

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. **PROJECT DATABASE**
that stores all relevant **configuration objects**
2. **VERSION MANAGEMENT CAPABILITY**
that **stores all versions** of a configuration object.
3. **A MAKE FACILITY**
that enables to **construct a specific version of the software**.
4. Version control and change control systems often implement a **ISSUES TRACKING** (also called **bug tracking**) **CAPABILITY**

VERSION CONTROL-Change set

- 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)**

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.

Engineering change order (ECO)

An **engineering change order (ECO)** is a documentation packet that outlines the

- proposed **change**,
- lists the product or part(s) that would be affected and
- requests review and
- approval from the individuals

who would be impacted or charged with implementing the **change**.

Elements of change management

1. Access Control

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

2. Synchronization Control

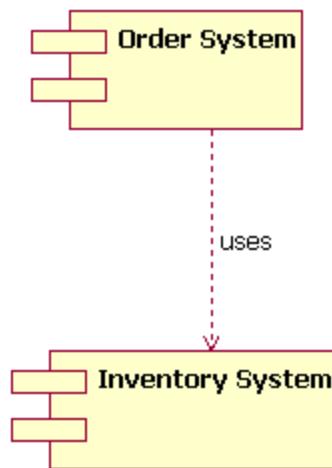
Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

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



COMPONENT DIAGRAM

THE COMPONENT DIAGRAM'S MAIN PURPOSE
IS TO SHOW THE STRUCTURAL RELATIONSHIPS
BETWEEN THE COMPONENTS OF A SYSTEM

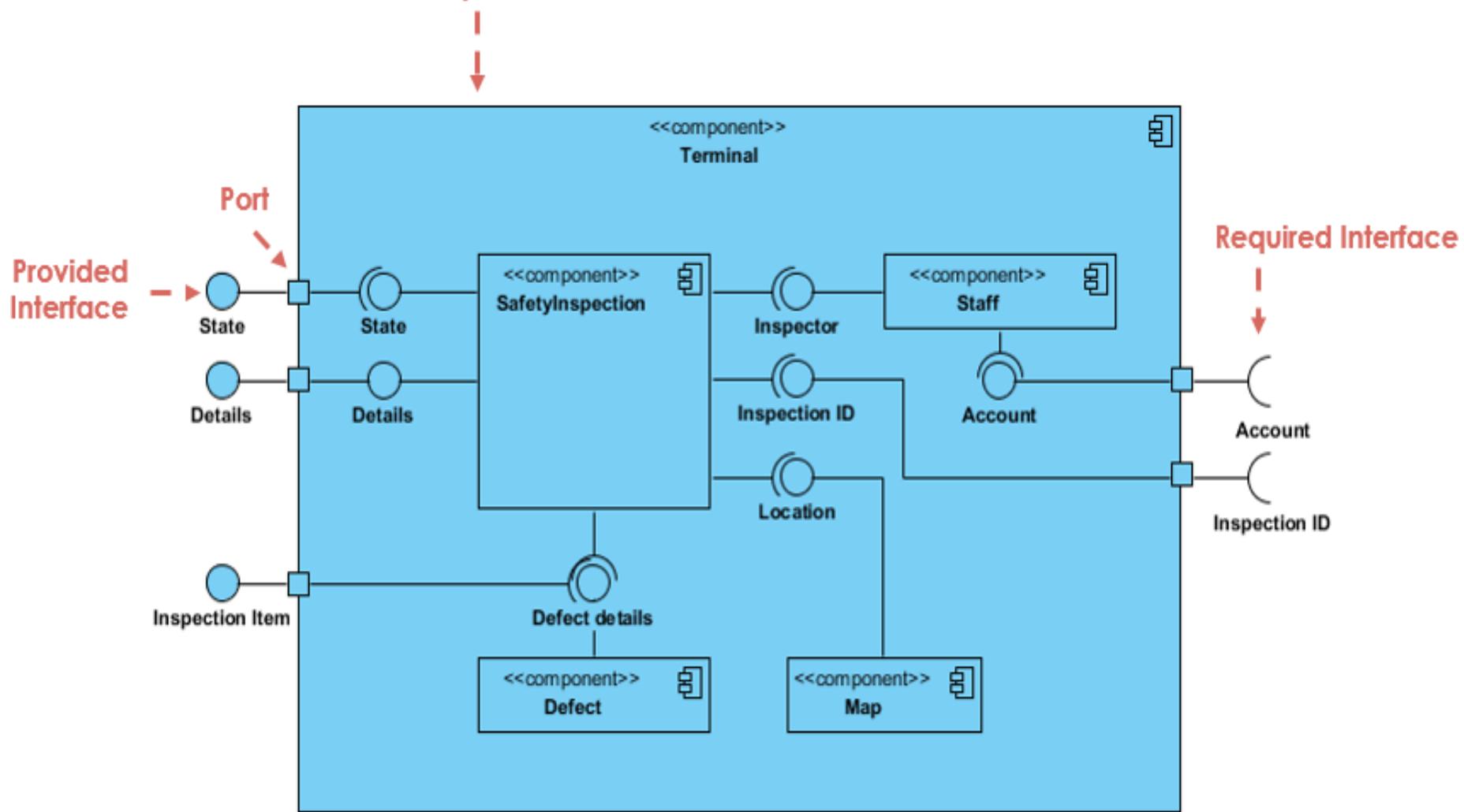


COMPONENT DIAGRAM

- 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

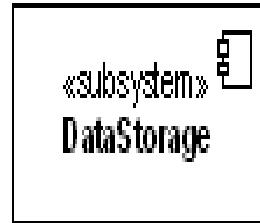
Component



INTERFACE

- **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 represent an interface that the component **requires** (in both cases, the interface's name is placed near the interface symbol itself).

