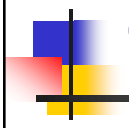


# Lecture 6
# Components and Implementation

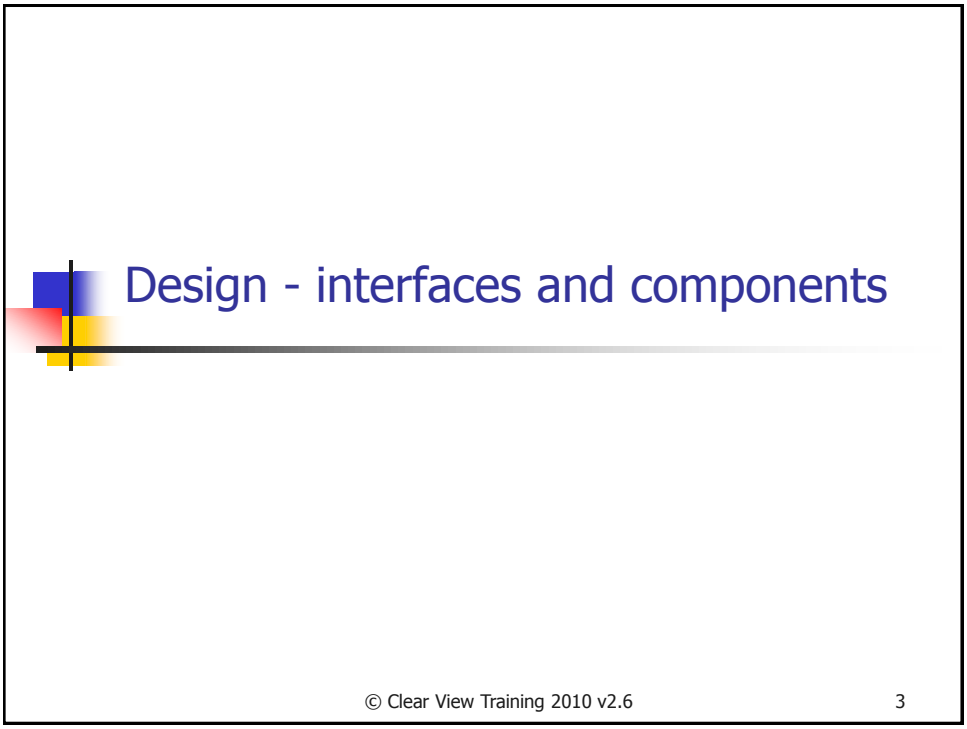
Dr Peter T. Popov
Centre for Software Reliability

