Module 9

Component Diagrams

Vasco Amaral
vma@fct.unl.pt

Before discussing components, let us talk about interfaces

What is an interface?

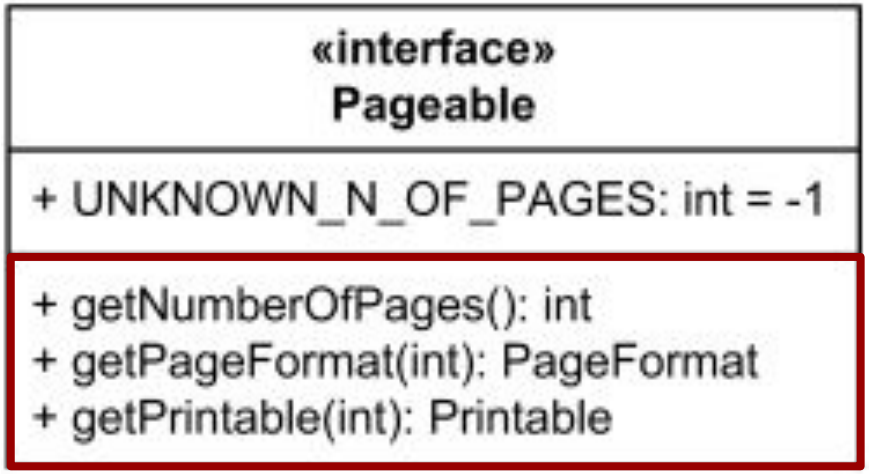
- Specifies a named set of public features
- Separates the specification of functionality from its implementation (by a class, or subsystem)
- Can't be instantiated, simply declaring a contract that may be realized by zero or more classifiers

Interfaces specify features

- Operations
- Attributes
- Associations
- Constraints
- Stereotypes
- Tagged values
- Protocols

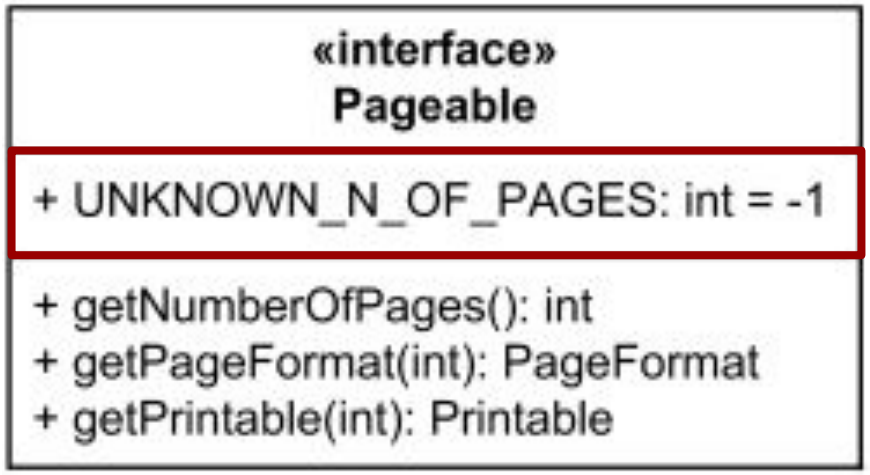
Interface operation is supported by a realizing classifier, which:

- Must have an operation with the same signature and semantics as the interface



Interface attribute is supported by a realizing classifier, which:

- Must have public operations to get and set the value of the attribute
  - The realizing classifier is not required to actually have the attribute specified by the interface, as long as it behaves as though it has it



Interface association is supported by a realizing classifier, which:

- Must have an association target classifier
  - If an interface specifies an association to another interface, the implementing classifiers of these interfaces must have an association between them

Interface constraint is supported by a realizing classifier, which:

- Must support the constraint

Interface stereotype is supported by a realizing classifier, which:

- Has the stereotype

Interface tagged value is supported by a realizing classifier, which:

- Has the tagged value

Interface protocol is supported by a realizing classifier, which:

- Must realize the protocol
  - This is defined as a protocol state machine

Interfaces need specifications of their features' semantics to guide implementers

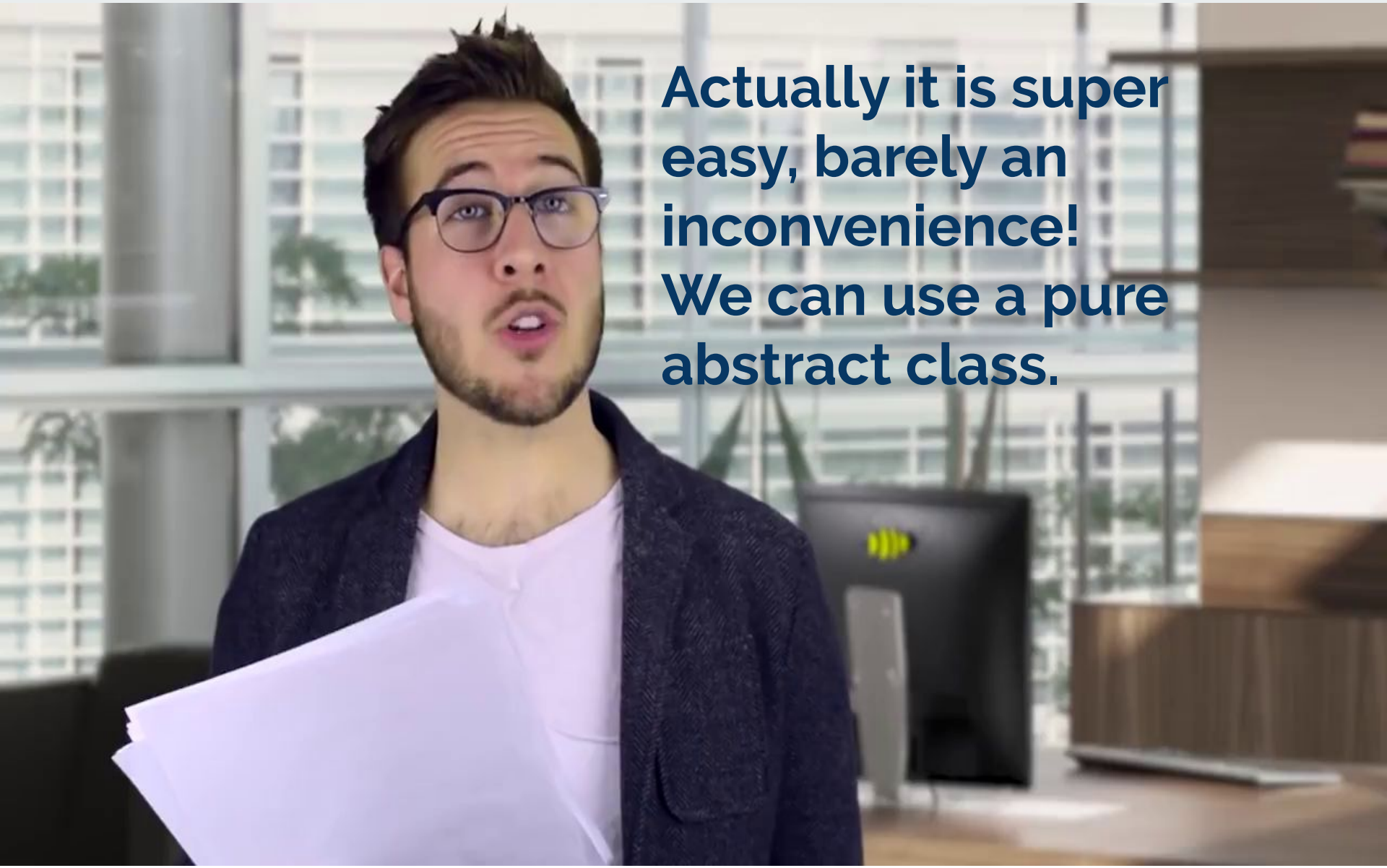
- Operations should include
  - The complete operation signature
    - Name, types of all parameters, return type
  - The semantics of the operation
    - Textual description, or pseudocode
- Attributes should include
  - Name and type
- Operations and attributes stereotypes, constraints and tagged values

Interfaces define a contract

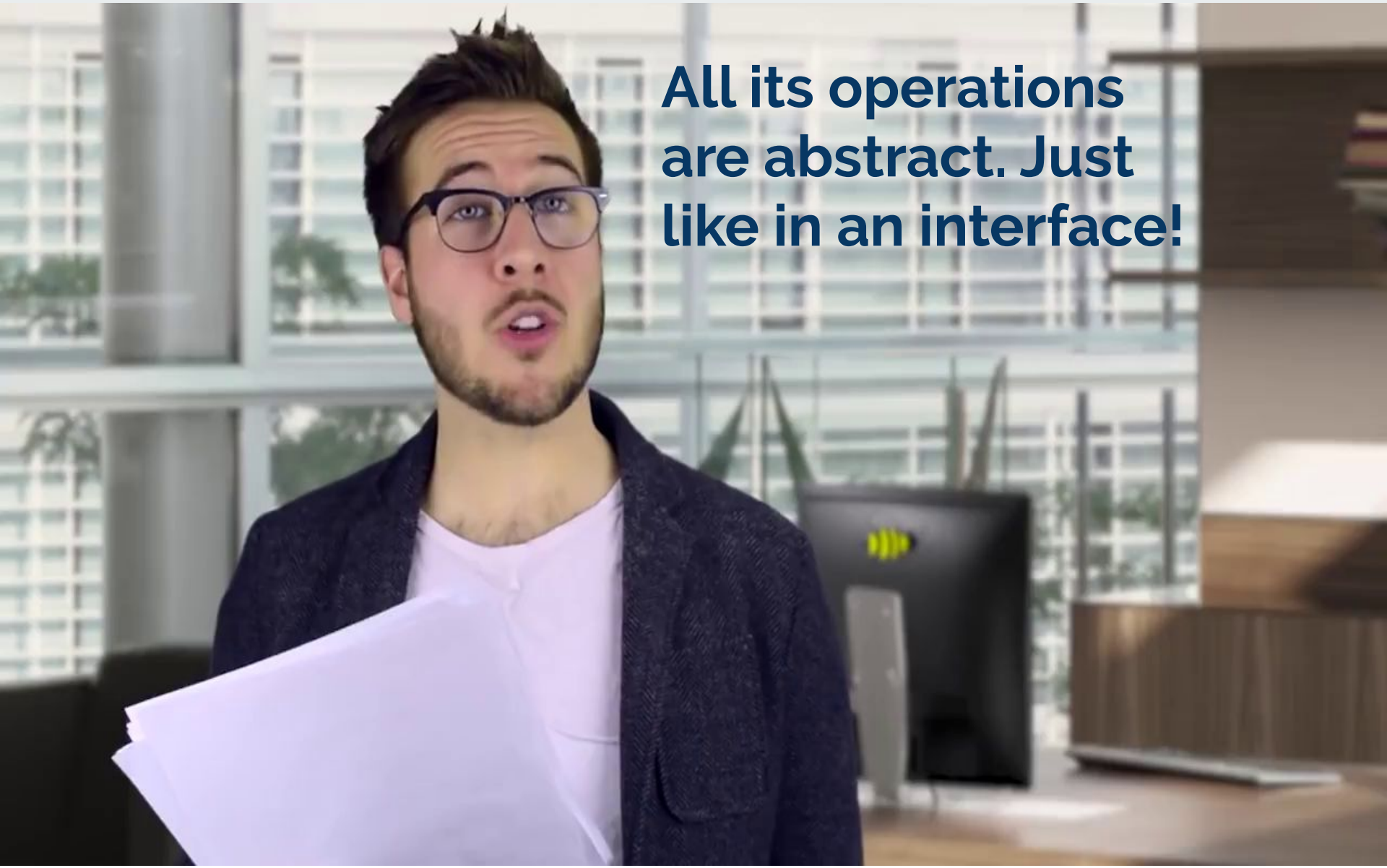
- So far, we have been **designing to an implementation**
- We can also **design to a contract** for **added flexibility**
  - This is what you have been doing since the Object-Oriented Programming course!
  - An interface can be realized (implemented) by several different classes

