

# Object Oriented Analysis & Design

Week 10

Package and Deployment Diagrams

Lecturer: Dr. Msagha J Mbogholi, PhD

# Flashback from Lesson 9

- The activity diagram together with the state diagram represent the dynamic model. The dynamic model also supports other diagrams such as sequence diagrams.
- Activity diagrams allow modelling of a process as an activity that consist of a collection of nodes that are connected at their edges.
- Activity nodes are divided into three categories: action nodes, control nodes, and object nodes.
- The UML2 does not restrict how to create activity partitions (swimlanes) as long as they fulfill the desired objective; consequently, the designer can partition along use cases, classes, departments, or even roles.
- Action nodes represent some work being carried out and are shown as a rectangle with rounded corners.
- Control nodes manage the flow in an activity.
- Object nodes are special nodes that are used to show when instances of a class (objects) are available at a specific point during an activity.

# Content

- Introduction
- Package diagram
- Component diagram
- Deployment diagram



# Part 1

## Introduction

# Introduction

- In the lessons covered hitherto the different UML diagrams have been used in helping to visualize, and by extension understand, the problem domain.
- Each diagram explains a different aspect of the system to be developed arising from the problem domain.
- However, we note that none of these diagrams have addressed the idea of the hardware or the software that will be used to develop the system.
- In this lesson we describe these by discussing the package, component and deployment diagrams.
- The package diagram breaks down the system into smaller parts called packages which together bring like classes together and show how they are related as packages.
- The component diagram shows the different modules that make up the system and their dependencies.
- The deployment diagram shows how the software components of the system are deployed.
- All the above diagrams are key components of the overall system architecture.



# Part 2

## Package Diagram

# Introduction

- During the heyday of the WWE (now called WWF) wrestling craze in my country there was a wrestler who went by the stage name 'Total package'.
- What did this mean to wrestlers and fans alike? To us kids then it meant this guy was everything, muscle, moves, and even looks! This was before somebody spoilt it and told us it's all acting! Our love for wrestling (my friends and I, at least) took a big dip from then.
- Surprisingly we all know what a package is since we use this term in everyday life; for example, you received a package from Anne. The concept in real life (yes you guessed it) is the same in object oriented analysis and design.
- Package diagrams bring together different related artefacts in OO into one single diagram.

# Introduction

- As per UML principles, the building blocks of UML consist of things (classes, actors, and so on), relationships and diagrams.
- A package can be looked at as a collection of elements; it is a grouping thing or a container which contains elements.
- The elements in packages are related to each other by some definition and this is why they are grouped together.
- By organizing these elements together they can be used more efficiently and the structure is more understandable.
- Every package has its own namespace (it's own unique name and names associated with it) and all the elements in the package have unique names. 8



# Introduction (cont'd)

- A package can be used to (Arlow and Neustadt, 2013):“
  - Provide an encapsulated namespace within which all names must be unique;
  - Group semantically related elements;
  - Define a semantic boundary in the model;
  - Provide units for parallel working and configuration management “
- In UML 2 packages are logical groupings and thus the elements of a package are semantically related; they are not physical (these are covered in a different part of this lesson).
- Every element belongs to one and only one package; further packages are arranged in a hierarchy, with the top most package denoted as **<<topLevel>>**. If an element is not assigned a package then by default it will belong to the topLevel package.

# Introduction (cont'd)

- The package hierarchy is also extended to the namespace with naming beginning at the topLevel namespace which is also the root.
- Packages may contain use cases, analysis cases, or use case realizations.
- The UML syntax allows you to describe a package at three levels, according to the level of detail that one desires to show. However, the package notation is shown simply as a folder regardless of the detail level.
- Fig 1 shows the UML notation for a package

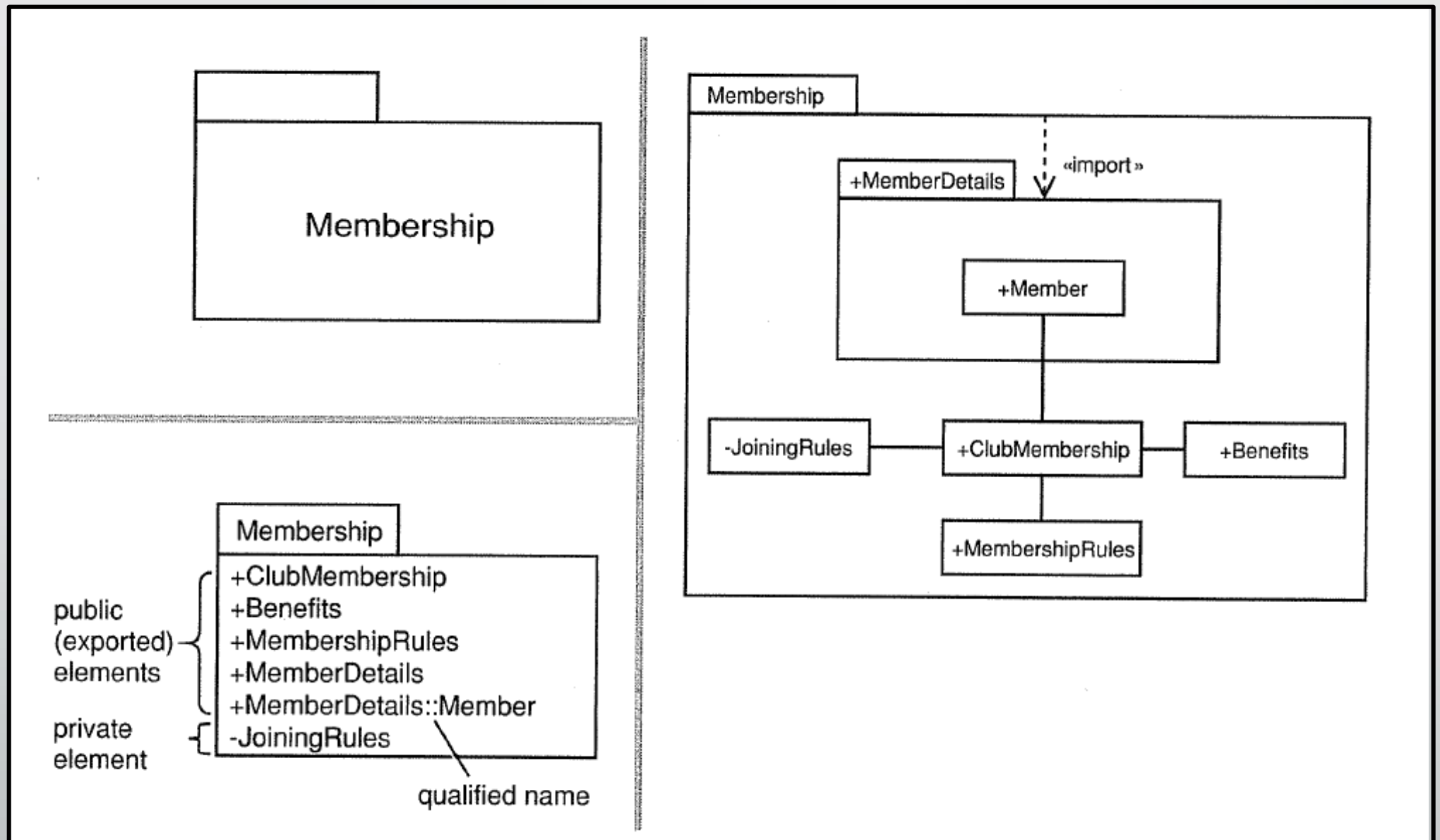


Fig 1. Package notation (Arlow and Neustadt, 2013)

# Introduction (cont'd)

- Package elements have visibility properties which declare their levels of visibility to clients of the package. These are summarized in table 1

Table 1. Visibility status of package elements (Arlow and Neustadt, 2013)

Symbol	Visibility	Semantics
+	public	Elements with public visibility are visible to elements outside the package – they are <i>exported</i> by the package
–	private	Elements with private visibility are completely hidden inside the package

# Introduction (cont'd)

- The exported elements of a package create an interface between the package and other elements and so it is advisable to have this interface exposure to be as limited as possible. One possible way to achieve this is to increase the package elements with private visibility and minimize those with public visibility.
- The UML provides two standard stereotypes for customizing packages for specific intentions. These are shown in table 2.

Table 2. Package stereotypes (Arlow and Neustadt, 2013)

Stereotype	Semantics
«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages

## 2.1 Packages and namespaces

- As described earlier all the element names in the package are semantically related.
- Further the boundaries of the package act like the system boundary in a use case diagram; it defines what belongs to the package, and what is outside of the package.
- When an element is one package and wishes to communicate with or refer to, another element in another package it must define the name of the element and the path to get to it (how to navigate to the other element).
- This path is referred to as the qualified name or pathname.
- A qualified name is created by prefixing the element name with the names of the packages in which it resides, separated by a double colon. The order is to start with the outer package name and work successively in the hierarchy till you arrive at the desired element.
- This works a lot like the directory structure in your local PC or laptop running some operating system such as Windows. An example is shown in fig 2. In this case the qualified name of the class Librarian is
  - `Library::Users::Librarian`

