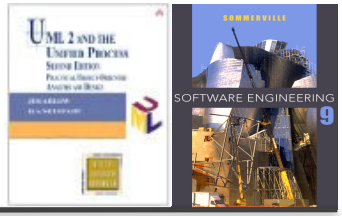


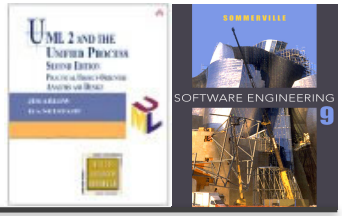
Lecture 8

ARCHITECTURE DESIGN

PB007 Software Engineering I
Faculty of Informatics, Masaryk University

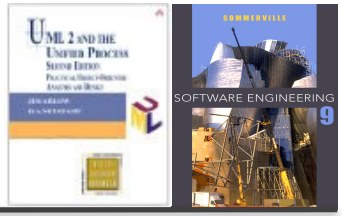
Fall 2016

Topics covered



- ✧ Architecture design

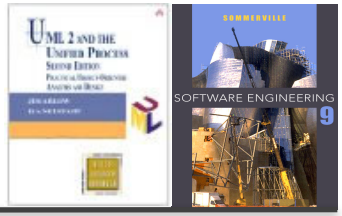
- ✧ UML Packages (Analysis)
- ✧ UML Component Diagram (Design)
- ✧ UML Deployment Diagram (Realisation)



Architecture Design

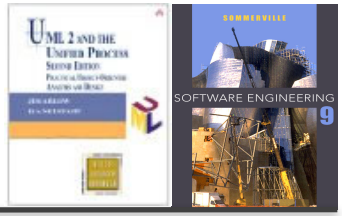
Lecture 8/Part 1

Topics covered



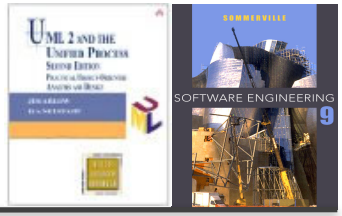
- ✧ Architectural views
- ✧ Architectural design decisions
- ✧ Architectural patterns
- ✧ Application architectures

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.

4 + 1 view model of software architecture



Process view shows how, at run-time, the system is composed of interacting processes.

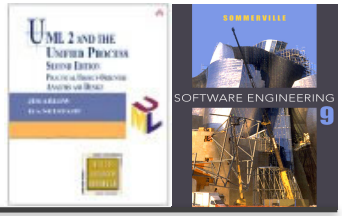
Logical view shows the key abstractions in the system as objects or object classes.

Use cases and scenarios view

Development view shows how the software is decomposed for development.

Physical view shows system hardware and how software components are distributed across system processors.

Architectural design decisions

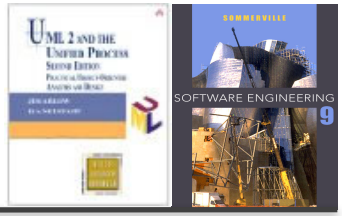


- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

- **Software architecture** gives answers to the most expensive questions.

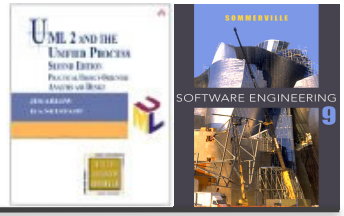
  – heard from O. Krajíček

Architectural design decisions



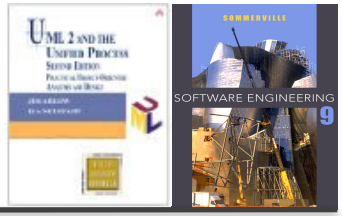
- **How will the system be decomposed into modules?**
- What approach will be used to structure the system?
- **What architectural styles are appropriate?**
- What control strategy should be used?
- **Is there a generic application architecture that can be used?**
- How should the architecture be documented?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

Architecture and system characteristics



- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of components.
- Reliability and Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

Architectural patterns

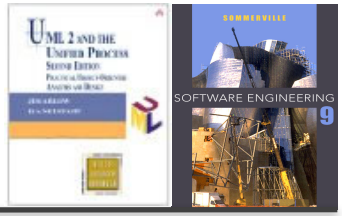


- Patterns are a means of representing, sharing and reusing knowledge.
- An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- Patterns should include information about when they are and when the are not useful.

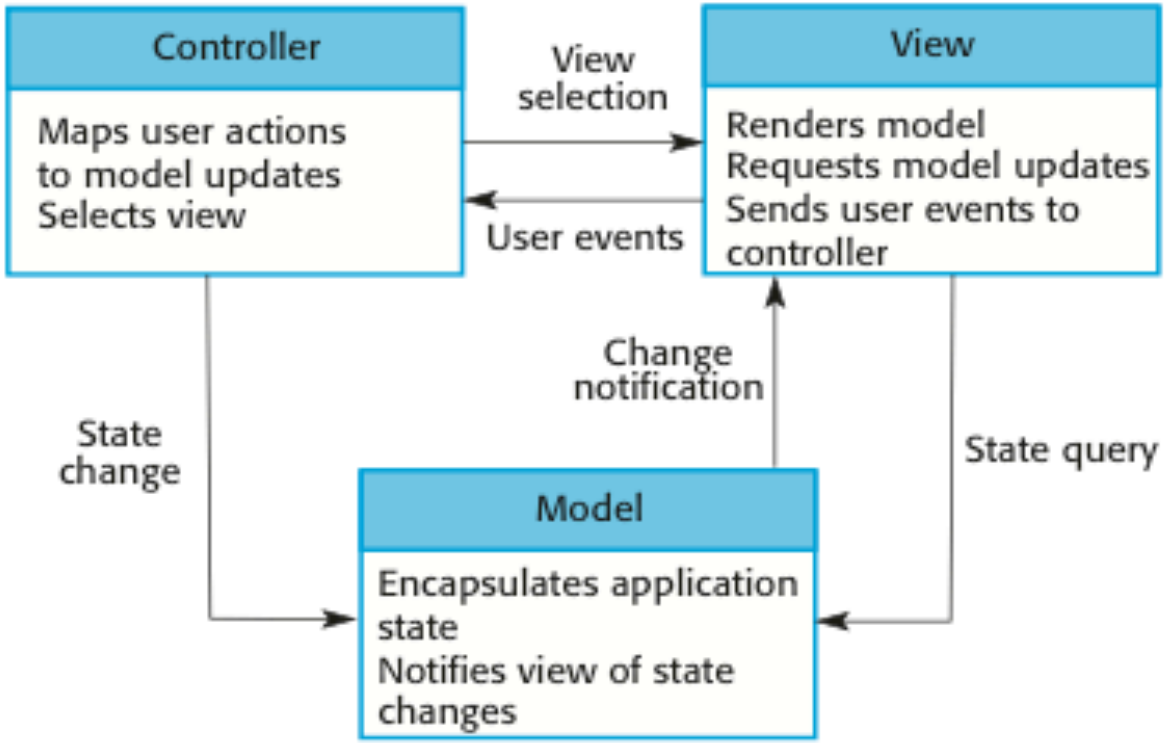
The zen of architecture = For the beginner architect there are so many ways and options of doing pretty much anything, but for a master, there only are a few.

– Juval Löwy

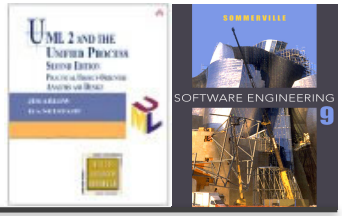
The Model-View-Controller (MVC) pattern



✧ Separates presentation and interaction from the system data.

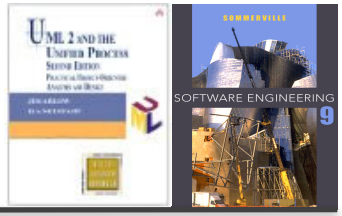


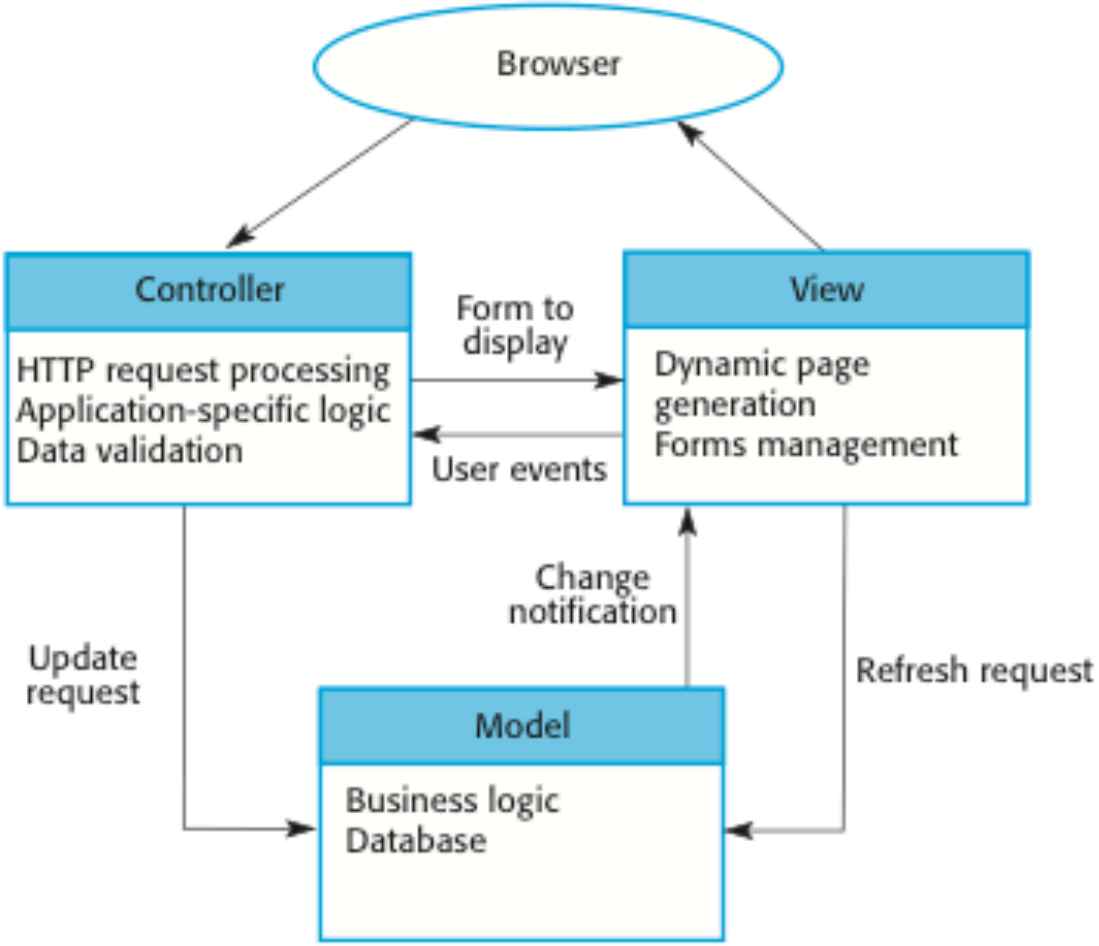
The Model-View-Controller (MVC) pattern



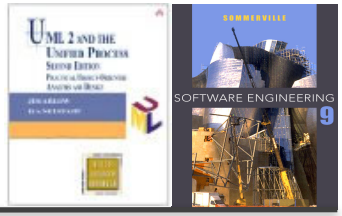
Name	MVC (Model-View-Controller)
Description	**Separates presentation and interaction from the system data**. The system is structured into three logical components that interact with each other. The **Model component** manages the system data and associated operations on that data. The **View component** 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

