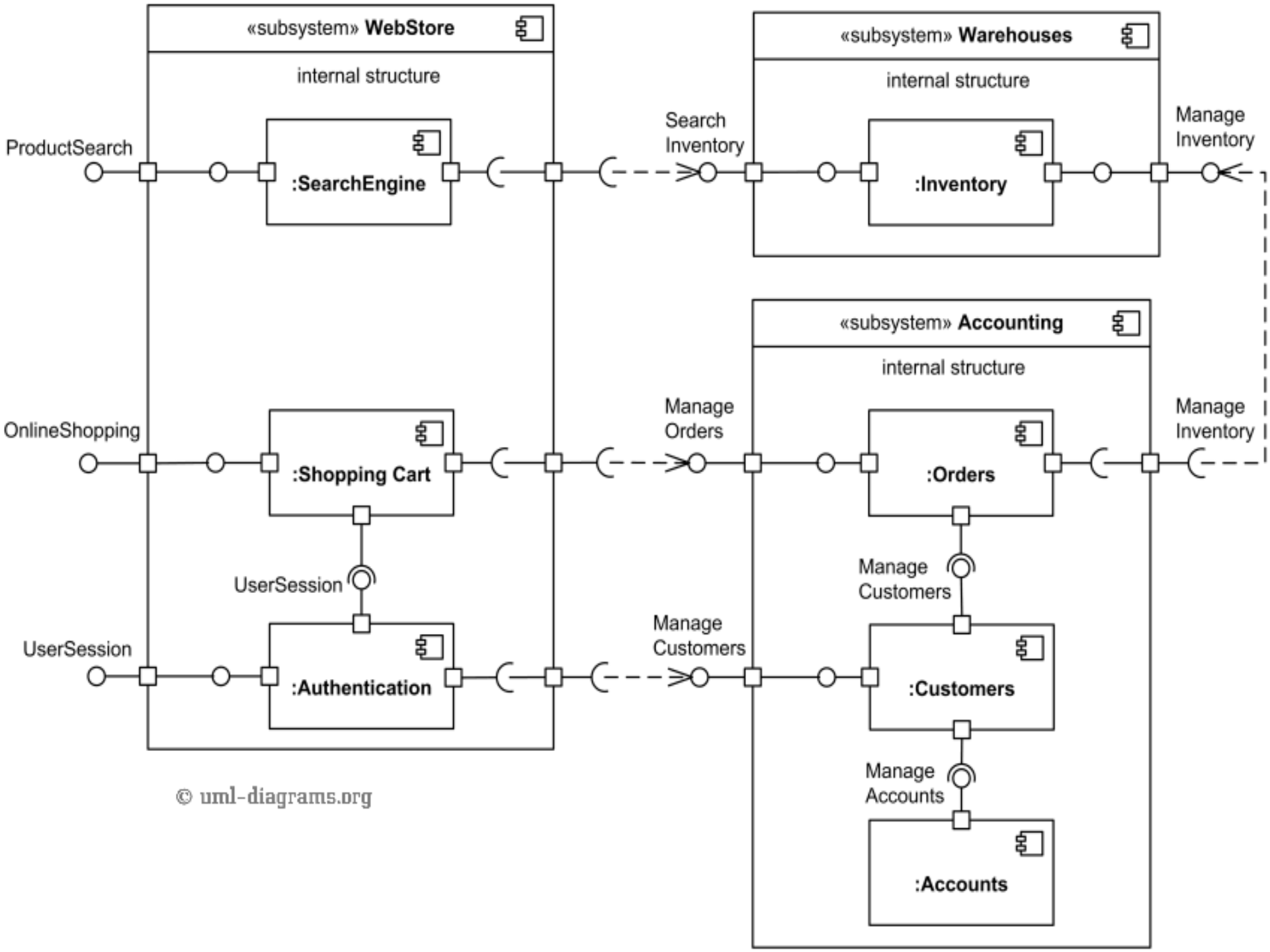
- This component's inner structure is composed of other components



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

Deployment Diagram

- 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.