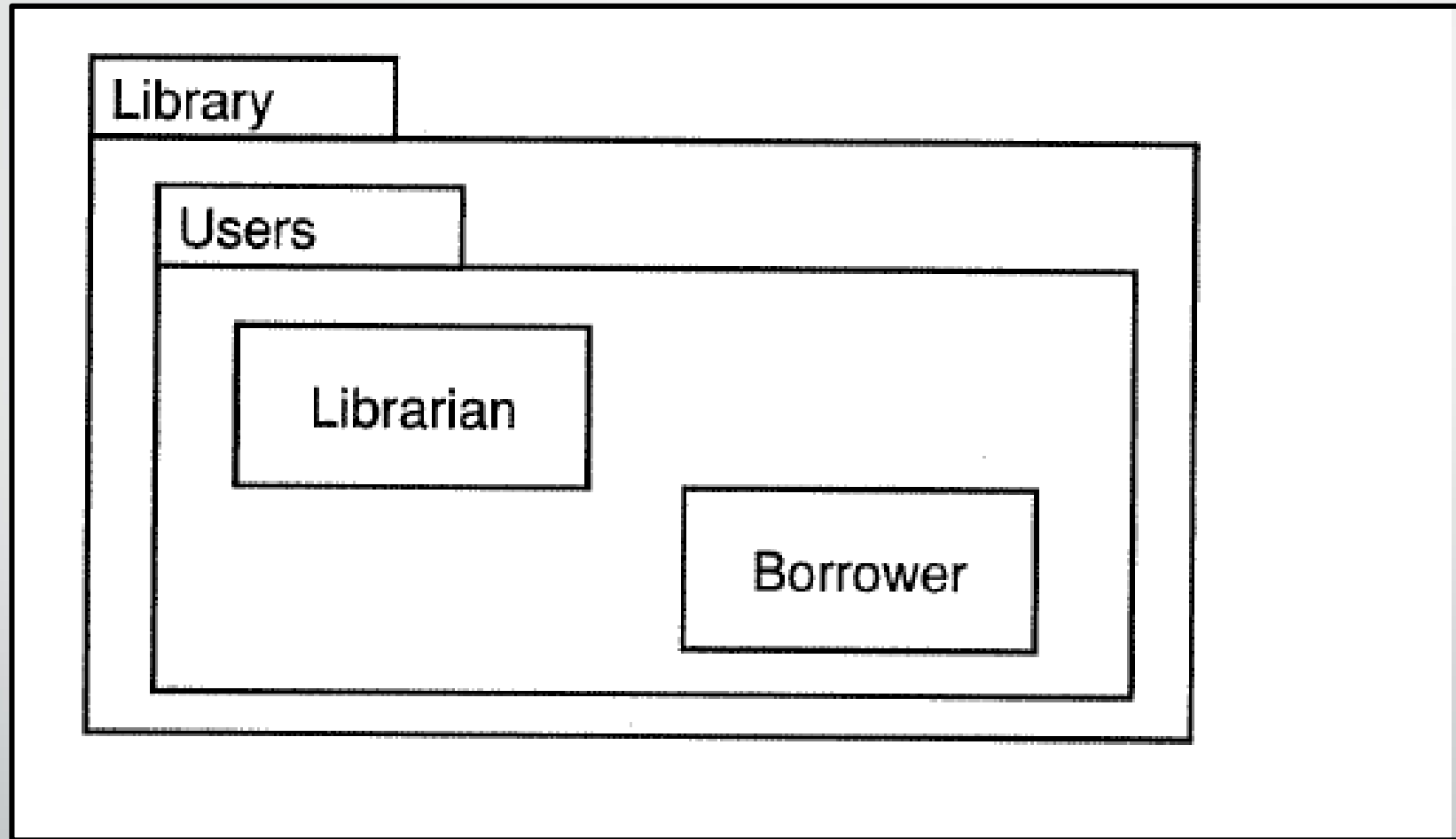


Fig 2. Librarian package (Arlow and Nuestadt, 2013)

## 2.2 Nested packages

- As you may have noticed the librarian class is actually nested within the library package.
- This presents one way of showing nested packages in a hierarchy; as long as the principle of containers is clear, then the nesting concept in fig 2 is rather evident.
- When there is a lot of nesting or when the nesting is complex then an alternative design is needed.
- Fig 3 shows the same library package of fig 2 with its elements using the alternative design



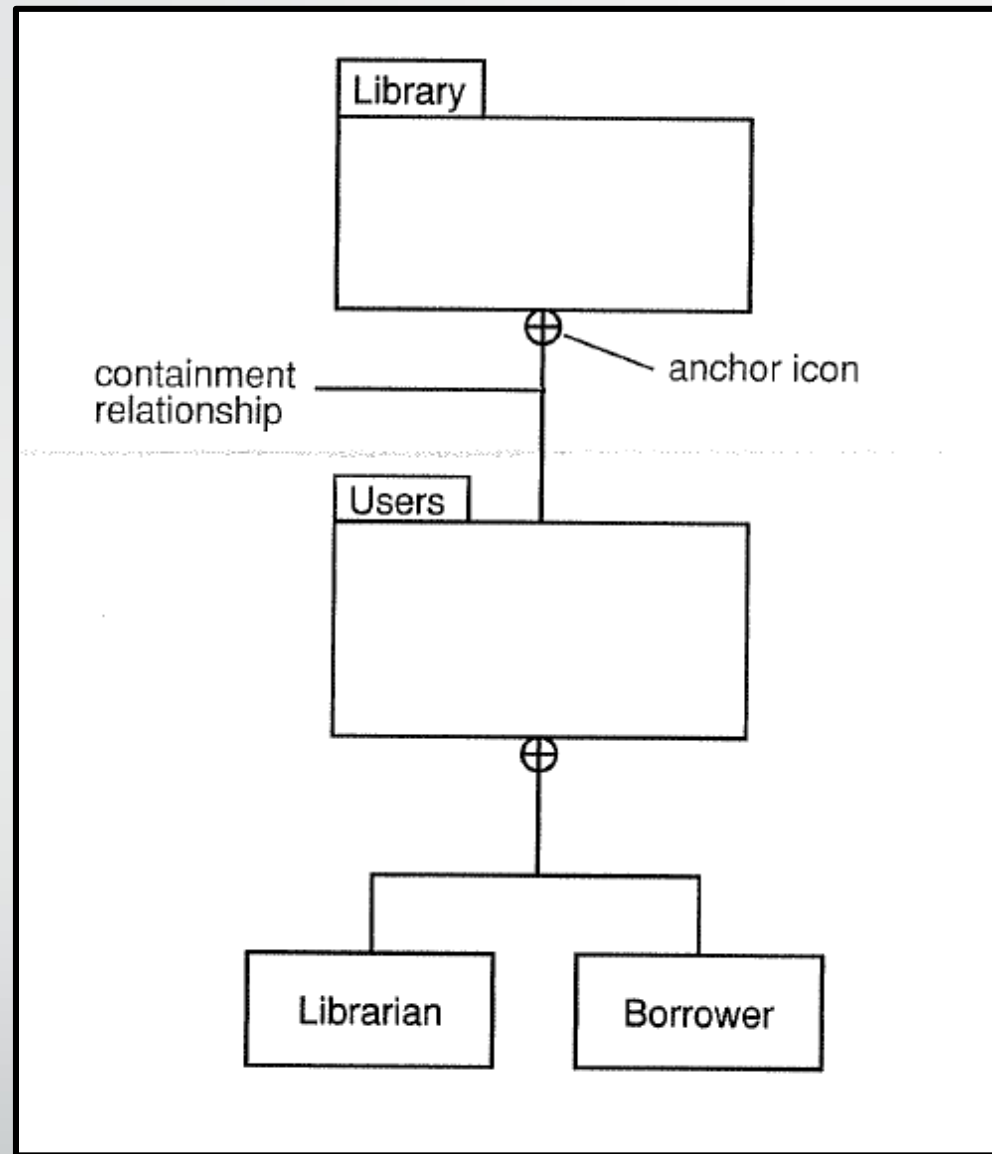


Fig 3. Nesting within a package (Arlow and Neustadt, 2013)

## 2.2 Nested packages (cont'd)

- The rules for nested packages are as follows (Arlow and Neustadt, 2013):
- Nested packages have access to the namespace of their owning package; for example, elements in the Users package have access to elements in the Library package using unqualified names (they can refer to them directly).
- However, elements in the owning package do not have this luxury; they must refer to elements in the owned packages by their pathnames; for example elements in the Library package will refer to elements in the Users package as `Users::Librarian` or `Users::Borrower` to access the two elements, respectively.

## 2.3 Package dependencies

- A package may be related to another package by dependency.
- Fig 4 shows a dependency relationship with the Membership package such that any package with this relationship will be able to see the public elements of the package but not the elements declared as private.
- There are different types of dependencies (just like in human relationships, you depend on friends for different things, parents for other things, siblings for yet other things, your lecturer for something different, and so on). These dependencies are defined in table 3.