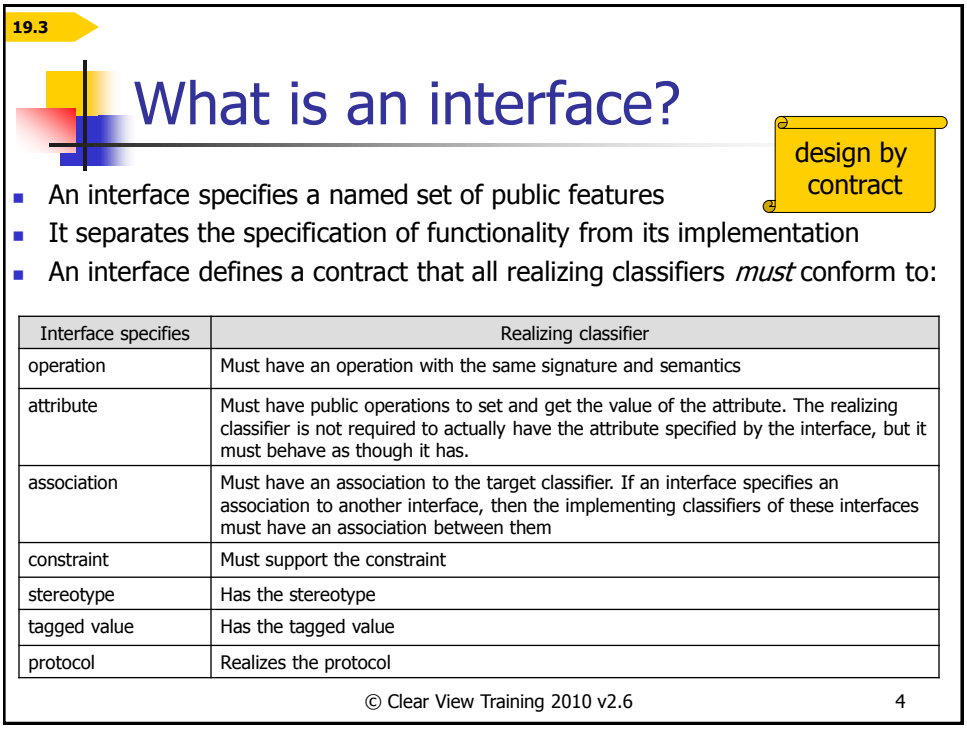
8nd November 2018

## Outline of the lecture

- We will cover two diagrams
  - Components diagrams
    - used to capture the high level software design (i.e. software architecture)
  - Deployment Diagrams
    - used to model the deployment of software on real hardware
- Examples of both, a component diagram and a Deployment diagram, will be developed in class

2

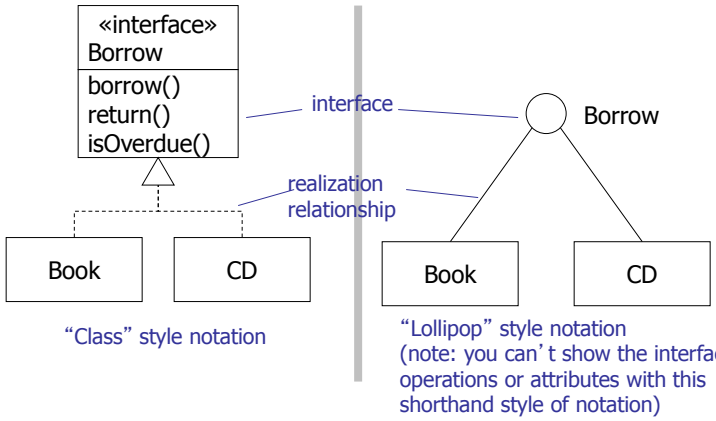


19.4

# Provided interface syntax

- A provided interface indicates that a classifier implements the services defined in an interface

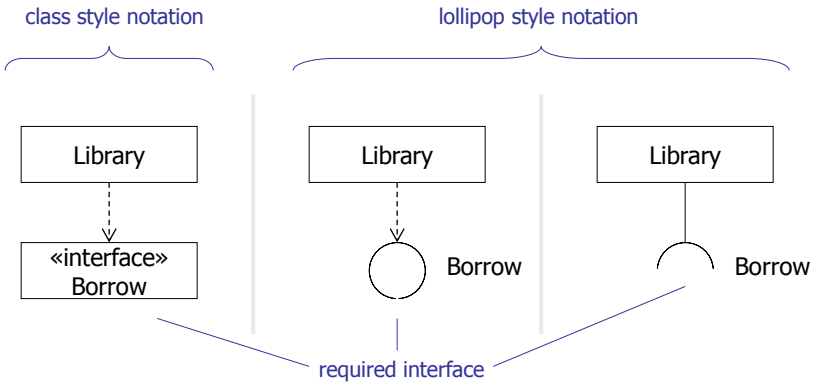



 5

19.4

# Required interface syntax

- A required interface indicates that a classifier uses the services defined by the interface

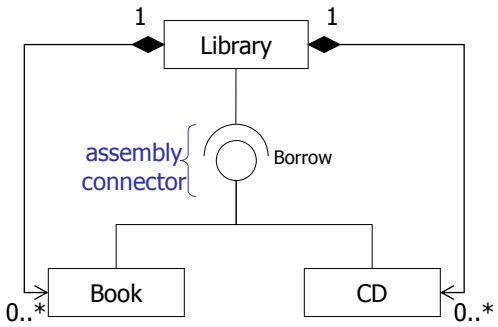



 6

19.4

# Assembly connectors

- You can connect provided and required interfaces using an ***assembly connector***