Deployment Diagram

- 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 or another item related to code.
- Notation of artifact consists of class rectangle name of artifact and label **<<artifact>>**

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.

Deployment Diagram

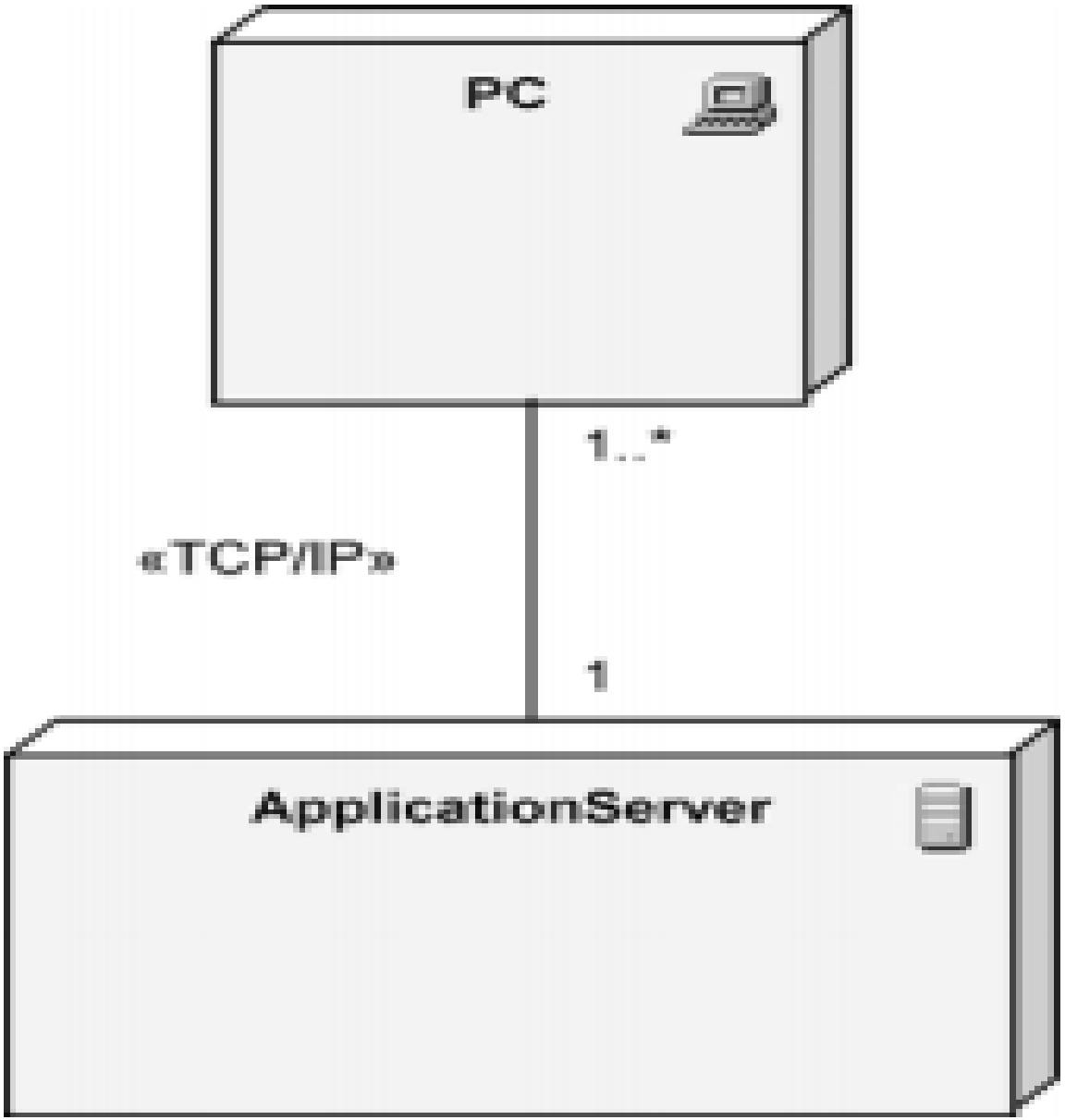
3. Connections

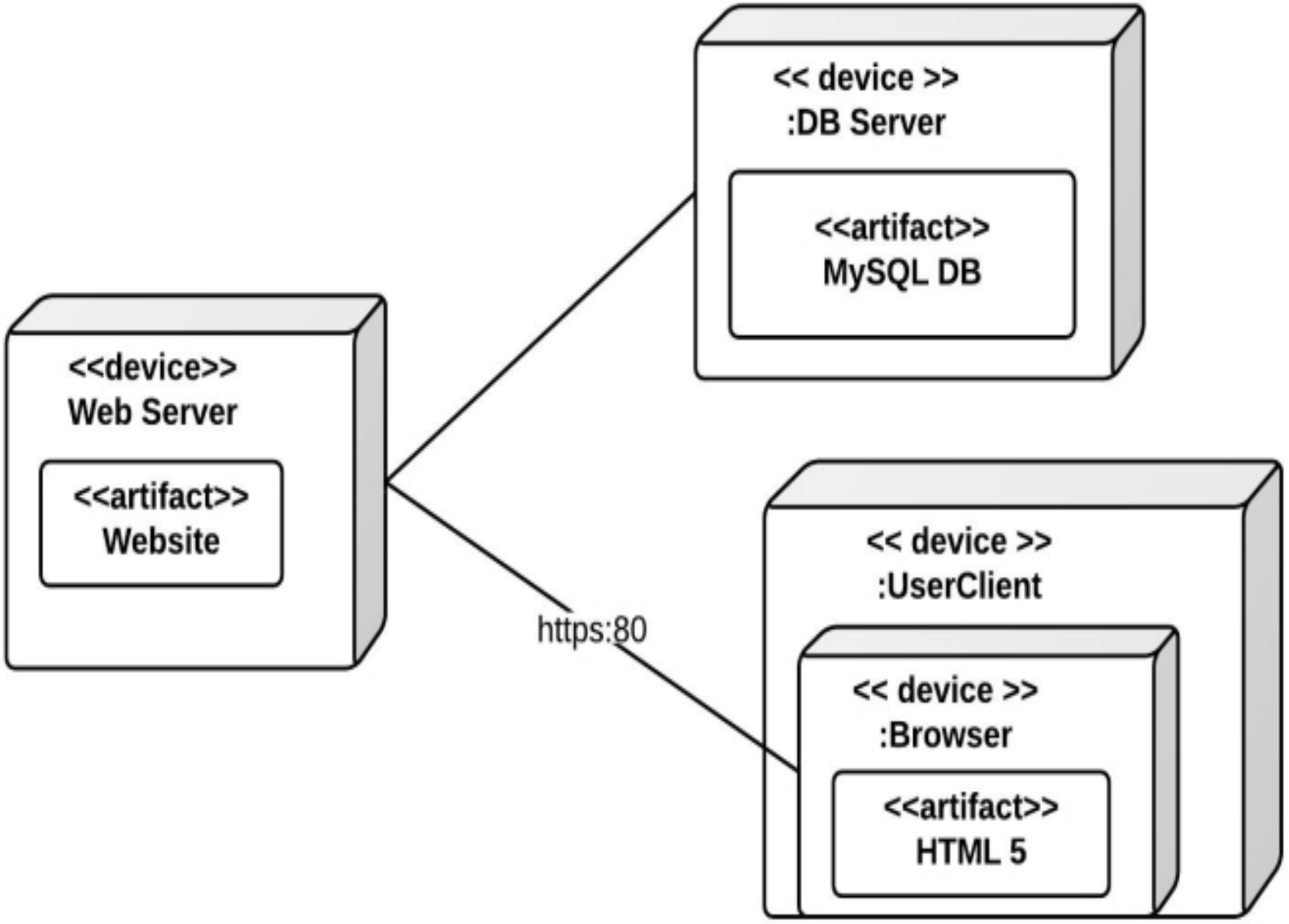
- Nodes communicate via messages and signals , through communication path indicated by solid line.