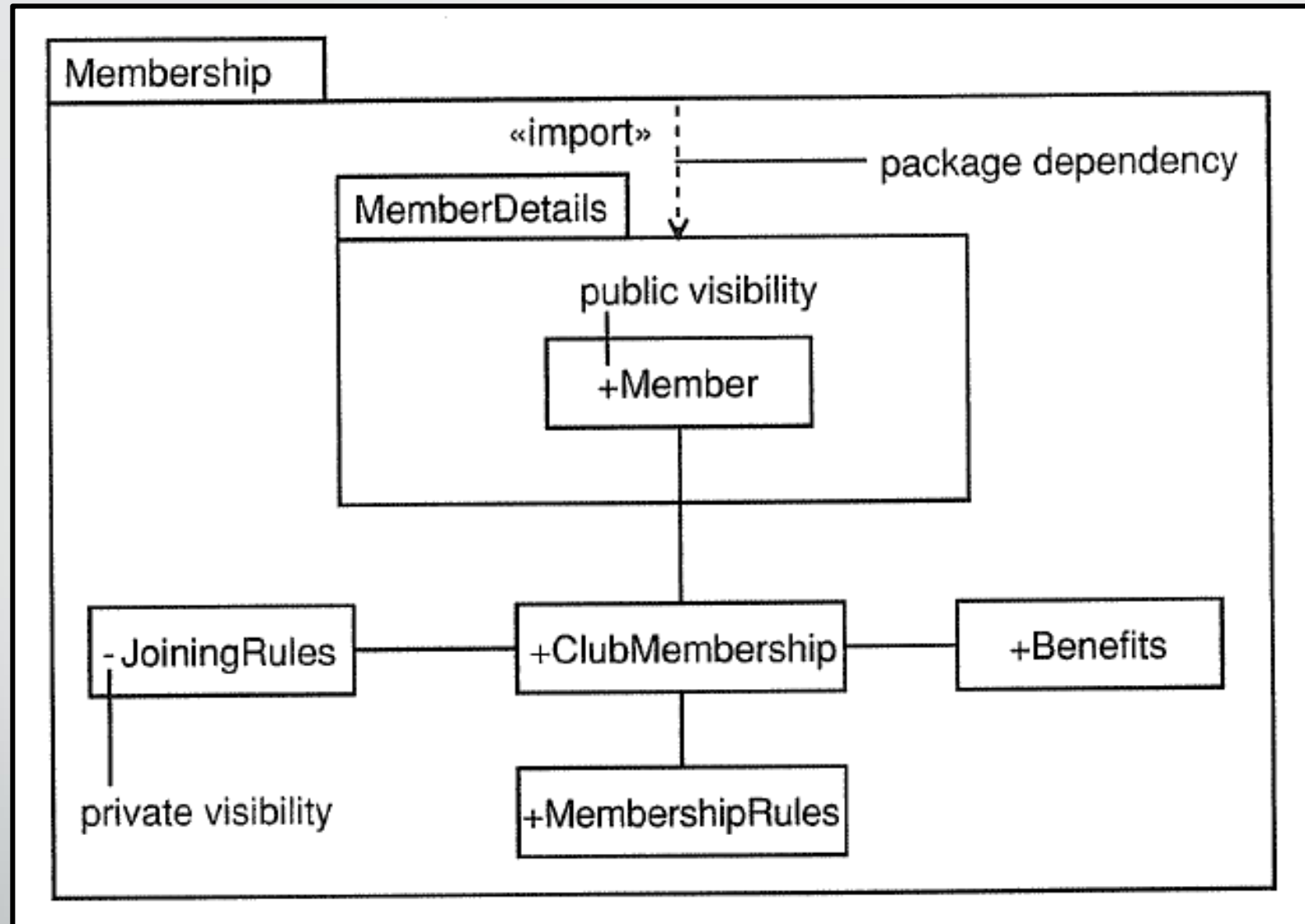



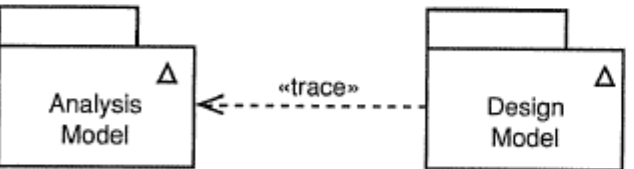



Fig 4. Element visibility in package (Arlow and Neustadt, 2013)

Table 3. Package dependencies (Arlow and Neustadt, 2013)

Package dependency	Semantics
 <pre> graph LR     Client -- «use» -.-&gt; Supplier             </pre>	<p>An element in the client package uses a public element in the supplier package in some way – the client depends on the supplier</p> <p>If a package dependency is shown without a stereotype, then «use» should be assumed</p>
 <pre> graph LR     Client -- «import» -.-&gt; Supplier             </pre>	<p>Public elements of the supplier namespace are added as public elements to the client namespace</p> <p>Elements in the client can access all public elements in the supplier using unqualified names</p>
 <pre> graph LR     Client -- «access» -.-&gt; Supplier             </pre>	<p>Public elements of the supplier namespace are added as private elements to the client namespace</p> <p>Elements in the client can access all public elements in the supplier using unqualified names</p>
 <pre> graph LR     DesignModel[Design Model Δ] -- «trace» -.-&gt; AnalysisModel[Analysis Model Δ]             </pre>	<p>«trace» usually represents a historical development of one element into another more developed version – it is usually a relationship between models rather than elements (an extra-model relationship)</p>
 <pre> graph LR     Client -- «merge» -.-&gt; Supplier             </pre>	<p>Public elements of the supplier package are merged with elements of the client package</p> <p>This dependency is only used in metamodeling – you should not encounter it in ordinary OO analysis and design</p>

## 2.3.1 Transitivity

- The transitivity rule states that if A is related to B, and B is related to C, then it follows that A is related to C. The relationship is usually depicted as follows:
  - If  $A \Rightarrow B$ , and  $B \Rightarrow C$ , then by transitivity  $A \Rightarrow C$ .
- In package dependency the following applies:
- `<<import>>` dependency is transitive; that is to say
  - If package A `<<import>>` B, and B `<<import>>` C, then A `<<import>>` C
- However, the `<<access>>` dependency is not transitive. This is demonstrated in fig 5.

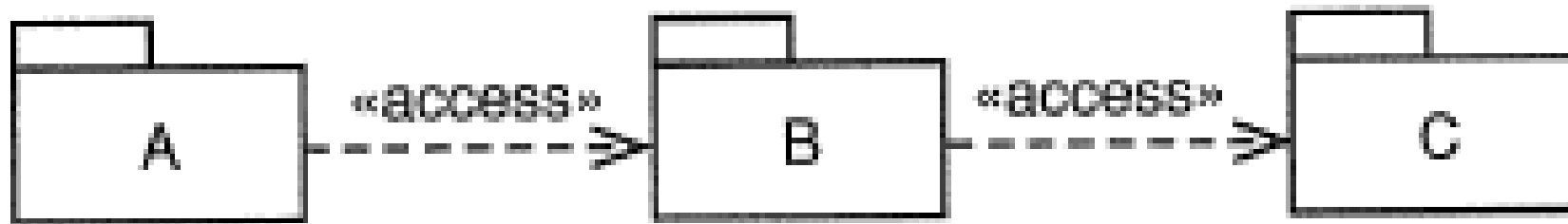


Fig 5. Transitive dependency in <<access>> (Arlow and Neustadt, 2013)

## 2.3.1 Transitivity (cont'd)

- In fig 5, package A accesses package B, and package B accesses package C.
- The following applies by the <<access>> dependency definition:
- Public elements in C become private elements in B;
- Public elements in B become private elements in A;
- This implies elements in A can't access elements in C.
- Using the <<access>> property nothing is accessed unless it is explicitly accessed.



## 2.4 Package generalization

- Packages are generalized much in the same way as it is done in the class diagram.
- Child package elements inherit the public elements of the parent package.
- Child packages may add new elements and may override elements of the parent package by providing an alternative element with the same name.
- Fig 6 demonstrates implementation of generalization. The two child classes inherit the public elements of the parent package and override the Item element with their own respective versions while adding their own elements to their respective packages.
- The substitutability principle must also apply like in class inheritance (Arlow and Neustadt, 2013) – anywhere the Product package is used, we should be able to use the child packages.

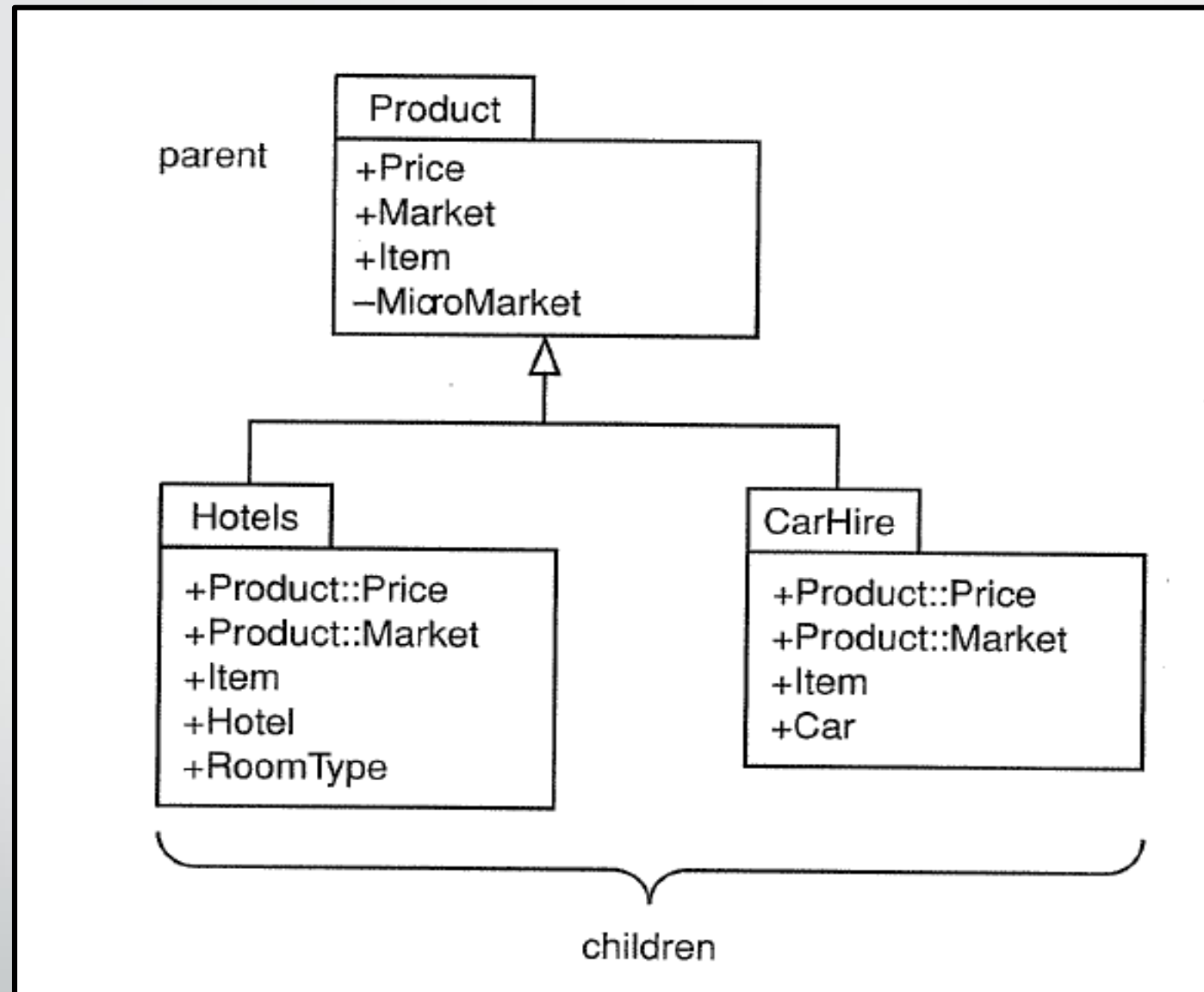


Fig 6. Package generalization (Arlow and Neustadt, 2013)

## 2.5 Guidelines

- Dennis et al. (2015) describe a five step process to creating a package diagram:
- Step 1: Set context (which context, for example public domain layer)
- Step 2: Cluster (group) the classes (using semantic relationships)
- Step 3: Create packages
- Step 4: Identify the dependencies (see table 3)
- Step 5: Lay out and draw the diagram



# Part 3

## Component Diagram

# Introduction

- A subsystem is an independent part of a system.
- Subsystems are designed to minimize coupling in a system by designing appropriate interfaces and to ensure that each subsystem correctly realizes the behavior specified by its interfaces. (Arlow and Neustadt, 2013).
- An interface is simply a point where two systems meet and interact; it may enhance the meeting experience or do nothing other than to provide the means for the communication.
- Interfaces and components are intertwined in the UML as shall be seen in this section; we can't discuss one without mentioning the other.