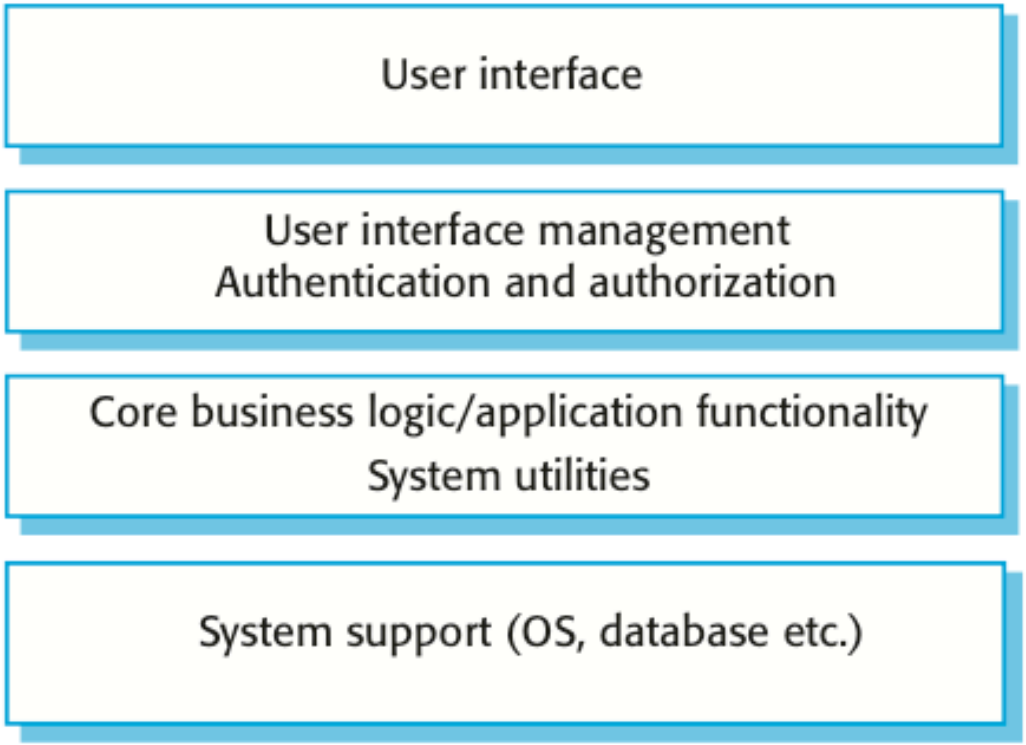




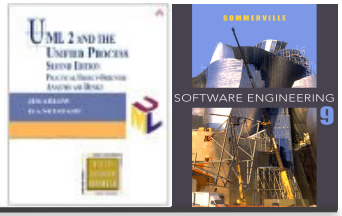
The Layered architecture pattern



- Organises the system into a set of layers with interfaces to other layers. Supports incremental development.

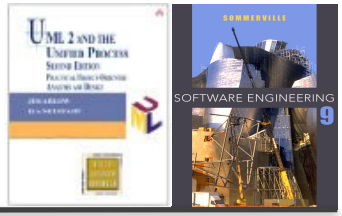


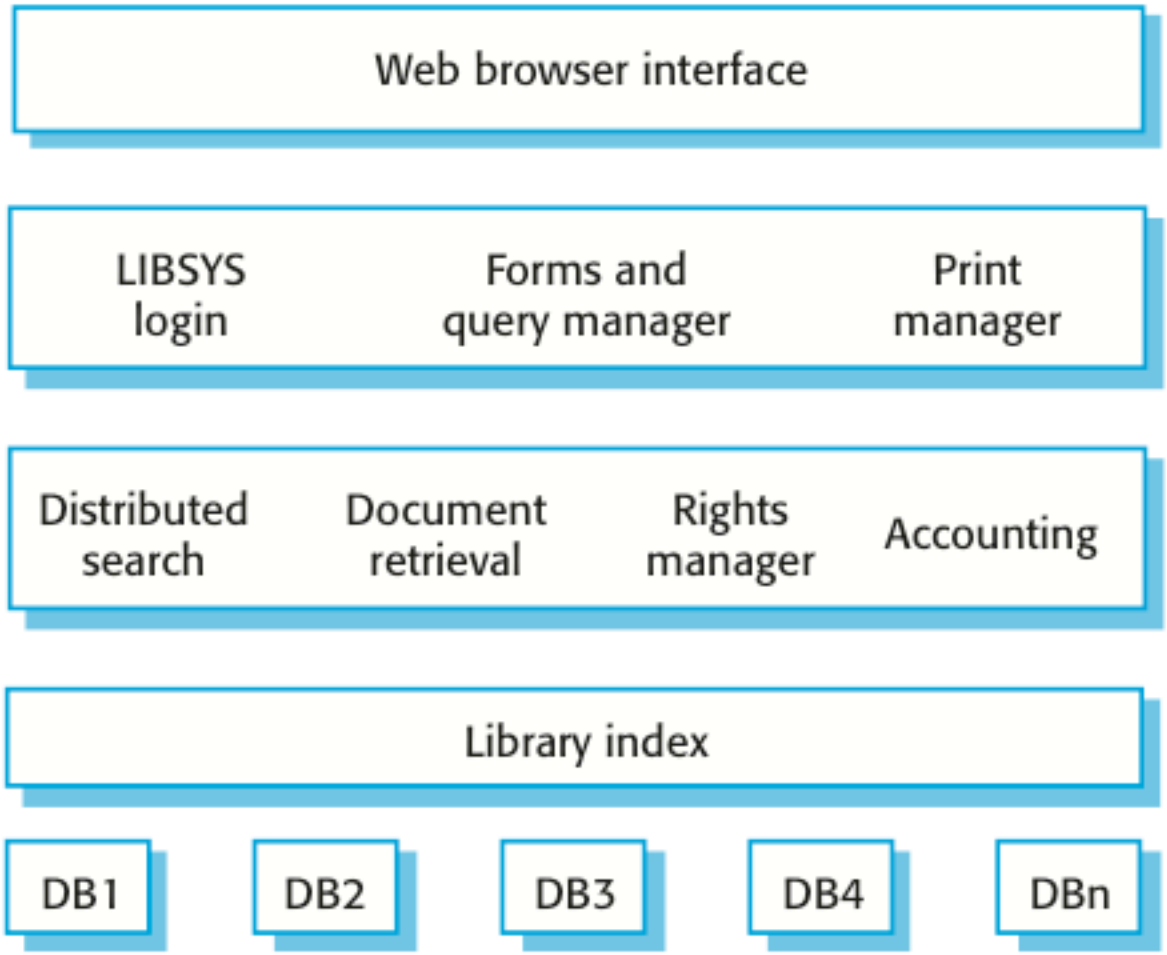
The Layered architecture pattern



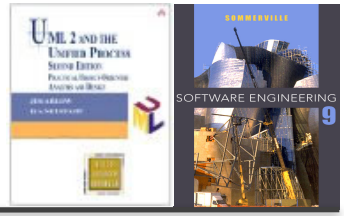
Name	Layered architecture
Description	Organizes the system into **layers with related functionality** associated with each layer. A layer **provides services to the layer above it** so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered 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

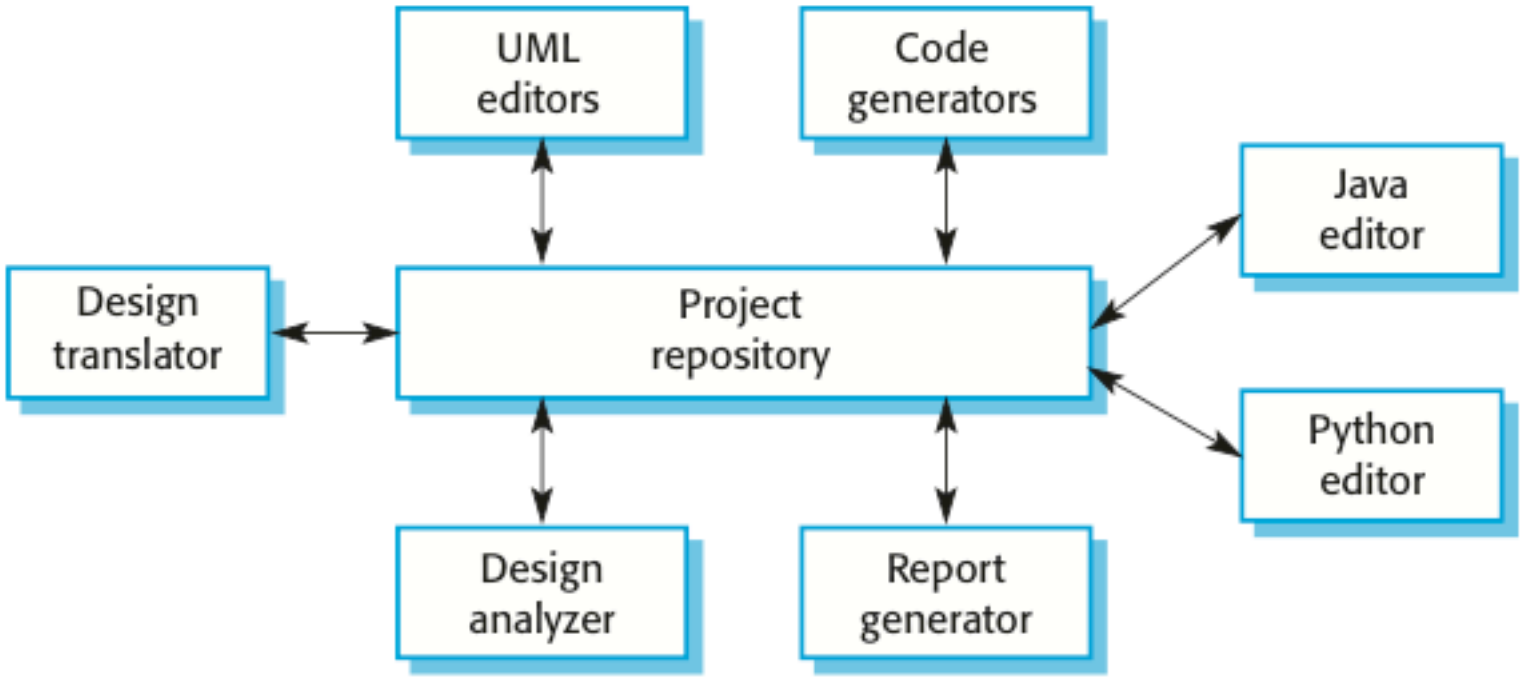




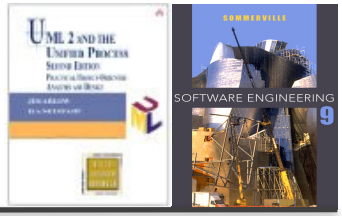
The Repository architecture pattern



- When large amounts of data are to be shared among subsystems, the repository model offers a solution.

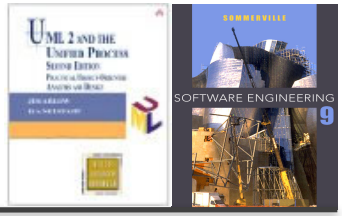


The Repository architecture pattern

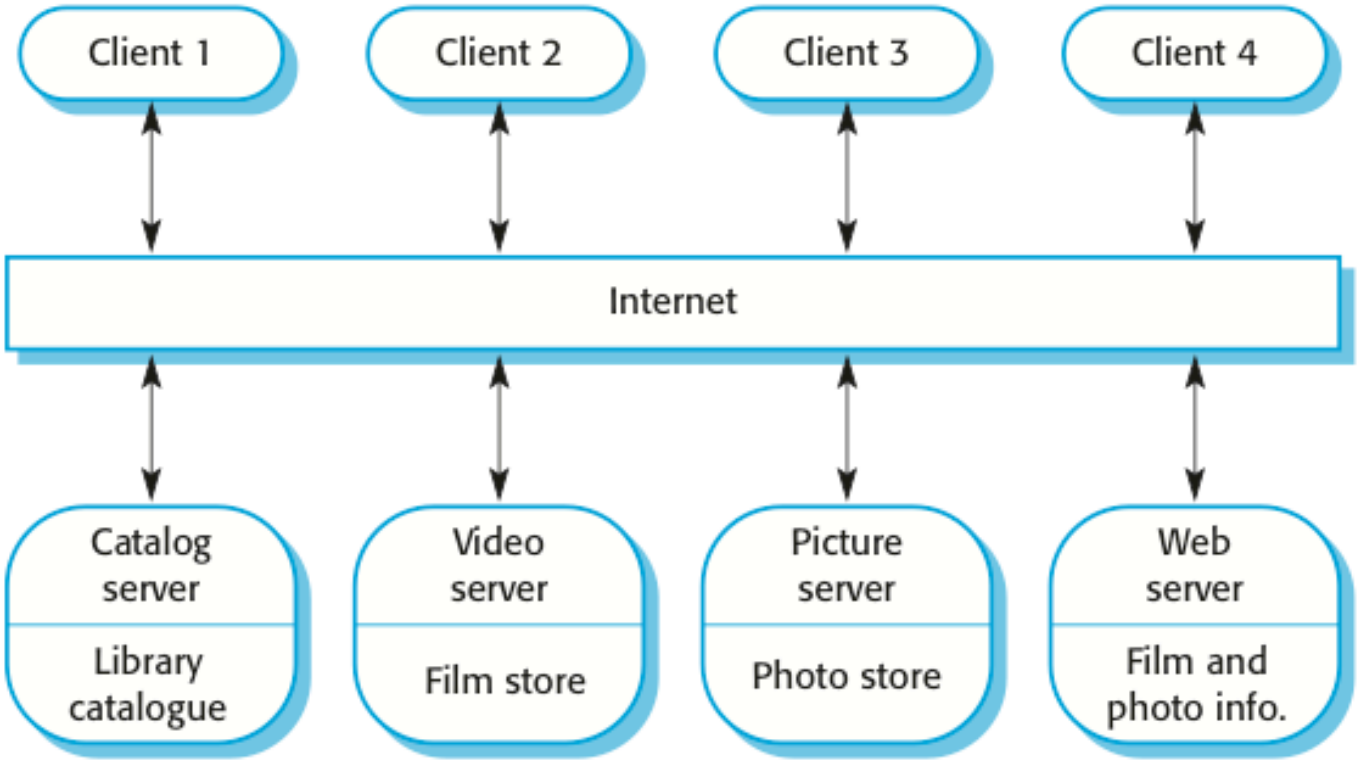


Name	Repository
Description	All data in a system is managed in a **central repository** that is accessible to all system components. Components **do not interact directly,** only through the repository.
Example	Figure on the previous slide is an example of an IDE where the components use a repository 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.