- 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.


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

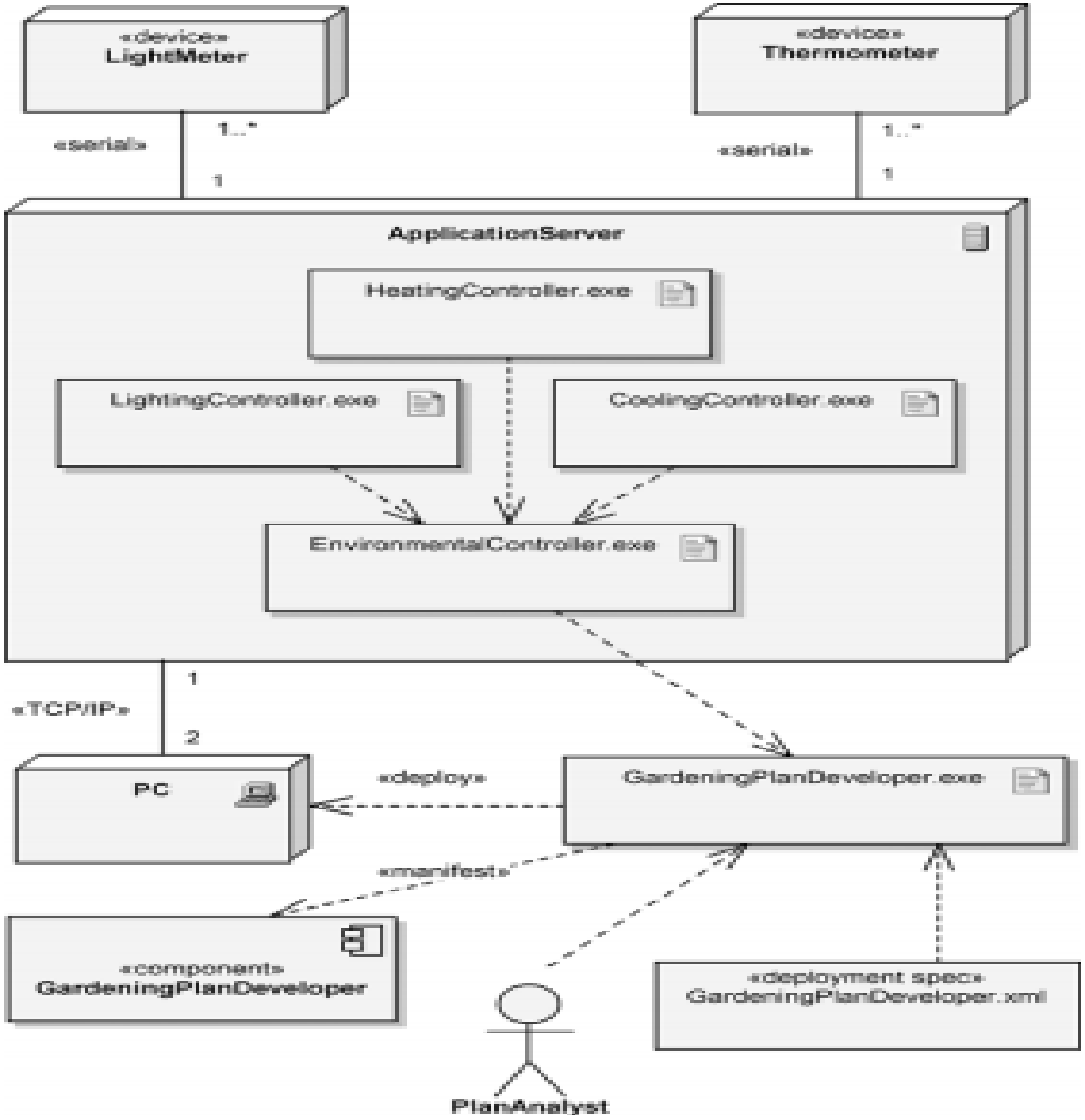


Figure 5–19 The Deployment Diagram for `EnvironmentalControlSystem`

Mapping Models to Code

- A **transformation** aims at improving one aspect of the model (e.g., its modularity) while preserving all of its other properties (e.g., its functionality).