## 3.1 Interfaces

- An interface has been defined in the introduction part of this section.
- In the UML an interface specifies a named set of public features
- The principle behind interfaces is to separate the specification of functionality from its implementation by a classifier such as a class or subsystem.
- Interfaces are not instantiable.
- Interfaces declare a contract that may be realized by a classifier (Arlow and Neustadt, 2013).
- Table 4 shows the features that can be specified by interfaces; it also shows the responsibilities of realizing classifiers with respect to the interface.

Table 4. Interfaces and realizing classifiers (Arlow and Neustadt, 2013)

Interface specifies	Realizing classifier
Operation	Must have an operation with the same signature and semantics
Attribute	Must have public operations to set and get the value of the attribute – the realizing classifier is <i>not</i> required to actually have the attribute specified by the interface, but it must behave as though it has
Association	Must have an association to the target classifier – if an interface specifies an association to another interface, the implementing classifiers of these interfaces must have an association between them
Constraint	Must support the constraint
Stereotype	Has the stereotype
Tagged value	Has the tagged value
Protocol (e.g., as defined by a protocol state machine – see Section 21.2.1)	Must realize the protocol

## 3.1 Interfaces (cont'd)

- The attributes and operations in an interface should be fully specified and Arlow and Neustadt (2013) aver should include the following:
- The complete operation signature (names, types of all parameters, and return type);
- Semantics of the operation (text or pseudocode);
- Attribute details, that is, name and type
- Any operation or attribute stereotypes, constraints, and tagged values.
- They further add that if an implementation language doesn't support interfaces, abstract classes may be used instead.



## 3.2 Provided and required interfaces

- Provided interfaces refers to the set of interfaces realized by a classifier.
- Required interfaces refers to the interfaces needed by a classifier to realize its operations.
- The UML provides three different types of syntax for interfaces – the Class notation (similar to the class diagram) , the lollipop notation, and the sockets notation.
- Fig 7 shows the first two notations taken from a simple library management system where the Library class maintains a collection of Books and CDs.
- Both classes realize the Borrow interface which specifies the protocol for a borrowable item.
- One key advantage of this approach is that the library management system will treat both classes in the same way as far as the borrowing protocol is concerned; we can add other classes such as a Journals class and simply utilize the same interface in the event that specific journals (or all for that matter) can be borrowed.

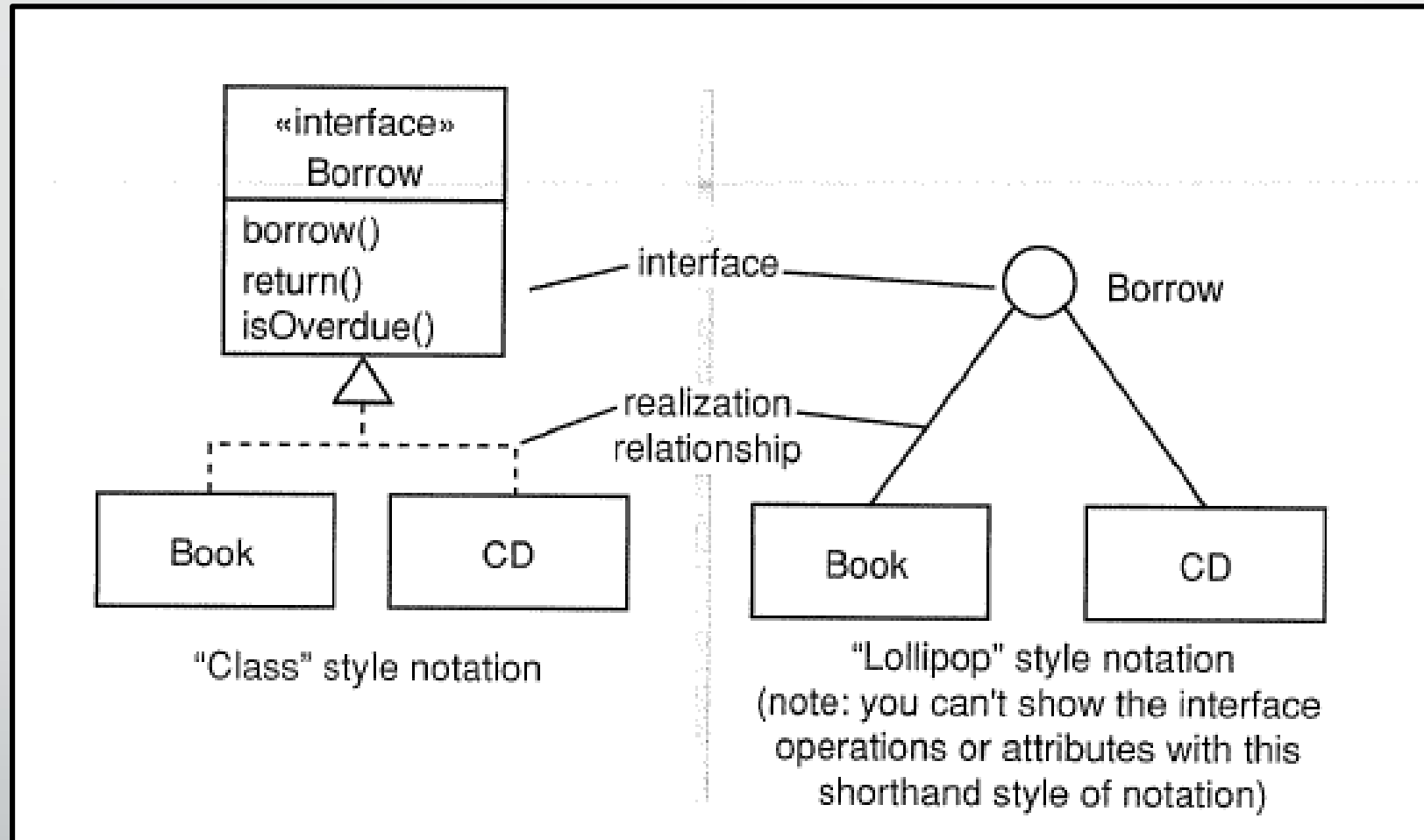


Fig 7. Interface notation using Class and lollipop (Arlow and Neustadt, 2013)

## 3.2 Provided and required interfaces

- Fig 8 shows the notation that is used by the Library class with the required interface Borrow.
- As can be seen all the three notations describe the position of the interface as far as the Borrow interface is concerned. The notation of the socket is to allow for “connectivity” (think ball and socket in a car or human anatomy) with other classes that utilize the same interface. It also shows that the Library class understands and requires the specific protocol defined by the Borrow interface.
- Further the socket shows that anything that requires the interface can be “plugged” into the socket (and Library) and Library will understand that it can be borrowed and returned.
- Fig 9 shows a complete diagram, showing the complete “ball and socket” between the Library class, Book class, and CD class.

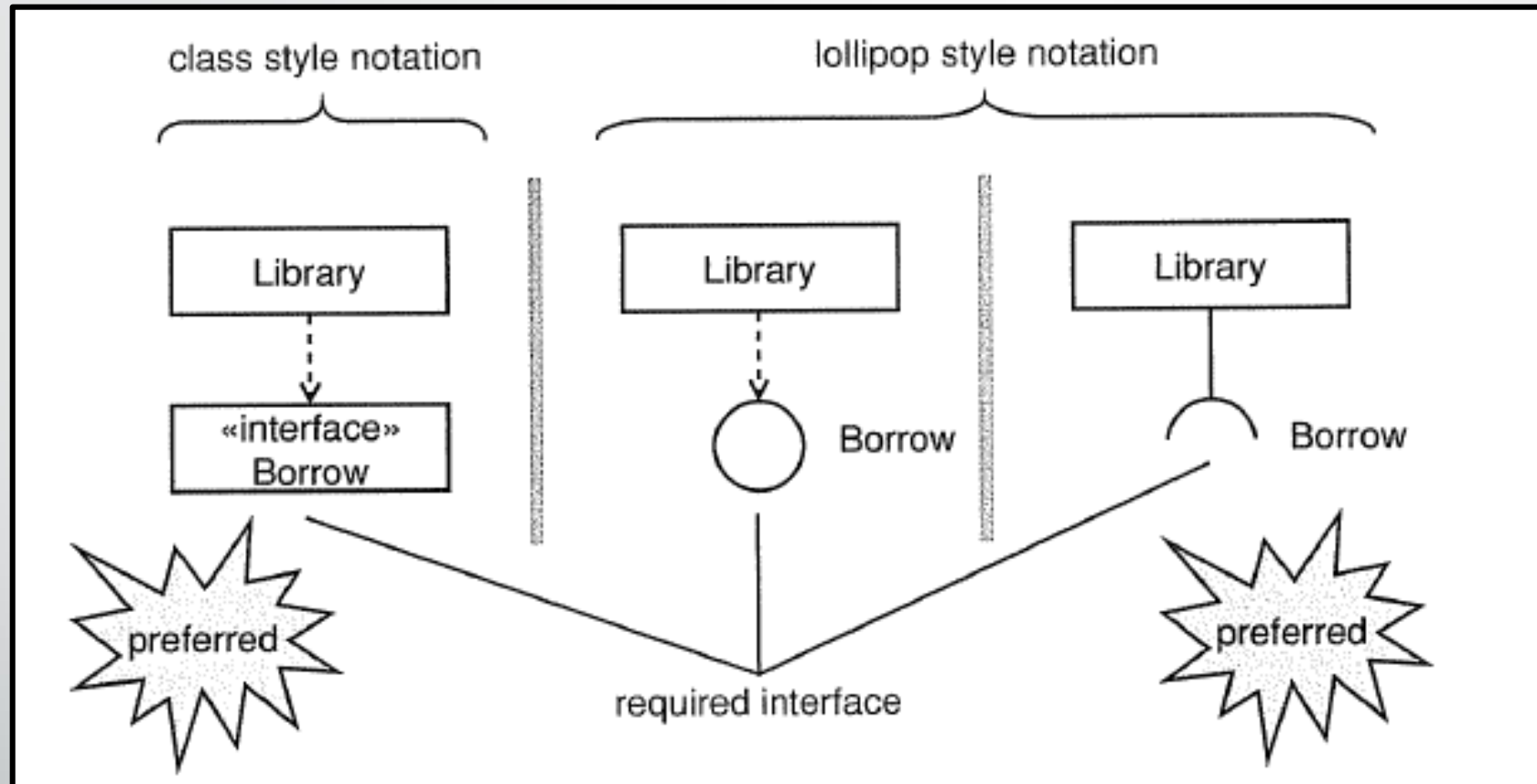


Fig 8. Notation used by Library (Arlow and Neustadt, 2013)

