Architecture Design and Implementation

Lecture 8

Topics covered

- Architectural design
- Implementation

- UML Packages (Analysis)
- UML Component Diagram (Design)
- UML Deployment Diagram (Implementation)

Architectural Design

Lecture 8/Part 1

Topics covered

- Architectural design decisions
- Architectural views
- Architectural patterns
- Application architectures

Architectural abstraction

- **Architecture in the small (analysis)** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large (design)** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

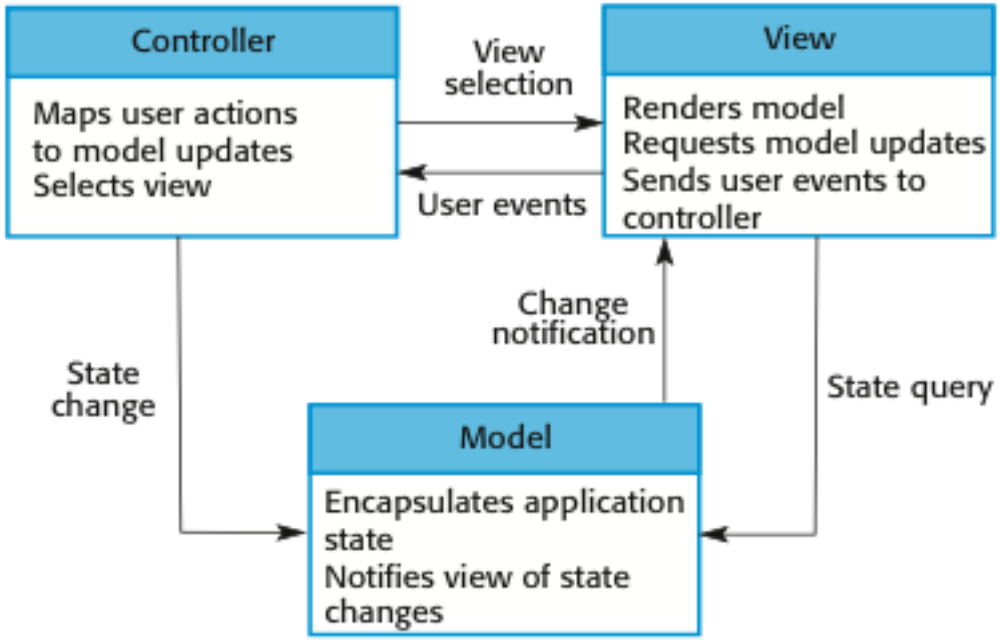
Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architectural patterns

- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.
- Patterns may be represented using tabular and graphical descriptions.

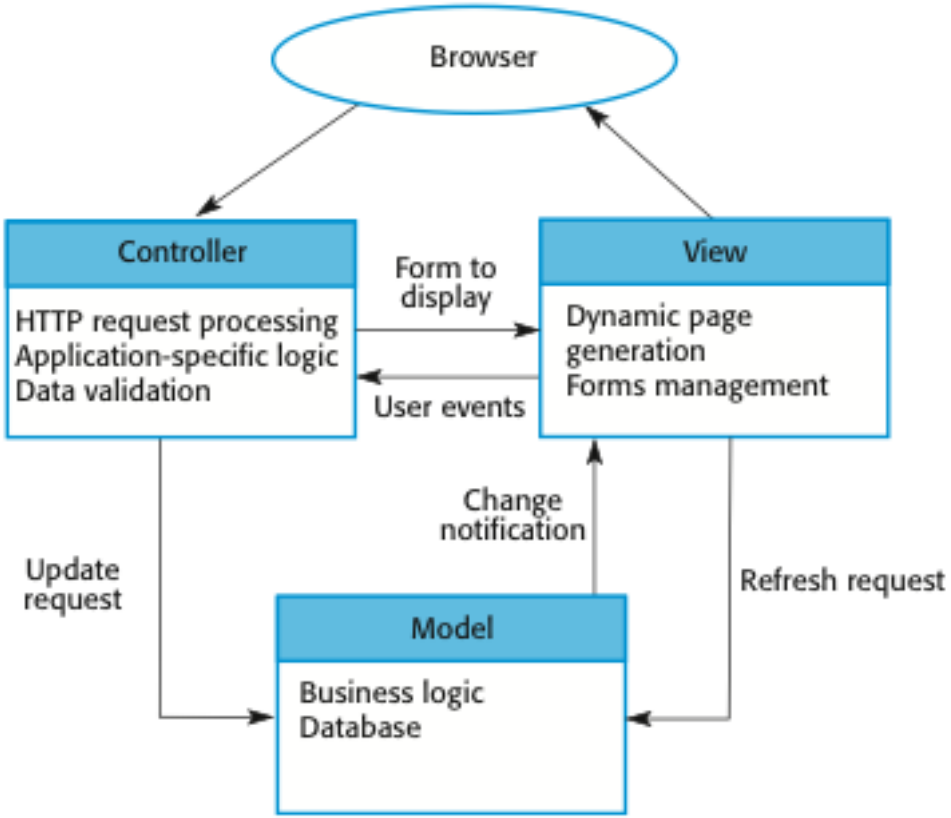
The Model-View-Controller (MVC) pattern



The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	Figure on the next slide shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