- Transformation Activities:

1. *Optimization*

- This activity addresses the performance requirements of the system model.

Mapping Models to Code

- **Examples**
- Reducing multiplicities
- adding redundant associations for efficiency,
- adding derived attributes to improve the access time to objects.

2. Realizing associations

- map associations to source code constructs,
- **Example**
- references and collections of references.

Mapping Models to Code

3. Mapping contracts to exceptions

- describe the behavior of operations when contracts are broken.

- This includes raising exceptions when violations are detected.

4. Mapping class models to a storage schema

- selected a persistent storage strategy, such as a database management system, a set of flat files, or a combination of both.

- map the class model to a storage schema, such as a relational database schema.

Mapping Models to Code

- **Four types of transformations:**

1. ***Model transformations***

- A **model transformation** is applied to an object model and results in another object model.

- purpose of object model transformation is to bringing it into closer compliance with all requirements.

- A transformation may add, remove, or rename classes, operations, associations, or attributes.

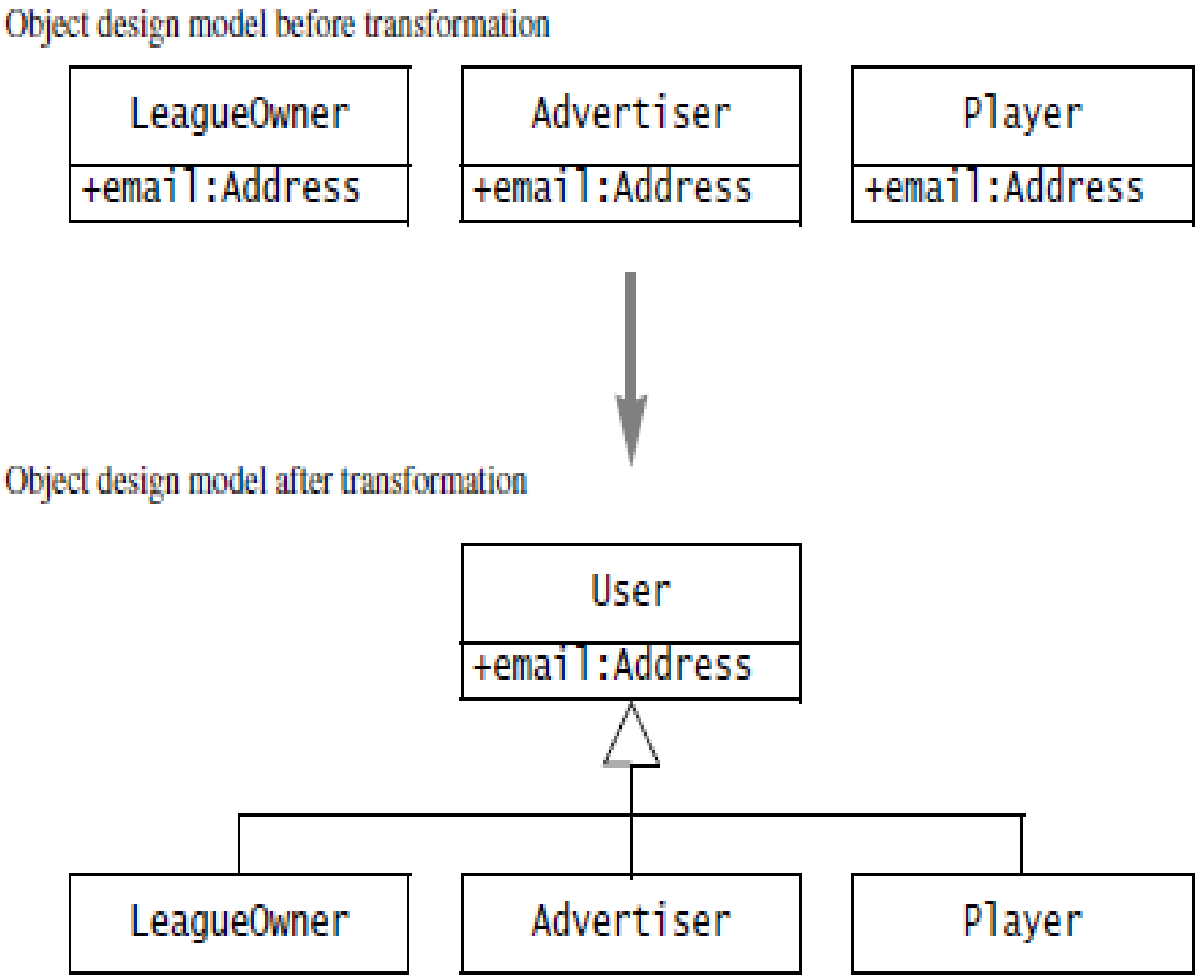


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

Mapping Models to Code

2. Refactoring