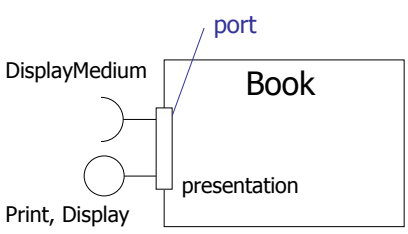


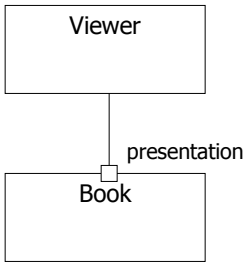

 7

19.6

# Ports: organizing interfaces

- A port specifies an interaction point between a classifier and its environment
- A port is typed by its *provided* and *required* interfaces:
  - It is a semantically cohesive set of provided and required interfaces
  - It may have a name
- If a port has a single *required* interface, this defines the type of the port
  - You can name the port portName:RequiredInterfaceName




 8

19.7

# Interfaces and CBD

- Interfaces are the key to *component based development* (CBD)
- CBD is about constructing software from *replaceable*, plug-in parts:
  - Plug – the provided interface
  - Socket – the required interface
- Consider:
  - Electrical outlets
  - Computer ports – USB, serial, parallel
- Interfaces define a contract so classifiers that realise the interface agree to abide by the contract and can be used interchangeably

 9

19.8

# What is a component?

- The 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 black-box whose external behaviour is completely defined by its provided and required interfaces
  - May be substituted for by other components provided they all support the same protocol

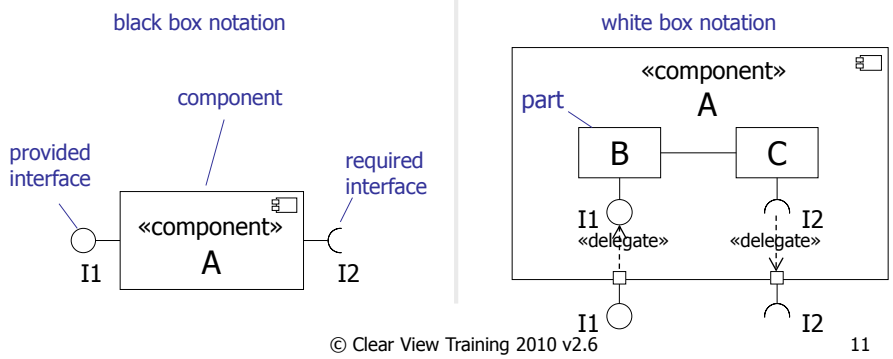
- Components can be:
  - Physical - can be directly instantiated at run-time e.g. an Enterprise JavaBean (EJB)
  - Logical - a purely logical construct e.g. a subsystem
    - only instantiated indirectly by virtue of its parts being instantiated

 10

19.8

# Component syntax

- Components may have provided and required interfaces, ports, internal structure
  - Provided and required interfaces usually delegate to internal parts
  - You can show the parts nested *inside* the component icon or *externally*, connected to it by dependency relationships




 11

19.9

# Standard component stereotypes

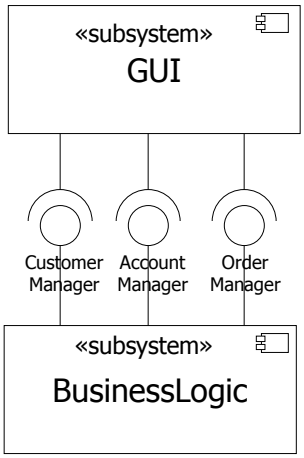
| Stereotype | Semantics |
|---|---|
| «buildComponent» | A component that defines a set of things for organizational or system level development purposes. |
| «entity» | A persistent information component representing a business concept. |
| «implementation» | A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a dependency. |
| «specification» | A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «specification» only has ***provided and required interfaces*** - no realizing classifiers. |
| «process» | A transaction based component. |
| «service» | A stateless, functional component (computes a value). |
| «subsystem» | A unit of hierarchical decomposition for large systems. |

 12

19.10

# Subsystems

- A subsystem is a component that acts as a unit of decomposition for a larger system
- It is a logical construct used to decompose a larger system into manageable chunks
- Subsystems *can't* be instantiated at run-time, but their contents can
  - i.e. the objects that make up the sub-system
- Interfaces connect subsystems together to create a ***system architecture***