- The interface defines a service offered by a class, subsystem, or component


What if the implementation language does not support interfaces, like C++?
I guess then you have a big problem to solve, right?


Actually it is super easy, barely an inconvenience! We can use a pure abstract class.

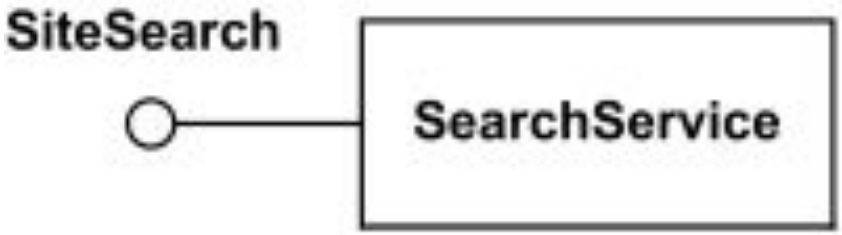

How does that work?


All its operations are abstract. Just like in an interface!


Abstract classes
are tight!

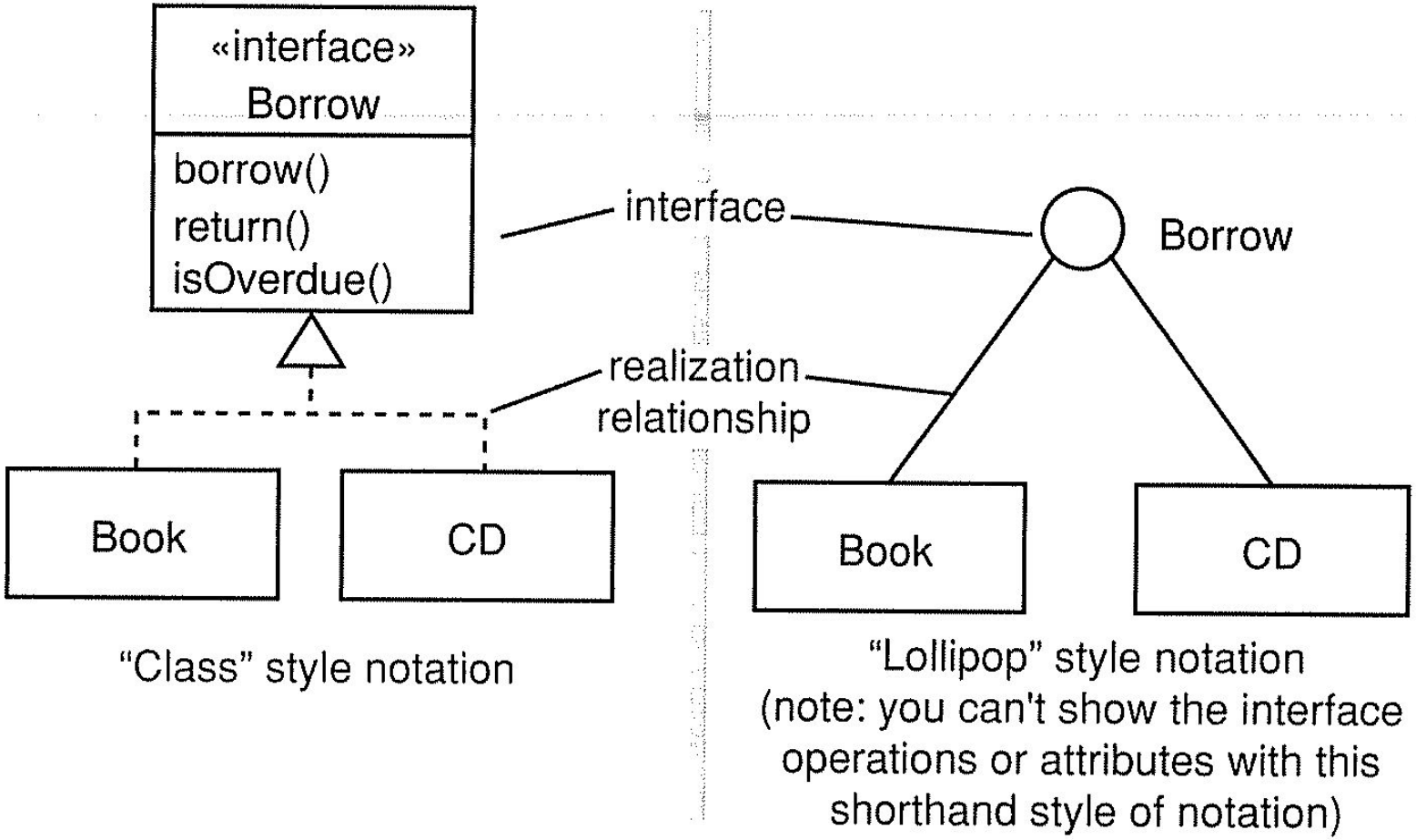
Provided interfaces are interfaces realized by a classifier

- Provided interfaces represent obligations of the instances of that classifier to their clients
- They describe services offered by those instances



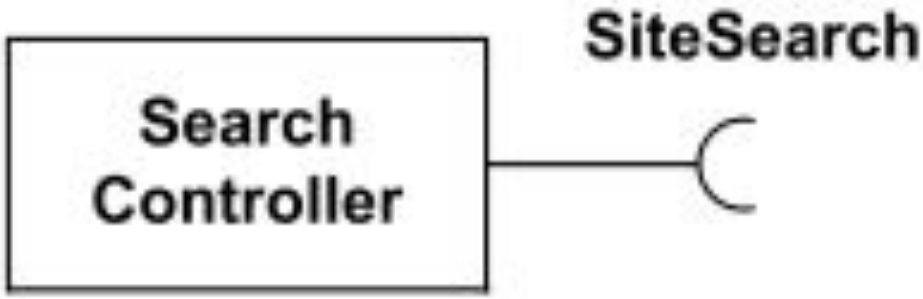
Interface SiteSearch is realized (implemented) by SearchService

Several presentation options for provided interfaces



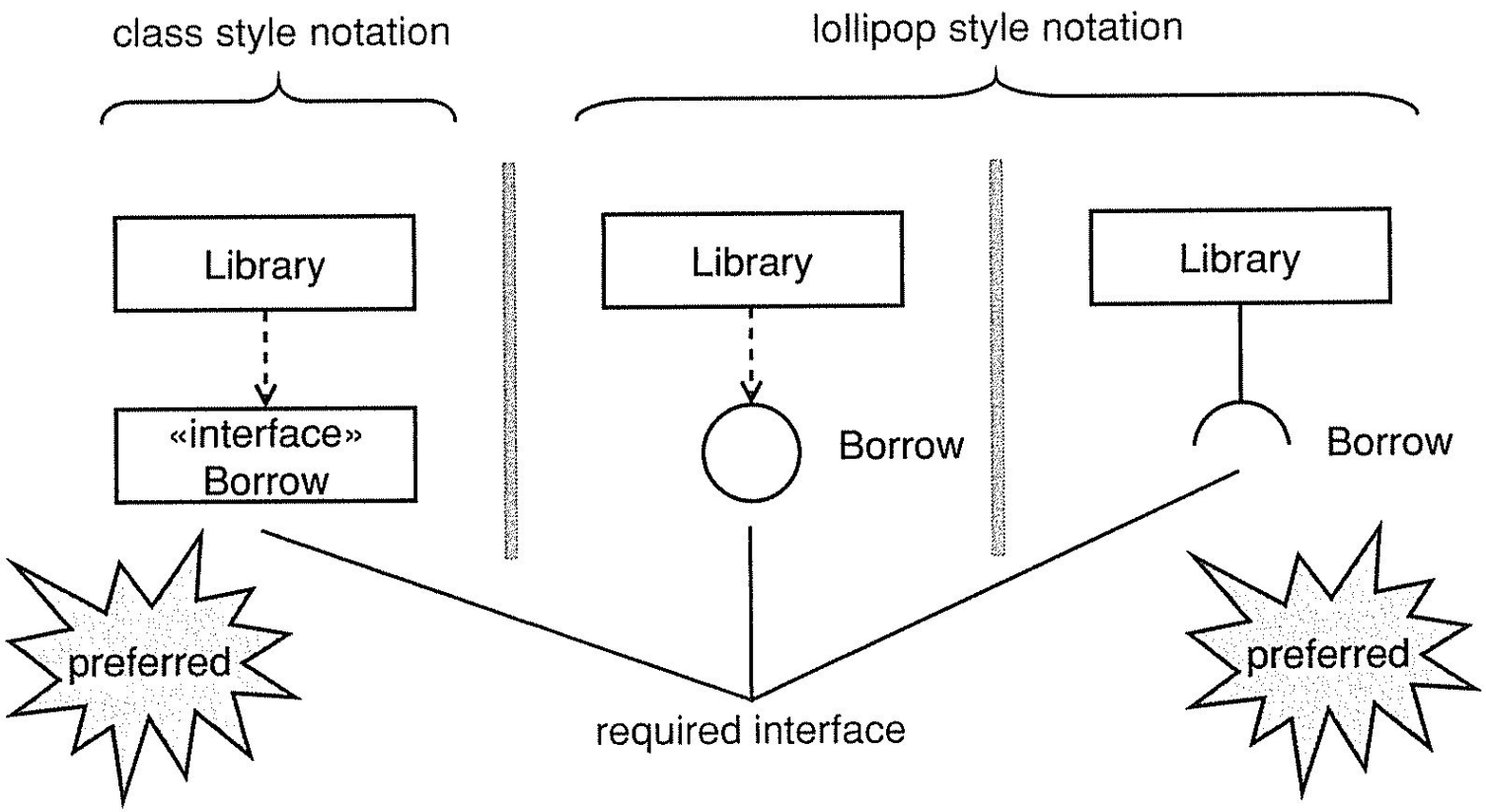
Required interfaces specify services that the classifier needs to perform its function