- A **refactoring** is a transformation of the source code that improves its readability or modifiability without changing the behavior of the system.

- Refactoring aims at improving the design of a working system.

- the refactoring is done in small incremental steps that are interleaved with tests.

Mapping Models to Code

- the object model transformation of Figure shown corresponds to a sequence of three refactorings.

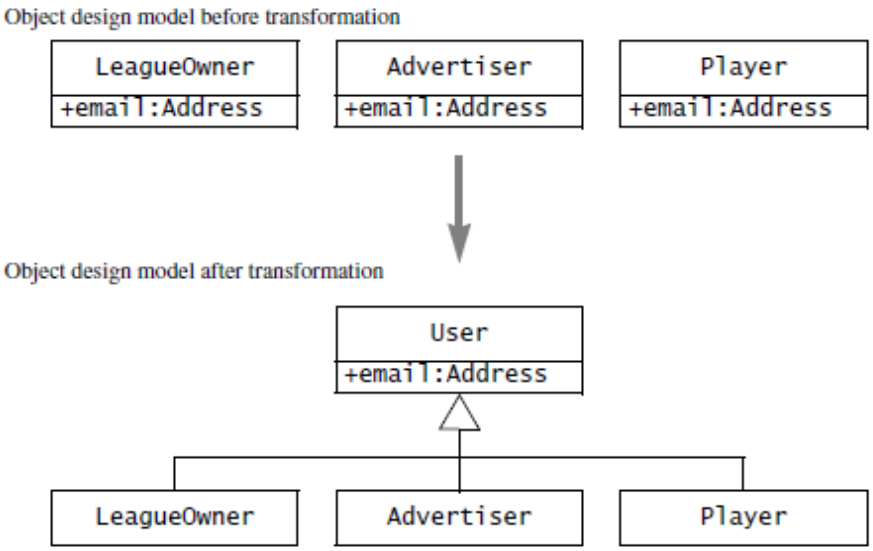


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

Mapping Models to Code

- The first one, Pull Up Field, moves the email field from the subclasses to the superclass User.

Pull Up Field relocates the `email` field using the following steps (Figure 10-3):

1. Inspect `Player`, `LeagueOwner`, and `Advertiser` to ensure that the `email` field is equivalent. Rename equivalent fields to `email` if necessary.
2. Create public class `User`.
3. Set parent of `Player`, `LeagueOwner`, and `Advertiser` to `User`.
4. Add a protected field `email` to class `User`.
5. Remove fields `email` from `Player`, `LeagueOwner`, and `Advertiser`.
6. Compile and test.

Mapping Models to Code

Before refactoring

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```

Figure 10-3 Applying the *Pull Up Field* refactoring.

Mapping Models to Code

- The second one, Pull Up Constructor Body, moves the initialization code from the subclasses to the superclass.