 13

19.11

# Finding interfaces and ports

***Interfaces***

- Challenge each association:
  - Does the association have to be to another class, or can it be a dependency to an interface?
- Challenge each message send:
  - Does the message send have to be to another class, or can it be to an interface?
- Many design patterns introduce interfaces
- Look for repeating groups of operations
- Look for groups of operations that might be useful elsewhere
- Look for possibilities for ***future expansion***

***Ports***

- Look for cohesive sets of provided and required interfaces and organize these into ***named ports***
- Look at the dependencies between subsystems - mediate these by an assembly connector where possible

 14

19.12

# Designing with interfaces

- Design interfaces based on ***common sets of operations***
- Design interfaces based on common roles
  - These roles may be between two classes or even within one class which interacts with itself
  - These roles may also be between two subsystems
- Design interfaces for new ***plug-in features***
- Design interfaces for plug-in algorithms
- The Façade Pattern - interfaces can be used to create "seams/layers" in a system:
  - Identify cohesive parts of the system
  - Package these into a «subsystem»
  - Define an interface to that subsystem
- Interfaces allow information *hiding* and *separation of concerns*

 15

19.12.2

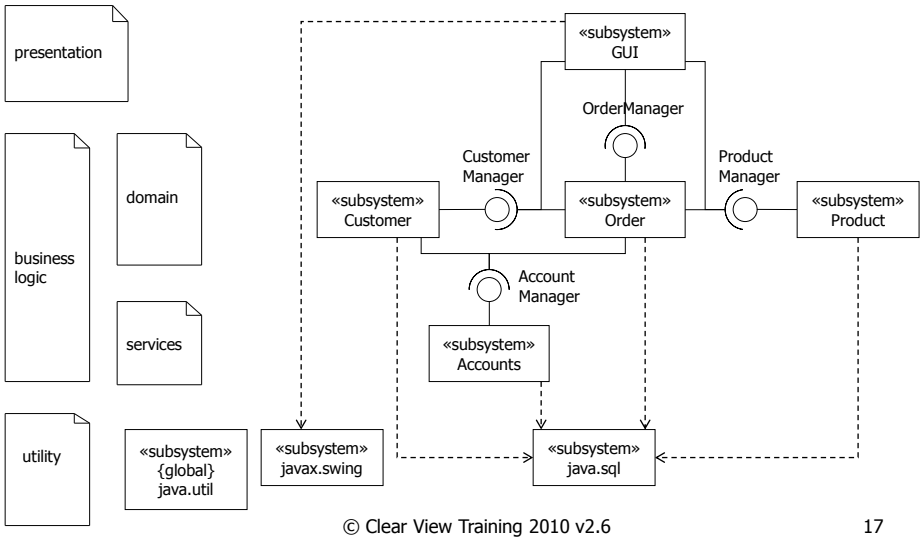
# Physical (software) architecture
## (high level design)

- Subsystems and interfaces comprise the ***physical architecture*** of our model
- We must now organise this collection of interfaces and subsystems to create a coherent architectural picture:
- We can apply the "layering" ***architectural pattern***
  - Subsystems are arranged into layers
  - Each layer contains design subsystems which are semantically cohesive e.g. Presentation layer, Business logic layer, Utility layer
  - Dependencies between layers are very carefully managed
  - Dependencies go one way
  - Dependencies are mediated by interfaces

 16

19.12.2

# Example layered architecture




 17

19.13

# Using interfaces

- Advantages:
  - When we design with classes, we are designing to *specific implementations*
  - When we design with interfaces, we are instead designing to contracts which may be realised by many different implementations (classes)
  - Designing to contracts frees our model from implementation dependencies and thereby *increases its flexibility and extensibility*
- Disadvantages:
  - Interfaces can add flexibility to systems BUT *flexibility may lead to complexity* (there is always an optimal trade off)
  - Too many interfaces can make a system too flexible!
  - Too many interfaces can make a system hard to understand

Keep it simple!