- Required interfaces are crucial so that the classifier may fulfill its obligations to its clients
- Required interfaces are specified by a usage dependency between the classifier and the corresponding interface

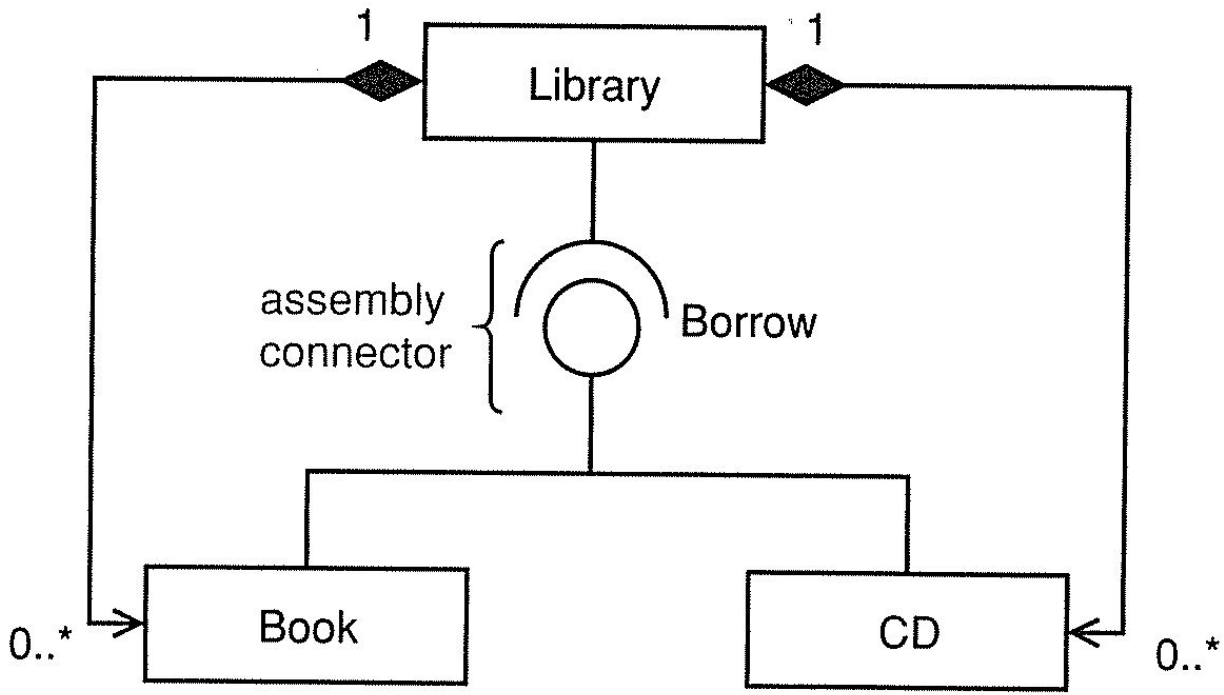


Interface SiteSearch is used (required) by SearchController

Several presentation options for required interfaces



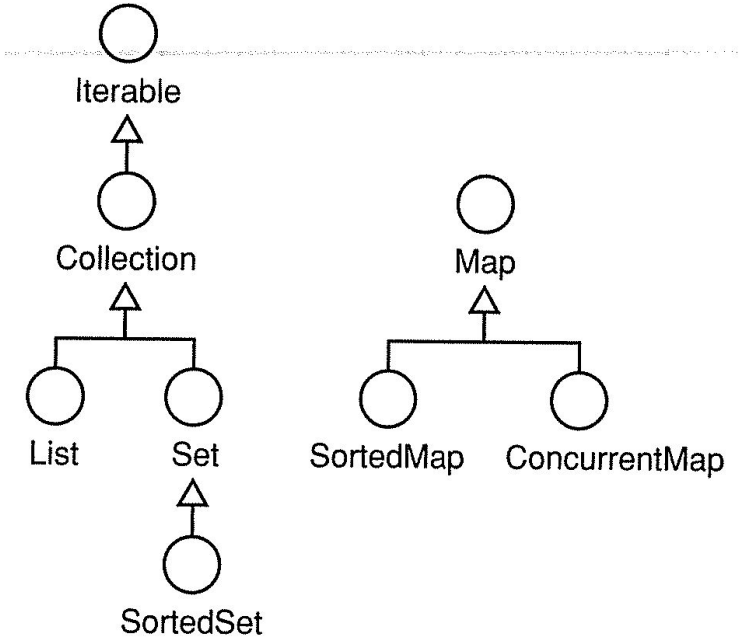
The assembly connector associates a required with a provided interface



Library uses the Borrow interface, which is provided both by Book and by CD

An example from the Java collections framework

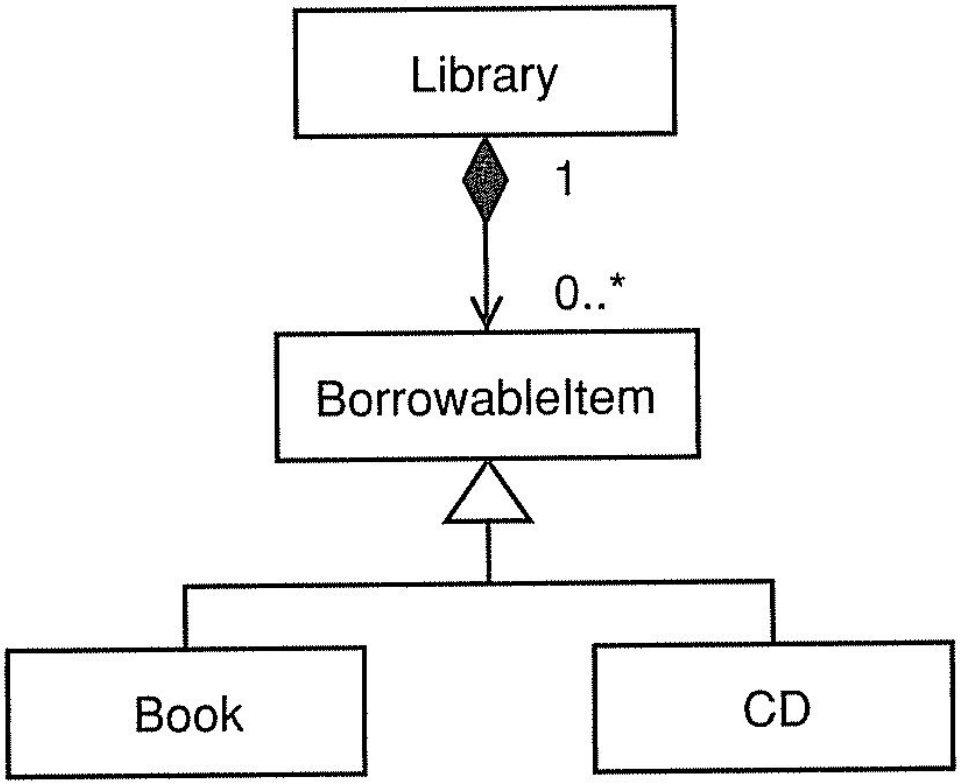
- Designing to an interface allows java developers to choose the implementation with the most adequate characteristics