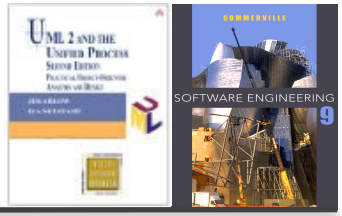
The Client-server architecture pattern



- Distribution of data and processing across stand-alone service-providing servers and clients calling the services.

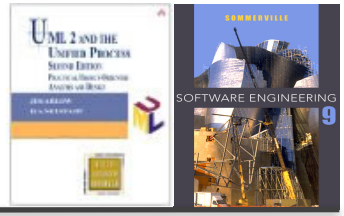


The Client–server pattern

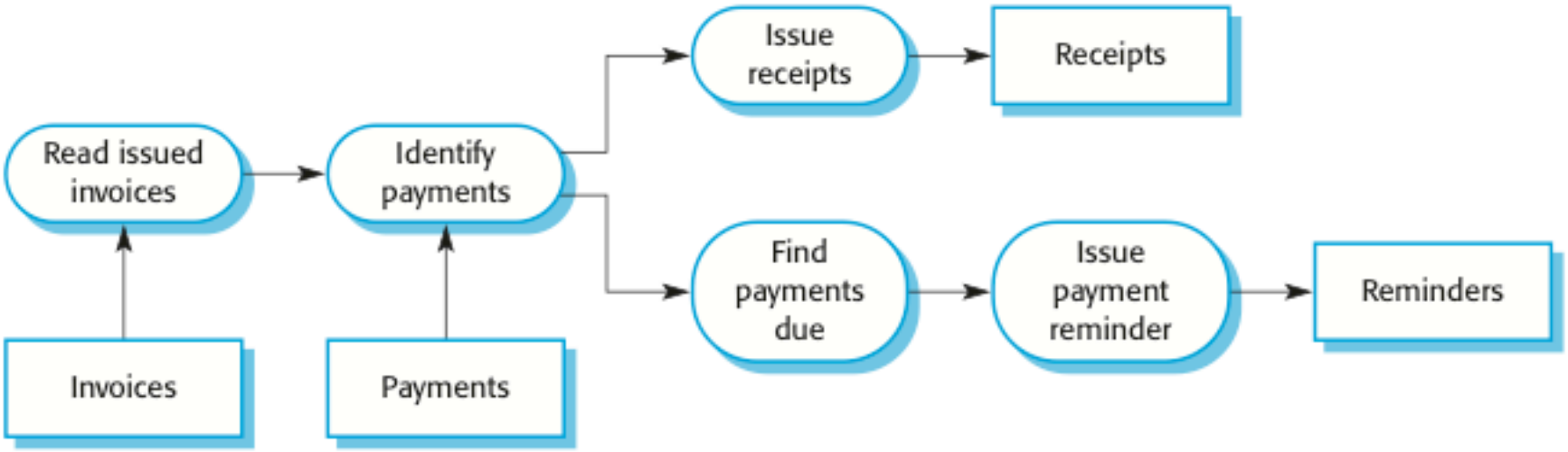


Name	Client-server
Description	In a client–server architecture, the functionality of the system is **organized into services**, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure on the previous slide is an example 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**.

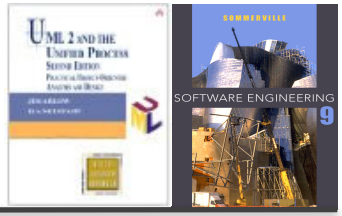
The Pipe and filter architecture pattern



- Functional transformations process their inputs to produce outputs.
- Variants of this approach are very common. When transformations are sequential, this is known as batch sequential model used in data processing systems.

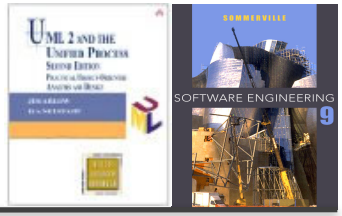


The Pipe and filter pattern



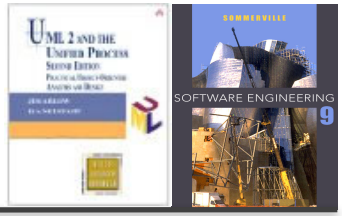
Name	Pipe and filter
Description	The processing of the data in a system is organized so that **each processing component (filter)** is discrete and carries out **one type of data transformation**. The **data flows (as in a pipe)** from one component to another for processing.
Example	Figure on the previous slide 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



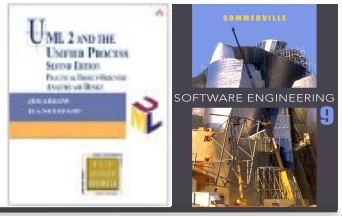
- Application systems are designed to meet an organisational need.
- As **businesses have much in common**, their application systems also tend to have a common architecture that reflects the application requirements.
- A **generic application architecture** is an architecture for a type of software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

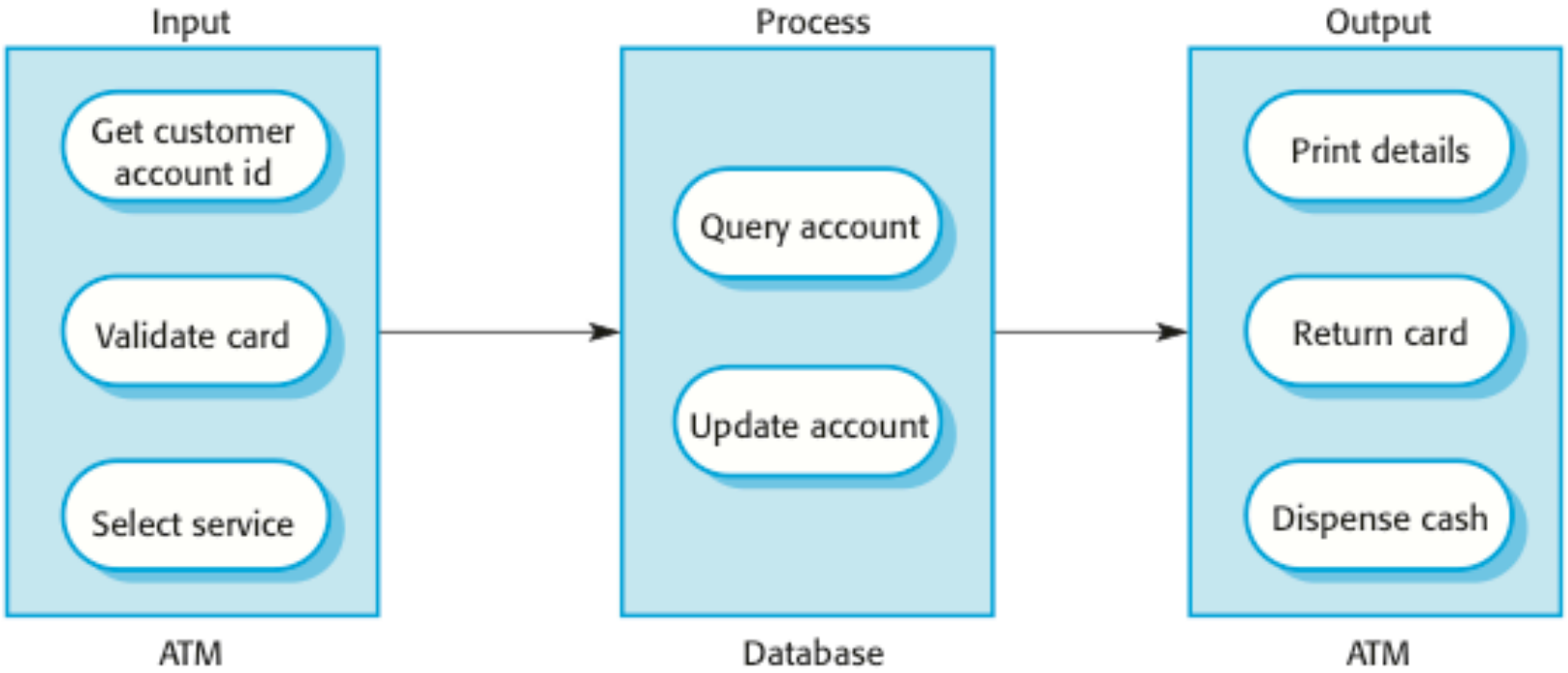
Examples of application types



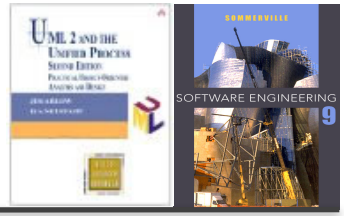
- ✧ Data processing applications
  - Data driven applications that process data in batches without explicit user intervention during the processing.
- ✧ Transaction processing applications
  - Data-centred applications that process user requests and update information in a system database.
- ✧ Event 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.

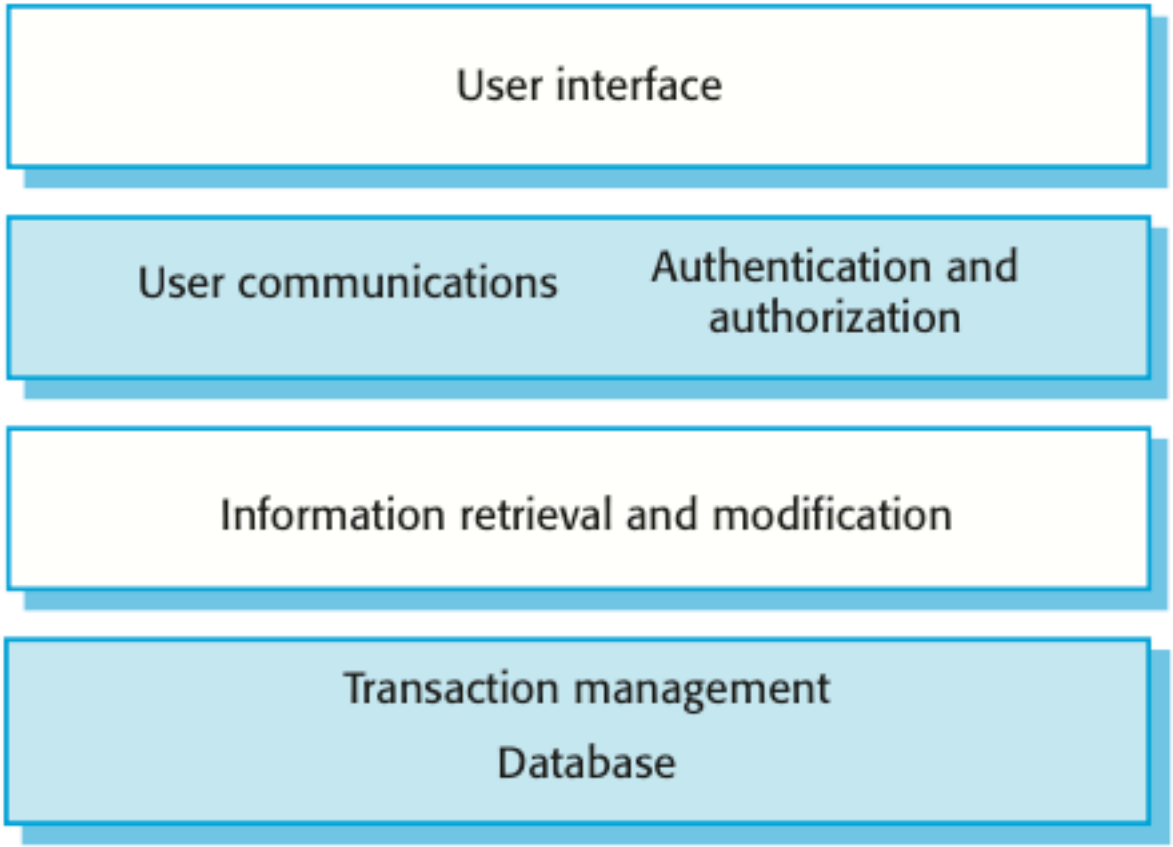
The architecture of a transaction processing application (ATM system) – a process view