Figure 6: An example of a subsystem element

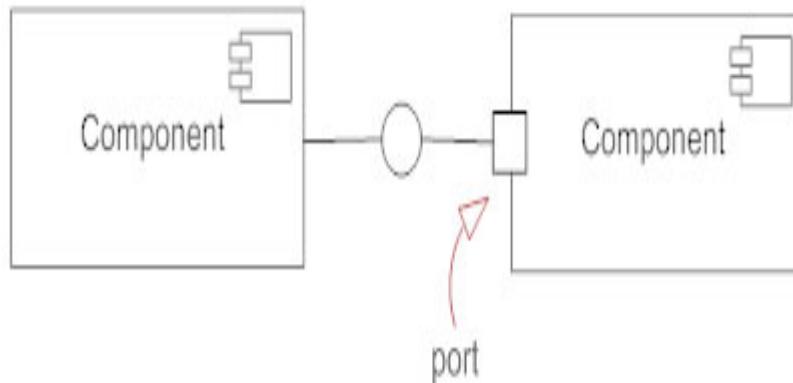


SUBSYSTEMS

- The subsystem classifier is a specialized version of a component classifier.
- The subsystem notation element inherits all the same rules as the component notation element.
 - The only difference is that a subsystem notation element has the keyword of subsystem instead of component.

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

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

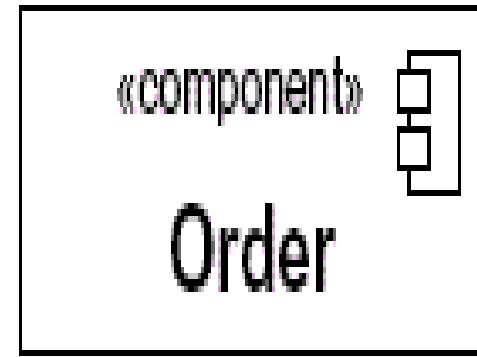
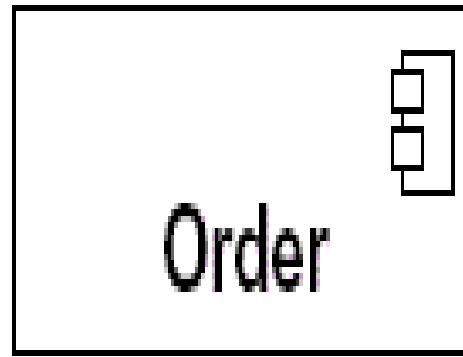


Figure 3: The additional compartment here shows the interfaces

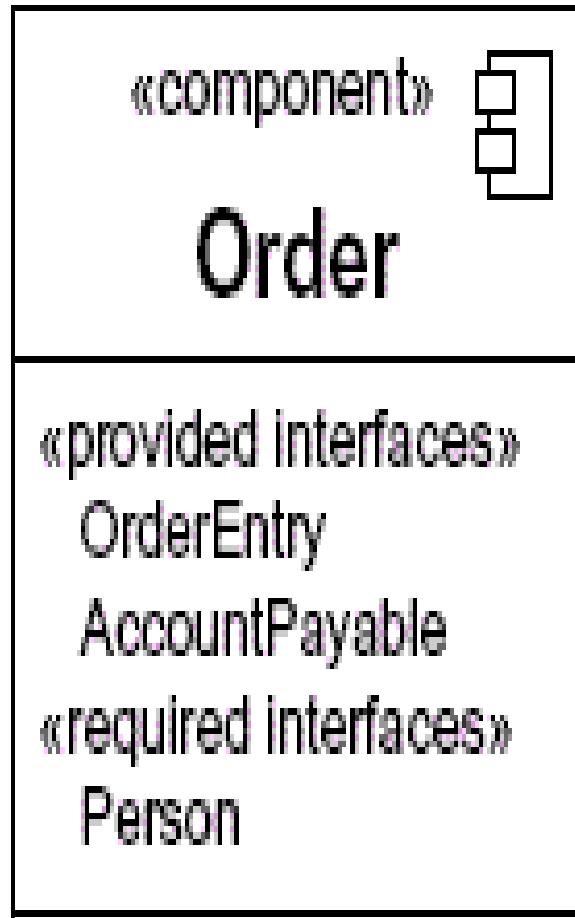
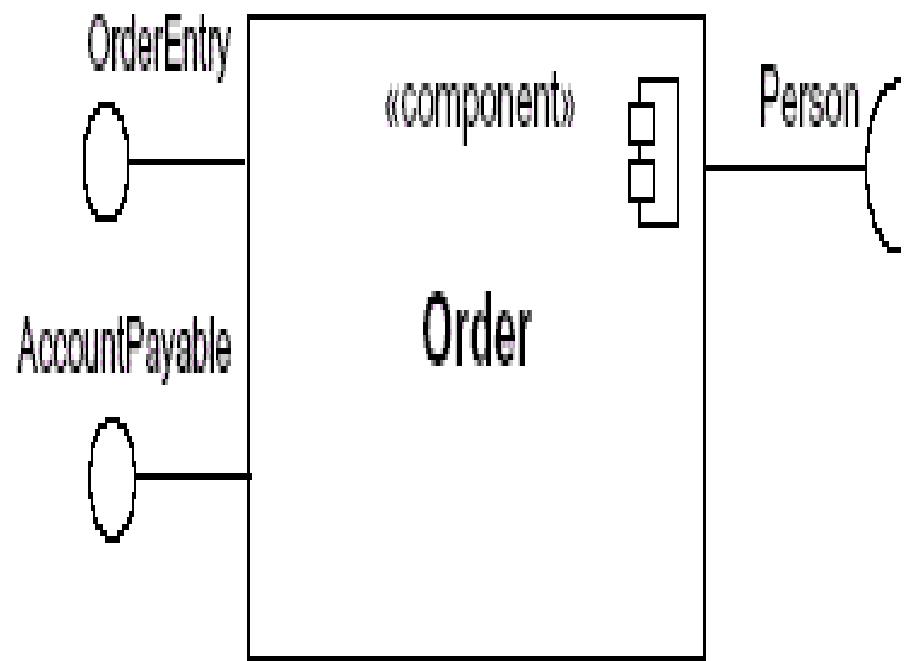