Then, we apply the *Pull Up Constructor Body* refactoring to move the initialization code for `email` using the following steps (Figure 10-4):

1. Add the constructor `User(Address email)` to class `User`.
2. Assign the field `email` in the constructor with the value passed in the parameter.
3. Add the call `super(email)` to the `Player` class constructor.
4. Compile and test.
5. Repeat steps 1–4 for the classes `LeagueOwner` and `Advertiser`.

Before refactoring

```
public class User {
    private String email;
}


public class Player extends User {
    public Player(String email) {
        this.email = email;
        //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        this.email = email;
        //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        this.email = email;
    //...
    }
}
```

After refactoring

```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);
    //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        super(email);
    //...
    }
}
```

Mapping Models to Code

- The third and final one, Pull Up Method, moves the methods manipulating the email field from the subclasses to the superclass.

1. Examine the methods of `Player` that use the `email` field. Note that `Player.notify()` uses `email` and that it does not use any fields or operations that are specific to `Player`.
2. Copy the `Player.notify()` method to the `User` class and recompile.
3. Remove the `Player.notify()` method.
4. Compile and test.
5. Repeat for `LeagueOwner` and `Advertiser`.

Mapping Models to Code

- Applying these three refactorings effectively transforms the ARENA source code in the same way the object model transformation

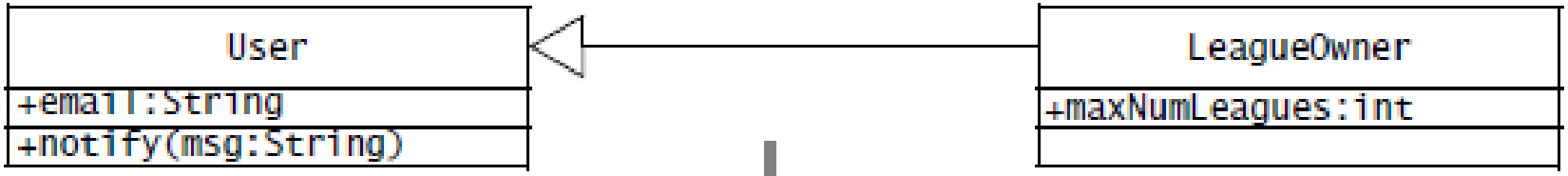
Mapping Models to Code

3. Forward Engineering

- **Forward engineering** is applied to a set of model elements and results in a set of corresponding source code statements, such as a class declaration, a Java expression, or a database schema.

- Purpose of forward engineering is **to maintain a strong correspondence between the object design model and the code**, and to **reduce implementation errors.**

Object design model before transformation



Source code after transformation

```
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String
value){
        email = value;
    }
    public void notify(String msg) {
        // ....
    }
    /* Other methods omitted */
}
```

```
public class LeagueOwner extends User
{
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues
                        (int value) {
        maxNumLeagues = value;
    }
    /* Other methods omitted */
}
```

Figure 10-5 Realization of the `User` and `LeagueOwner` classes (UML class diagram and Java excerpts). In this transformation, the public visibility of `email` and `maxNumLeagues` denotes that the methods for getting and setting their values are public. The actual fields representing these attributes are private.

Mapping Models to Code

4. Reverse Engineering

- **Reverse engineering** is applied to a set of source code elements and results in a set of model elements.

- **purpose** - to recreate the model for an existing system, either because the model was lost or never created, or because it became out of sync with the source code.

- Reverse engineering is essentially an inverse transformation of forward engineering.

Mapping Models to Code

- **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.***

2. ***Each transformation must be local***. A transformation should change only a few methods or a few classes at once.