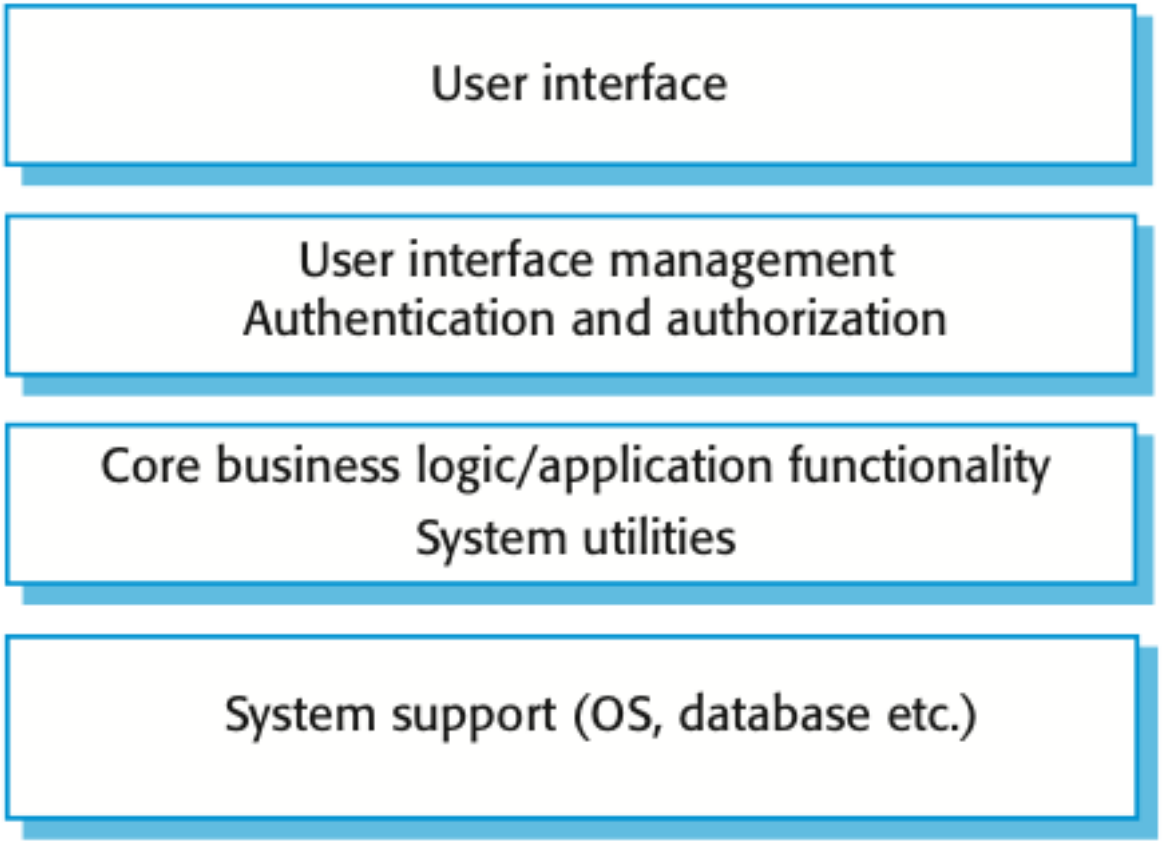
Web application architecture using MVC



The Layered architecture pattern

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

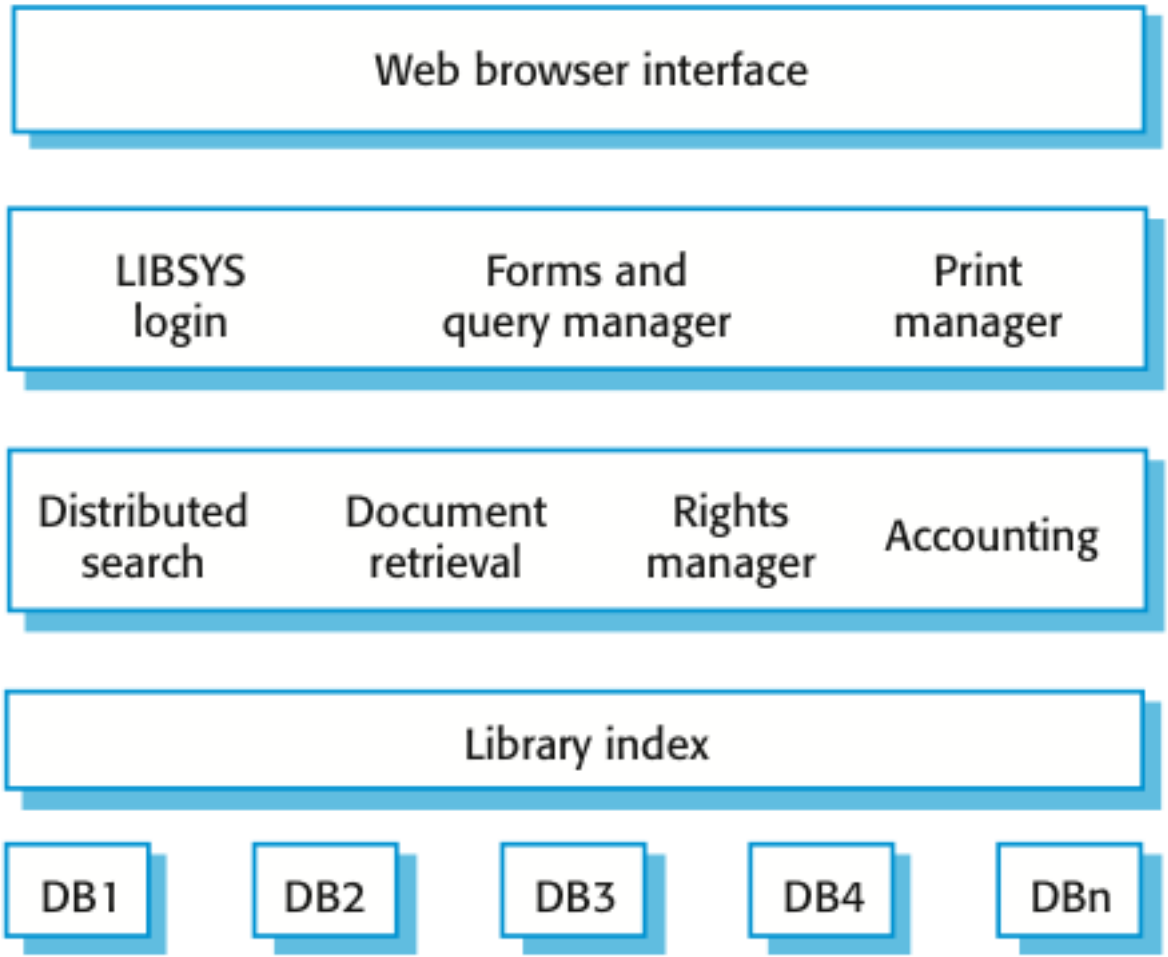
A generic layered architecture



The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

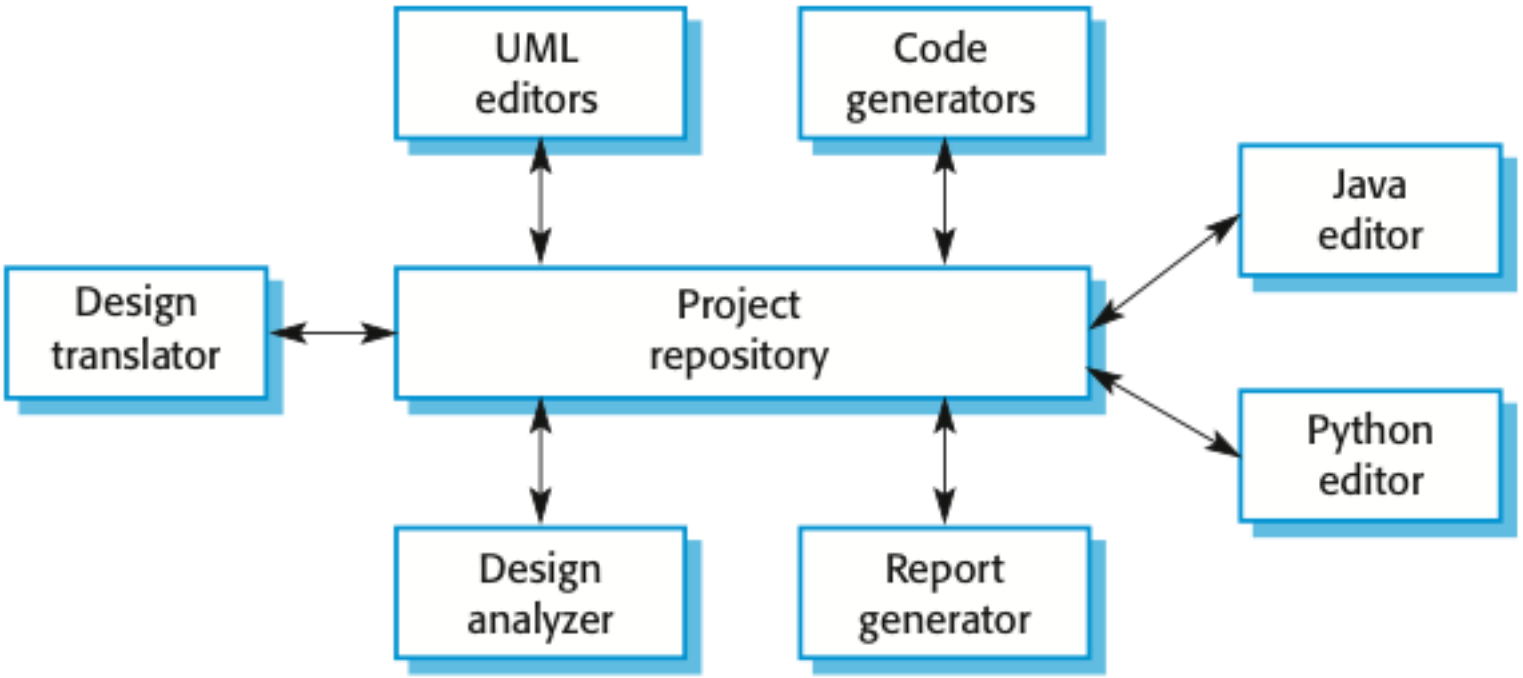
The architecture of the LIBSYS system



The Repository architecture pattern

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

A repository architecture for an IDE



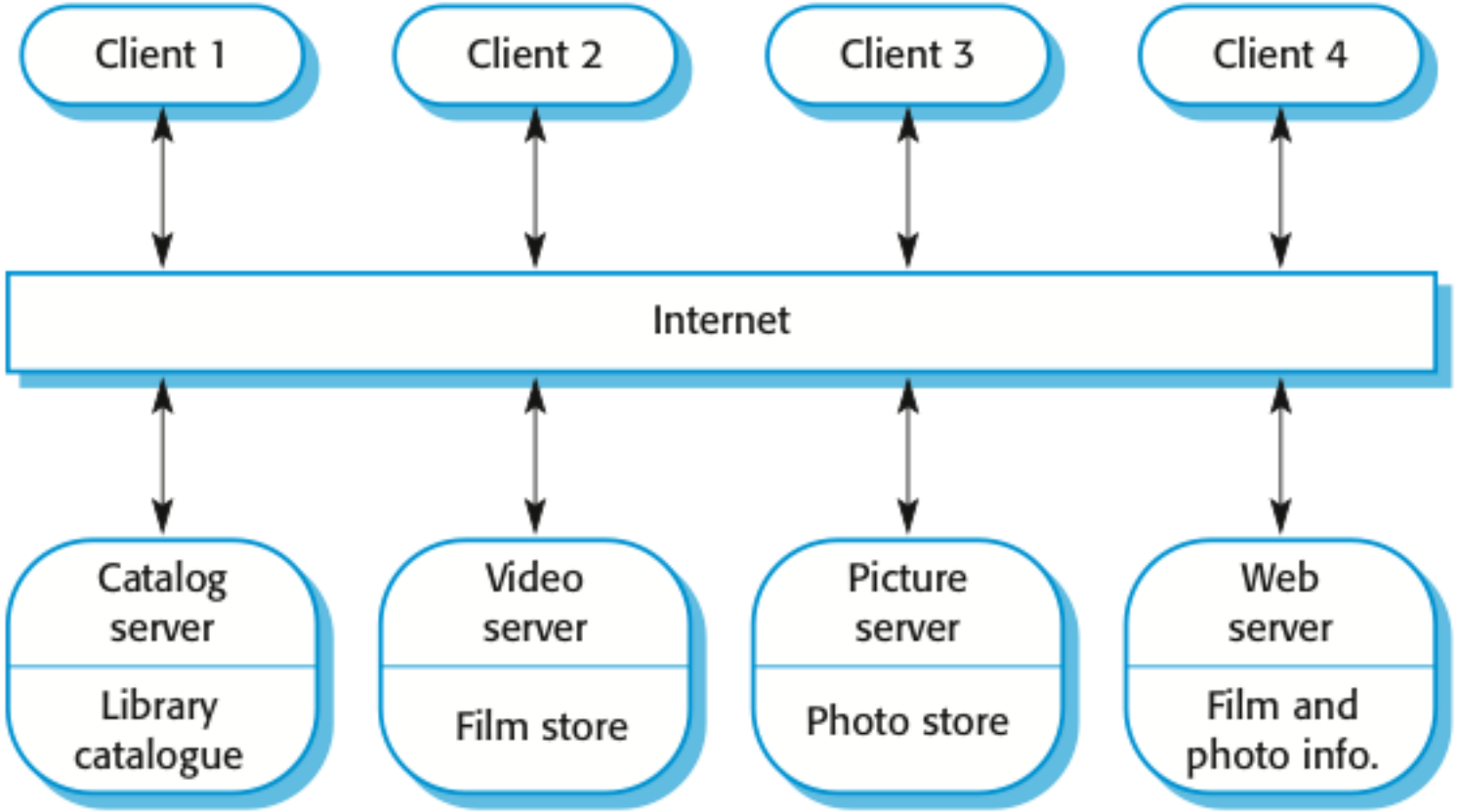
The Repository architecture pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure above is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

The Client-server architecture pattern

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

A client–server architecture for a film library



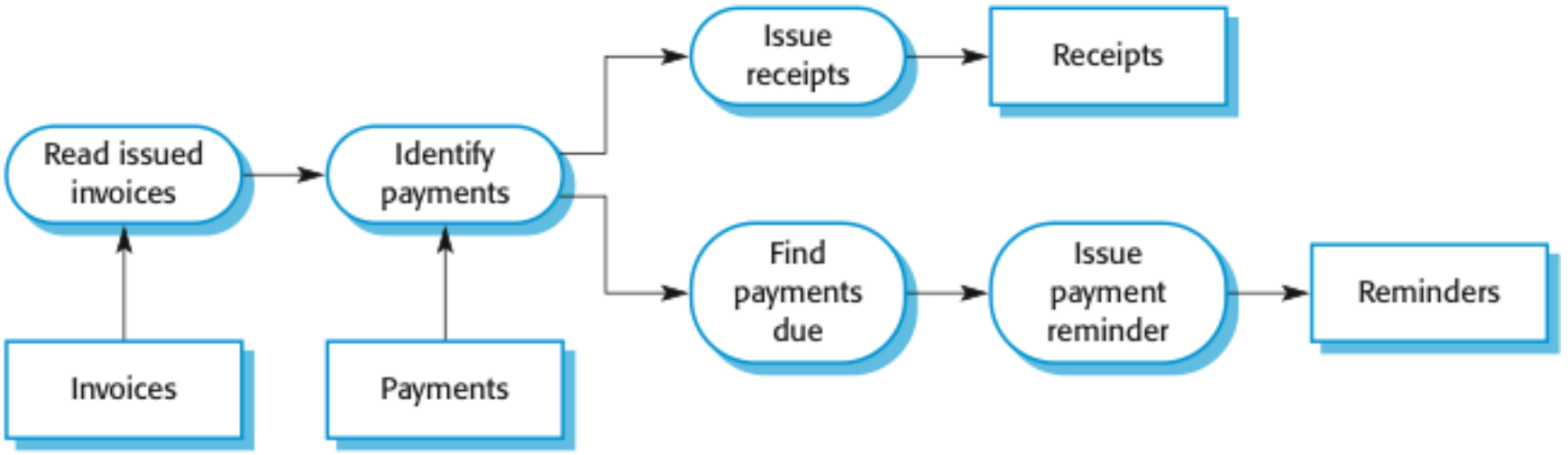
The Client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

The Pipe and filter architecture pattern

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipe and filter architecture



The Pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure above is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

Application architectures

- Application systems are designed to meet an organisational need.
- As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
- A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

Examples of application types

- Data processing applications
  - Data driven applications that process data in batches without explicit user intervention during the processing.
- Transaction processing applications
  - Data-centred applications that process user requests and update information in a system database.
- Event processing systems
  - Applications where system actions depend on interpreting events from the system's environment.
- Language processing systems
  - Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

Key points

- A software architecture is a description of how a software system is organized.
- Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.
- Application systems architectures embody a common architecture that the businesses have in common.