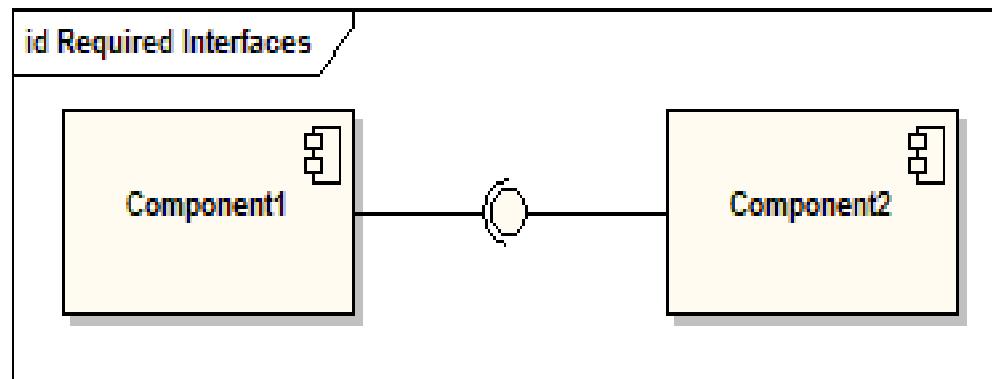
Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols



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



Component diagram

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

1) WEBSTORE SUBSYSTEM CONTAINS THREE COMPONENTS RELATED TO ONLINE SHOPPING

- **Search Engine**,
- **Shopping Cart**, and
- **Authentication**.

Search Engine component allows to search or browse items by exposing **provided interface**

- **Product Search** and uses **required interface**
- **Search Inventory** provided by **Inventory** component.

Shopping Cart component uses **Manage Orders interface** provided by **Orders** component during checkout.

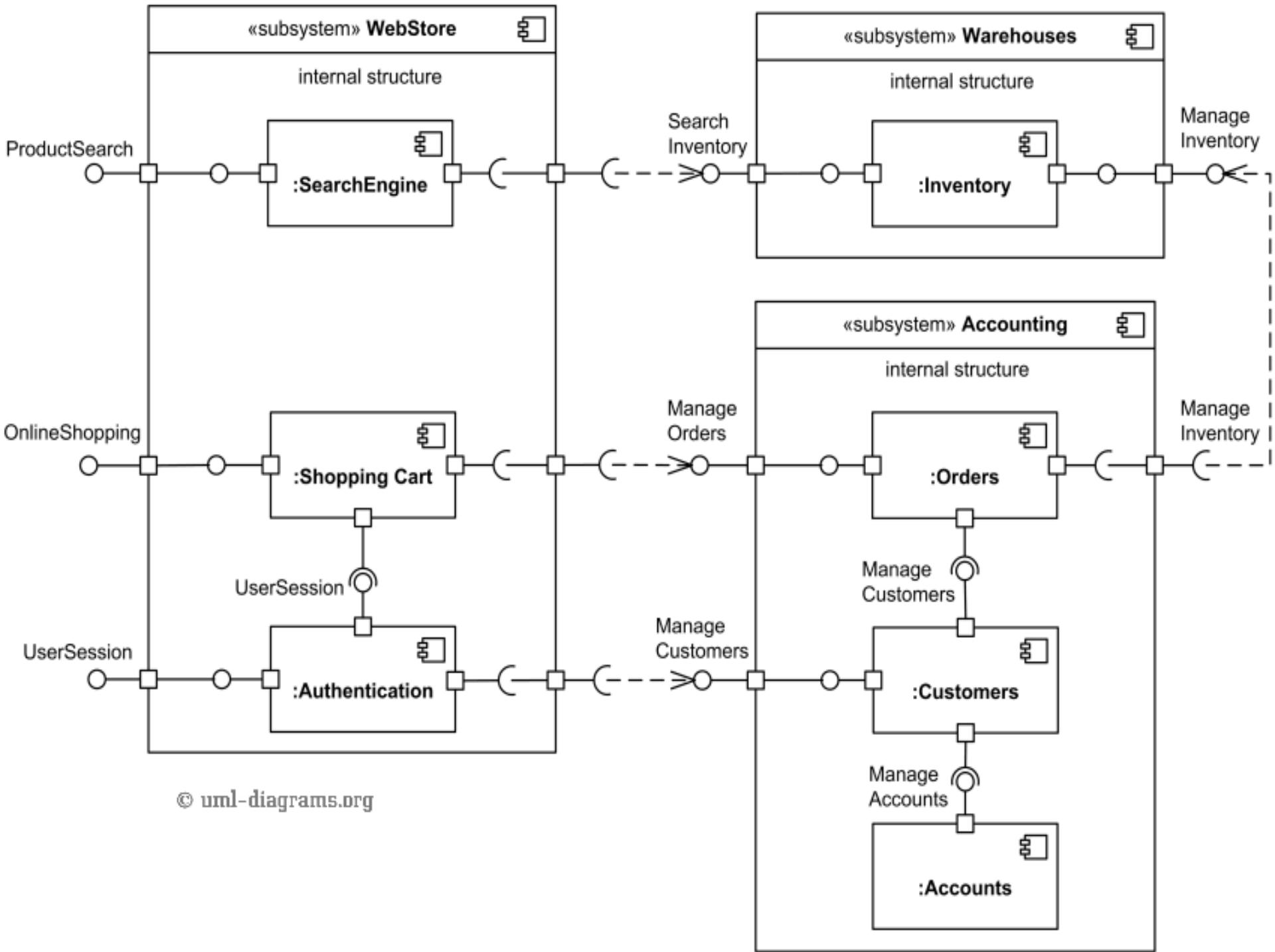
Authentication component allows customers to create account, login, or logout and binds customer to some account.