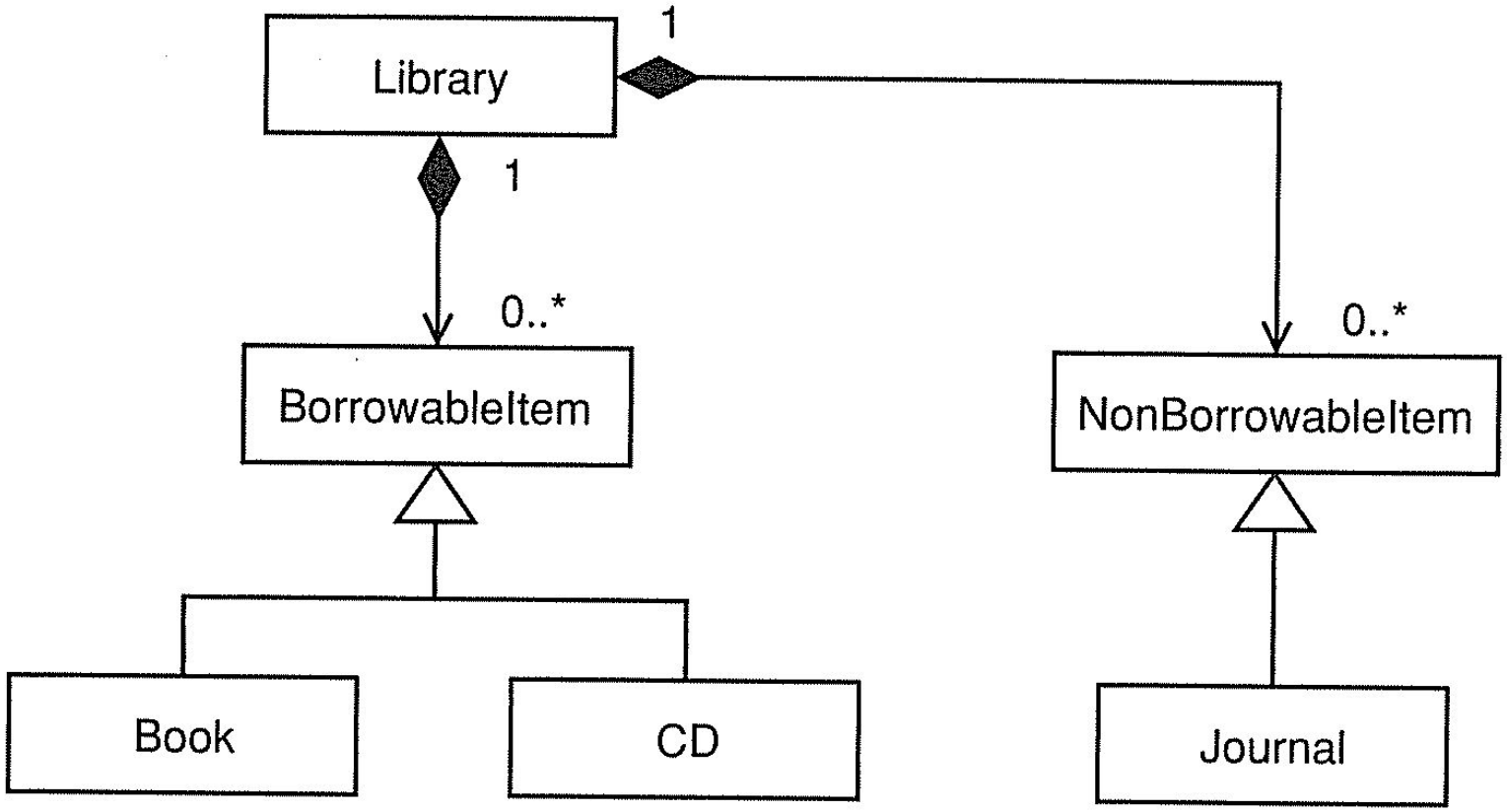
Interface realization *vs.* inheritance

- Interface realization
  - “realizes a contract specified by”
- Inheritance
  - “is a”

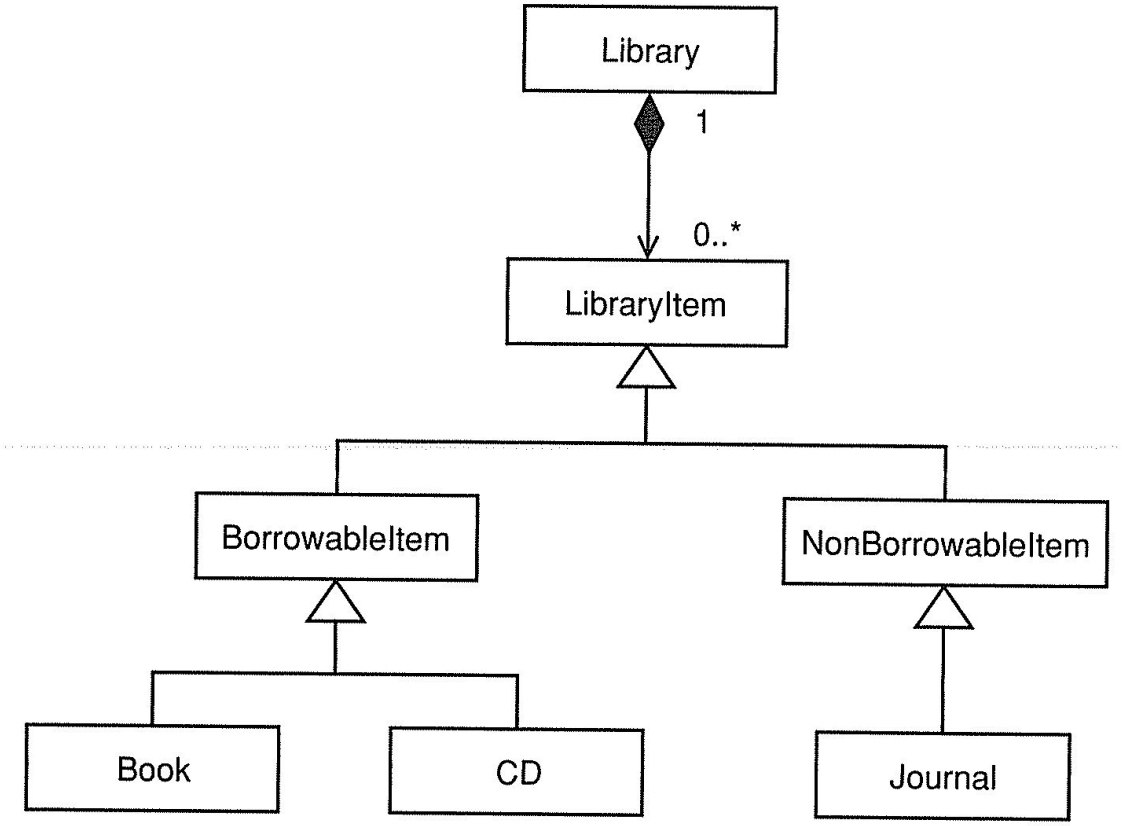
Consider the following example



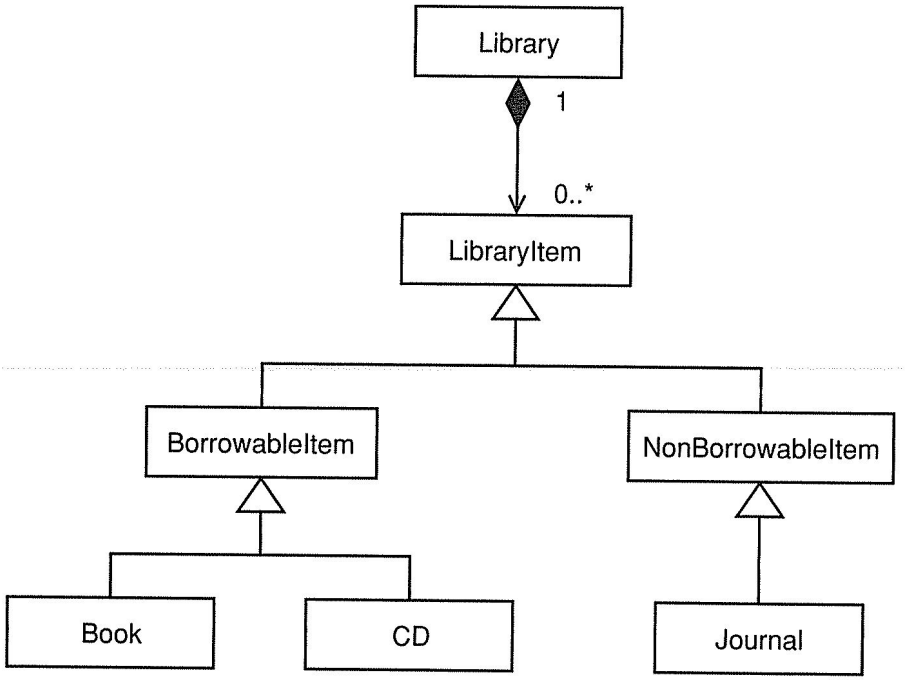
Now, suppose you also have items you don't want to borrow