Implementation

Lecture 8/Part 2

Implementation

- Purpose:
- To convert the design model into an executable system
- I.e. to implement the design classes and components

- Artifacts:
  - Code
  - UML Component diagrams
  - UML Deployment diagrams

Implementation issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - **Reuse** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - **Configuration management** During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - **Host-target development** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels

- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management

- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Configuration management activities

- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Development platform tools

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

Integrated development environments (IDEs)

- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Open source development

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.

Open source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open source licensing models

- The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

Implementation good practices

Dependable programming guidelines

1. **Limit the visibility of information in a program**
2. **Check all inputs for validity**
3. **Provide a handler for all exceptions**
4. **Minimize the use of error-prone constructs**
5. **Provide restart capabilities**
6. **Check array bounds**
7. **Include timeouts when calling external components**
8. **Name all constants that represent real-world values**

Control the visibility of information in a program

- Program components should only be allowed access to data that they need for their implementation.
- This means that accidental corruption of parts of the program state by these components is impossible.
- You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().

Check all inputs for validity

- All program take inputs from their environment and make assumptions about these inputs.
- However, program specifications rarely define what to do if an input is not consistent with these assumptions.
- Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
- Consequently, you should always check inputs before processing against the assumptions made about these inputs.

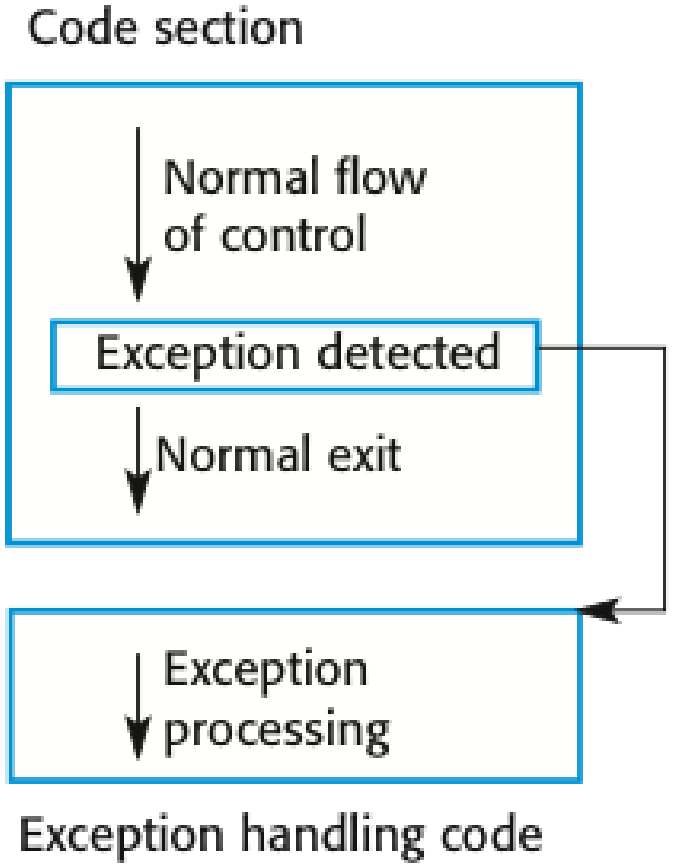
Validity checks

- Range checks
  - Check that the input falls within a known range.
- Size checks
  - Check that the input does not exceed some maximum size e.g. 40 characters for a name.
- Representation checks
  - Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.
- Reasonableness checks
  - Use information about the input to check if it is reasonable rather than an extreme value.

Provide a handler for all exceptions

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions needs many additional statements to be added to the program. This adds a significant overhead and is potentially error-prone.

Exception handling



Exception handling

- Three possible exception handling strategies
  - Signal to a calling component that an exception has occurred and provide information about the type of exception.
  - Carry out some alternative processing to the processing where the exception occurred. This is only possible where the exception handler has enough information to recover from the problem that has arisen.
  - Pass control to a run-time support system to handle the exception.
- Exception handling is a mechanism to provide some fault tolerance

Minimize the use of error-prone constructs

- Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
- This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don't check for mistakes when they could do so.
- Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

Error-prone constructs

- ✧ Unconditional branch (goto) statements
- ✧ Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
- ✧ Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- ✧ Dynamic memory allocation
  - Run-time allocation can cause memory overflow.

Error-prone constructs

- Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- Recursion
  - Errors in recursion can cause memory overflow as the program stack fills up.
- Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
- Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Error-prone constructs

- Aliasing
  - Using more than 1 name to refer to the same state variable.
- Unbounded arrays
  - Buffer overflow failures can occur if no bound checking on arrays.
- Default input processing
  - An input action that occurs irrespective of the input.
  - This can cause problems if the default action is to transfer control elsewhere in the program. In incorrect or deliberately malicious input can then trigger a program failure.

Provide restart capabilities

- For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
- Restart depends on the type of system
  - Keep copies of forms so that users don't have to fill them in again if there is a problem
  - Save state periodically and restart from the saved state

Check array bounds

- In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- This leads to the well-known 'bounded buffer' vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

Include timeouts when calling external components

- In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- To avoid this, you should always include timeouts on all calls to external components.
- After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

Name all constants that represent real-world values

- Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- This means that when these ‘constants’ change (for sure, they are not really constant), then you only have to make the change in one place in your program.

Clean code by Robert C. Martin

- A handbook of agile software craftsmanship
- Guidelines for:
  - Meaningful names
  - Functions
  - Comments
  - Formatting
  - Objects and data structures
  - Error handling
  - Concurrency
  - … and others
- Smells and heuristics

Key points

- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

UML Packages (Analysis)

Lecture 8/Part 3

Packages

- A package is a *general purpose* mechanism for organising model elements into groups
  - Group semantically related elements
  - Define a "semantic boundary" in the model
  - Provide units for parallel working and configuration management
  - Each package defines an *encapsulated namespace* i.e. all names must be unique within the package
- In UML 2 a package is a purely logical grouping mechanism
  - Use components for physical grouping
- Analysis packages contain:
  - Use cases, analysis classes, use case realizations, analysis packages