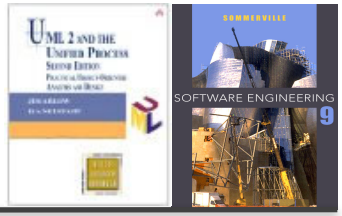


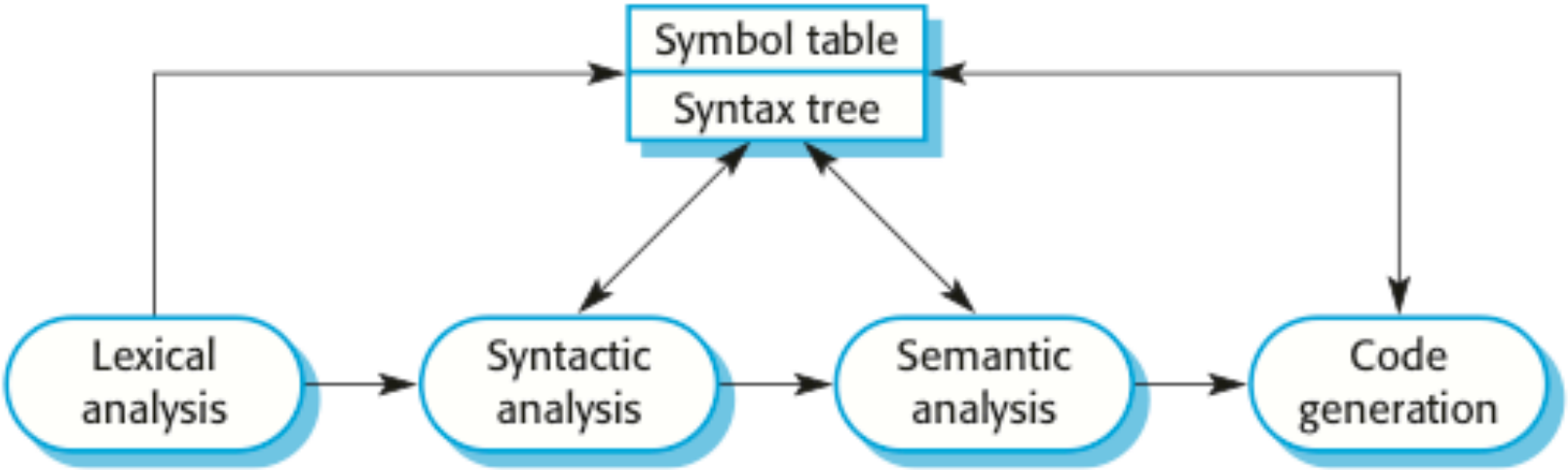
Layered architecture of a transaction processing application (Information system)



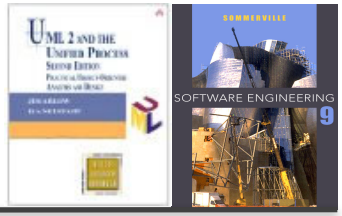


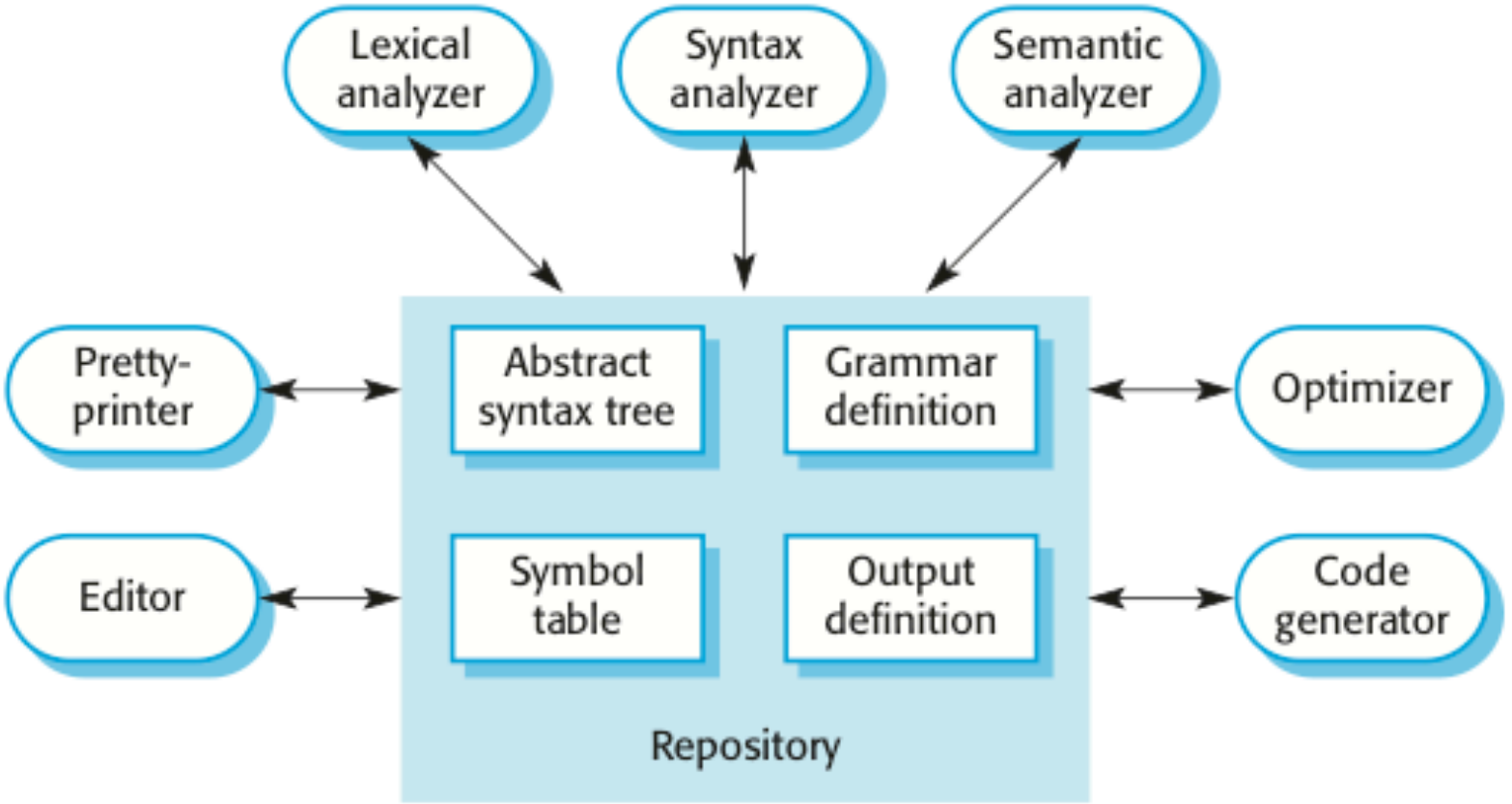
Pipe and filter architecture of a language processing system (Compiler)



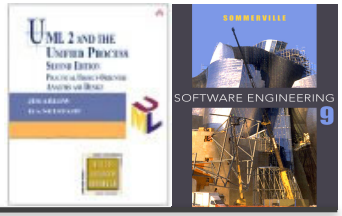


Repository architecture of a language processing system

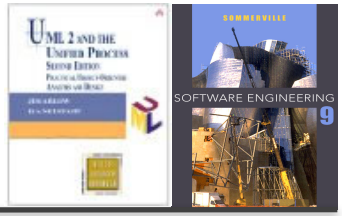




Key points



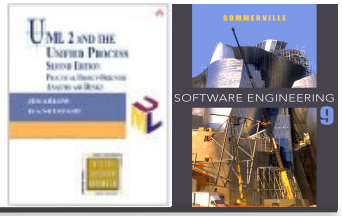
- A **software architecture** is a description of how a software system is organized.
- Architectural **design decisions** include decisions on the type of application, the distribution of the system, the architectural styles to be used.
- **Architectural patterns** are a means of reusing knowledge about 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.



UML Packages (Analysis)

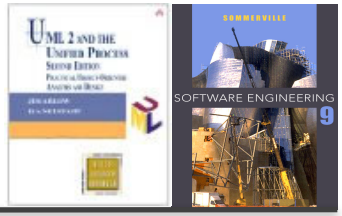
Lecture 8/Part 2

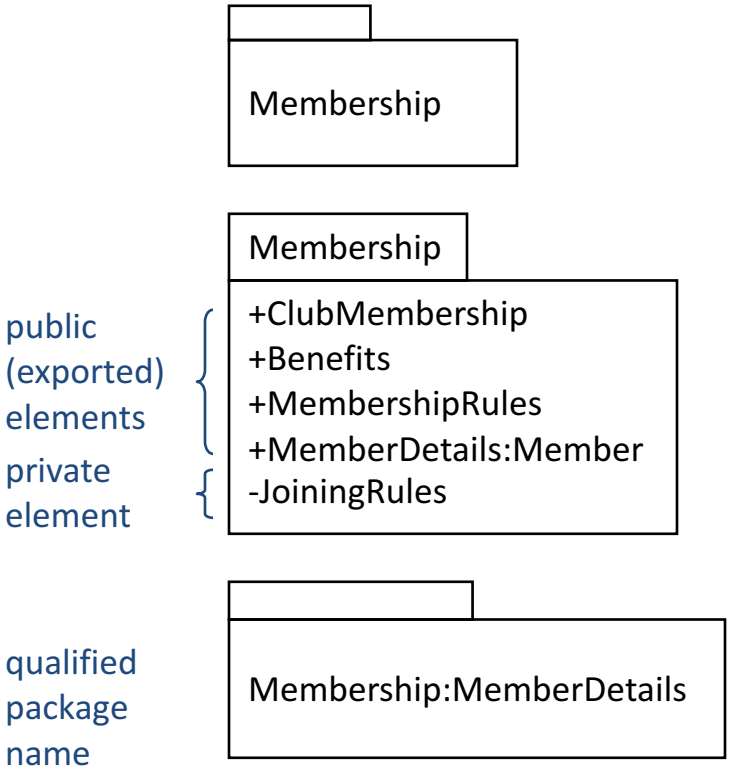
Packages

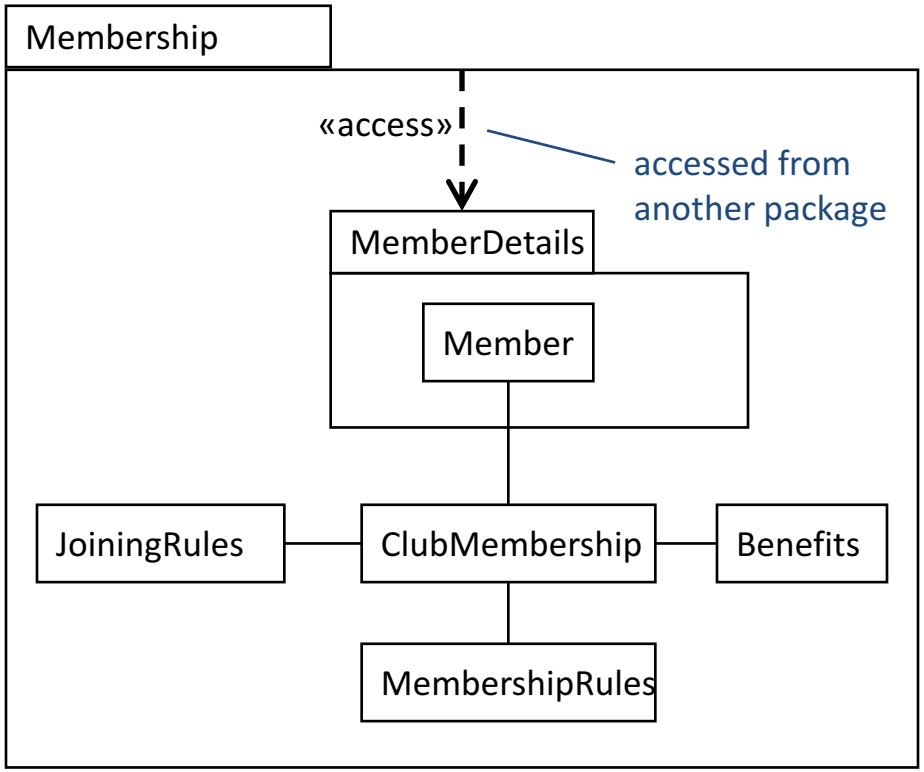


- A package is a general purpose mechanism for **organising model elements** into groups
  - Group semantically related elements
  - Define a “semantic boundary” in the model
  - Provide units for parallel working and configuration management
- In UML 2 a package is a purely **logical** grouping mechanism
  - Use components for **physical** grouping
- Analysis packages contain:
  - Analysis classes, analysis packages, use cases, use case realizations.