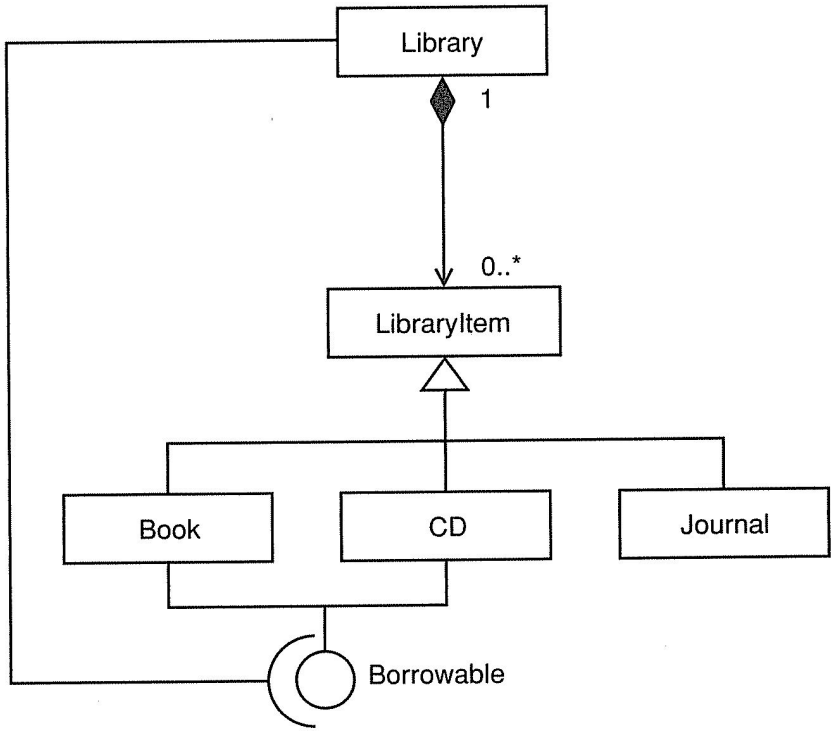


There are some commonalities between borrowable and non-borrowable items



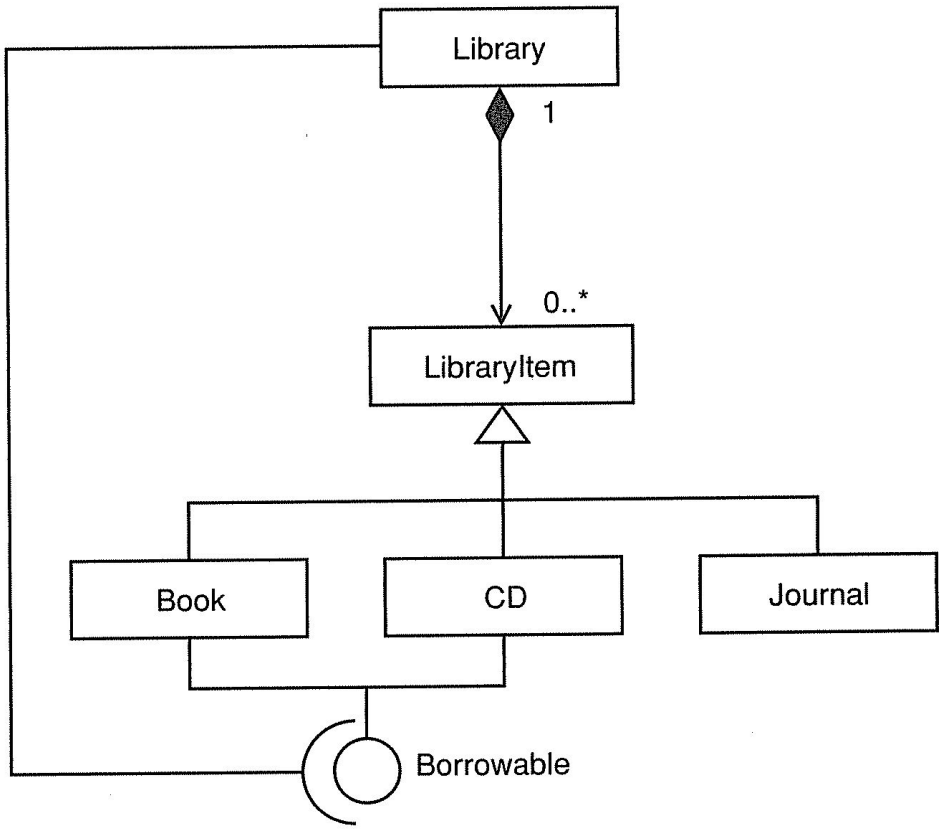
An interface-based solution is more elegant, simpler and with better semantics



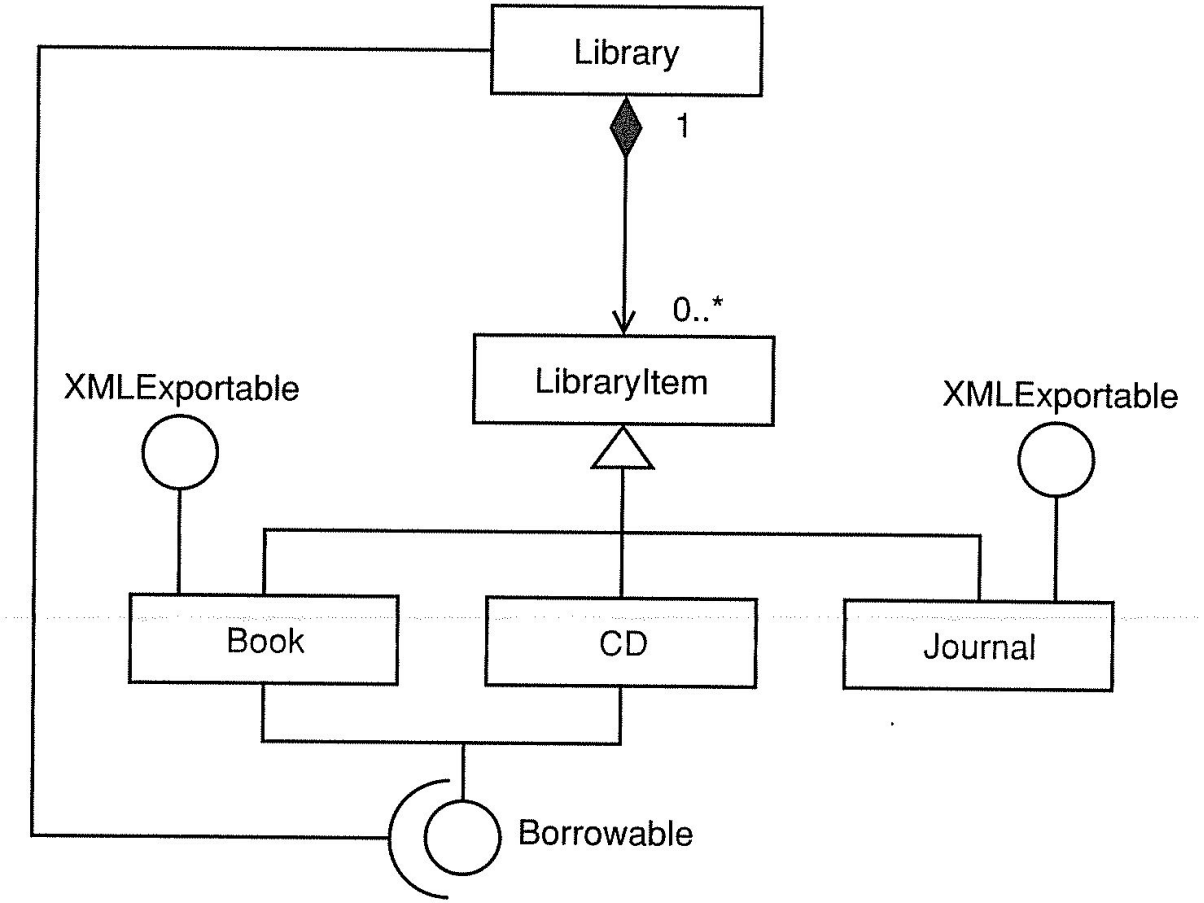


An interface-based solution is more elegant, simpler and with better semantics

- Every item in the Library is a LibraryItem
- “borrowability” was factored out into a separate interface
- Fewer classes than before (5 vs. 7)
- Fewer composition relationships (1 vs. 2)
- Simpler inheritance hierarchy (2 levels vs. 3 levels)
- Fewer inheritance relationships (3 vs. 5)

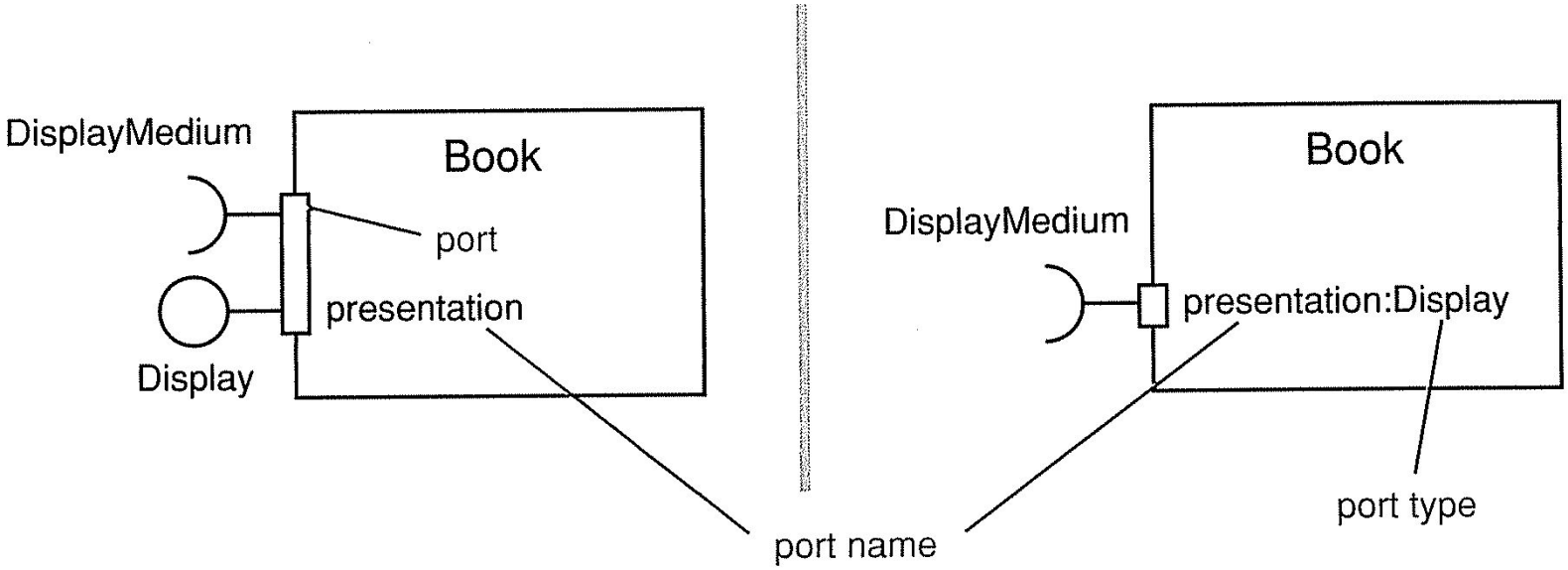


Use interfaces to specify the common protocols of classes that should <u>not</u> normally be related by inheritance



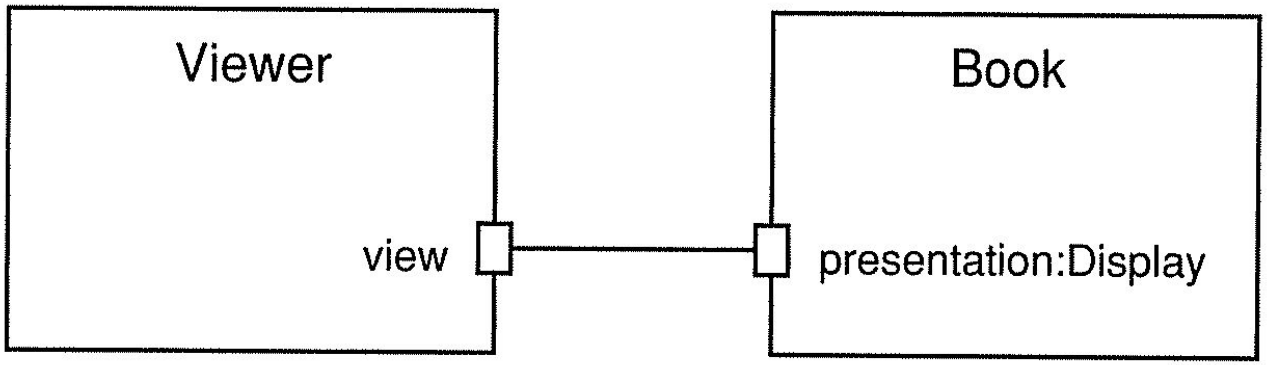
Ports group semantically cohesive sets of provided and required interfaces