2) ACCOUNTING SUBSYSTEM PROVIDES TWO INTERFACES

- **Manage Orders** and **Manage Customers**.
- **Delegation connectors** link these external contracts of the subsystem to the realization of the contracts by **Orders** and **Customers** components.

3) WAREHOUSES SUBSYSTEM PROVIDES TWO INTERFACES

- **Search Inventory** and **Manage Inventory** used by other subsystems and wired through **dependencies**



DEPLOYMENT DIAGRAM

A **deployment diagram** is a UML **diagram** type that shows the **EXECUTION ARCHITECTURE** of a system, including **nodes** such as hardware or software execution environments, and the **middleware connecting** them.

Deployment diagrams are typically used to visualize the physical hardware and software of a system.

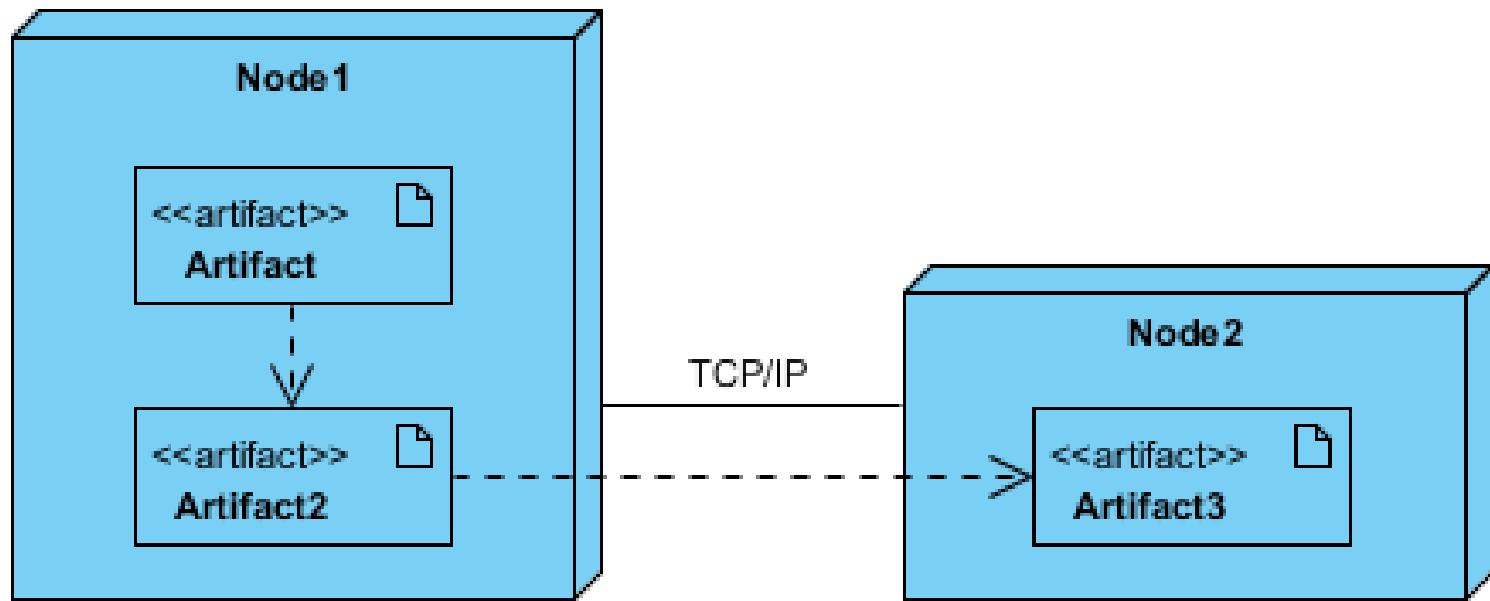
DEPLOYMENT DIAGRAM

- 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**.

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>>



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.