Package syntax

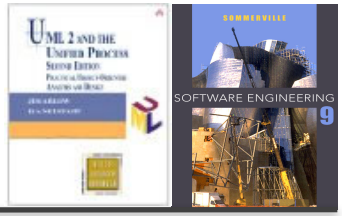




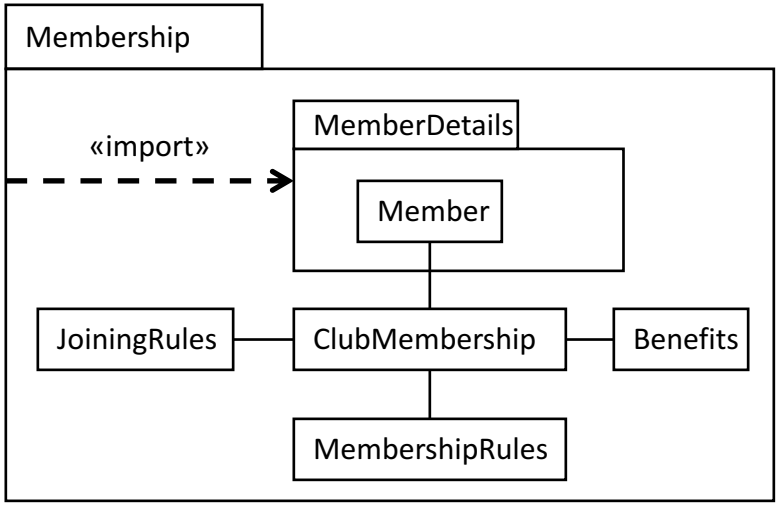


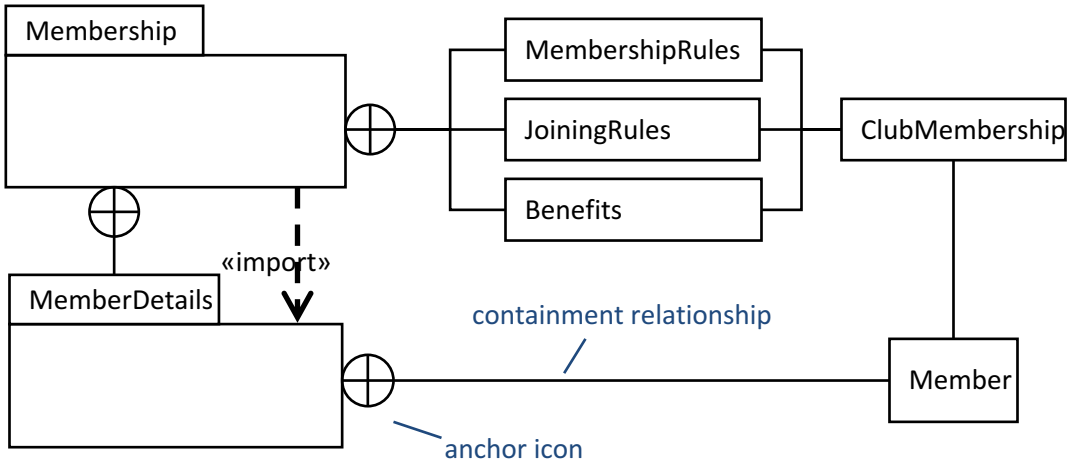
- Use stereotypes to distinguish different package purposes.

Nested packages

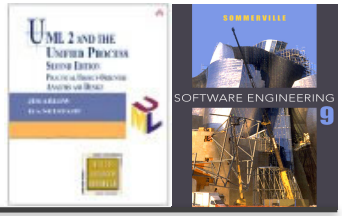


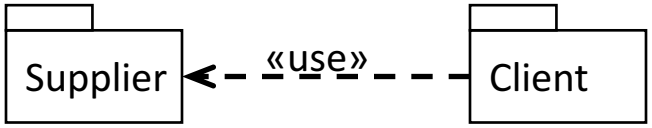
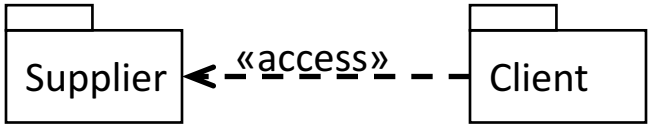
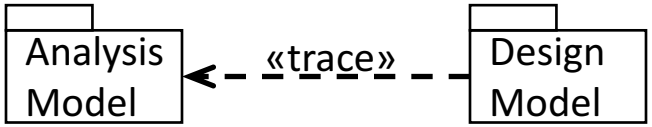
- Each package defines an encapsulated namespace i.e. all names must be unique within the package
- If an element is visible within a package then it is visible within all nested packages
  - e.g. Benefits is visible within MemberDetails
- Show containment using nesting or the containment relationship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

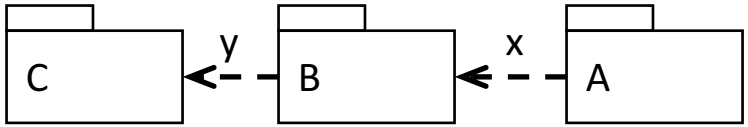




Package dependencies

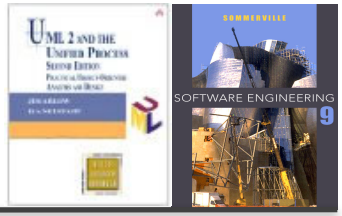


dependency	semantics
	An element in the client uses an element in the supplier in some way. The client depends on the supplier. **Transitive**.
	Public elements of the supplier namespace are added as public elements to the client namespace. **Transitive**.
	Public elements of the supplier namespace are added as private elements to the client namespace. **Not transitive**.
	«trace» usually represents a historical development of one element into another more refined version. It is an extra-model relationship. **Transitive**.

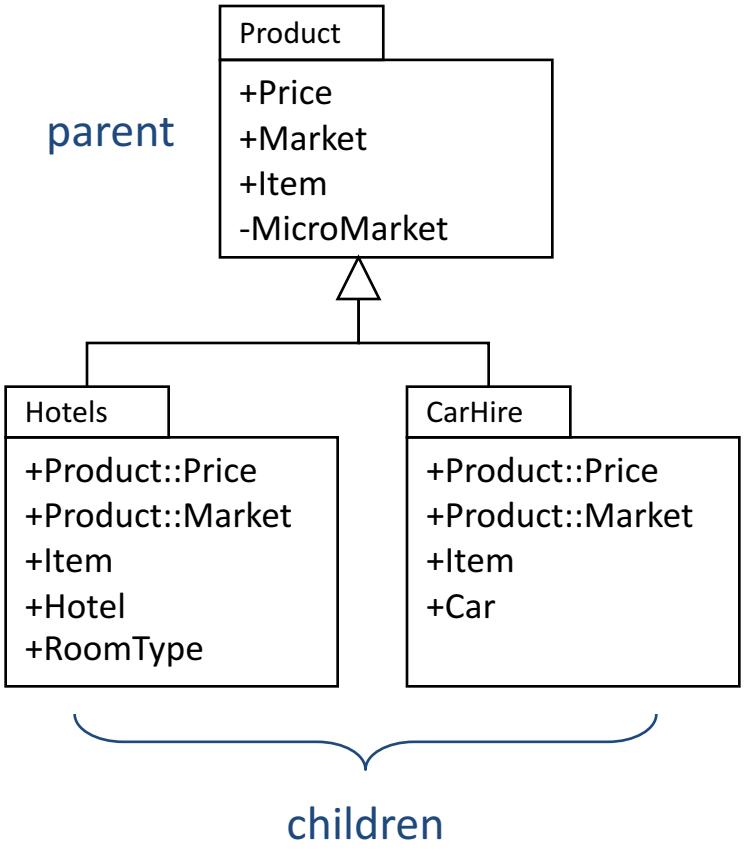


transitivity - if dependencies x and y are transitive, there is an implicit dependency between A and C

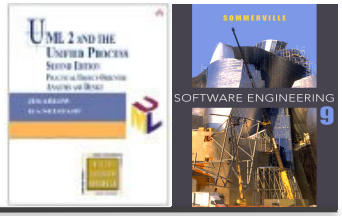
Package generalisation

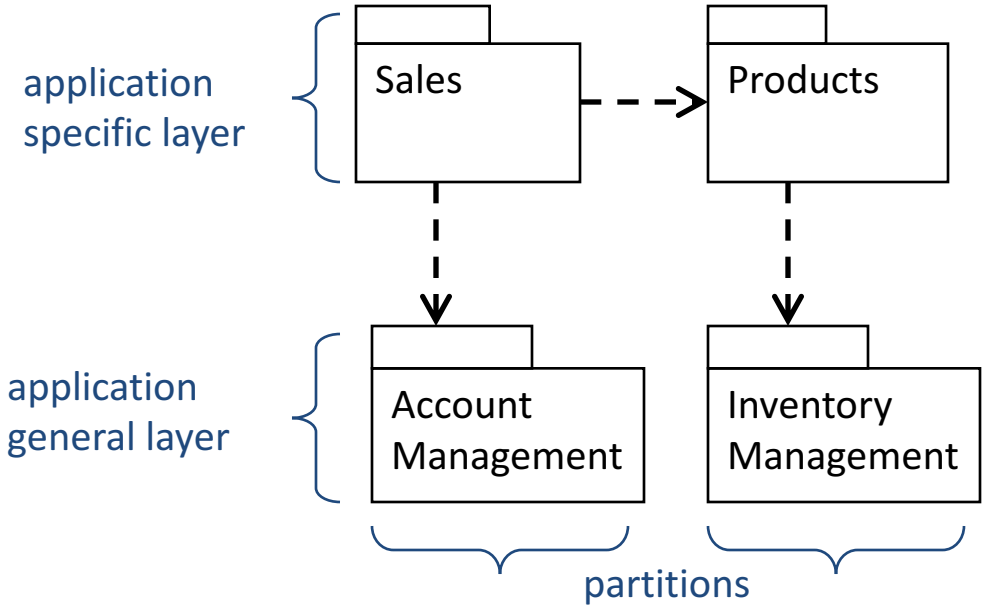


- The more specialised child packages **inherit** the public and protected elements in their parent package
- Child packages may **override** elements in the parent package. Both *Hotels* and *CarHire* packages override *Product::Item*
- Child packages may **add new elements**. *Hotels* adds *Hotel*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