- A port is a specific point of interaction between a classifier and its environment
- For ports to be connected, their provided and required interfaces must match



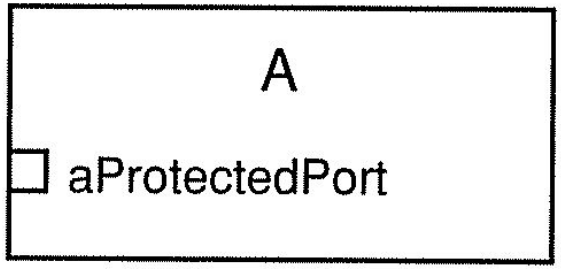
Ports may be used to simplify a diagram

- For ports to be connected, their provided interfaces must match
- Using ports is more concise than explicitly representing all the provided and required interfaces
  - This may make the diagram harder to interpret
- In this example, Viewer and Book are connected via the interfaces specified in the view and presentation ports



Ports have visibility and multiplicity

- If the port is over the classifier boundary, as in the previous examples, it is public
- If it is totally inside the boundary, then it is either protected (this is the default) or private - the actual visibility is only specified in the port specification

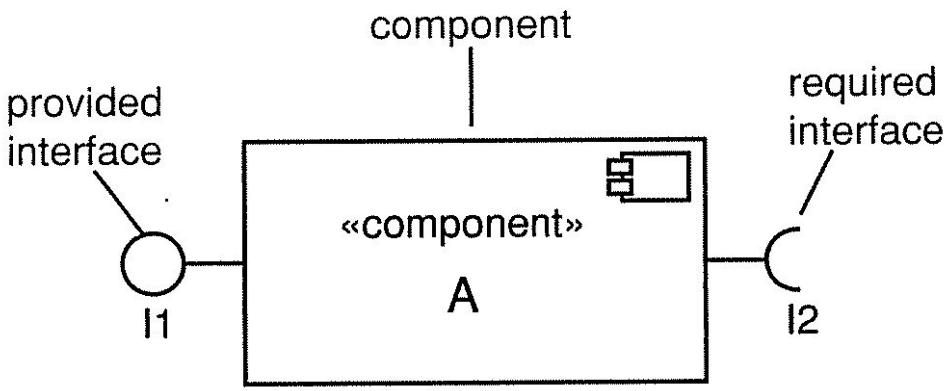


- Multiplicity is shown in square brackets after the port name (e.g., presentation:Display[1])

Component-based development is about constructing software from plug-in parts

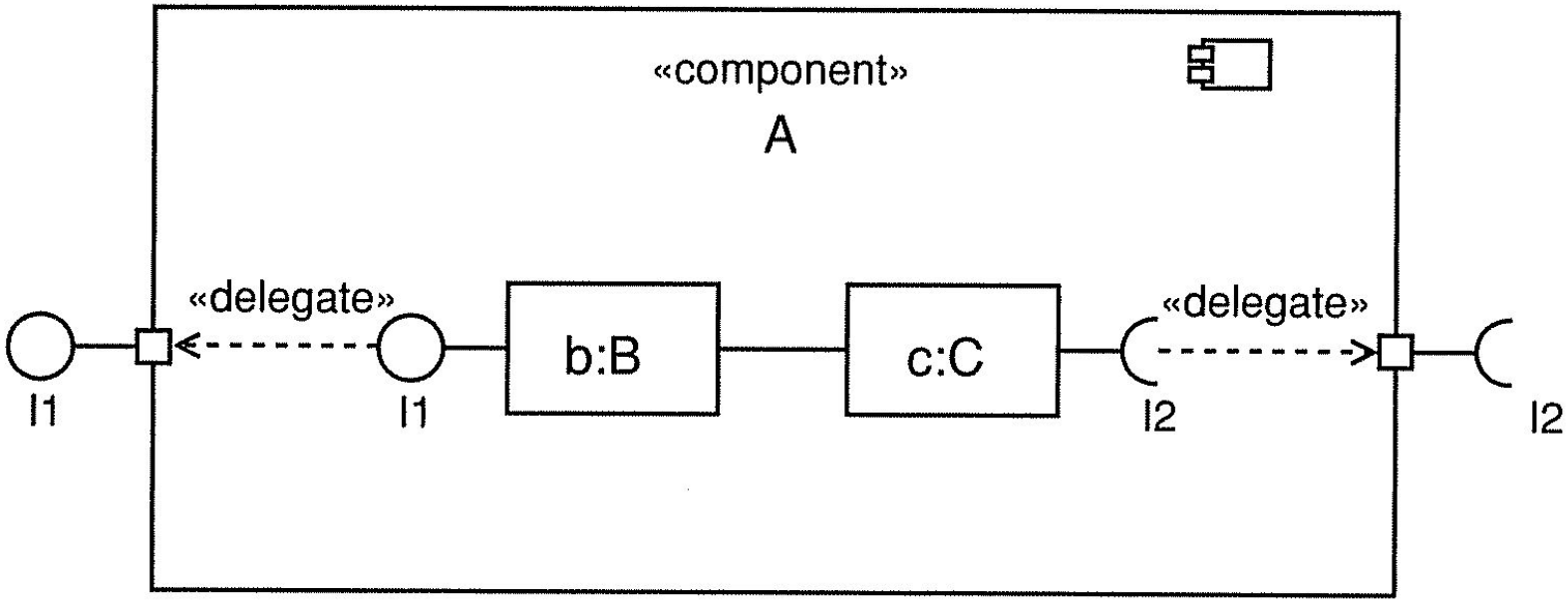
What is a UML component?

- A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment
- Acts as a black-box whose behavior is defined by its required and provided interfaces
  - A component may be replaced by another one, as long as it supports the same protocol

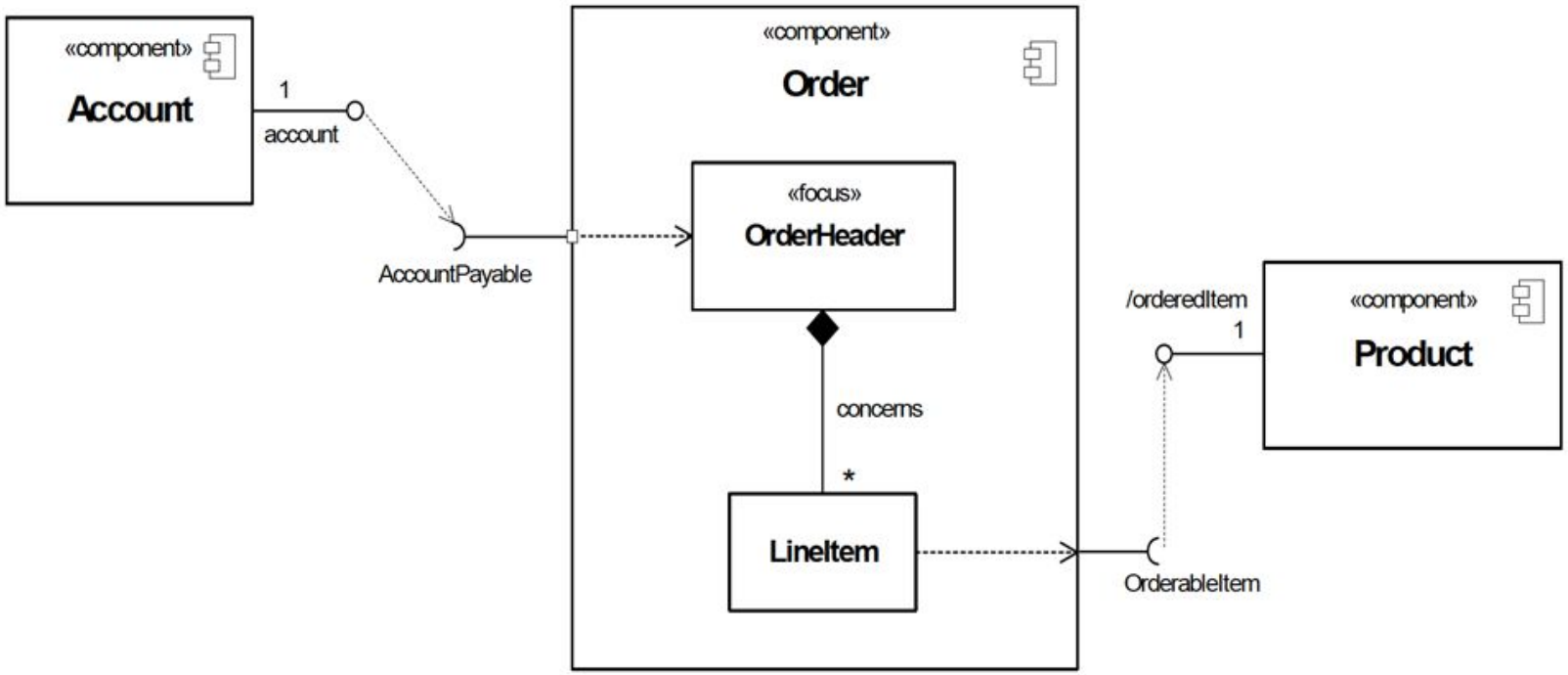


Components may have an internal structure

- The internal parts of the components can use <<delegate>> dependencies to connect to provided and required interfaces through ports