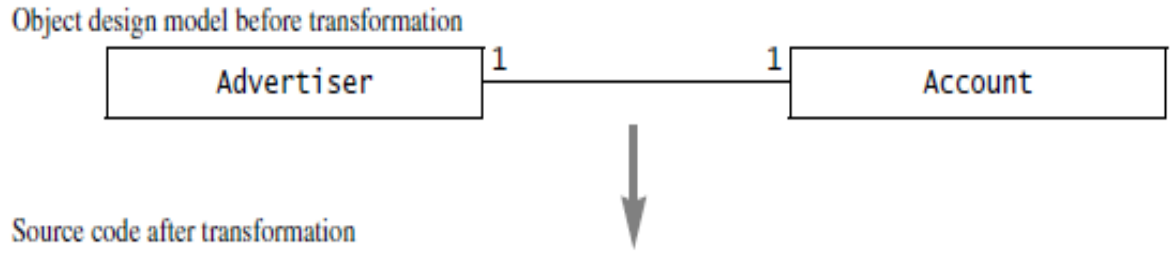
3. ***Each transformation must be applied in isolation to other changes*.**

4. ***Each transformation must be followed by a validation step*.**

Mapping Models to Code

- ***Mapping Associations to Collections***

1. Unidirectional one-to-one associations.



Source code after transformation

```
public class Advertiser {
    private Account account;
    public Advertiser() {
        account = new Account();
    }
    public Account getAccount() {
        return account;
    }
}
```

Figure 10-8 Realization of a unidirectional, one-to-one association (UML class diagram and Java).

Mapping Models to Code

2. Bidirectional one-to-one associations.

Object design model before transformation

Source code after transformation

```
public class Advertiser {
    /* The account field is initialized
     * in the constructor and never
     * modified. */
    private Account account;

    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}
```

```
public class Account {
    /* The owner field is initialized
     * during the constructor and
     * never modified. */
    private Advertiser owner;

    public Account(Advertiser owner) {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}
```

Figure 10-9 Realization of a bidirectional one-to-one association (UML class diagram and Java excerpts).

Mapping Models to Code

3. One-to-many associations.

Object design model before transformation

Source code after transformation

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount
            (Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner
            (Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
              newOwner.addAccount(this);
            if (old != null)
              old.removeAccount(this);
        }
    }
}
```

Figure 10-10 Realization of a bidirectional, one-to-many association (UML class diagram and Java).

Mapping Models to Code

4. Many-to-many associations.

Object design model before transformation

Source code after transformation

```
public class Tournament {
    private List players;
    public Tournament() {
        players = new ArrayList();
    }
    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }
}
```

```
public class Player {
    private List tournaments;
    public Player() {
        tournaments = new ArrayList();
    }
    public void addTournament
            (Tournament t) {
        if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
        }
    }
}
```

Figure 10-11 Realization of a bidirectional, many-to-many association (UML class diagram and Java).

Mapping Models to Code

- ***Mapping Contracts to Exceptions***

1. **Checking preconditions.**

- Preconditions should be checked at the beginning of the method, before any processing is done.

- There should be a test that checks if the precondition is true and raises an exception otherwise.

- Each precondition corresponds to a different exception

Mapping Models to Code

2. Checking post conditions.

- Post conditions should be checked at the end of the method, after all the work has been accomplished and the state changes are finalized.

- Post condition corresponds to a Boolean expression in an if statement that raises an exception if the contract is violated.

Mapping Models to Code

3. Checking invariants.

- When treating each operation contract individually, invariants are checked at the same time as post conditions.

4. Dealing with inheritance.

- The checking code for preconditions and post conditions should be encapsulated into separate methods that can be called from subclasses.

Mapping Models to Code

5. Coding effort.

- In many cases, the code required for checking preconditions and post conditions is longer and more complex than the code accomplishing the real work.

6. Increased opportunities for defects.

- Checking code can also include errors, increasing testing effort.

Mapping Models to Code

7. Performances drawback.

Checking systematically all contracts can significantly slow down the code, sometimes by an order of magnitude. Although correctness is always a design goal, response time and throughput design goals would not be met.

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p);
        }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

Figure 10-14 Example of exception handling in Java. `TournamentForm` catches exceptions raised by `Tournament` and `TournamentControl` and logs them into an error console for display to the user.

```
public class Tournament {
//...
    private List players;

    public void addPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //
```