Package syntax

Membership

Membership

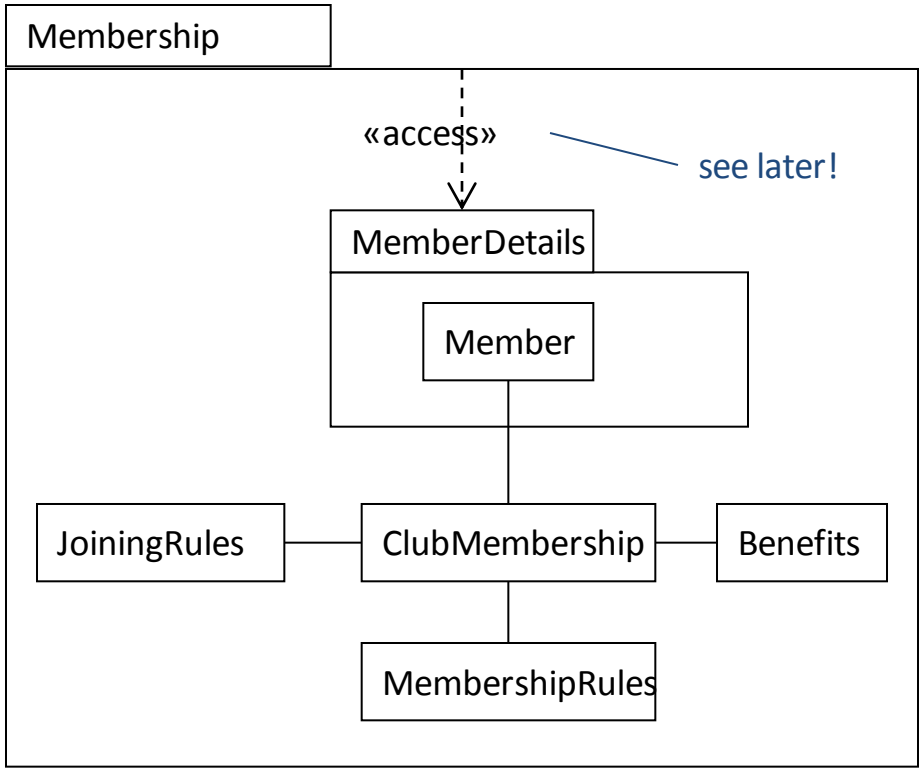
public (exported) elements

private element

+ClubMembership
+Benefits
+MembershipRules
+MemberDetails:Member
-JoiningRules

qualified package name

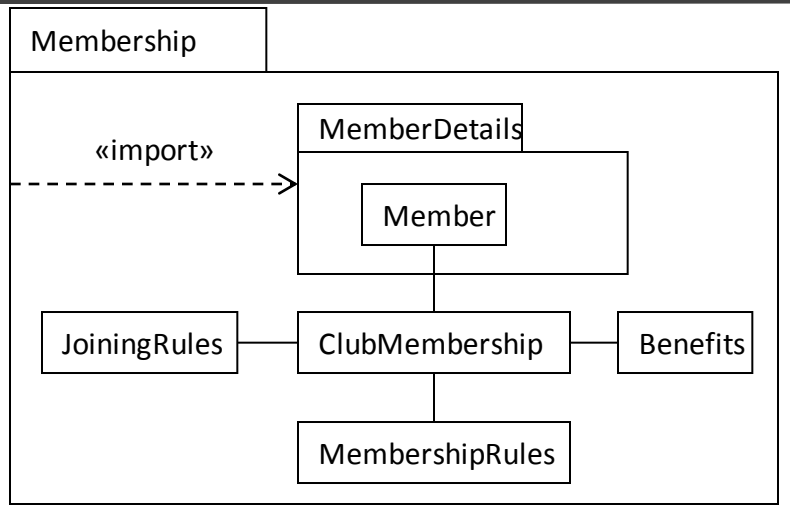
Membership:MemberDetails

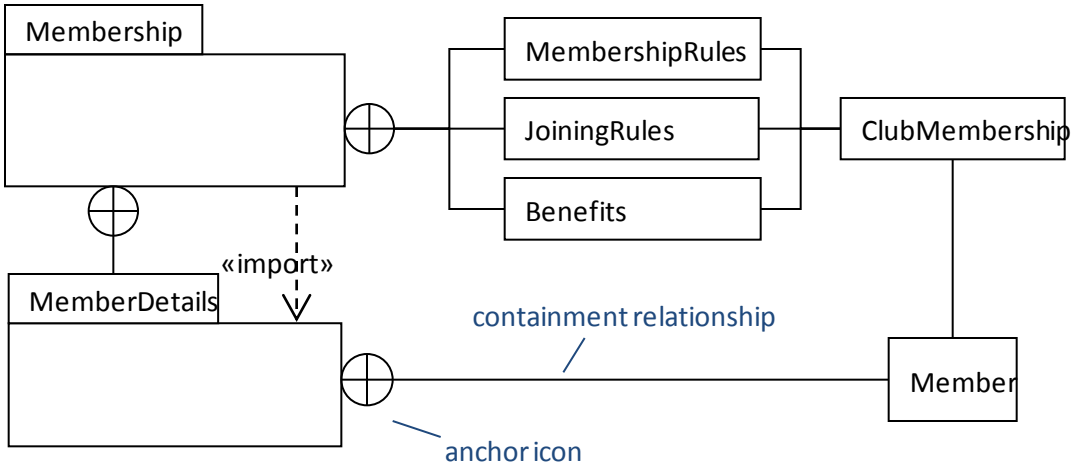


standard UML 2 package stereotypes	
«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages Analogous to a class library in Java, C# etc.

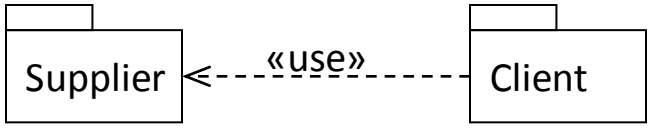
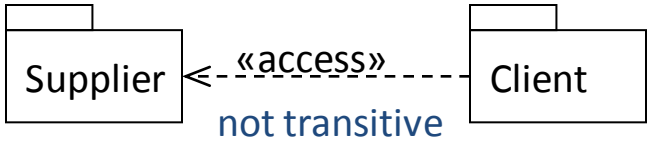
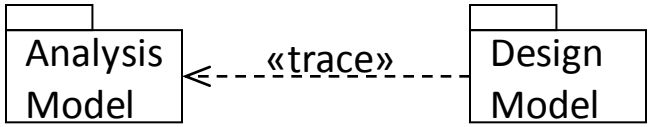
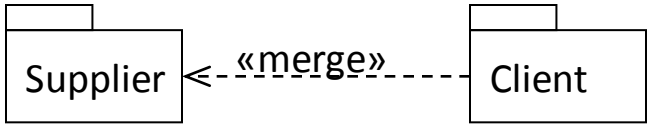
Nested packages

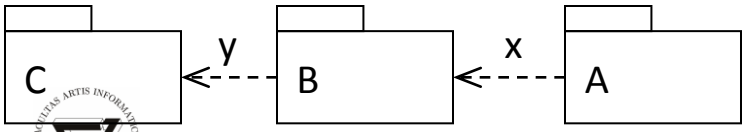
- If an element is visible within a package then it is visible within all nested packages
  - e.g. Benefits is visible within MemberDetails
- Show containment using nesting or the containment relationship
- Use «access» or «import» to merge the namespace of nested packages with the parent namespace





Package dependencies

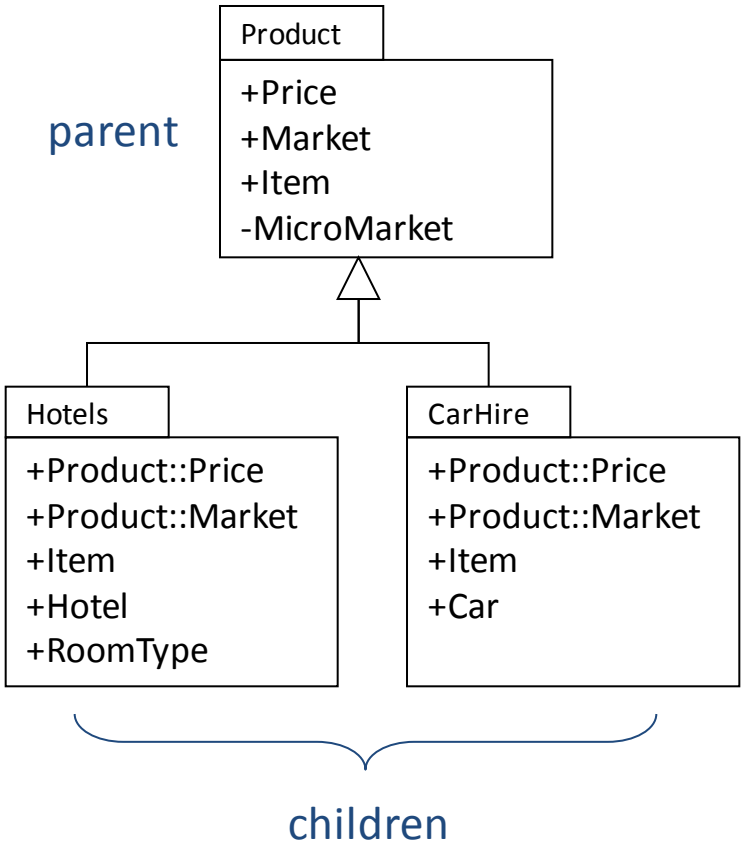
dependency	semantics
	An element in the client uses an element in the supplier in some way. The client depends on the supplier. Transitive.
	Public elements of the supplier namespace are added as public elements to the client namespace. Transitive.
	Public elements of the supplier namespace are added as private elements to the client namespace. Not transitive.
	«trace» usually represents an historical development of one element into another more refined version. It is an extra-model relationship. Transitive.
	The client package merges the public contents of its supplier packages. This is a complex relationship only used for metamodeling - you can ignore it.



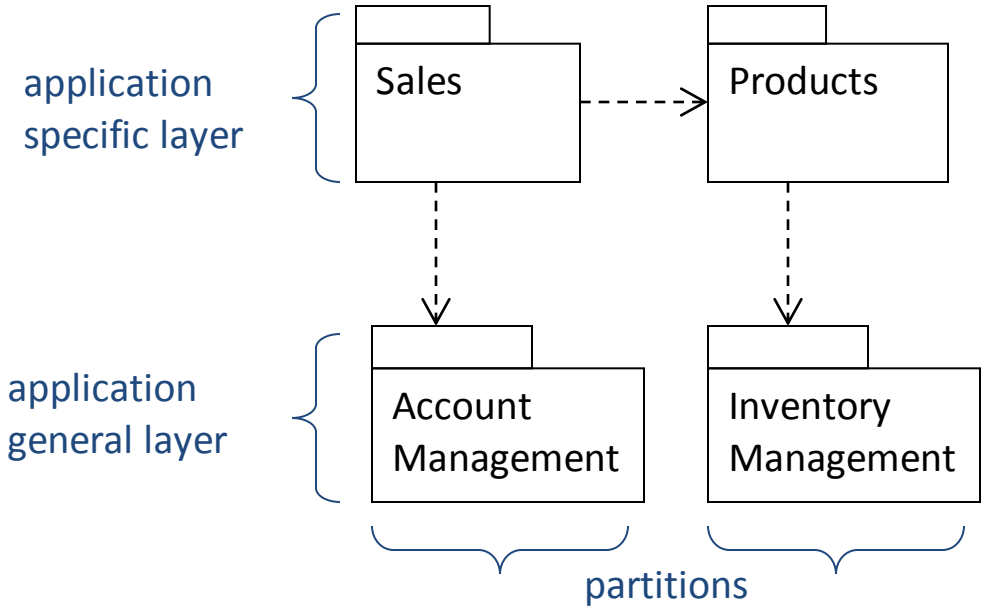
transitivity - if dependencies x and y are transitive, there is an *implicit* dependency between A and C

Package generalisation

- The more specialised child packages *inherit* the public and protected elements in their parent package
- Child packages may *override* elements in the parent package. Both Hotels and CarHire packages override Product::Item
- Child packages may *add* new elements. Hotels adds Hotel and RoomType, CarHire adds Car



Architectural analysis



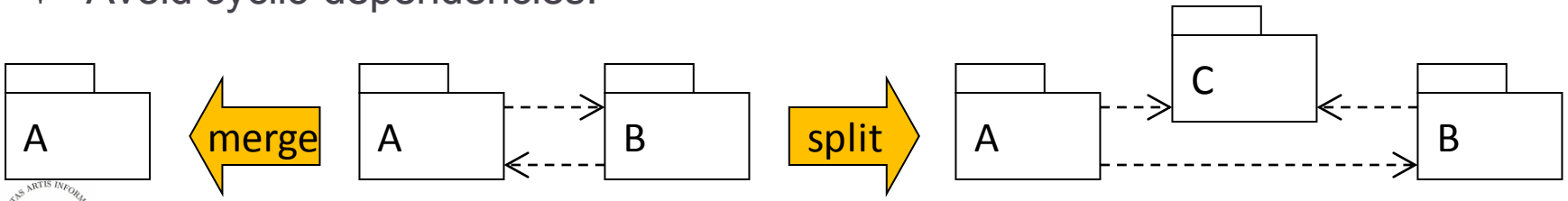
- This involves organising the analysis classes into a set of cohesive packages
- The architecture should be *layered* and *partitioned* to separate concerns
  - It's useful to layer analysis models into application specific and application general layers
- Coupling between packages should be minimised
- Each package should have the minimum number of public or protected elements

Finding analysis packages

- These are often discovered as the model matures
- We can use the natural groupings in the use case model to help identify analysis packages:
  - One or more use cases that support a particular business process or actor
  - Related use cases
- Analysis classes that realise these groupings will often be part of the same analysis package
- Be careful, as it is common for use cases to *cut across* analysis packages!
  - One class may realise several use cases that are allocated to different packages

Analysis packages: guidelines

- A cohesive group of closely related classes or a class hierarchy
- Minimise dependencies between packages
- Localise business processes in packages where possible
- Minimise nesting of packages
- Don't worry about dependency stereotypes
- Don't worry about package generalisation
- Refine package structure as analysis progresses
- 4 to 10 classes per package
- Avoid cyclic dependencies!