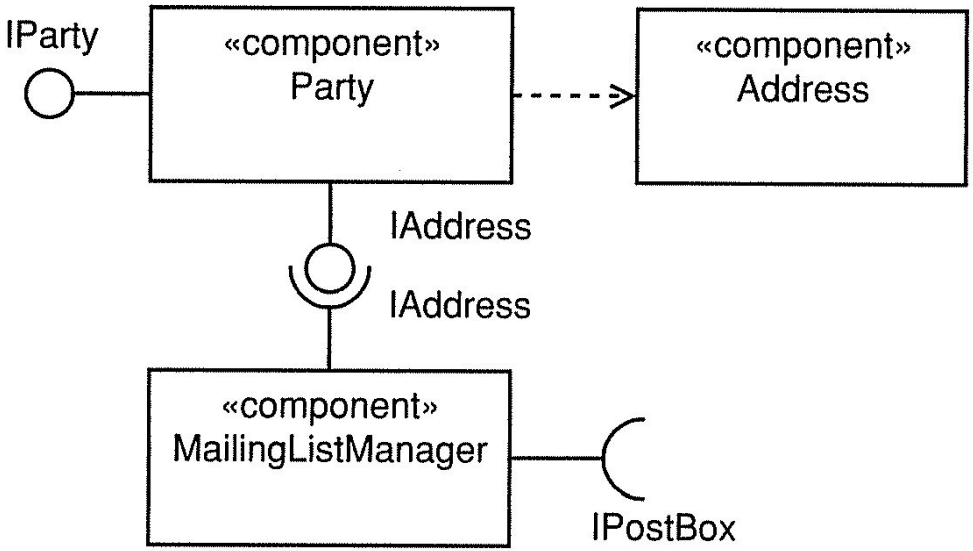


<<delegate>> connectors link external interfaces to the component parts that implement or require them



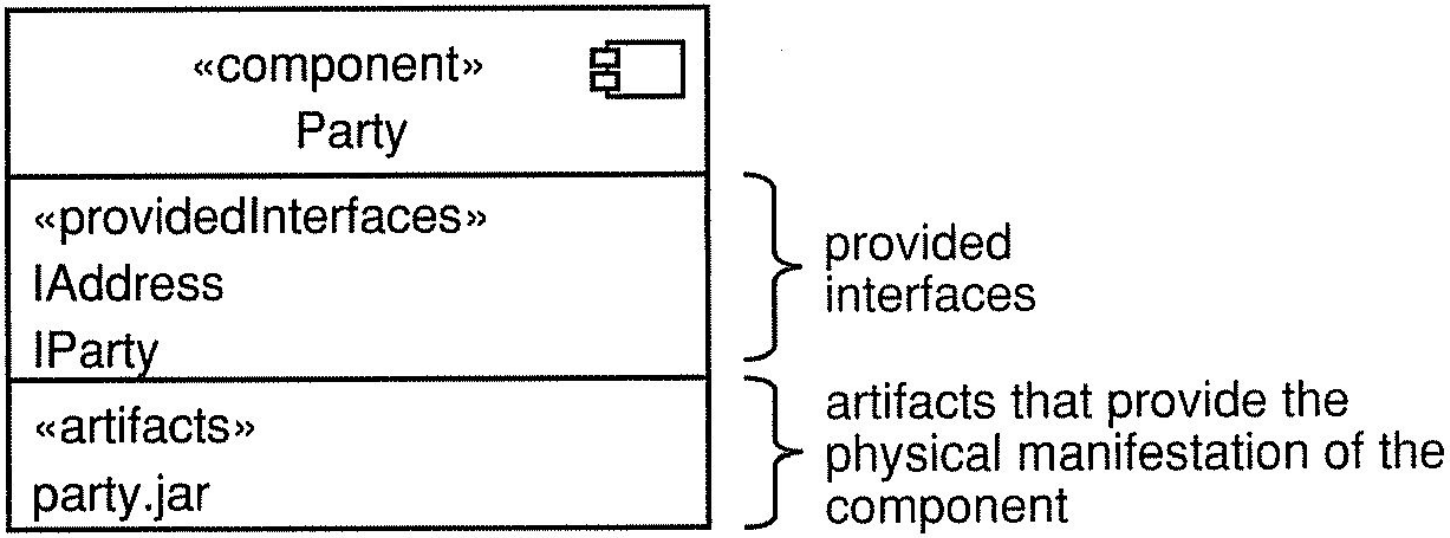
Interfaces allow you to connect components in a flexible way

- Components may depend on other components
- To decouple components, you ***always*** mediate the dependency with an interface
- When a component requires an interface, this can be presented either with a dependency, or using an assembly connector

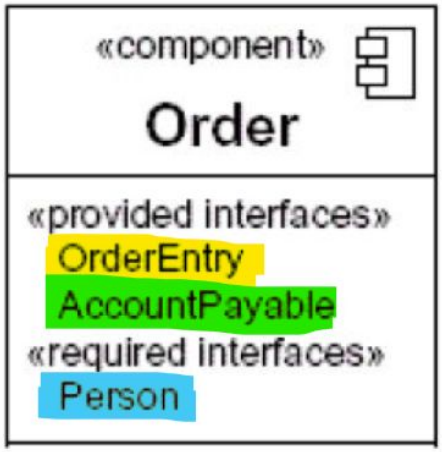


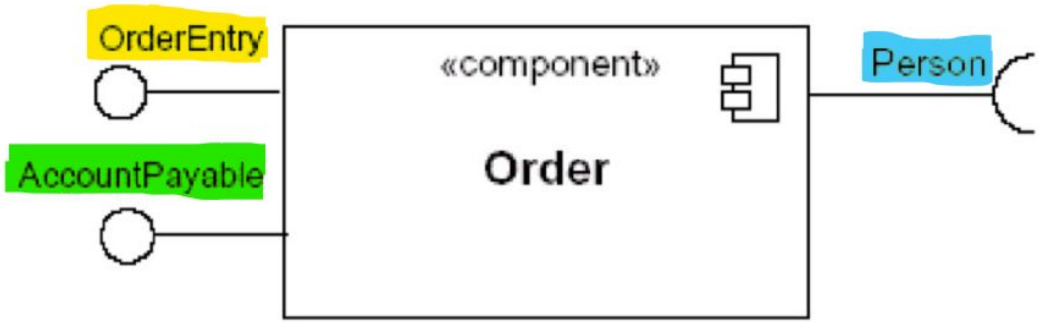
Components may be presented as white boxes

- The white-box view exposes internal details of the component
  - Provided and required interfaces, realizations, associated artifacts