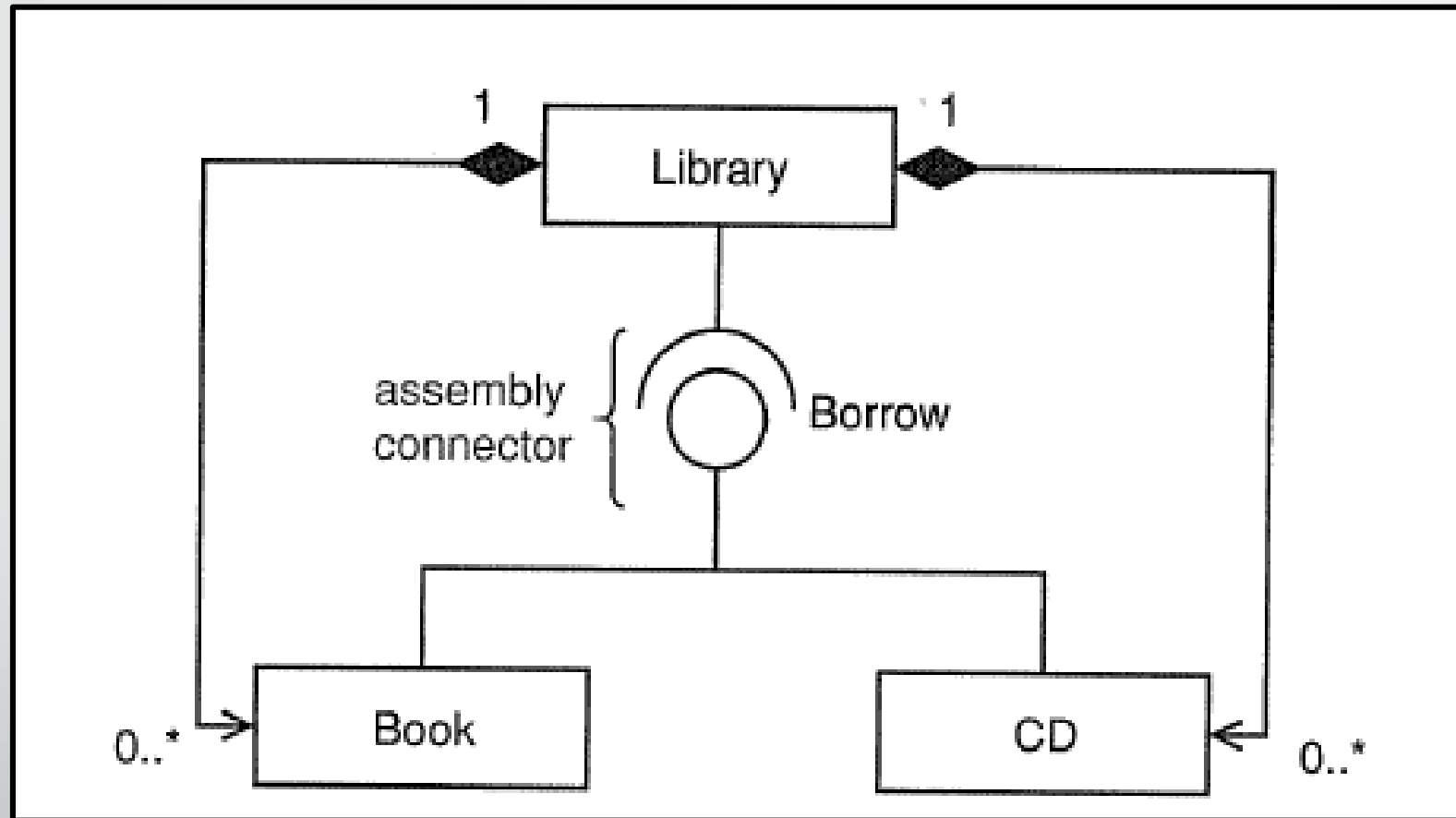


Fig 9. Assembled library system (Arlow and Neustadt, 2013)

## 3.3 Components

- A port groups a semantically cohesive set of provided and required interfaces. it indicates a specific point of interaction between a classifier and its environment (Arlow and Neustadt, 2013).
- Ports are very useful in showing the required and provided interfaces of a given classifier. For ports to be connected the required and provided interfaces must match (recall the ball and socket analogy)
- Fig 10 shows two different notations that are used to show ports in the UML.
- Fig 11 shows a Viewer class that connects to the presentation port of the Book class.
- When a port is drawn overlapping the boundary of a classifier then that means it is public (meaning provided and required interfaces are both public); conversely if the port rectangle is inside the classifier then the port has either private or protected (default) visibility.
- Ports may also have multiplicity, which is shown in square brackets; for example, (window: GUI [1]), where window is the port name, and GUI is the port type.

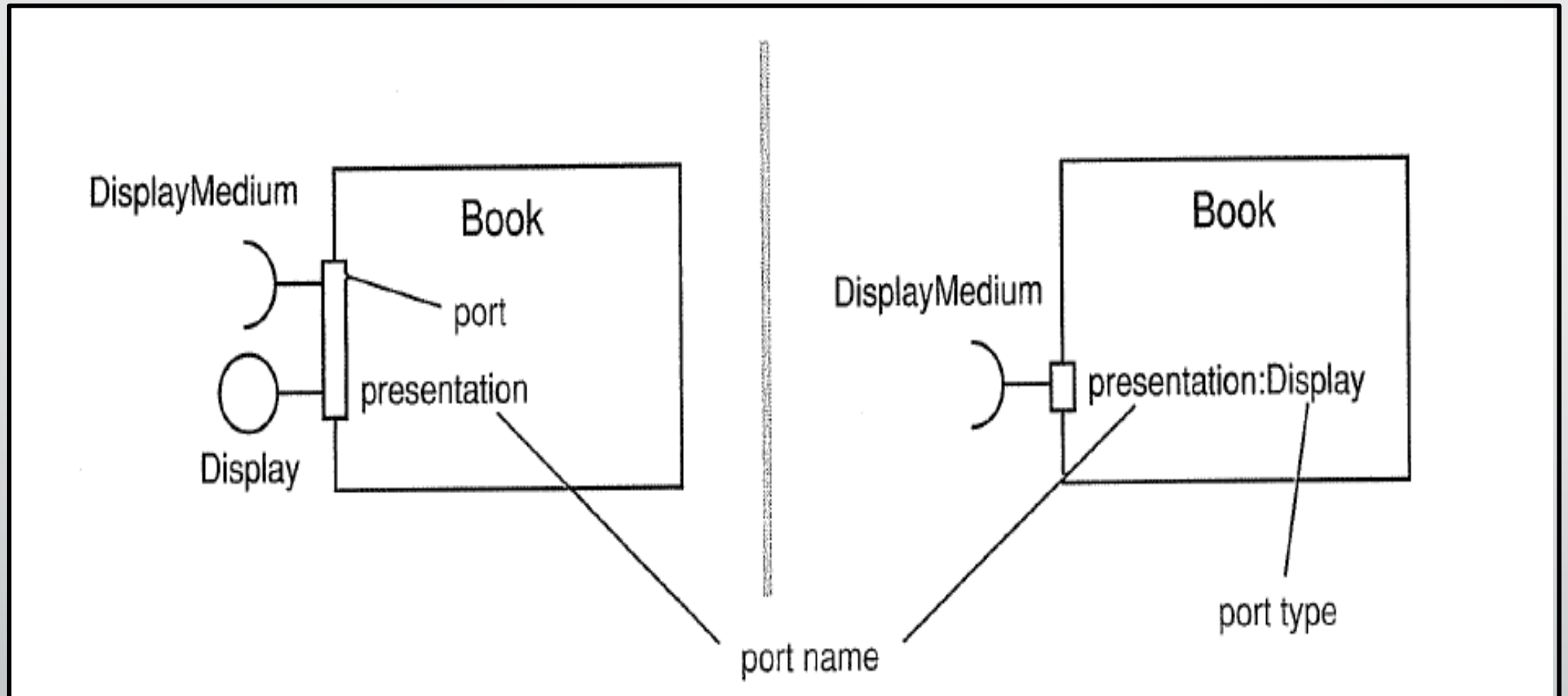


Fig 10. Notation for ports (Arlow and Neustadt, 2013)