 18

19.14

# Summary

- Interfaces specify a named set of public features:
  - They define a contract that classes and subsystems may realise
  - Programming to interfaces rather than to classes *reduces dependencies* between the classes and subsystems in our model
  - Programming to interfaces *increases flexibility and extensibility*
- Design subsystems and interfaces allow us to:
  - Componentize our system
  - Define software architecture

This part of the lecture follows closely Chapter 19 of Arlow's book.

 19

# An example (advanced topic):
## Redesigning a system with Hibernate components

- Old system: Consider the example shown on slide 17
  - JDBC drivers are used to implement java.sql API.
  - The old system (i.e. the application code) is developed to work with a particular DB server, e.g. Oracle and ***uses the proprietary syntax of SQL statements*** (e.g. of outer joins)
- Consider redesigning the system on slide 17, so that it can work with a range of DBs *without changing the applications code*
  - The redesign will require a layer of abstraction between the application code and the persisitence layer, e.g. using ***Hibernate***, the object-relational mapping (ORM) library
  - Consider replacing the subsystems with dependency on java.sql with subsystems that depend on the Hibernate library instead.
- Draw a component diagram in which Hibernate is introduced as a component.
  - ***Hint 1:*** The components that use directly java.sql (i.e. have dependency relationship with it) will have to be replaced with components that use the Hibernate library instead.
  - ***Hint 2:*** These new components (let us call them H_Customer, H_Product and H_Accounts) must ***provide*** the same interfaces as the components that they replace (i.e. Customer, Product and Accounts).

20

## Persistence with Hibernate (2)

- ***Hibernate*** is an object-relational mapping (ORM) library
  - Introduces a layer of abstraction between the objects and the persistent layer (i.e. how the data is stored in a DB).
  - Introduces its own Hibernate Query Language (HQL) against Hibernate's data objects
  - The applications must be written to use HQL and the Hibernate library (instead of using a JDBC API)
  - Hibernate itself uses a JDBC driver (e.g. from Oracle) to work with a particular DB server.
  - Hibernate provides a translation from HQL to the particular SQL syntax implemented by the chosen SQL server
    - Most of the statements (e.g. the commonly used SQL statements) require a minimal transformation from HQL to SQL.
- Java Persistence API is an extension of the Hibernate concept
  - Consider redesigning the architecture on slide 17 with JPA.

21

## Implementation - introduction



22

23.2

# Implementation - purpose

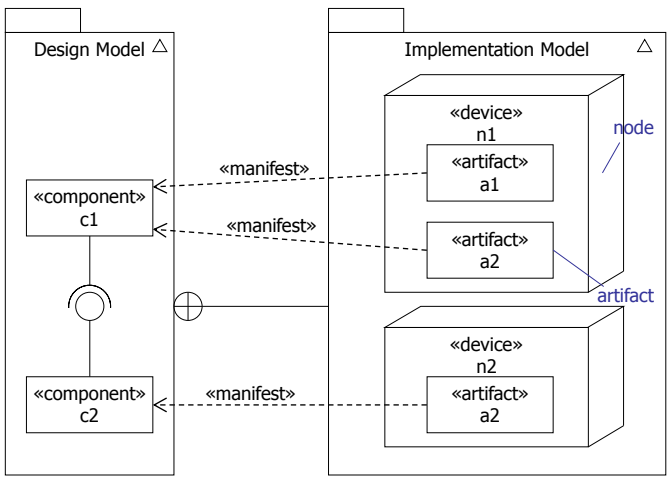
- To implement the design classes and components
  - To create an implementation model
- To convert the Design Model into an executable program

 23

23.3

# Implementation artifacts - metamodel

- The implementation model is part of the Design Model. It comprises:
  - Component diagrams showing components and the artifacts that realize them
  - Deployment diagrams showing artifacts deployed on nodes
- Components are manifest by artifacts
- Artifacts are *deployed* on nodes




 24

23.6

# Summary

- Software implementation is the primary focus for software construction.
- Purpose – to create an executable system
- artifacts:
  - component diagrams
    - components and artifacts
  - deployment diagrams
    - nodes and artifacts