1..*

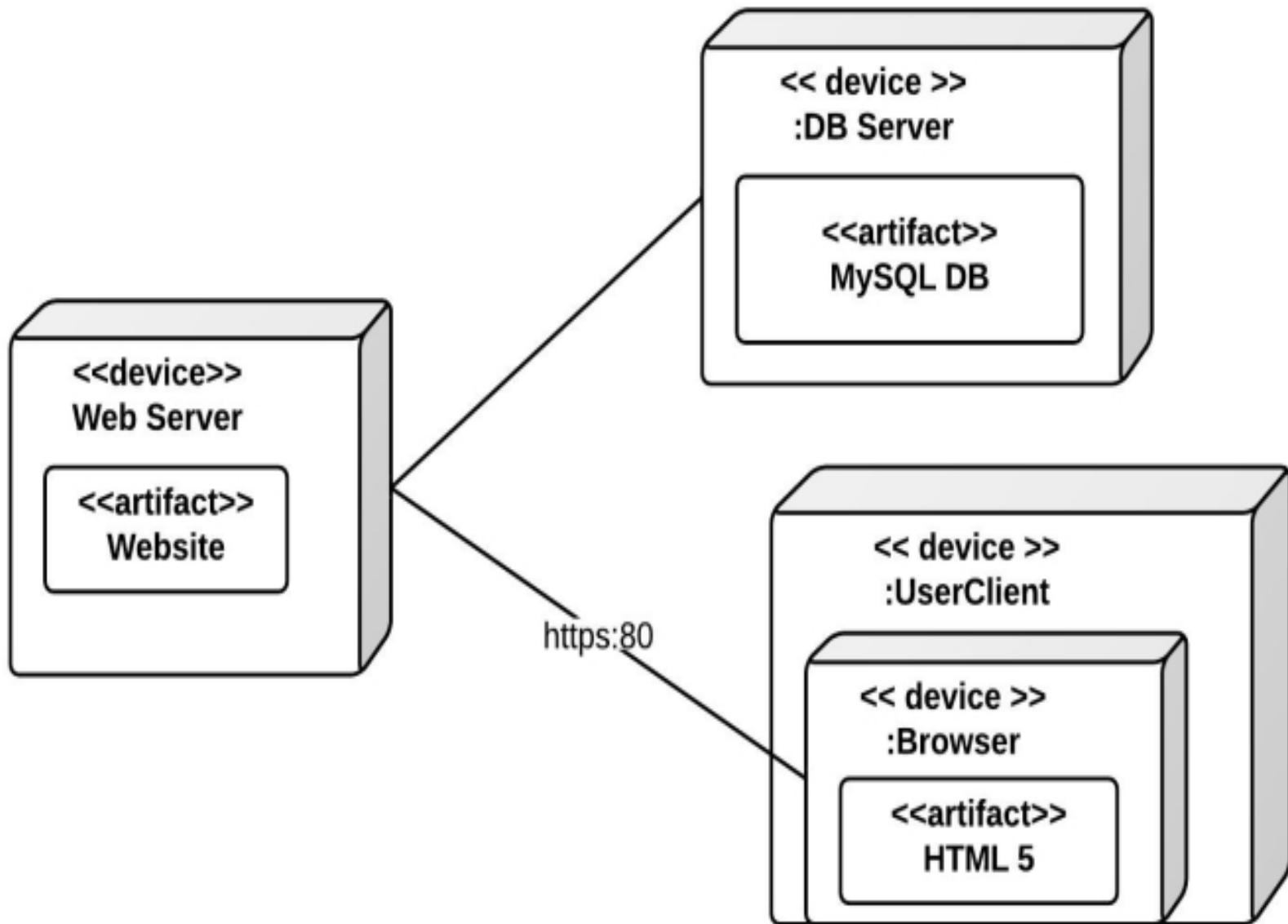
«TCP/IP»

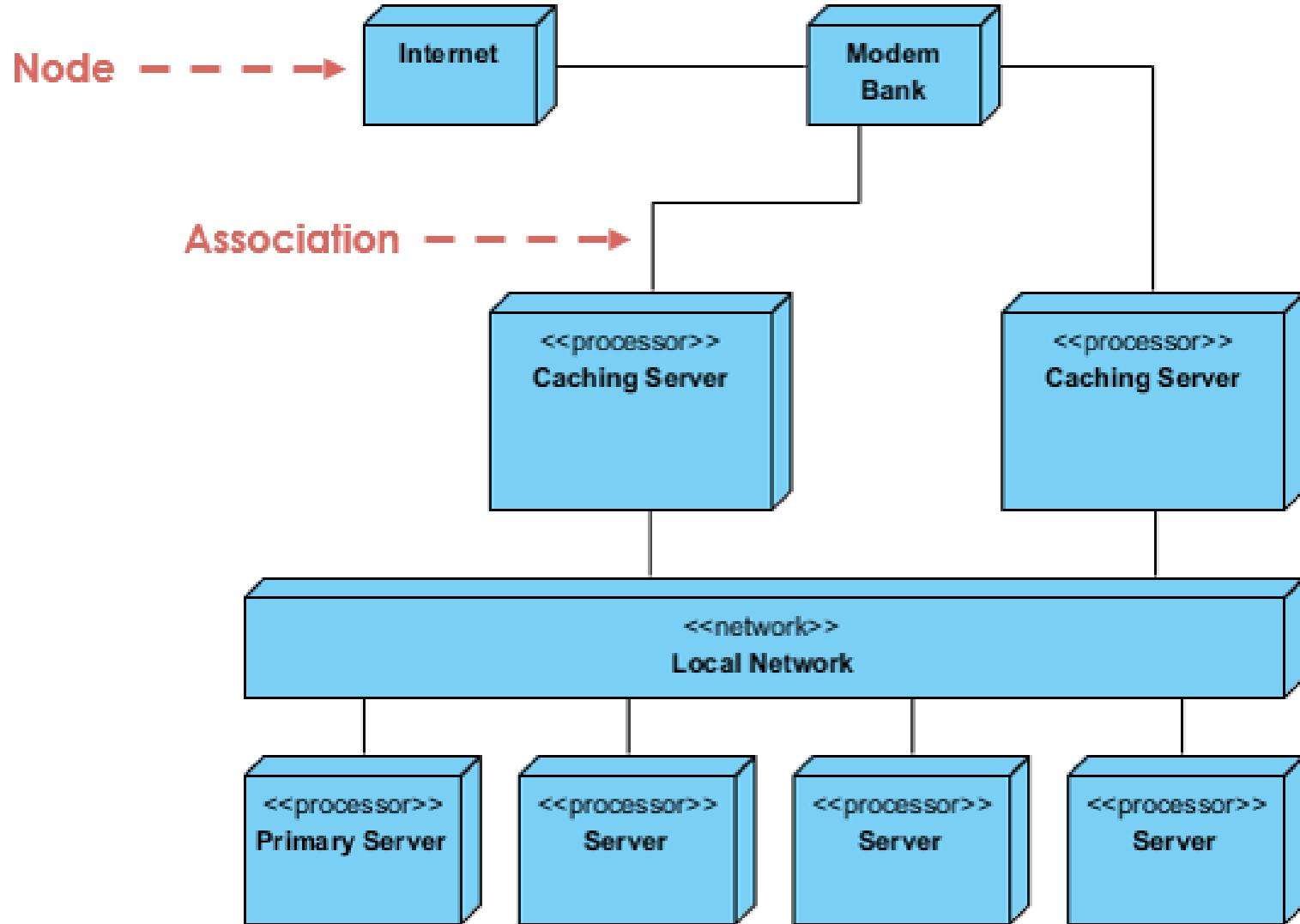
1



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.