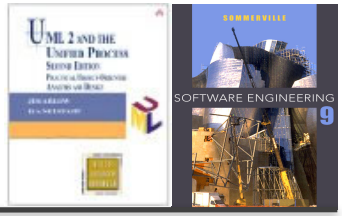
Architectural analysis



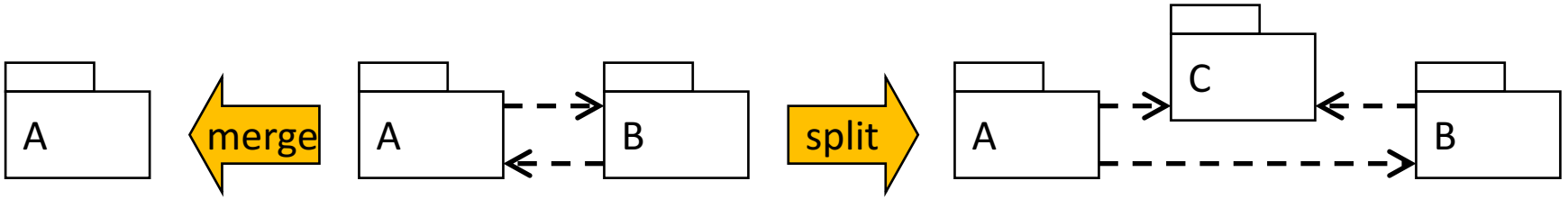


- This involves organising the analysis classes into a set of cohesive packages
  - The architecture should be partitioned to **separate concerns**, such as to specific and application general layers
  - Coupling between packages should be minimised

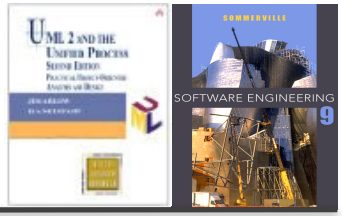
Finding analysis packages



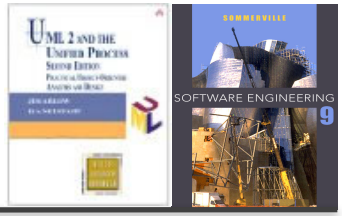
- A cohesive group of closely related classes or a class hierarchy
- 4 to 10 classes per package
- Minimise dependencies between packages
- Localise business processes in packages where possible
- Minimise nesting of packages
- Don't worry about dependency stereotypes and package generalisation
- Refine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



Key points



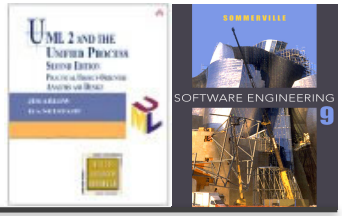
- Packages are the UML way of grouping modeling elements
- There are dependency and generalisation relationships between packages
- The package structure of the analysis model defines the logical system architecture

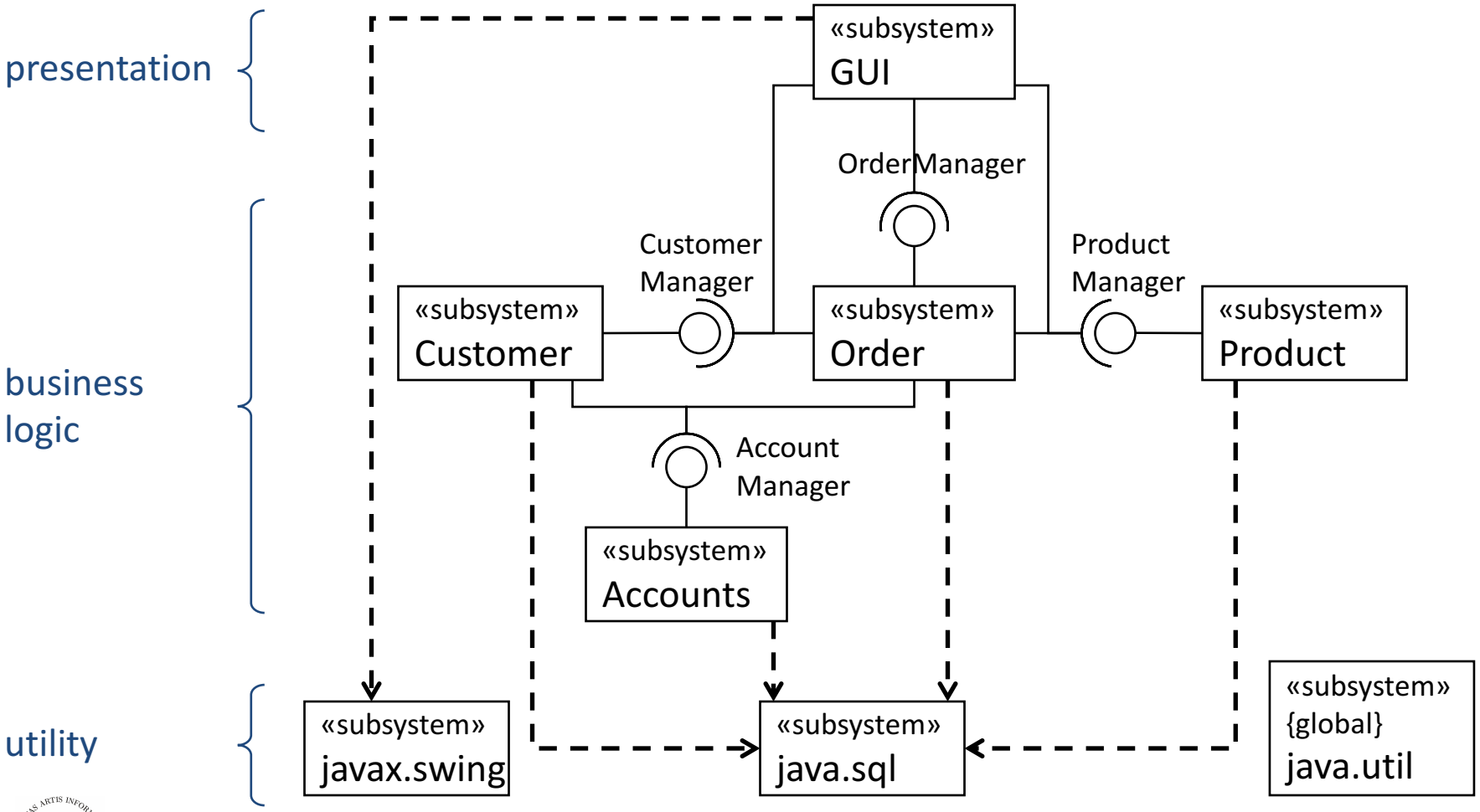


UML Component Diagram (Design)

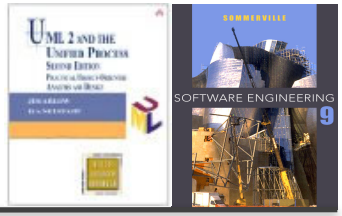
Lecture 8/Part 3

Example of a (layered) architecture



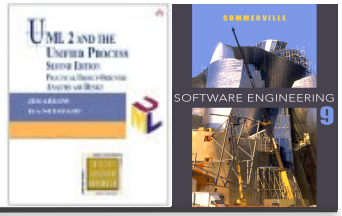


What is a component?



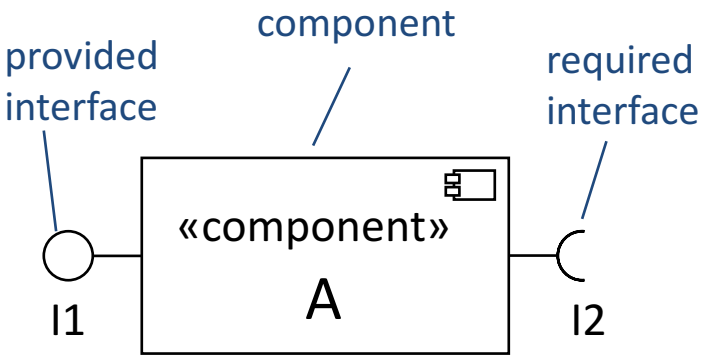
- The UML 2.0 specification states that, "A component represents a modular part of a system that **encapsulates its contents** and whose manifestation is **replaceable** within its environment"
  - A **black-box** whose external behaviour is completely defined by its **provided and required interfaces**
  - 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

Component syntax

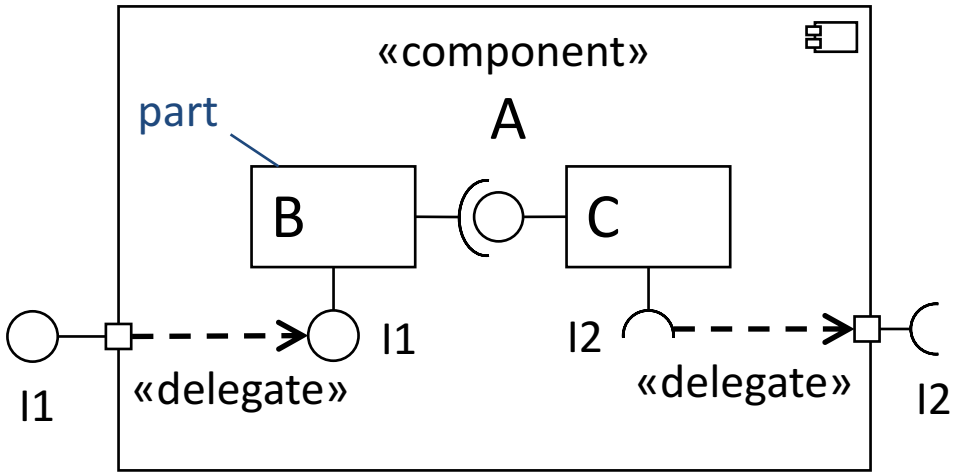


- Components may have provided and required interfaces, ports, internal structure
  - Provided and required interfaces usually delegate to internal parts
  - You can show the parts nested inside the component icon or externally, connected to it by dependency relationships

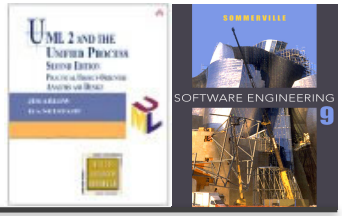
black box notation



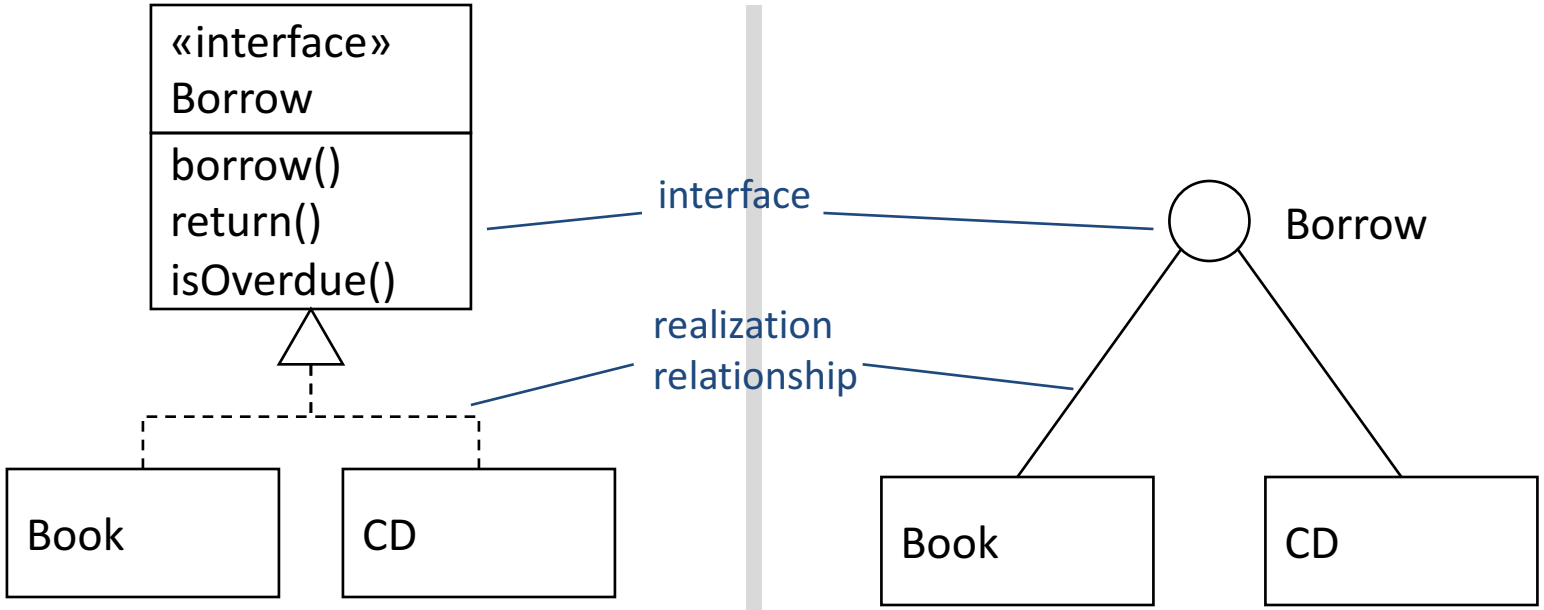
white box notation



Provided interface syntax



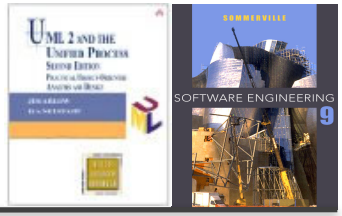
- A provided interface indicates that a classifier implements the services defined in an interface