 25

# Implementation - deployment

 26

24.3

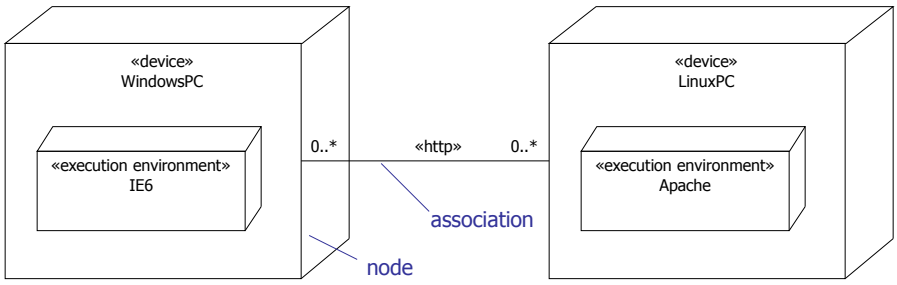
# Deployment model

- The deployment model is an object model that describes how ***functionality is distributed*** across ***physical nodes***
  - It models the mapping between the ***software architecture*** and the ***physical system architecture***
- It models the system's physical architecture as artifacts deployed on nodes
  - Each node is a type of computational resource
  - Nodes have relationships that represent methods of communication between them e.g. http, iiop, netbios
  - Artifacts represent ***physical software*** (files) e.g. a JAR file or .exe file
- Design - we may create a *first-cut deployment diagram*:
  - Focus on the big picture - nodes or node instances and their connections
  - Leave detailed artifact deployment to actual implementation workflow
- Implementation - finish the deployment diagram:
  - Focus on artifact deployment on nodes

27

24.4

# Nodes – descriptor form




- A node represents a type of computational resource
  - e.g. a WindowsPC
- Standard stereotypes are «device» and «execution environment»

28

# Devices

- A device refers to a piece of computing *hardware*:
  - PC,
  - a laptop,
  - a mobile device,
- Devices contain execution environment(s).

29

# Execution Environment

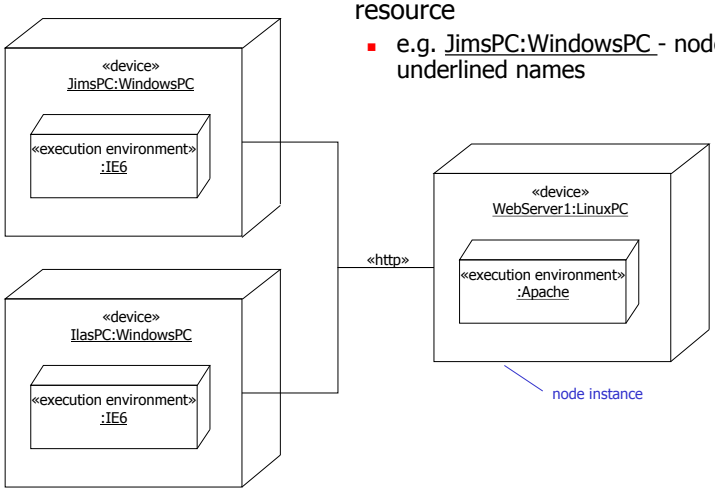
- An execution environment is a piece of "system" software, in which other ("application") software can be run:
  - Operating system (one can run applications, services),
  - Web-browser – one can run in it other application code, e.g. written using scripting languages such as JavaScript.
  - RDBMS – the users writes:
    - SQL statements, which are executed by RDBMS
    - Some RDBMS also offer other features, triggers (e.g. to react to data changes – on update, on delete, etc.) or stored procedures, in which proprietary languages are used (e.g. Oracle's PL/SQL, Microsoft's Transact SQL, etc.)
  - Virtual environment (e.g. hypervisors or containers such as Docker) offers an API for starting/stopping virtual machines (VM)
- Execution environments may be *nested*, e.g.
  - OS contains an RDBMS, web browser, etc.
  - Hypervisor can contain multiple operating systems (OSs), etc.