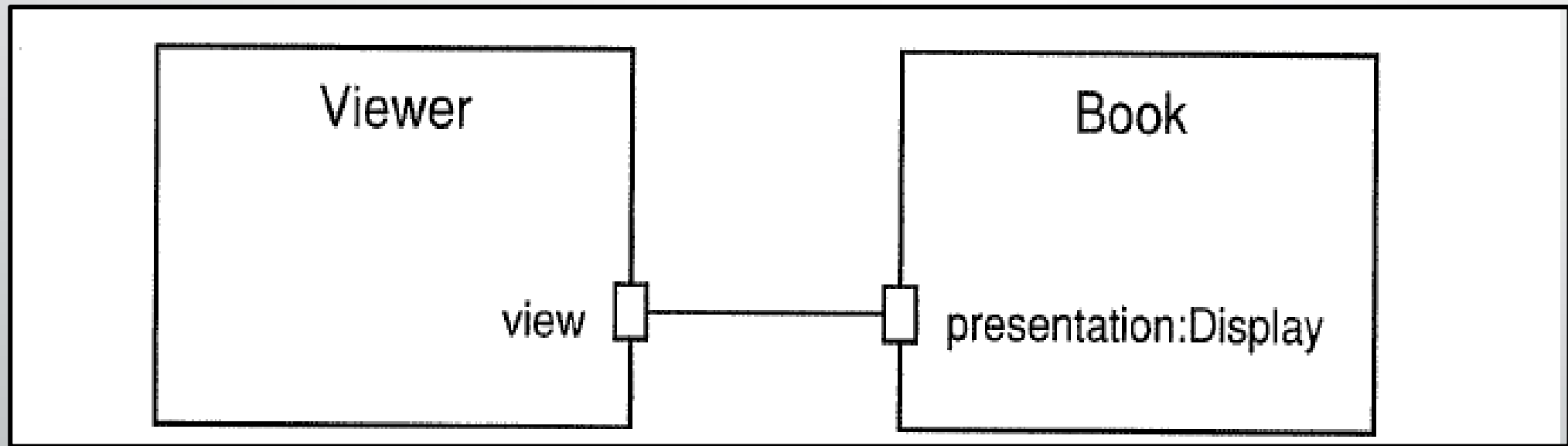


Fig 11. Viewer class. (Arlow and Neustadt, 2013)



## 3.3 Components (cont'd)

- UML 2.0 specification states that , “a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.”
- A component acts as a black box whose external behavior is completely defined by its required and provided interfaces. (Arlow and Neustadt, 2013)
- A component is a logical part of a system usually larger than a single class, which can (in principle) be replaced, reused, or sold separately. (Stevens and Pooley, 2006).
- Components are shown as rectangles with the component symbol in the top right hand corner. The notation for a component is shown in fig 12.
- Fig 13 shows an example of a component diagram; an explanation is provided thereafter.

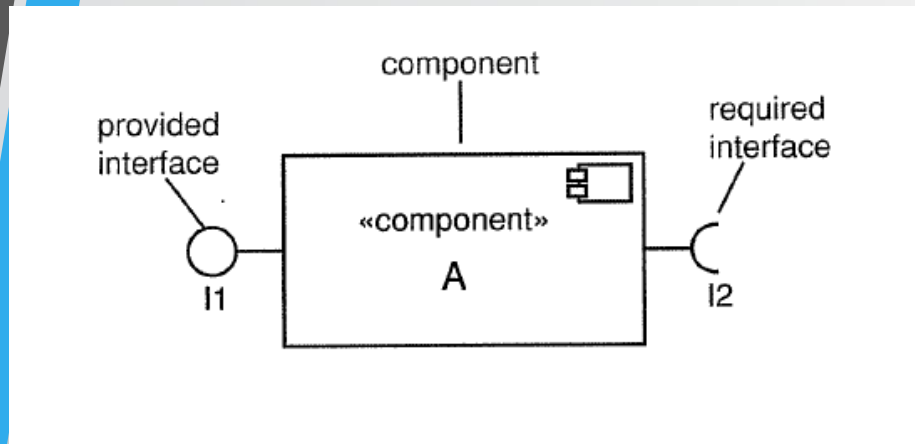


Fig 12. Component notation  
(Arlow and Neustadt, 2013)

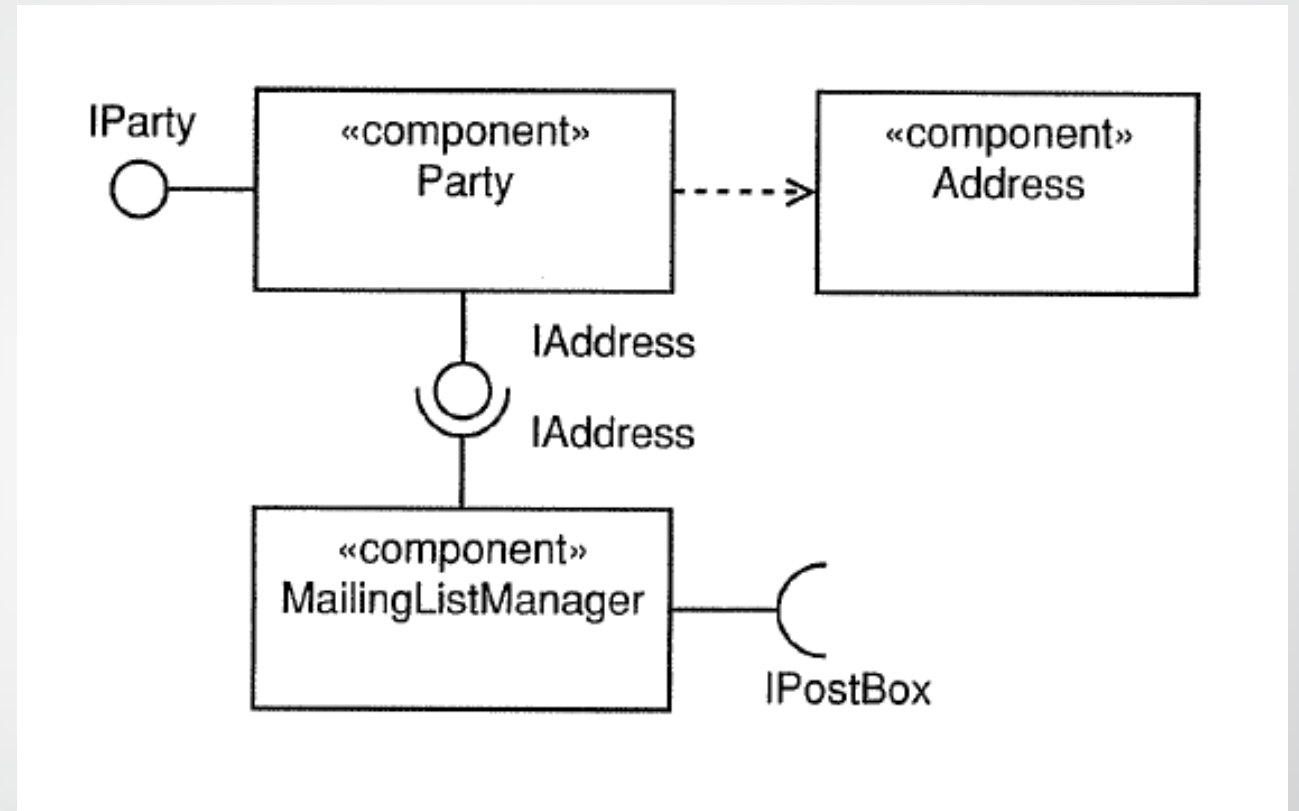


Fig 12. Component diagram (Arlow and Neustadt, 2013)

## 3.3 Components (cont'd)

- Arlow and Neustadt (2013) describe fig 12 as follows: "
  - The Party component provides two interfaces of type IParty and IAddress; these interfaces are represented as balls.
  - The MailingListManager component requires two interfaces of type IAddress and IPostBox. These are represented as sockets.
  - There is an assembly connector between the Party component and the MailingListManager component. This shows that the MailingListManager is communicating with the Party component via the provided IAddress interface.
  - In this model the Party component is acting as a façade to decouple the MailingListManager component from the details of the Address component."
- A Facade Pattern says that just "just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client". (javatpoint.com)
- Fig 13 shows a complete component diagram.

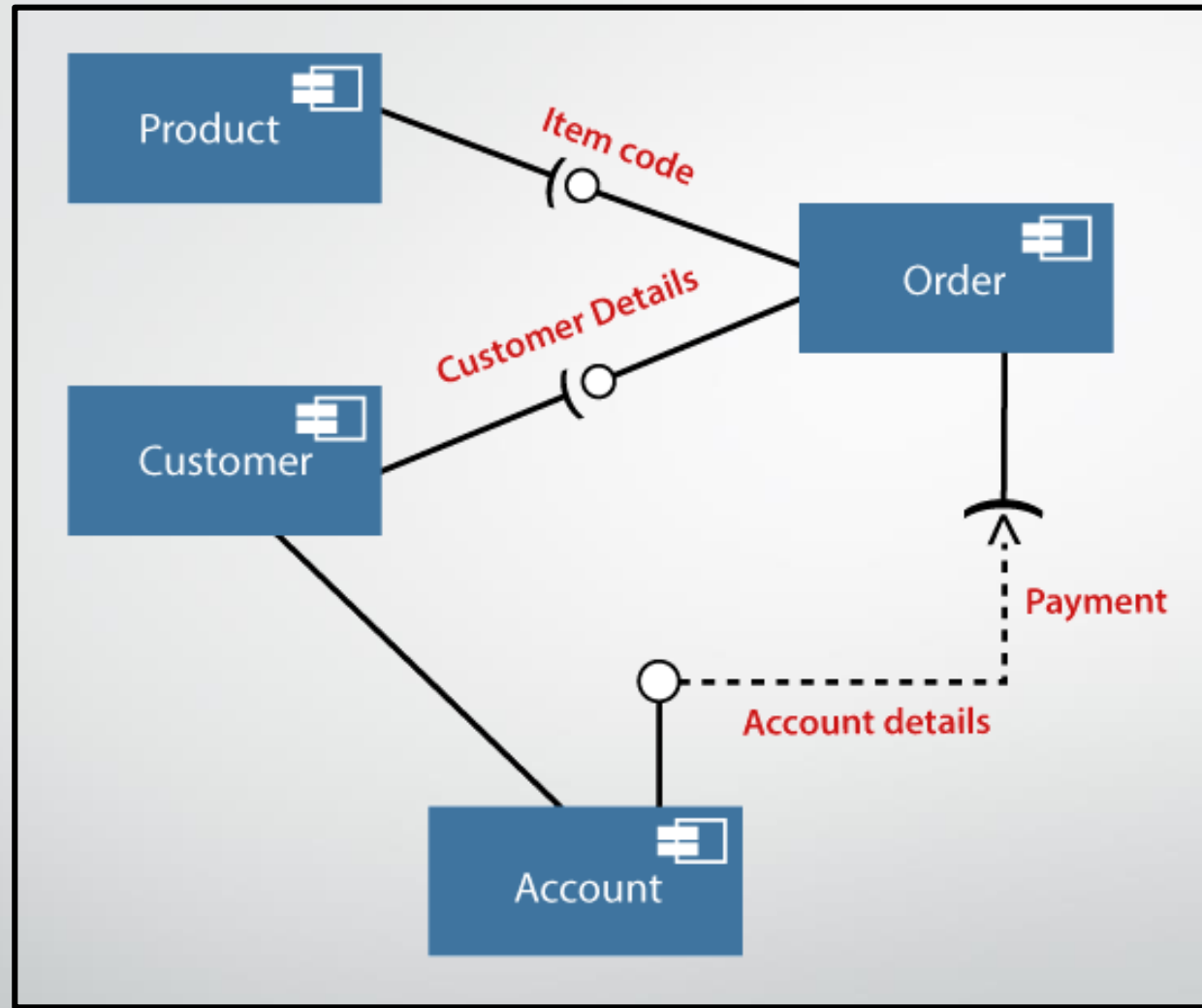


Fig 14. Component diagram (javatpoint.com)