Key points

- Packages are the UML way of grouping modeling elements
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture

UML Component Diagram (Design)

Lecture 8/Part 4

What is an interface?

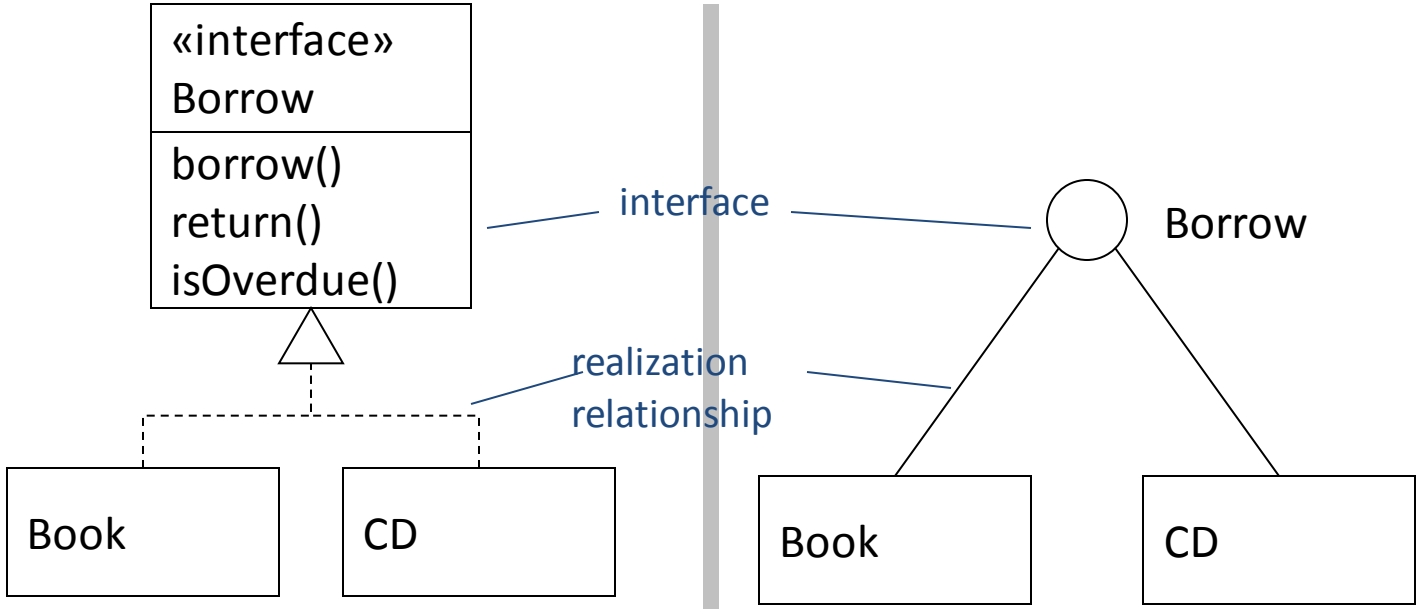
design by contract

- An interface specifies a named set of public features
- It separates the specification of functionality from its implementation
- An interface defines a contract that all realizing classifiers *must* conform to:

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 not 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, then 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	Realizes the protocol

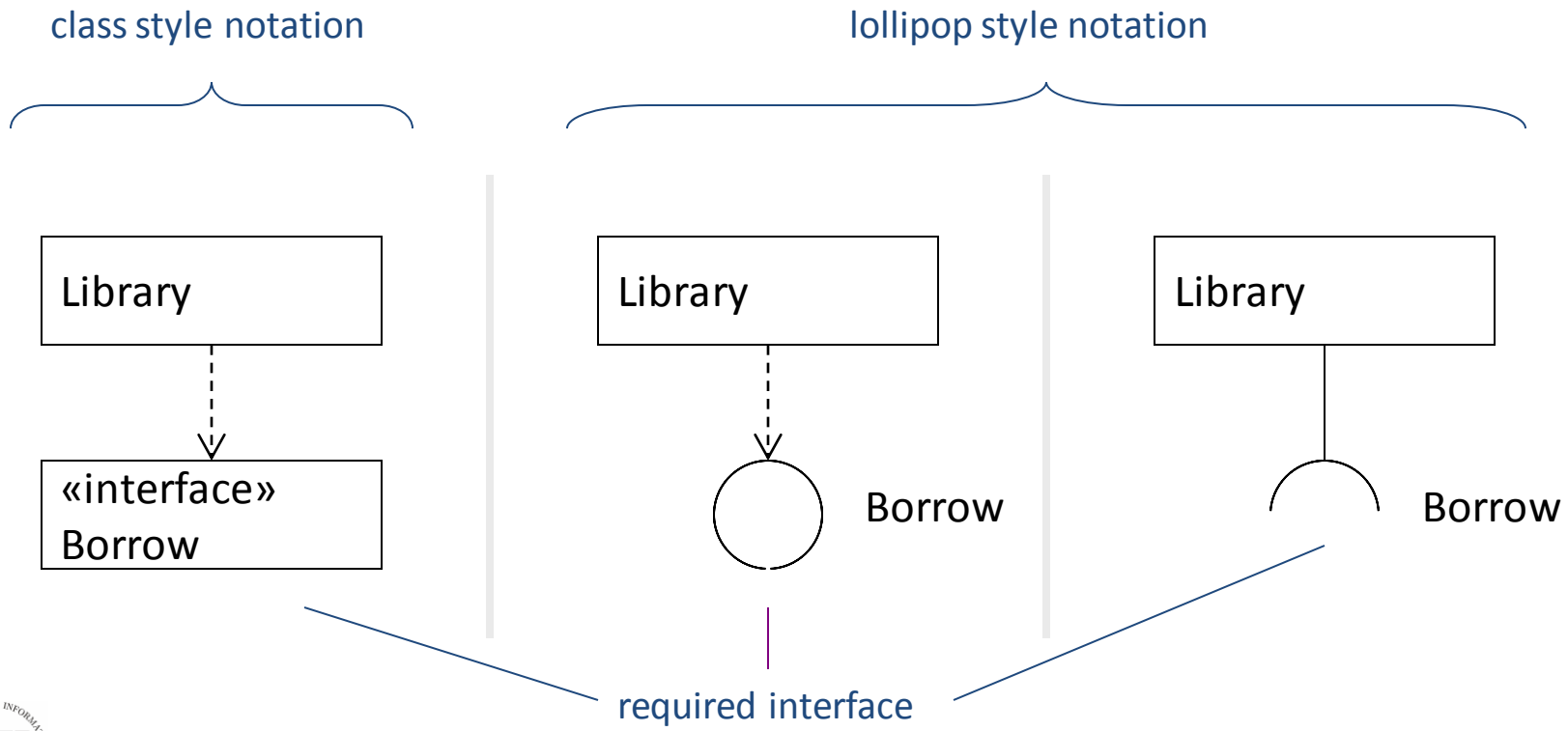
Provided interface syntax

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



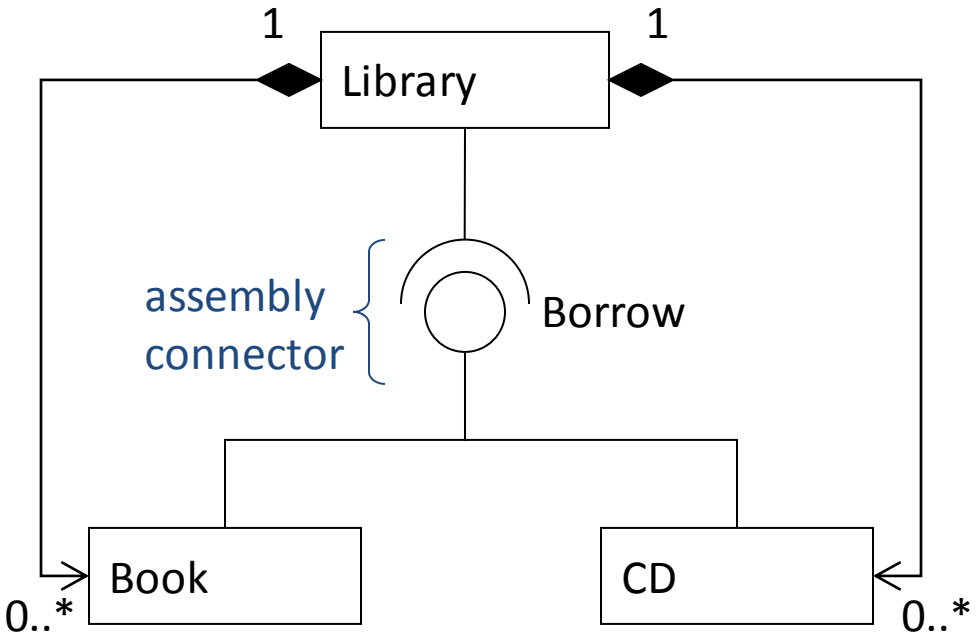
Required interface syntax

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



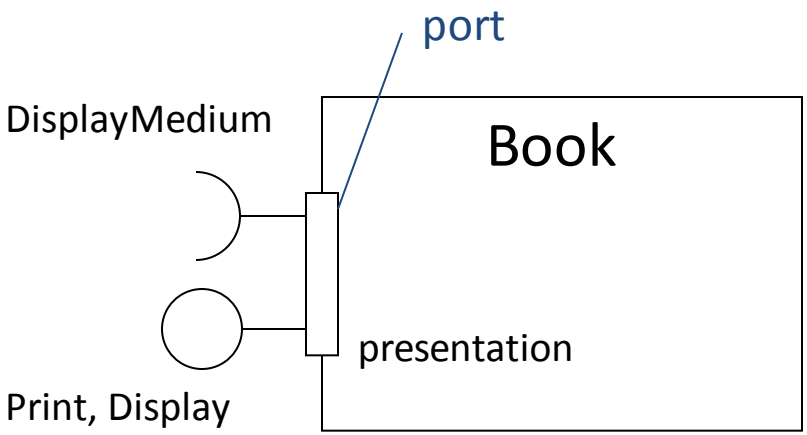
Assembly connectors

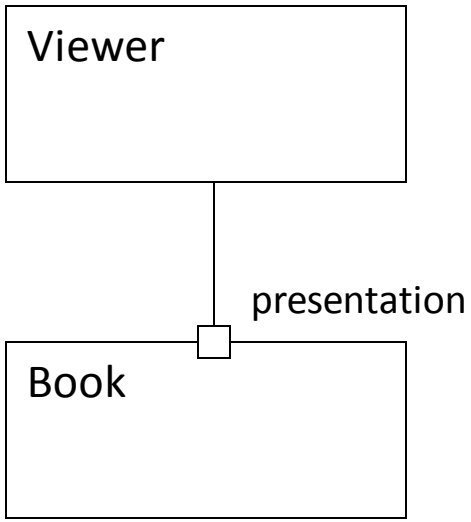
- You can connect provided and required interfaces using an assembly connector



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





Interfaces and component-based development (CBD)

- Interfaces are the key to component based development
- This is 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

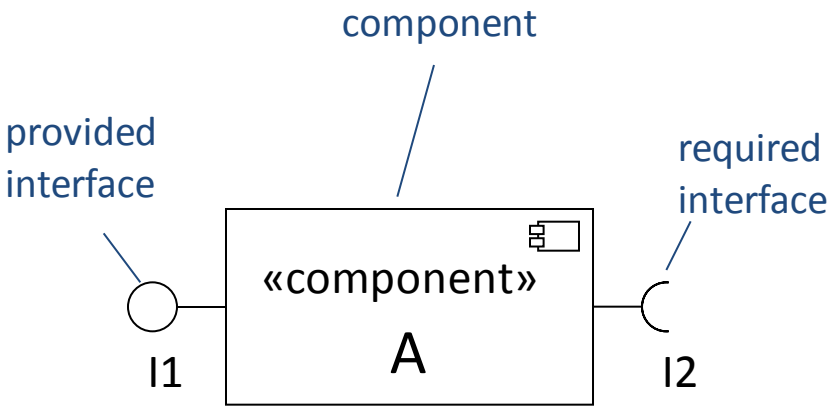
What is a component?