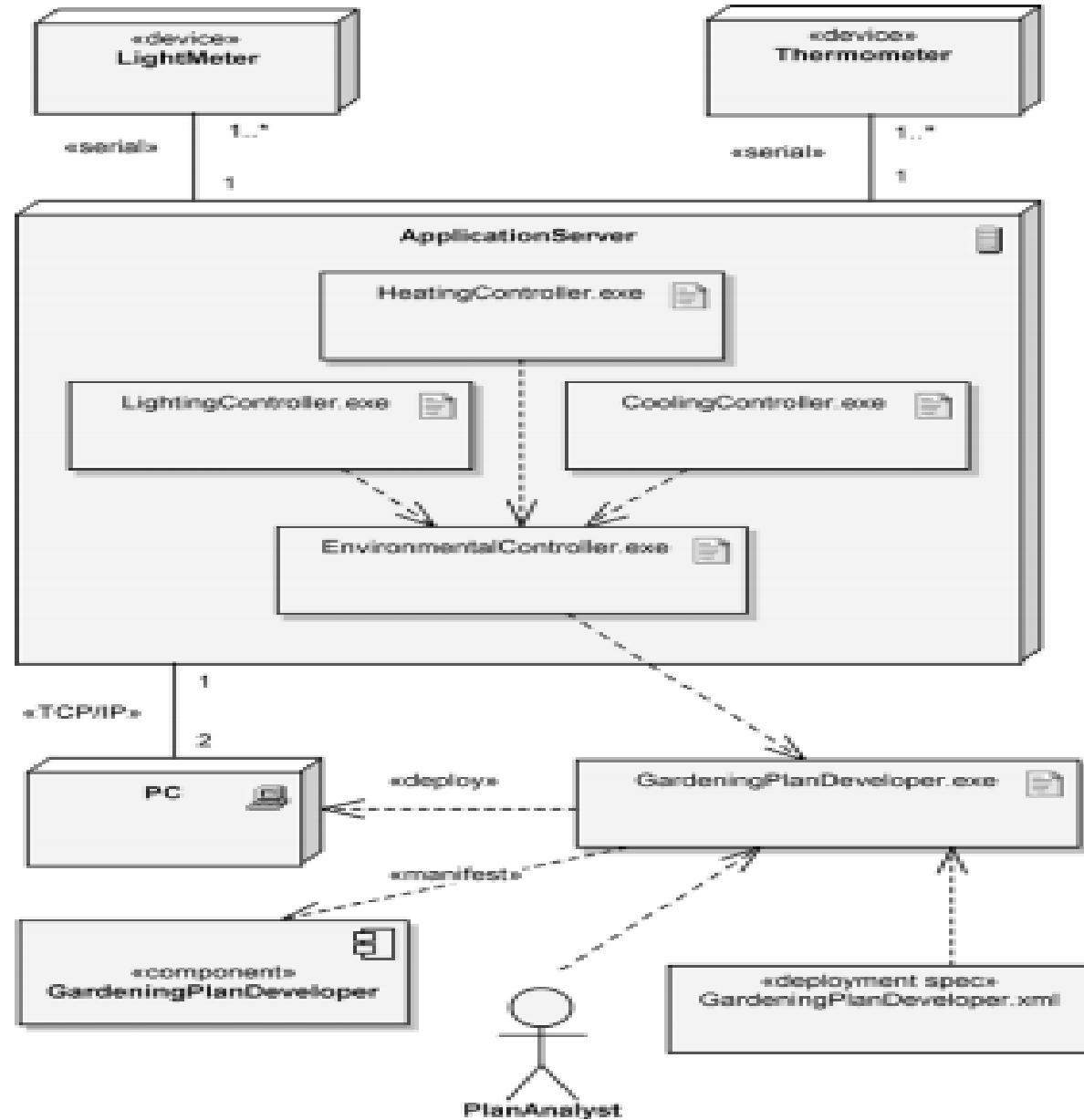
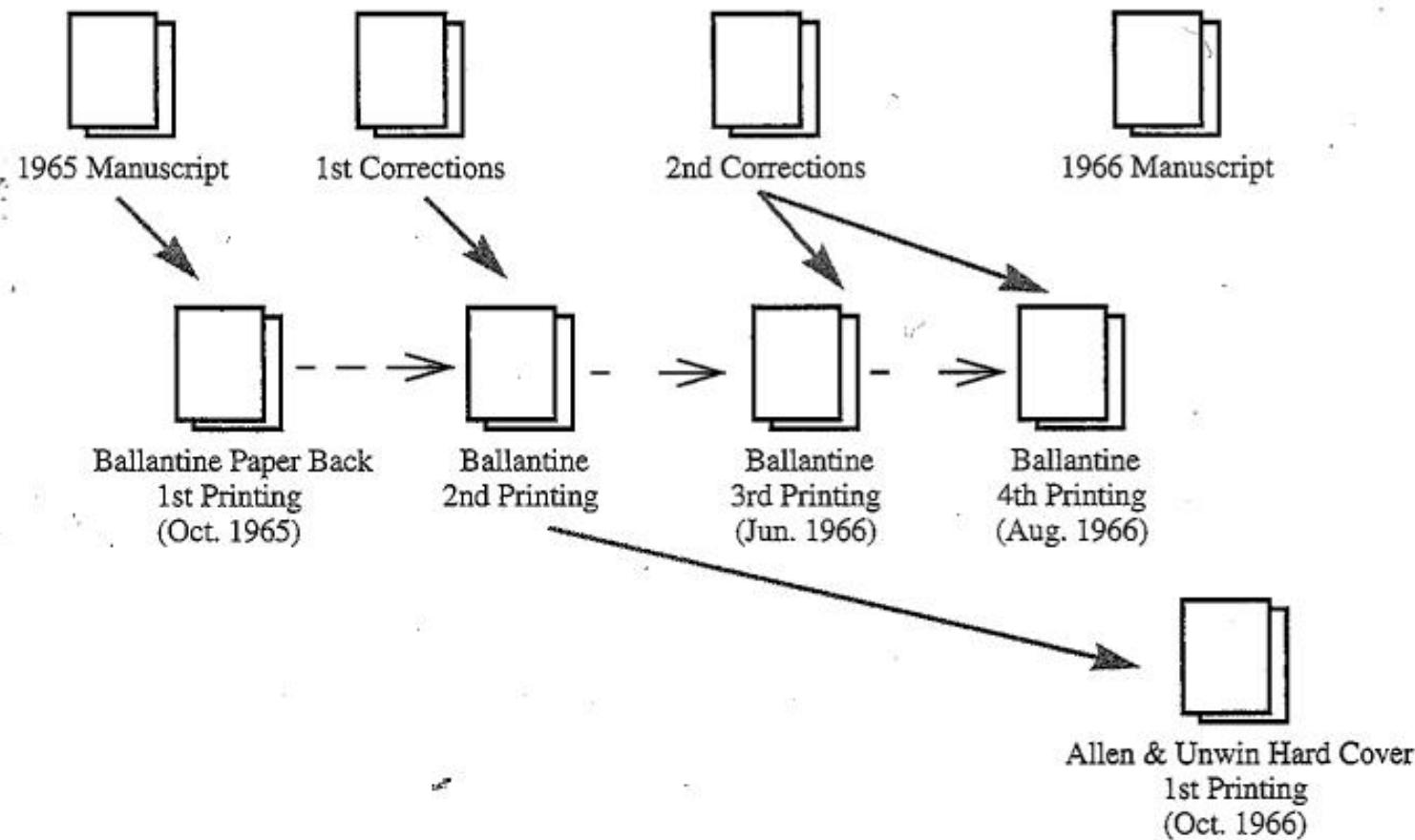