30

24.4

# Nodes – instance form

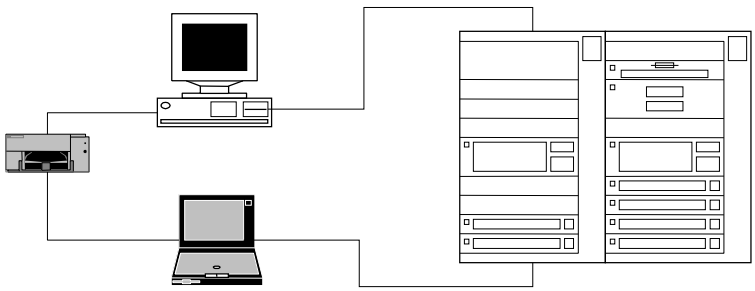
- A node instance represents an actual physical resource
  - e.g. JimsPC:WindowsPC - node instances have underlined names




 31

24.4

# Stereotyping nodes



- It's very useful to use lots of stereotyping on the deployment diagram to make it as clear and readable as possible

 32

24.5

# Artifacts

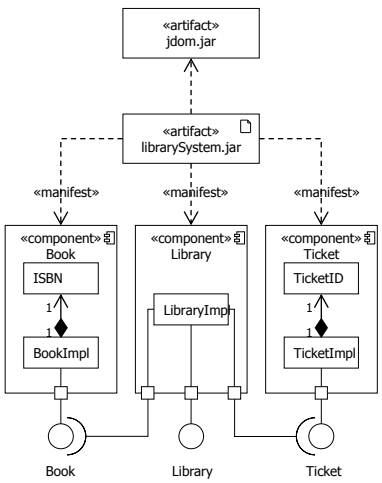
- An artifact represents a type of concrete, real-world thing such as a ***file***
  - Can be deployed on nodes (typically on an <<execution environment>>)
- Artifact instances represent particular *copies* of artifacts
  - Can be deployed on node instances
- An artifact can manifest one or more components
  - The artifact represents the thing that is the physical manifestation of the component (e.g. a JAR file)

 33

24.5

# Artifacts and components

- Artifacts provide the physical manifestation for one or more components
- Artifacts may have the artifact icon in their upper right hand corner
- Artifacts can contain other artifacts
- Artifacts can depend on other artifacts

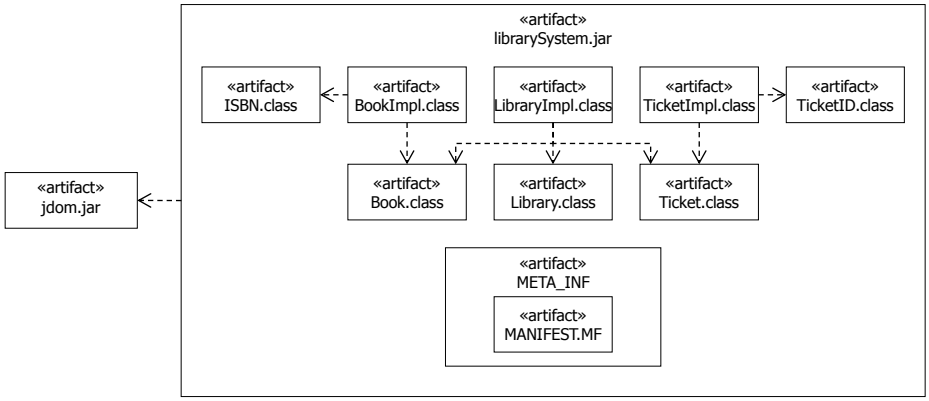



 34

24.5

# Artifact relationships

- An artifact may depend on other artifacts when a component in the client artifact depends on a component in the supplier artifact in some way