- 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

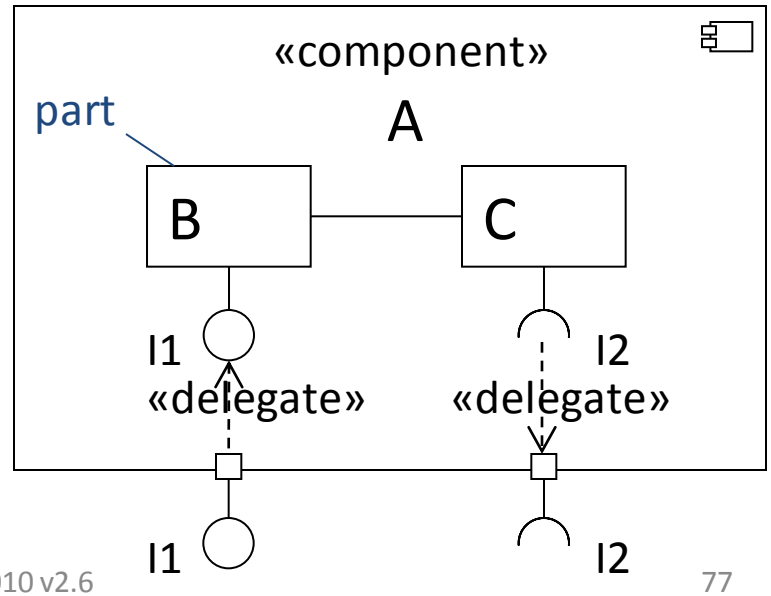
Component syntax

- 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

black box notation



white box notation

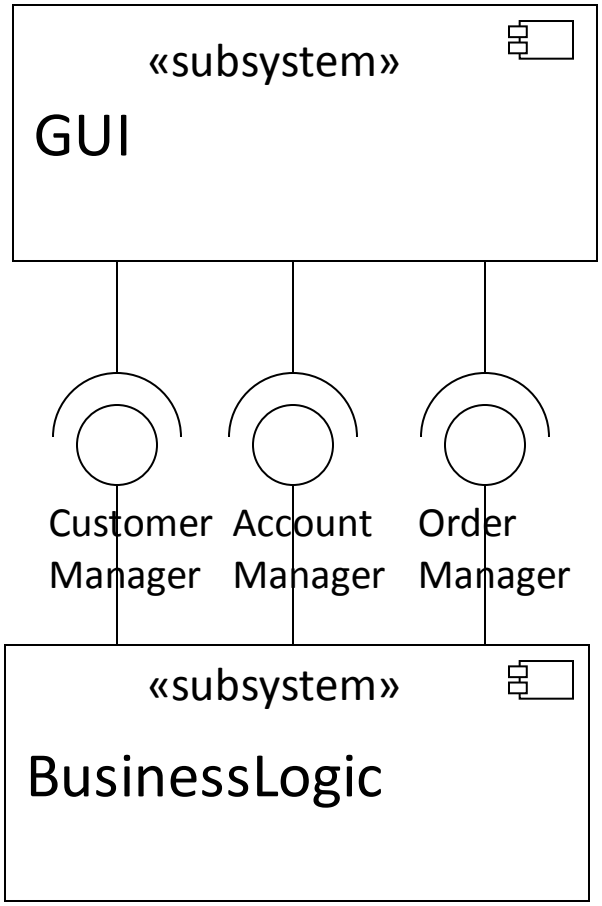


Standard component stereotypes

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.

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
- Interfaces connect subsystems together to create a system architecture



Finding interfaces and ports

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

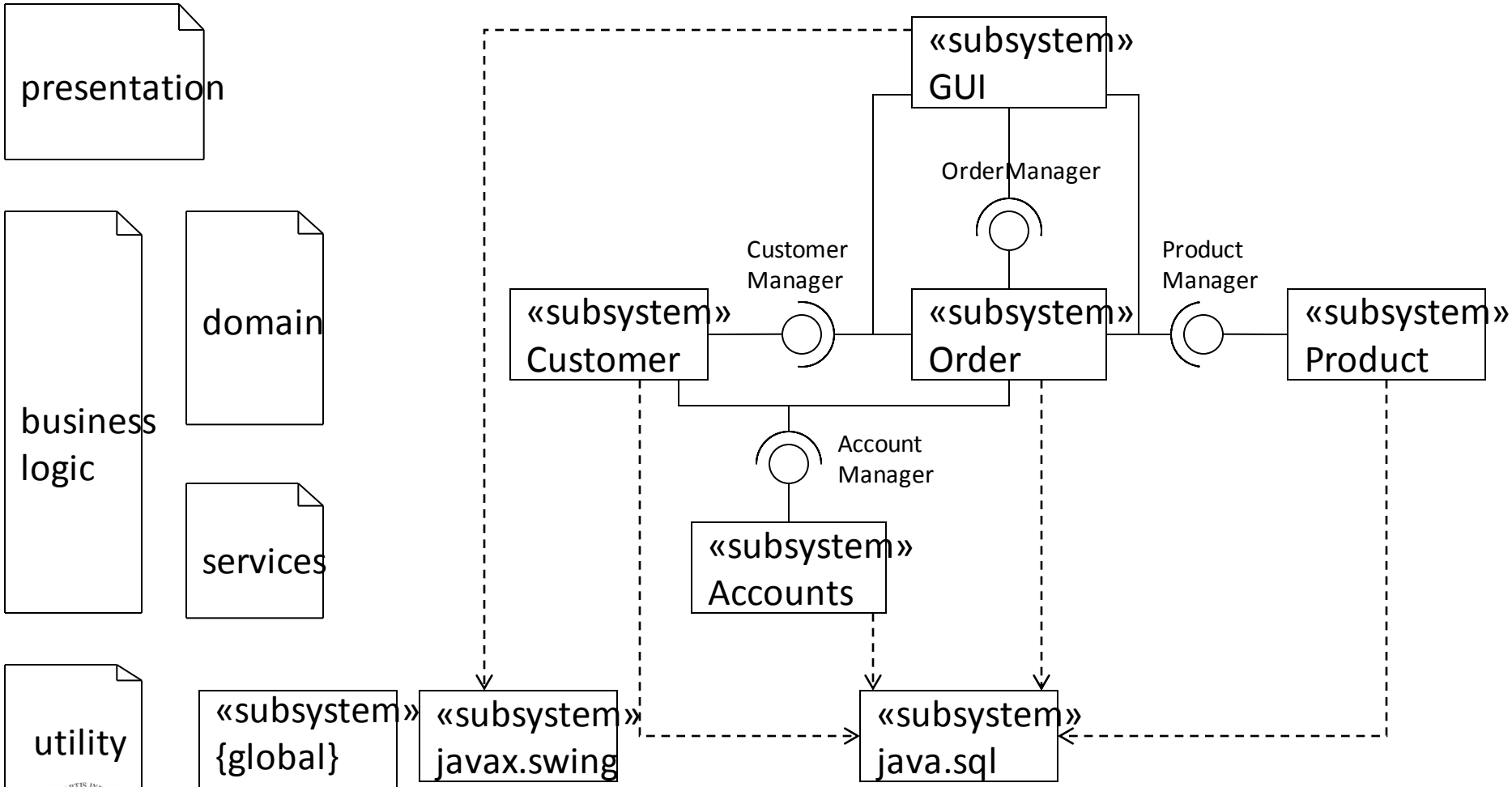
Designing with interfaces

- 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
- Façade Pattern – interfaces to create "seams" in a system
  - Identify cohesive parts of the system
  - Package these into a «subsystem»
  - Define an interface to that subsystem
- Interfaces for information hiding and separation of concerns

Physical architecture

- 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

Example layered architecture



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
  - Too many interfaces can make a system too flexible!
  - Too many interfaces can make a system hard to understand

Keep it simple!