Figure 5–19 The Deployment Diagram for EnvironmentalControlSystem

MAPPING MODELS TO CODE



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.
- **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.

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

- Select 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.

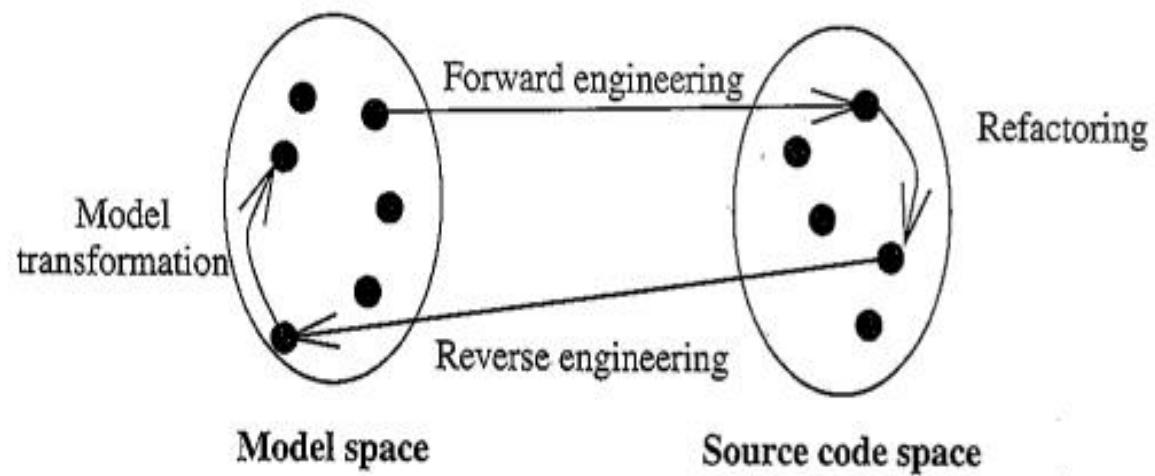


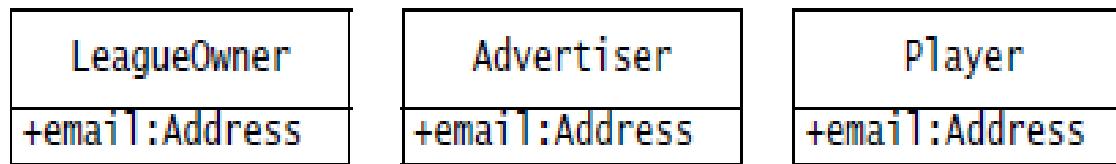
Figure 10-1 The four types of transformations described in this chapter: model transformations, refactorings, forward engineering, and reverse engineering.

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.

Object design model before transformation



Object design model after transformation

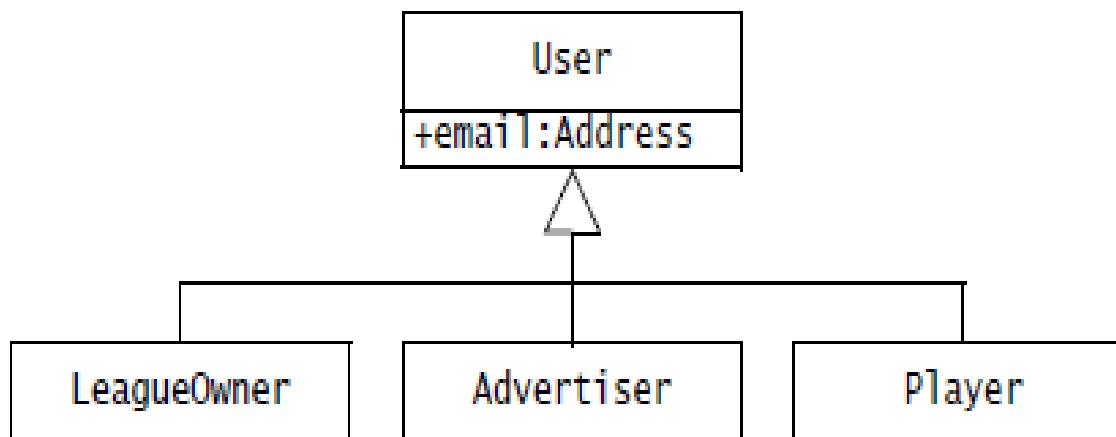


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

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.