 35

24.5

# Artifact standard stereotypes

- UML 2 provides a small number of standard stereotypes for artifacts

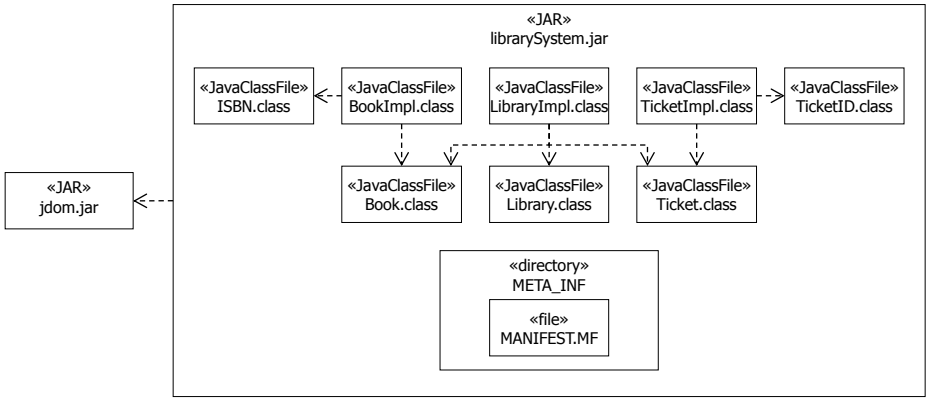
| artifact stereotype | semantics |
|---|---|
| «file» | A physical file |
| «deployment spec» | A specification of deployment details (e.g. web.xml in J2EE) |
| «document» | A generic file that holds some information |
| «executable» | An executable program file |
| «library» | A static or dynamic library 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 |

 36

24.5

# Stereotyping artifacts

- Applying a UML profile can clarify component diagrams
  - e.g. applying the example Java profile from the UML 2 specification.

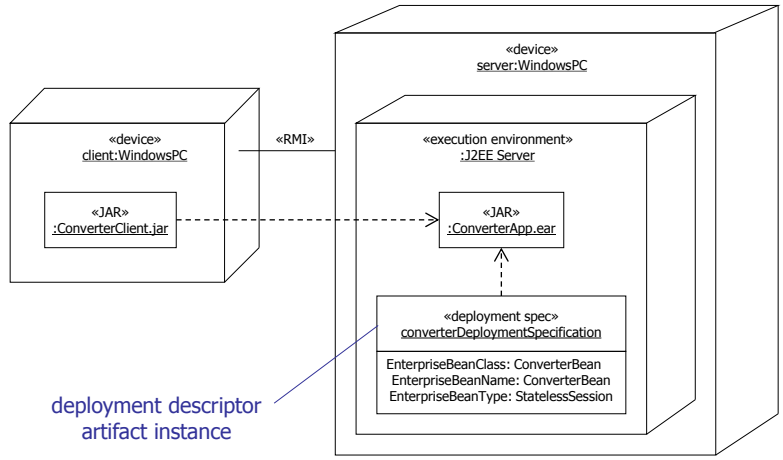



 37

24.6

# Deployment

- Artifacts are deployed on nodes, artifact instances are deployed on node instances




 38

24.7

# Summary

- The descriptor form deployment diagram
  - Allows you to show how functionality represented by artefacts is distributed across nodes
  - Nodes represent types of physical hardware or execution environments
- The instance form deployment diagram
  - Allows you to show how functionality represented by artefact instances is distributed across node instances
  - Node instances represent actual physical hardware or execution environments

The material on Deployment Diagrams follows closely Chapter 24 of Arlow's book.

 39

# An Example: BAGS system

**Baggage Automatic Guardian System (BAGS)**

BAGS consists of a Central Administration of Left Luggage (CALL) subsystem, deployed on a server, CALL_Server, running Linux Red Hat and a number of Locally Operated Control Kit (LOCK) subsystems, each operating a rack of lockers available to the public to store their luggage.

LOCK subsystems is deployed on a PC compliant industrial controllers, LOCK_1, LOCK_2, etc., each running Windows 7 as an operating system.
Software deployed on CALL_Sever is a Linux application, CALL_app . LOCK_X machines run LOCK_app.exe software.
The communications between CALL_app and LOCK_app.exe uses a BAGS proprietary protocol over TCP/IP connection.
CALL software uses a MySQL 5.1 server deployed on a CALL_DB machine running Linux Red Hat as an operating system.
The database that contains the tables used by the CALL_app is called CALL_db and is accessed via the MySQL server (i.e. MySQL is the execution environments for CALL_db).

Draw a deployment diagram of the BAGS system showing:
- the nodes,
- the execution environments
- the artifacts (CALL_app, CALL_db and LOCK_app.exe) deployed in their respective execution environments and
- the associations/dependencies between artifacts/nodes.

40