Key points

- 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 an architecture

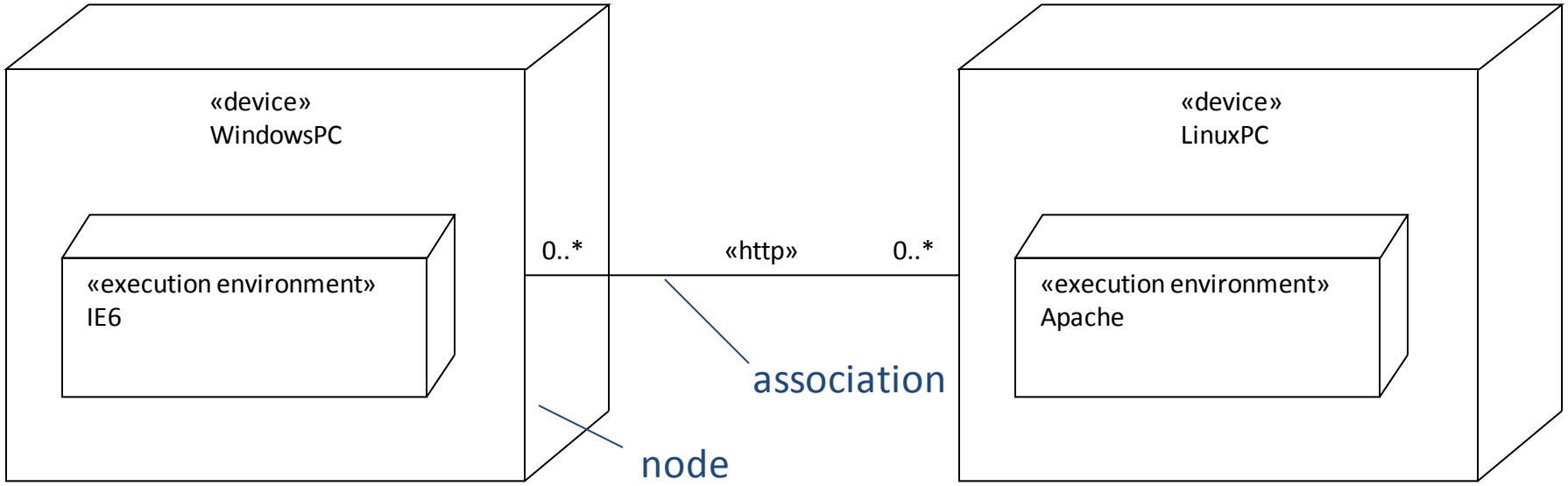
UML Deployment Diagram (Implementation)

Lecture 8/Part 5

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 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 the implementation workflow
- Implementation - finish the deployment diagram:
  - Focus on artifact deployment on nodes

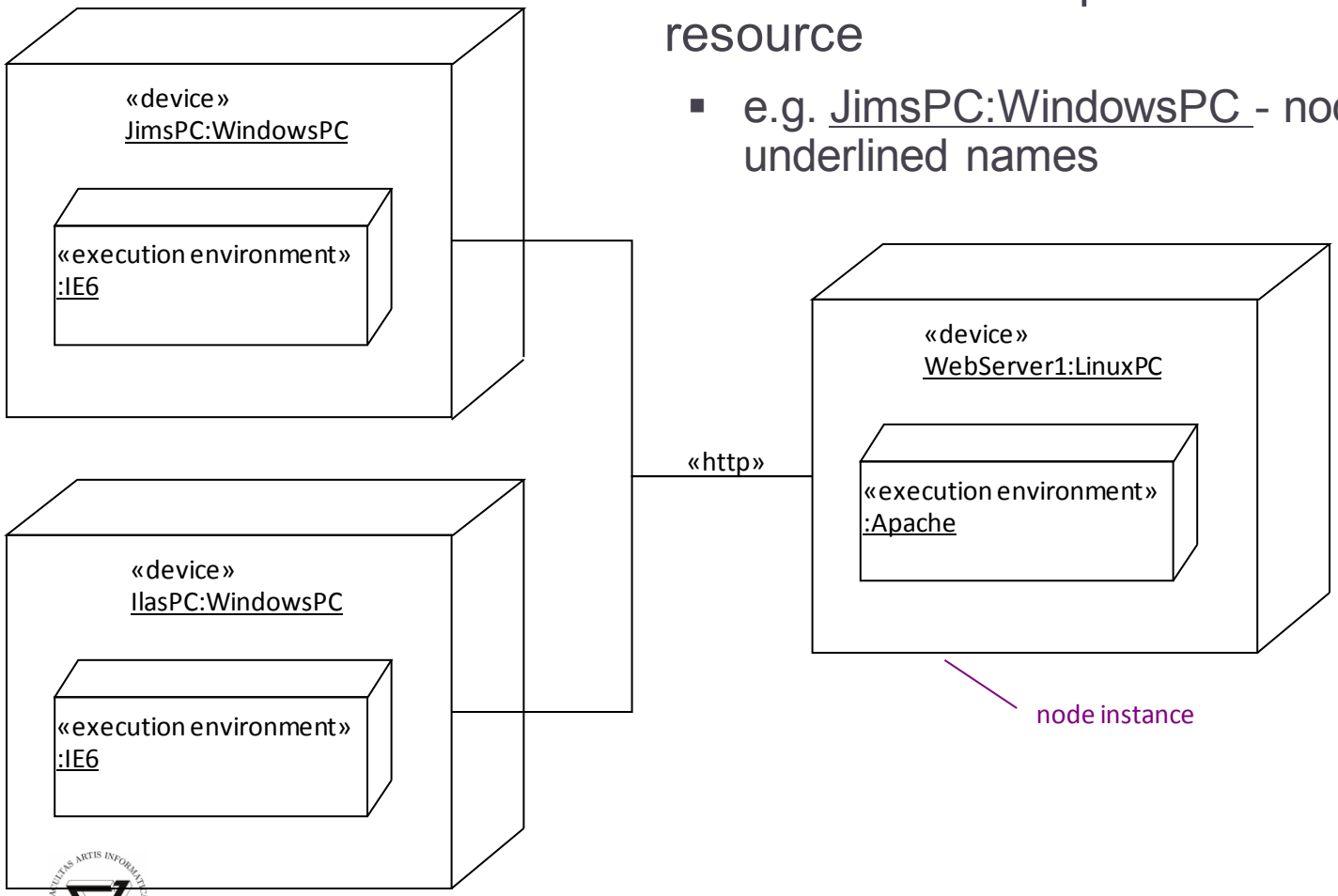
Nodes – descriptor form



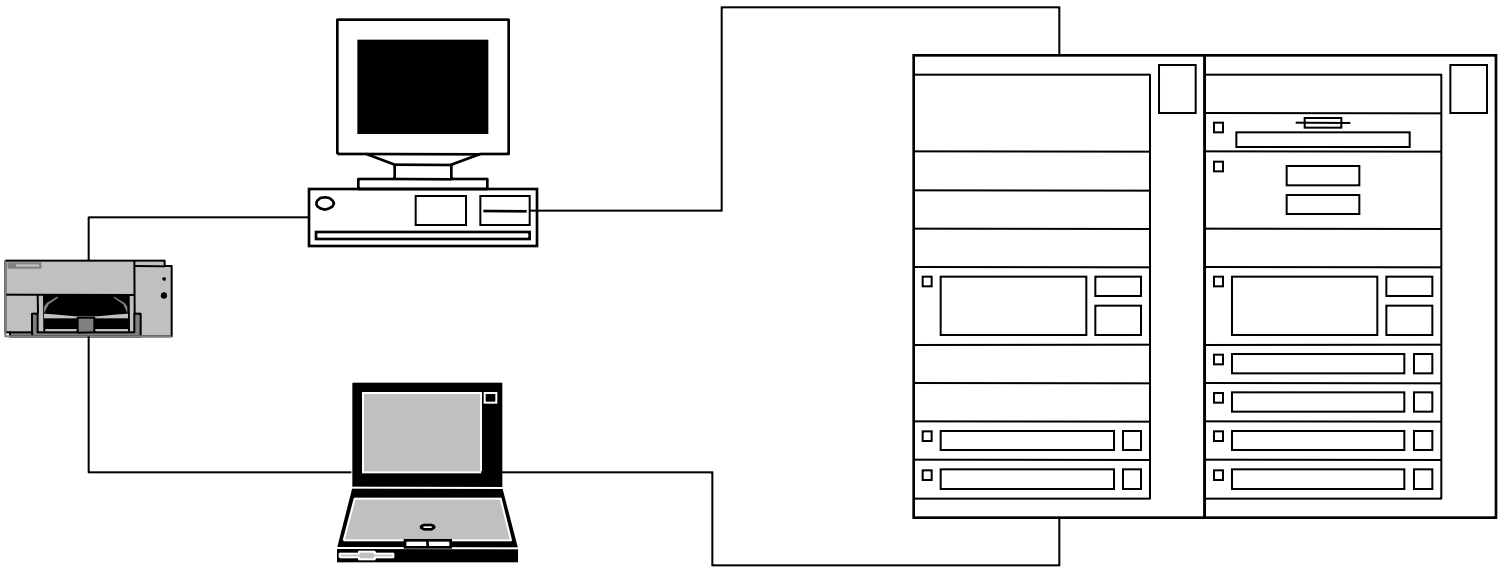
- A node represents a type of computational resource
  - e.g. a WindowsPC
- Standard stereotypes are «device» and «execution environment»

Nodes – instance form

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



Stereotyping nodes



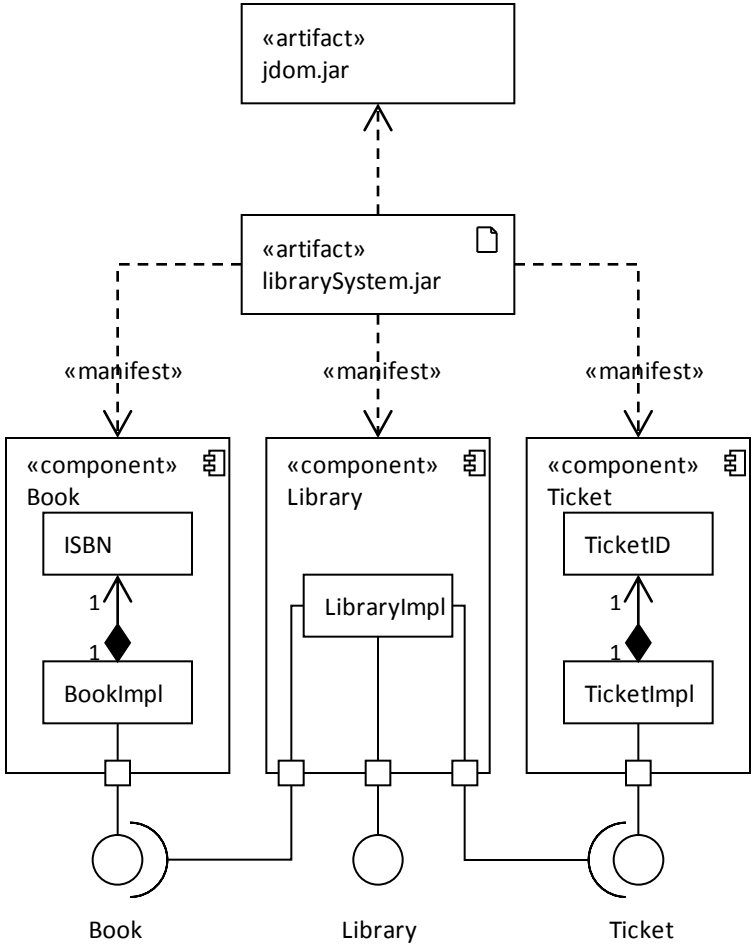
- It's very useful to use lots of stereotyping on the deployment diagram to make it as clear and readable as possible