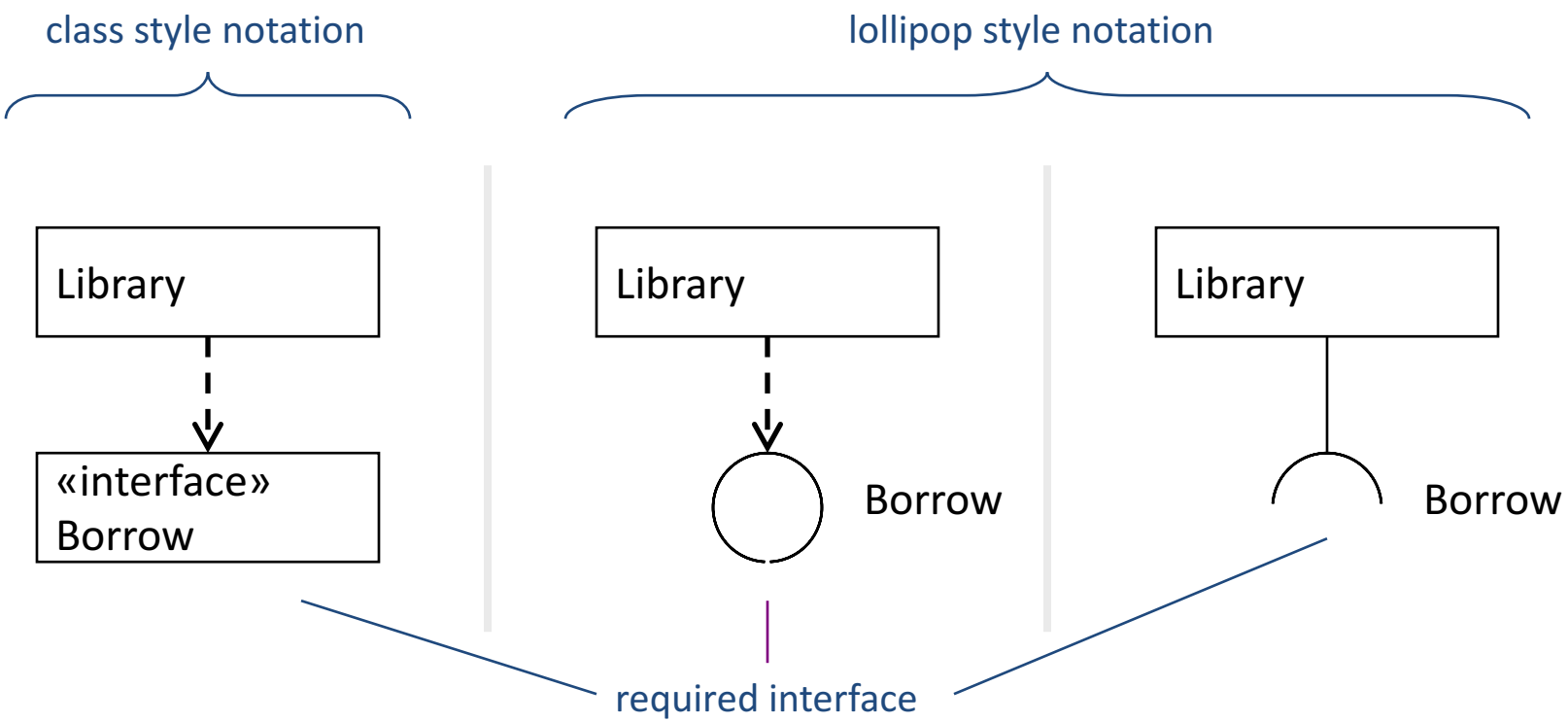
"Class" style notation

"Lollipop" style notation
(note: you can't show interface operations or attributes with this notation)

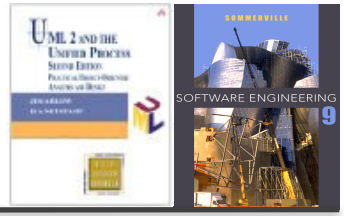
Required interface syntax



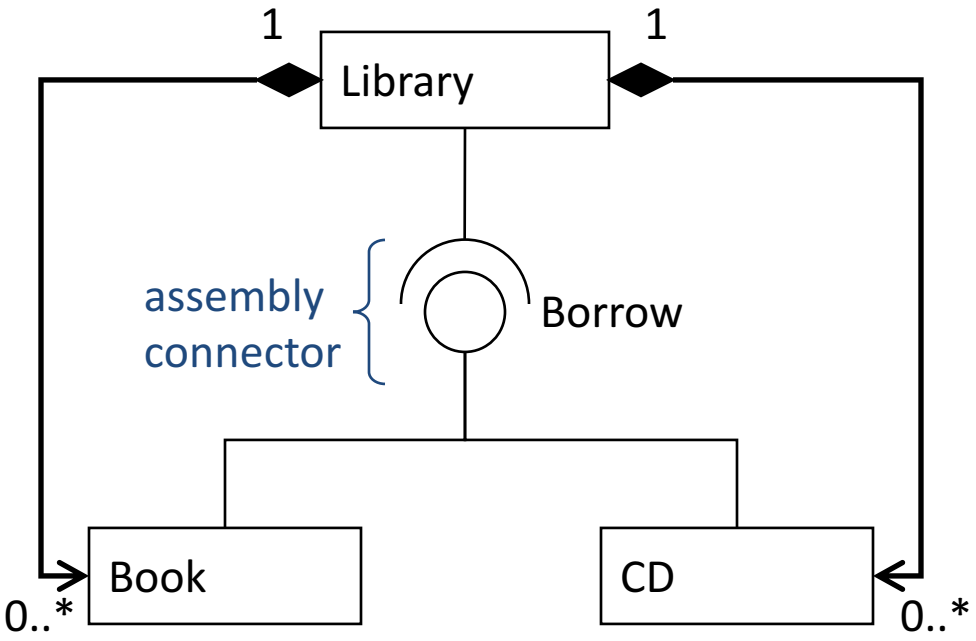
- A required interface indicates that a classifier uses the services defined by the interface



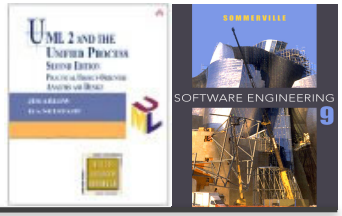
Assembly connectors



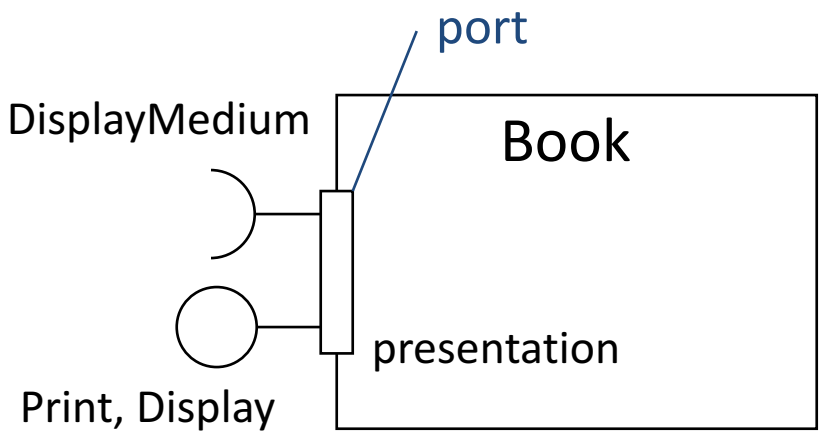
- You can connect provided and required interfaces using an assembly connector

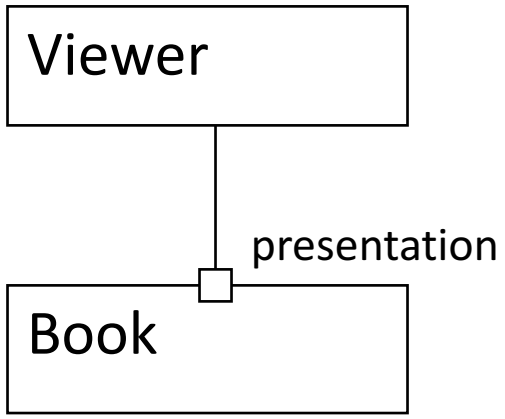


Ports for organizing interfaces

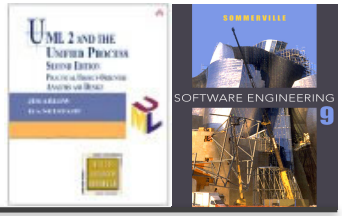


- A port specifies an interaction point between a classifier and its environment
- A port may have a name and is typed by its provided and required interfaces:
  - It is a semantically cohesive set of provided and required interfaces



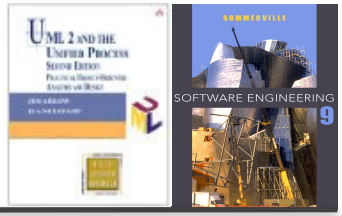


Using interfaces

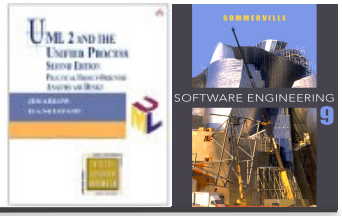


- Advantages:
  - When we design with classes, we are designing to specific implementations
  - When we design with interfaces, we are instead designing to contracts which may be realised by many different implementations (classes)
  - Designing to contracts frees our model from implementation dependencies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

Key points



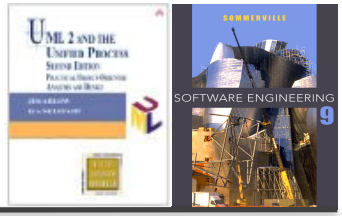
- ✧ Interfaces specify a named set of public features:
  - They define a contract that classes and subsystems may realise
  - Programming to interfaces rather than to classes reduces dependencies between the classes and subsystems in our model
  - Programming to interfaces increases flexibility and extensibility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



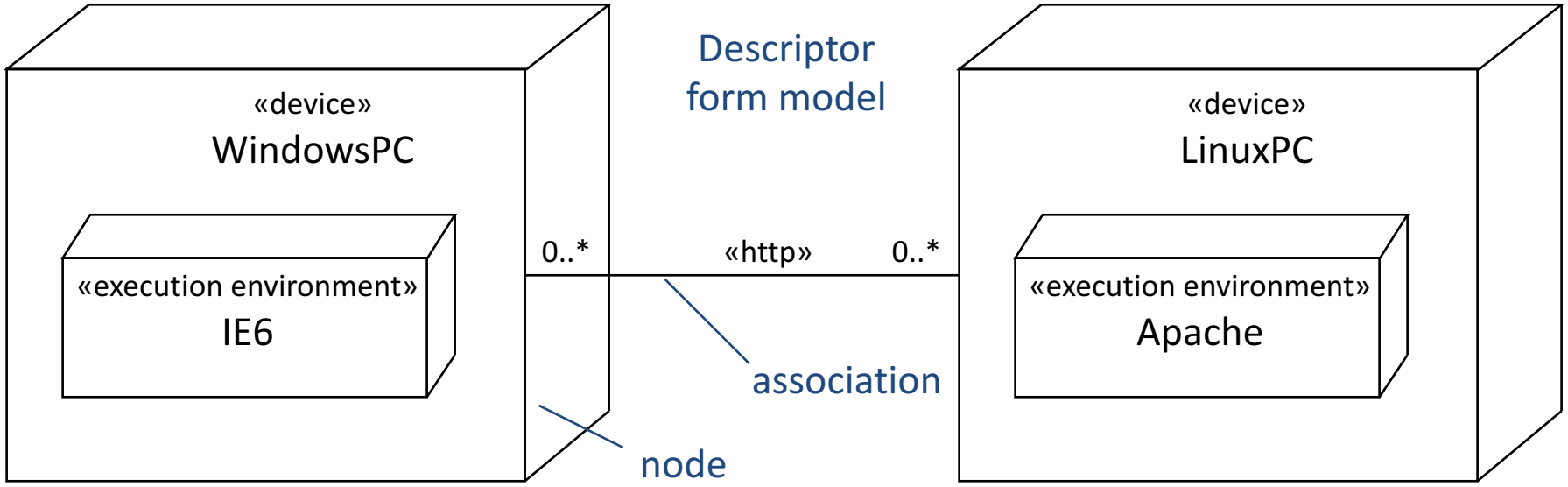
UML Deployment Diagram (Realisation)