# Part 4

## Deployment Diagram

## 4.1 The deployment diagram

- The deployment diagram shows how the software will be deployed in the system, and how that hardware is connected.
- It answers the simple question “how will the software be deployed?”
- In a distributed system the deployment diagram models the distribution of the software across physical nodes.
- There are two forms of deployment diagram as described by Arlow and Neustadt (2013):
- Descriptor form – contains nodes (that is, different types of hardware), relationship between nodes, and artifacts (physical software artifacts such as Java JAR files)
- Instance form – contains node instances (specific nodes such as John’s laptop), relationships between node instances and artifact instances (specific software such as a specific copy of moviemaker).
- The construction of the deployment diagram is a two step process (Arlow and Neustadt, 2013):
- In design workflow, focus on node or node instances and connections.
- In implementation workflow, focus on assigning artifact instances to node instances (instance form), or artifacts to nodes (descriptor form)

## 4.2 Nodes

- A deployment diagram consists of components, artifacts, interfaces and nodes.
- The first three have already been defined and explained in other sections.
- The UML 2.0 specification states “a node represents a type of computational resource upon which artifacts can be deployed for execution.” The UML describes two standard stereotypes for nodes:
  - <<device>> - a physical device such as a laptop or server
  - <<execution environment>> - type of execution environment for software such as Windows server or a Python container.
- Nodes can also be nested in other nodes; fig 15 shows an example of this, while fig 16 shows the instance form of fig 15.
- The UML also allows you to stereotype your deployment diagram, thereby assigning your own icons in the diagram. By using icons that for example, represent the actual hardware being deployed, it makes the diagram that much easier to understand.
- Fig 17 and fig 18 show two deployment diagrams implementing custom stereotypes.

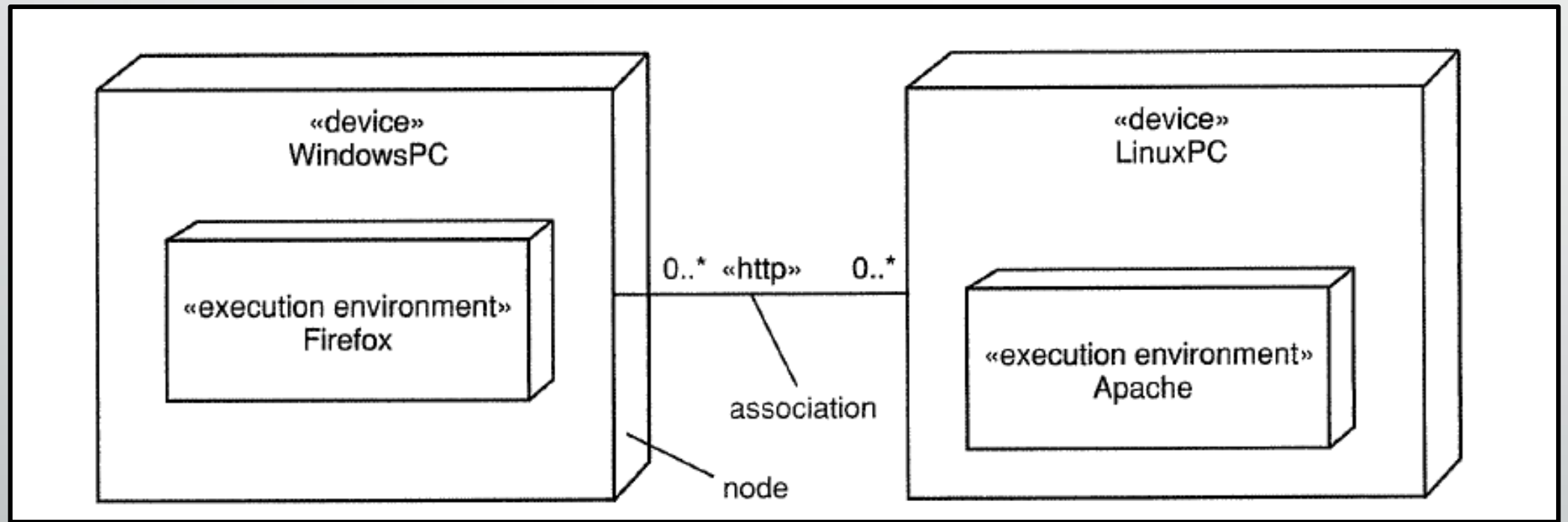


Fig 15. Nested nodes descriptor form (Arlow and Neustadt, 2013)



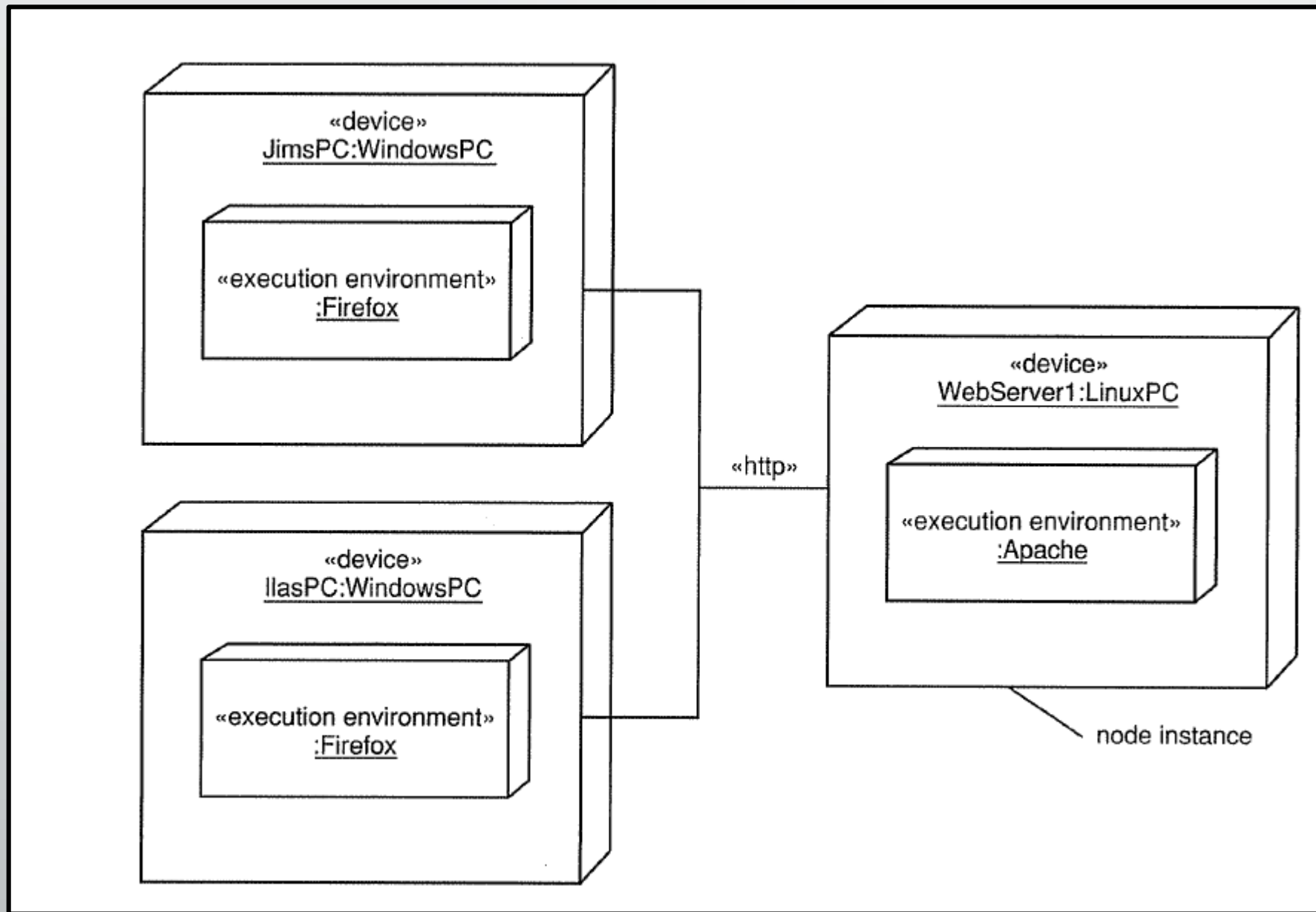


Fig 16. Nested nodes instance form (Arlow and Neustadt, 2013)