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

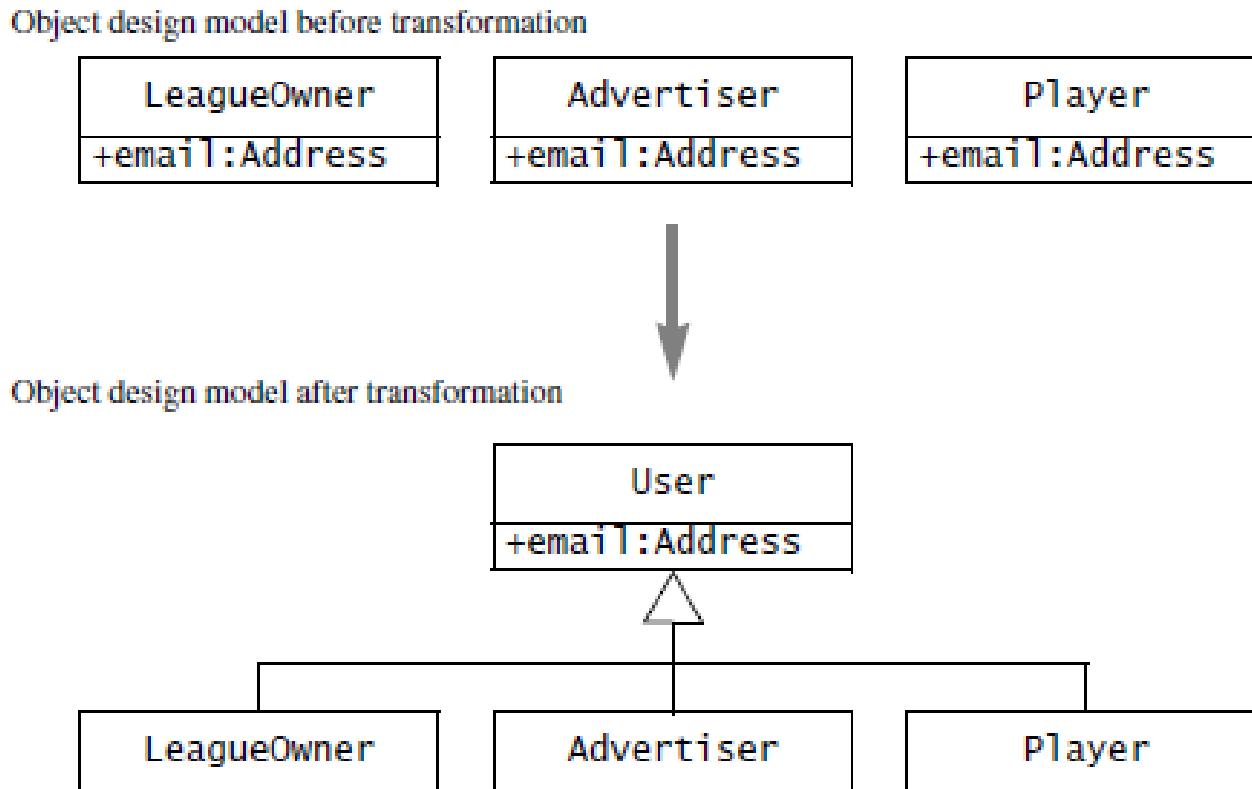


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

- The first one, **Pull Up Field**, moves the email field from the subclasses to the superclass User.
- Create a constructor in a superclass.
- Extract the common code from the beginning of the constructor of each subclass to the superclass constructor. Before doing so, try to move as much common code as possible to the beginning of the constructor.
- Place the call for the superclass constructor in the first line in the subclass constructors.

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.

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.

- 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`.

Problem

- Your subclasses have constructors with code that's mostly identical.

```
class Manager extends Employee { public Manager(String name, String id, int grade) { this.name = name; this.id = id; this.grade = grade; } // ... }
```

Solution

- Create a superclass constructor and move the code that's the same in the subclasses to it. Call the superclass constructor in the subclass constructors.

```
class Manager extends Employee { public Manager(String name, String id, int grade) { super(name, id); this.grade = grade; } // ... }
```

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

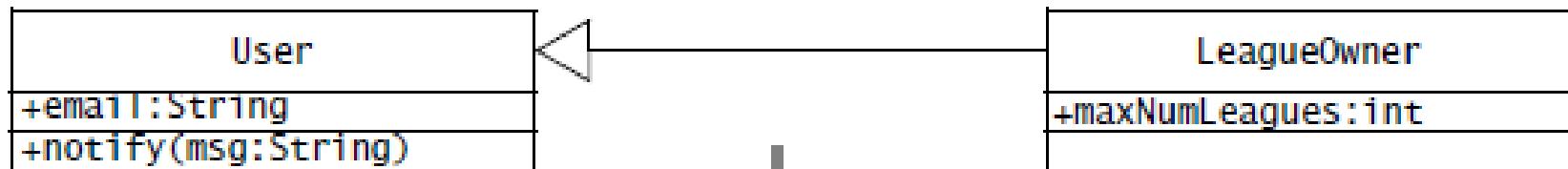
```
public class User {  
    public User(String email) {  
        this.email = email;  
    }  
}  
public class Player extends User {  
    public Player(String email) {  
        super(email);  
        //...  
    }  
}  
public class LeagueOwner extends User {  
    public LeagueOwner(String email)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

- 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.

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.

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.

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:

- 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 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

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.

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.

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.

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.