Lecture 8/Part 4

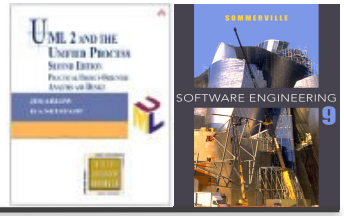
Deployment model



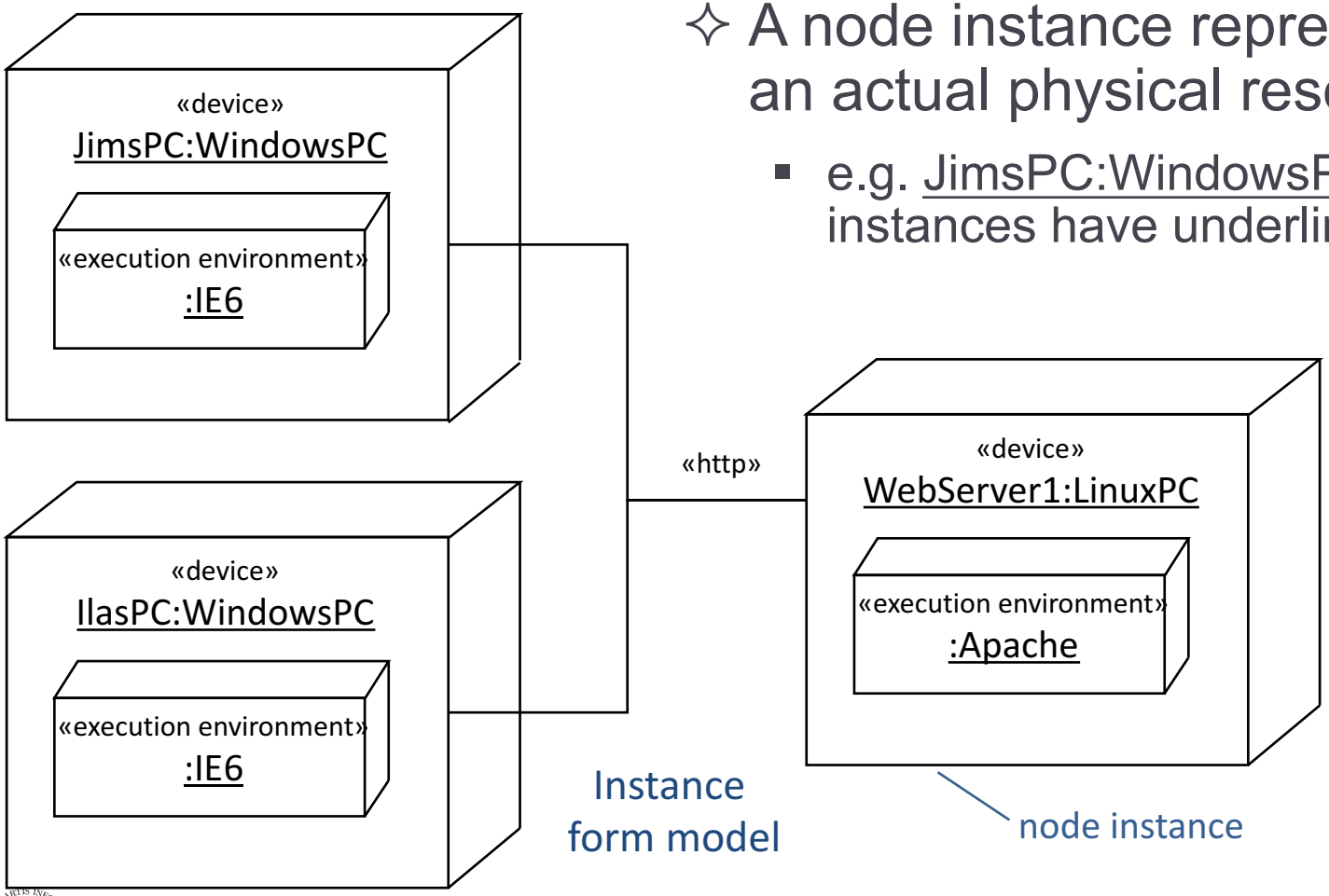
- The deployment model models **system's physical architecture** and the **mapping of the software architecture** to the physical nodes
  - Each node is a type of computational resource
  - Nodes have relationships that represent methods of communication
  - Artifacts represent physical software e.g. a JAR file or .exe file



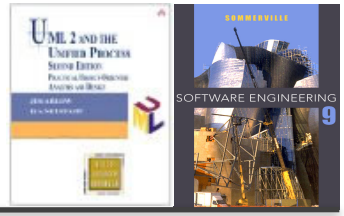
Instance form model



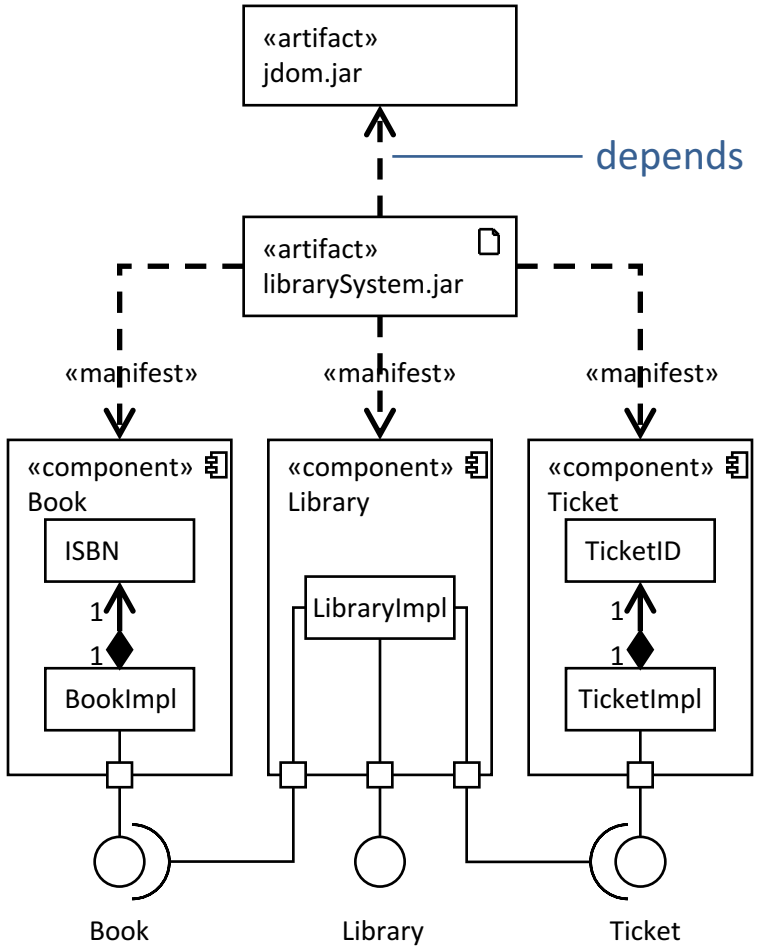
- A node instance represents an actual physical resource
  - e.g. JimsPC:WindowsPC - node instances have underlined names



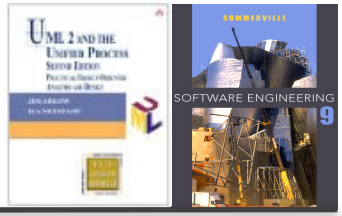
Artifacts and components



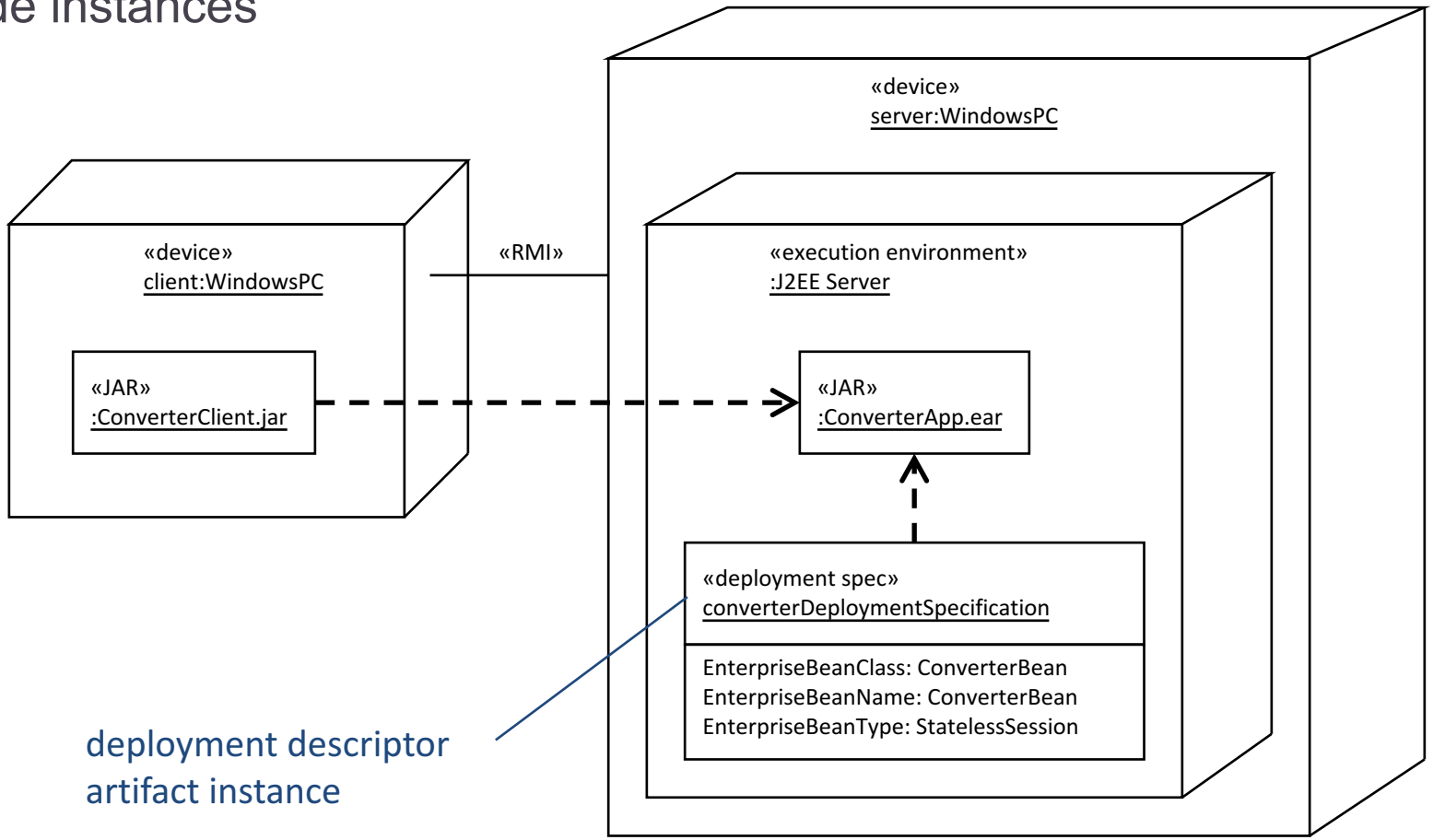
- Artifacts and components represent the software deployed on physical nodes
- An artifacts represents a concrete deployed real-world thing, such as a file
  - Artifacts = Physical level
- Artifacts provide the physical manifestation for one or more components
  - Components = Logical level



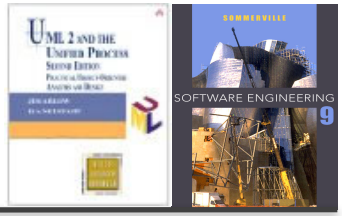
Example



- 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 artifacts 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 artifact instances is distributed across node instances
  - Node instances represent actual physical hardware or execution environments