Artifacts

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

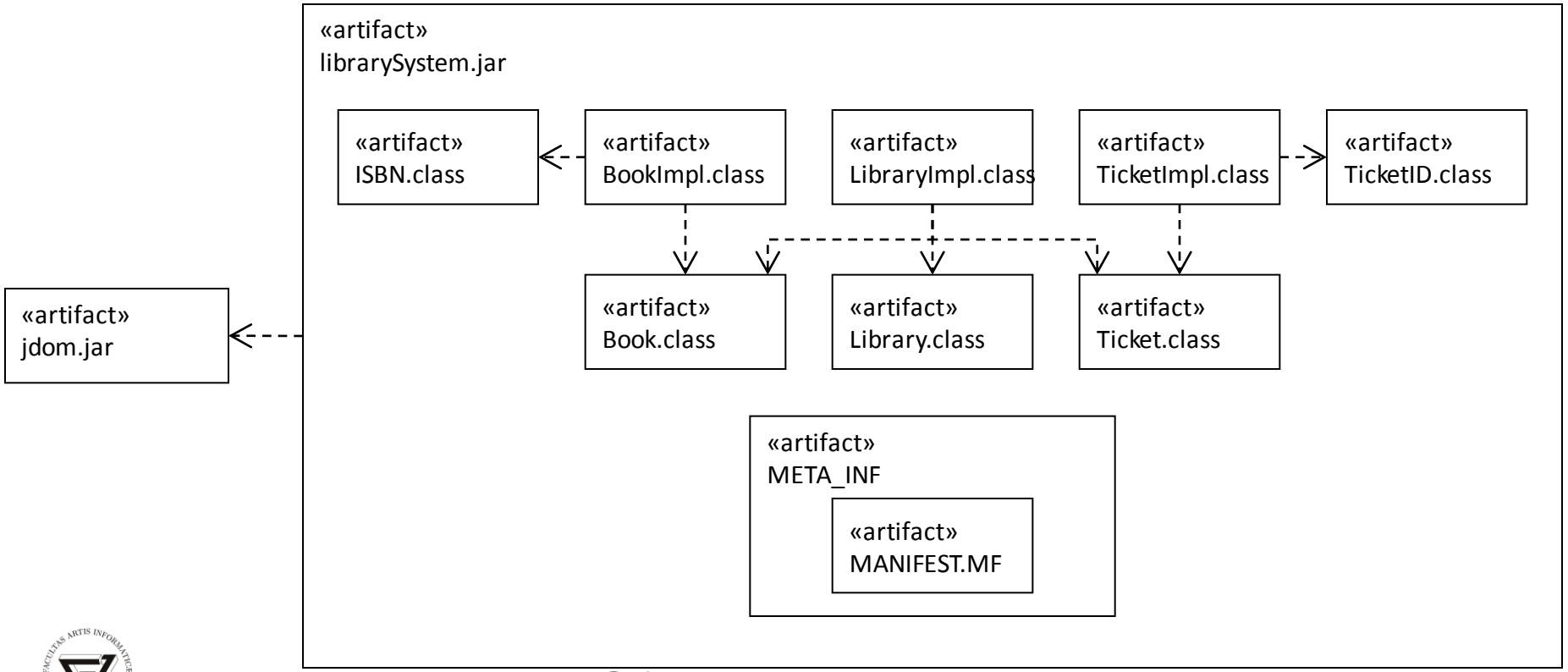
Artifacts and components

- 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



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



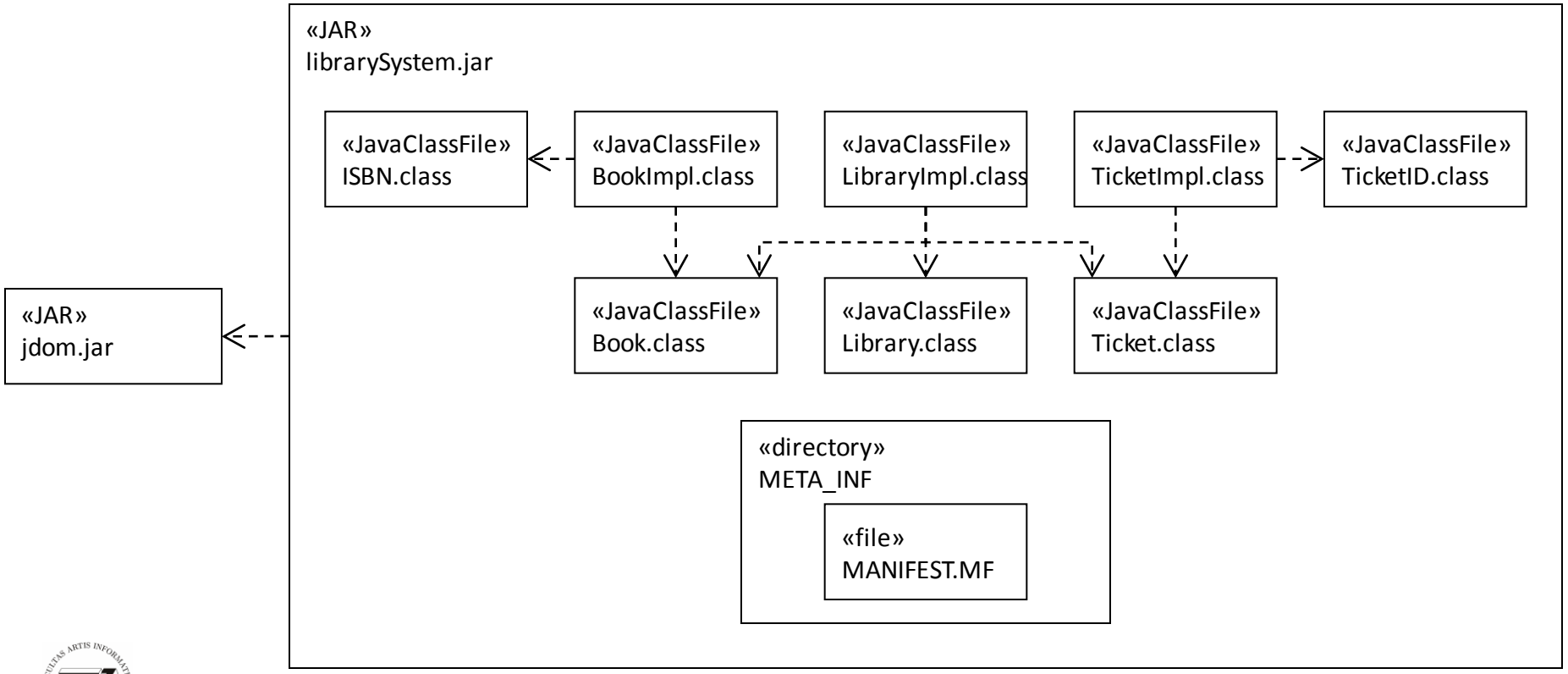
Artifact standard stereotypes

- 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

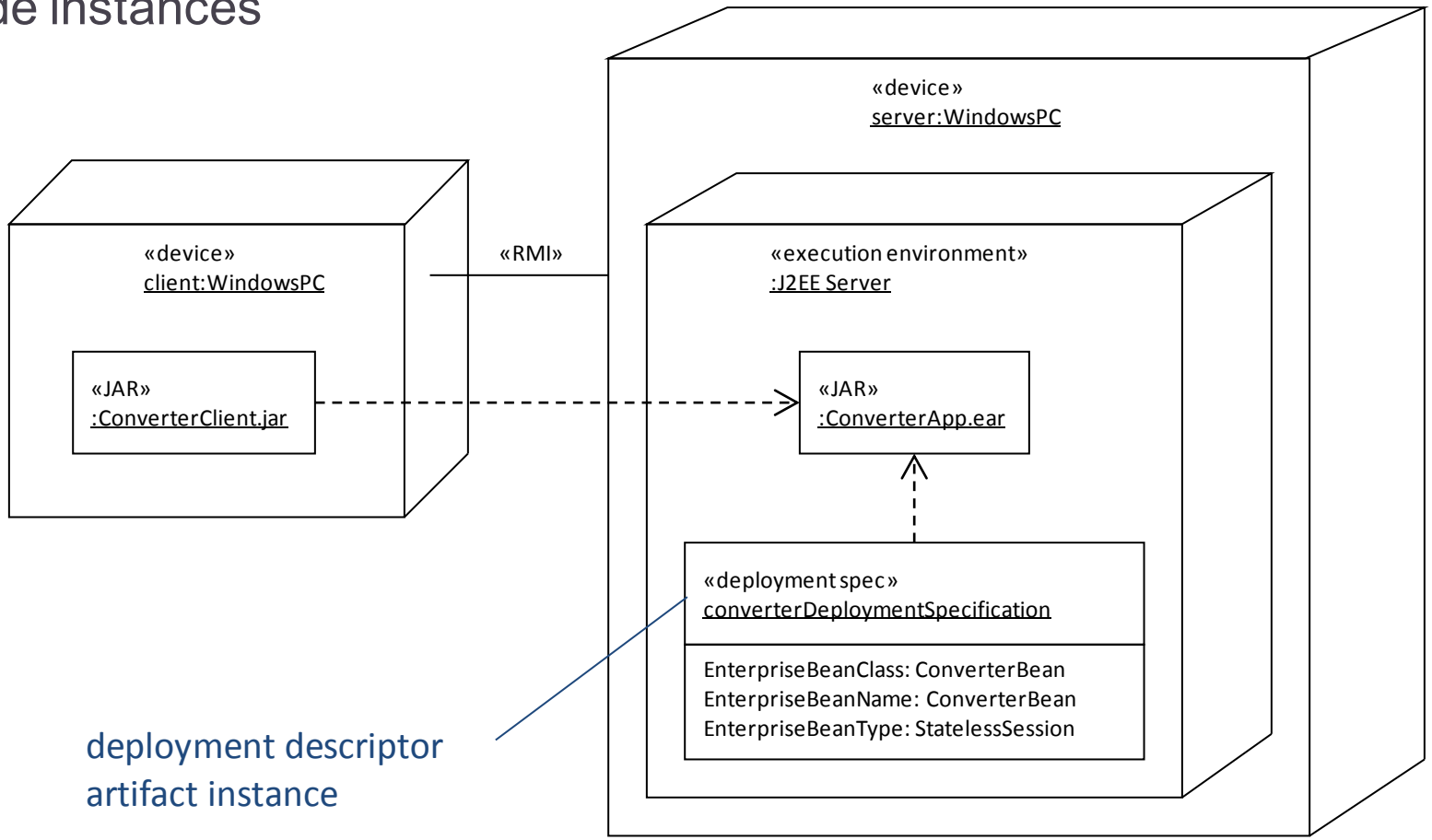
Stereotyping artifacts

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



Deployment

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



Key points

- 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