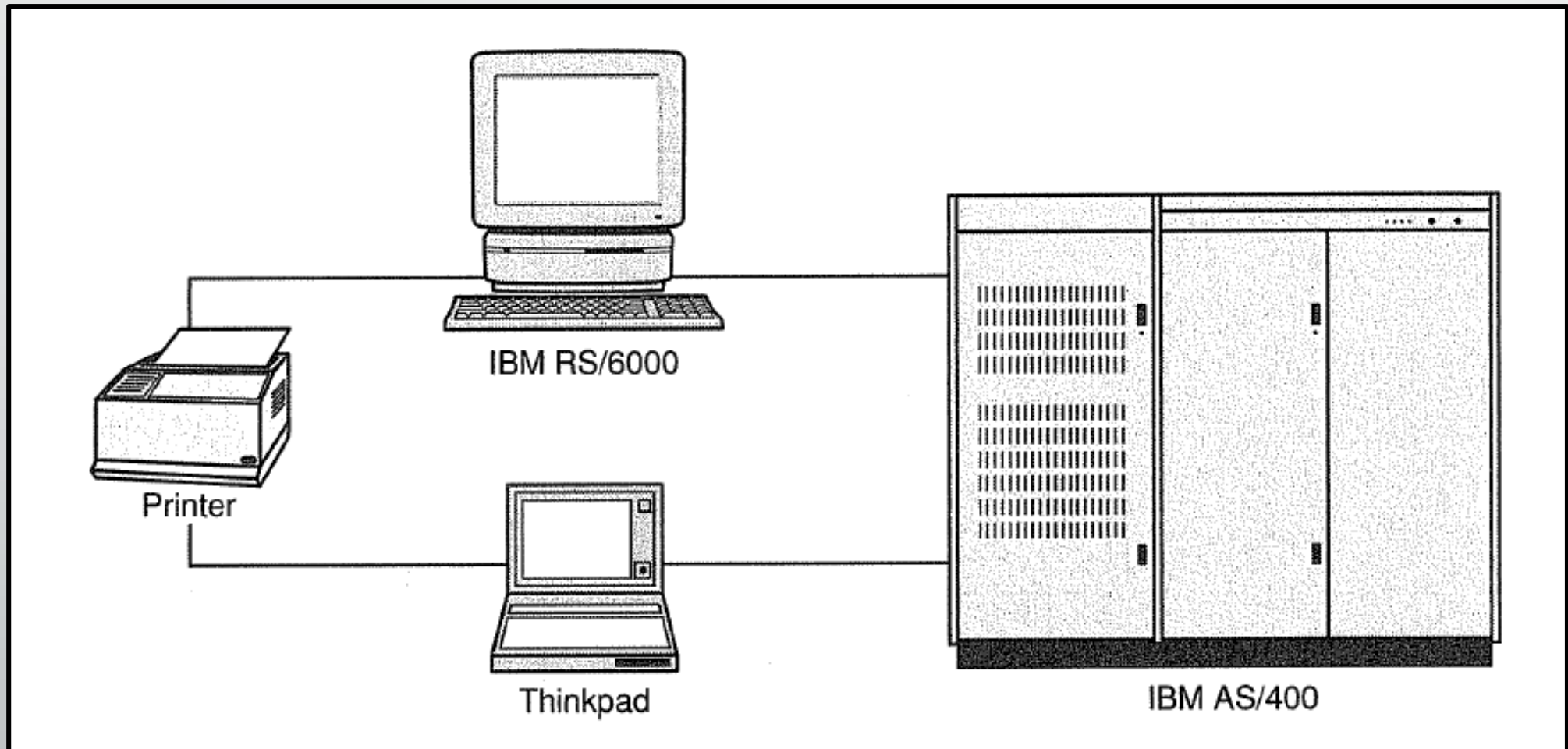


Fig 17. Stereotype deployment diagram<sub>1</sub> (Arlow and Neustadt, 2013)

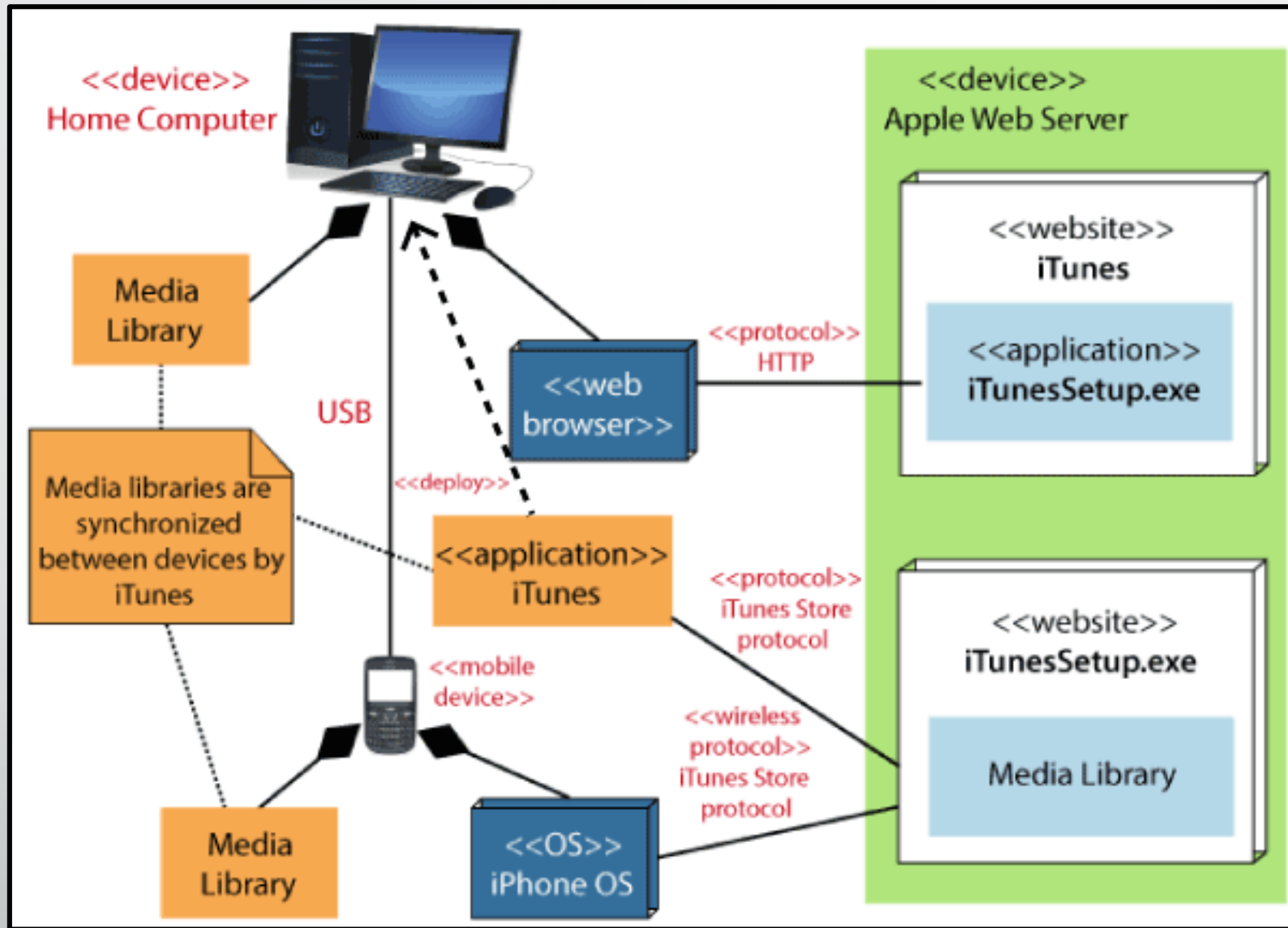


Fig 17. Stereotype deployment diagram2 (javatpoint.com)

## 4.3 Artefacts

- Artefacts represent a real world thing such as a file; they are deployed on nodes.
- Artefact instances are specific instances of particular artefacts such as a specific copy of a file deployed on a device; for example a specific copy of ooad.py
- Examples of artefacts include source files, executable files, scripts, database tables and documents (Arlow and Neustadt, 2013)
- The UML provides standard artefact stereotypes that represent different types of files; these are shown in table 5.

Table 5. UML standard artefact stereotypes (Arlow and Neustadt, 2013)

Artifact stereotype	Semantics
«file»	A physical file
«deployment spec»	A specification of deployment details (e.g., web.xml in J2EE)
«document»	A generic file that holds some information
«executable»	An executable program file
«library»	A static or dynamic library such as a dynamic link library (DLL) or Java Archive (JAR) file
«script»	A script that can be executed by an interpreter
«source»	A source file that can be compiled into an executable file

## 4.4 When and where

- Javatpoint.com describe the instances when deployment diagrams should be used:
- “The deployment diagram is mostly employed by network engineers, system administrators, etc. with the purpose of representing the deployment of software on the hardware system. It envisions the interaction of the software with the hardware to accomplish the execution. The selected hardware must be of good quality so that the software can work more efficiently at a faster rate by producing accurate results in no time.”
- They further add that deployment diagrams can be used for the following:
  - “To model the network and hardware topology of a system.
  - To model the distributed networks and systems.
  - Implement forwarding and reverse engineering processes.
  - To model the hardware details for a client/server system.
  - For modeling the embedded system.” (javatpoint.com)

# Summary

- In UML 2 packages are logical groupings and thus the elements of a package are semantically related; they are not physical.
- Packages may contain use cases, analysis cases, or use case realizations.
- Generalization can be applied in packages in the same way as the class diagram.
- The principle behind interfaces is to separate the specification of functionality from its implementation by a classifier such as a class or subsystem.
- Provided interfaces refers to the set of interfaces realized by a classifier; required interfaces refers to the interfaces needed by a classifier to realize its operations
- A component is a logical part of a system usually larger than a single class, which can (in principle) be replaced, reused, or sold separately.
- The deployment diagram shows how the software will be deployed in the system, and how that hardware is connected.
- There are two forms of deployment diagrams – descriptor form and instance form.



# References

- Arlow, J., & Neustadt, I. (2013). *Uml 2 and the unified process: Practical object-oriented analysis and Design* (Second). Addison-Wesley.
- Dennis, A., Wixom, B. H., Tegarden, D. P., & Seeman, E. (2015). *System analysis & design: An object-oriented approach with Uml*. Wiley.
- Stevens, P., & Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components*. Pearson Educaion Limited.
- *UML Component Diagram - Javatpoint*. (n.d.). Www.javatpoint.com. Retrieved November 5, 2022, from <https://www.javatpoint.com/uml-component-diagram>
- *UML Deployment Diagram - Javatpoint*. (n.d.). Www.javatpoint.com. Retrieved November 5, 2022, from <https://www.javatpoint.com/uml-deployment-diagram>