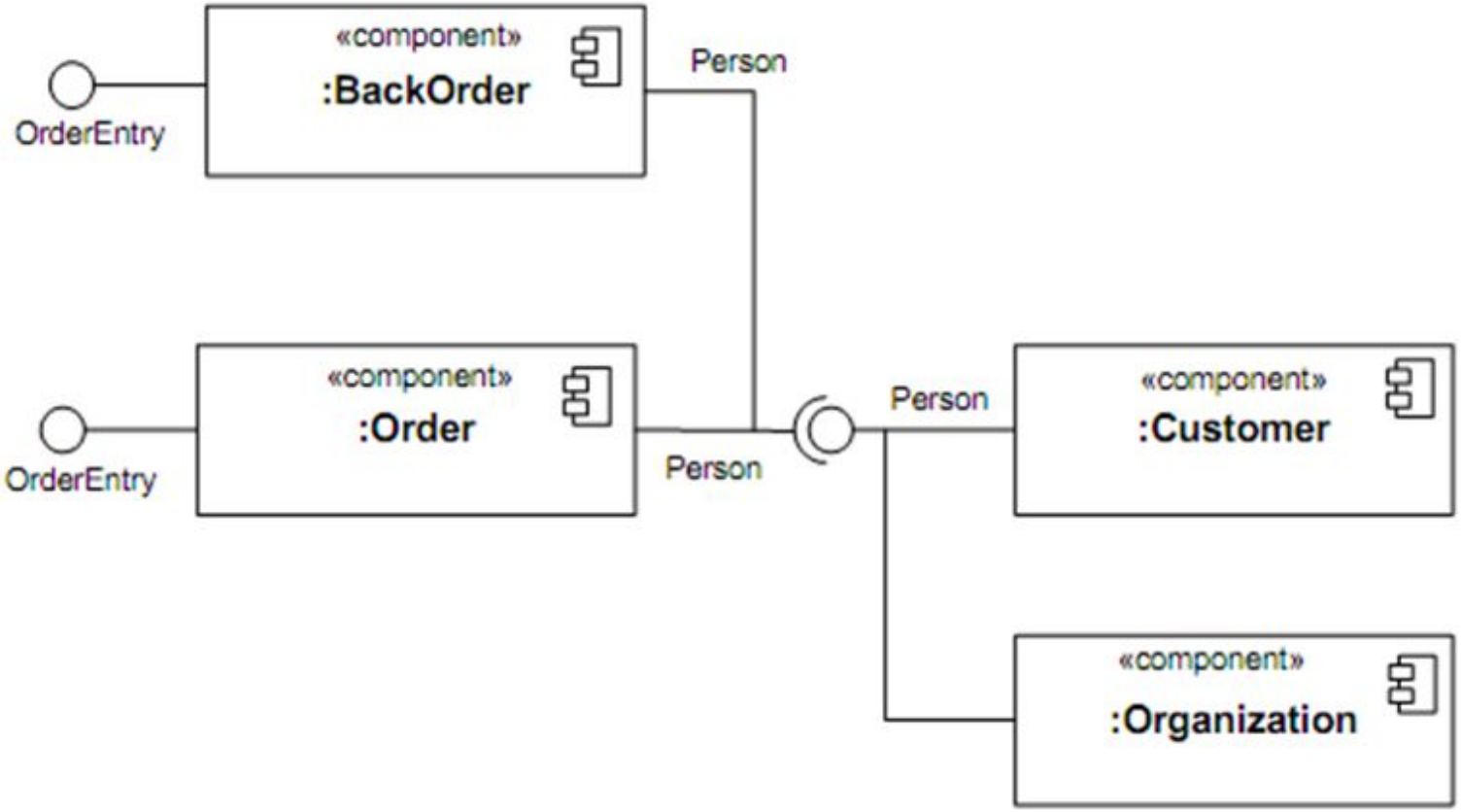


Contrasting component alternative notations



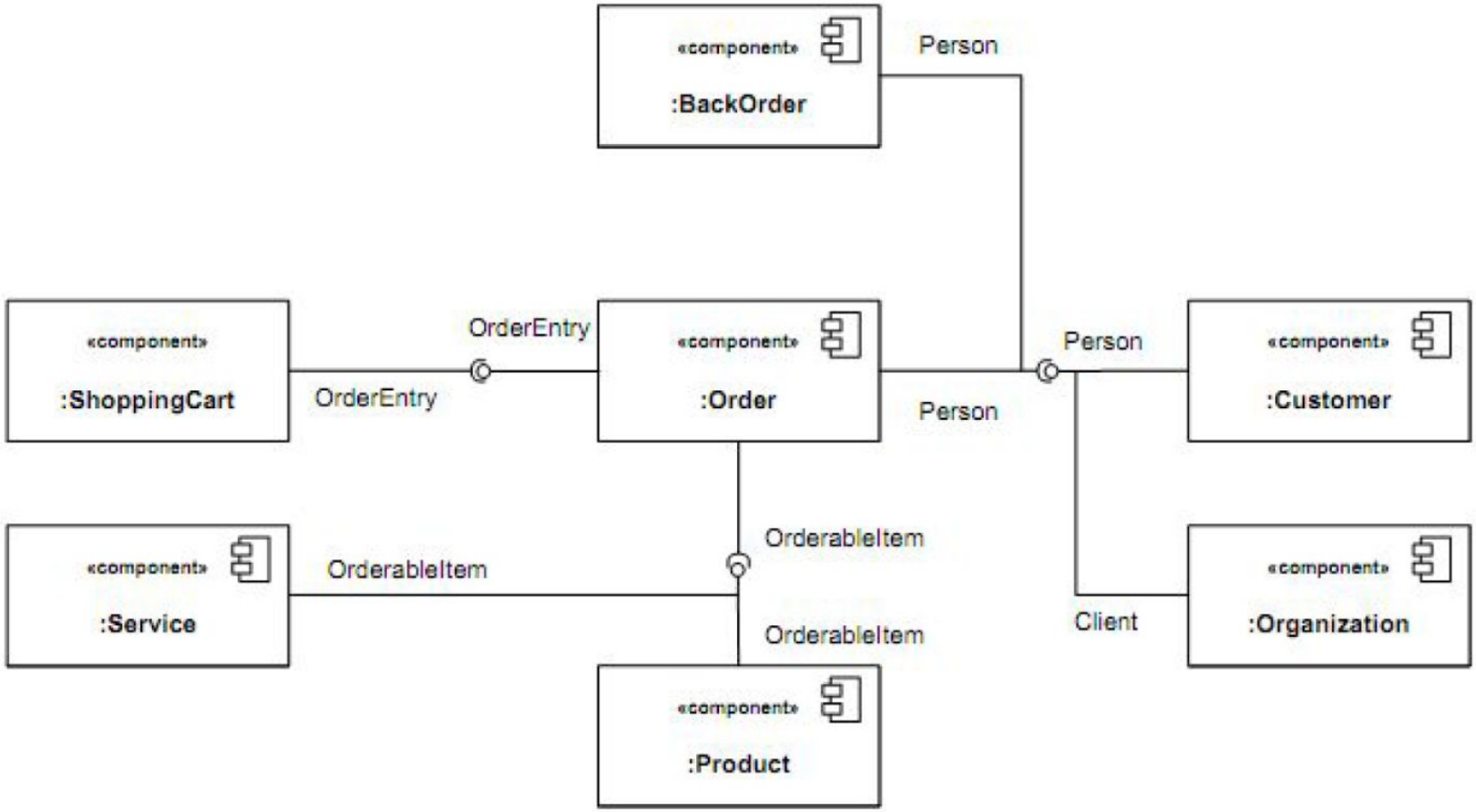


A connector can link multiple providers and customers that share the same provided and required interface



Note: Client interface is a subtype of Person interface

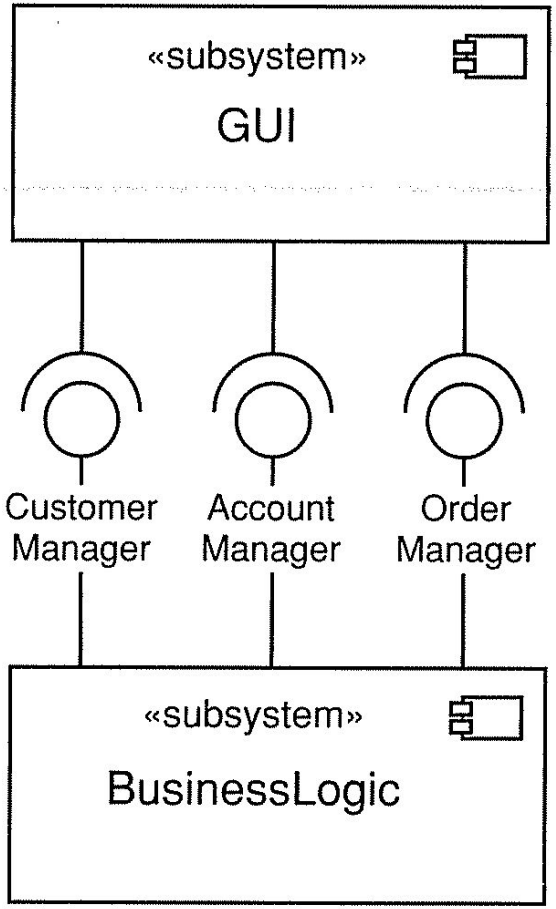
Use a component infrastructure (assembly) to specify how components connect



A <<subsystem>> is a component that acts as a unit of decomposition for a larger system

Use interfaces to hide the implementation details of subsystems

- The GUI only knows BusinessLogic through its provided interfaces
- We can completely change the BusinessLogic component for another one providing exactly the same interfaces
- We can completely change the GUI component for another one requiring exactly the same interfaces



How do you find interfaces?

- Challenge each association
  - Should it be to a particular class of objects, or more flexible (i.e., to an interface)?
- Challenge each sent message
  - Should it be sent to a particular class of objects, or more flexible (i.e., to an interface)?
- Factor out groups of operations that might be reusable elsewhere
- Factor out sets of operations that repeat in more than one class

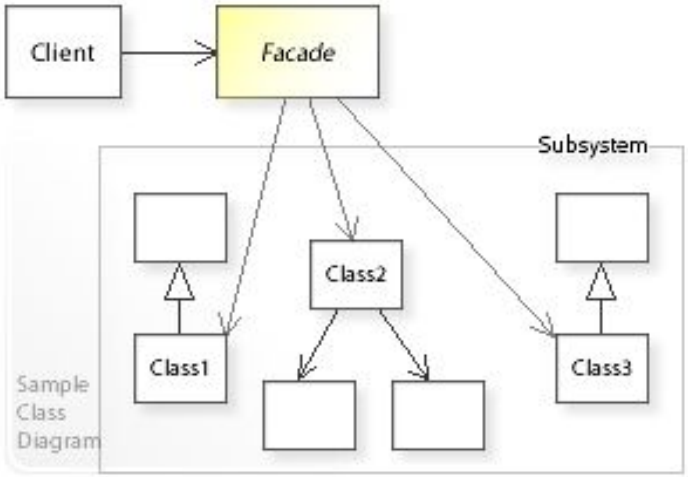
How do you find interfaces?

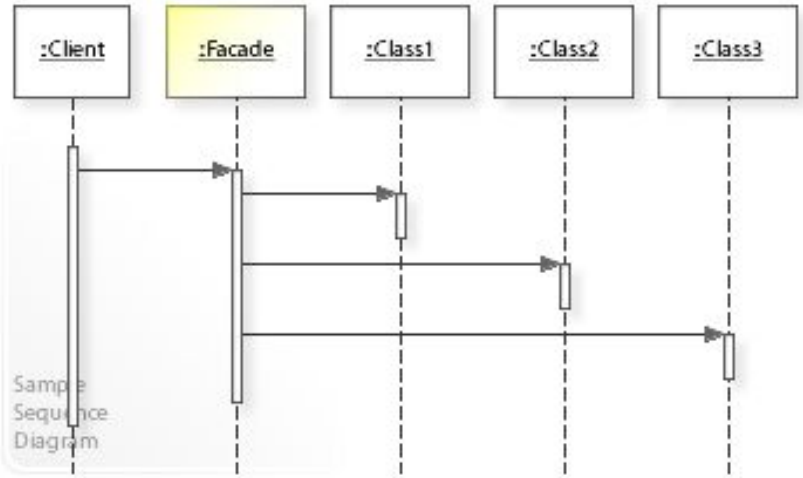
- Factor out sets of attributes that repeat in more than one class
- Look for classes that play the same role in a system
  - The role may be a good candidate for interface
- Look for future expansion possibilities
- Look at the dependencies between components and mediate these with assembly connectors, when possible

Designing with interfaces

The façade pattern

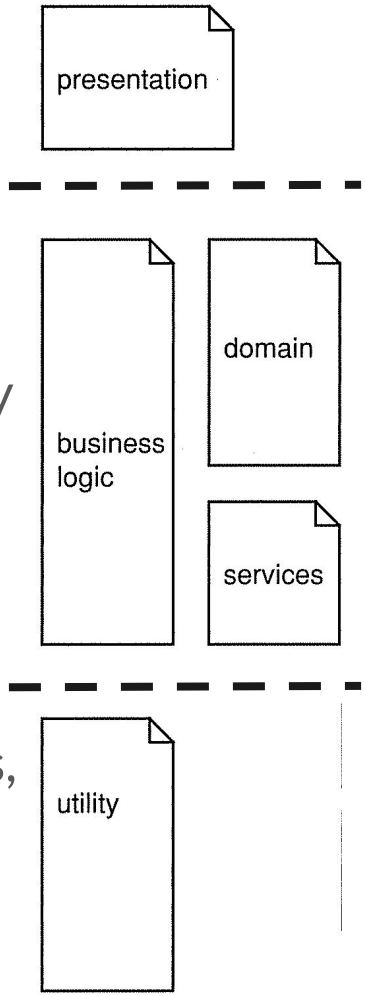
- Hides a complex implementation behind a simple interface
  - Good for information hiding and separation of concerns
    - Identify cohesive parts of the system
    - package these into a <<subsystem>>
    - define an interface to that <<subsystem>>



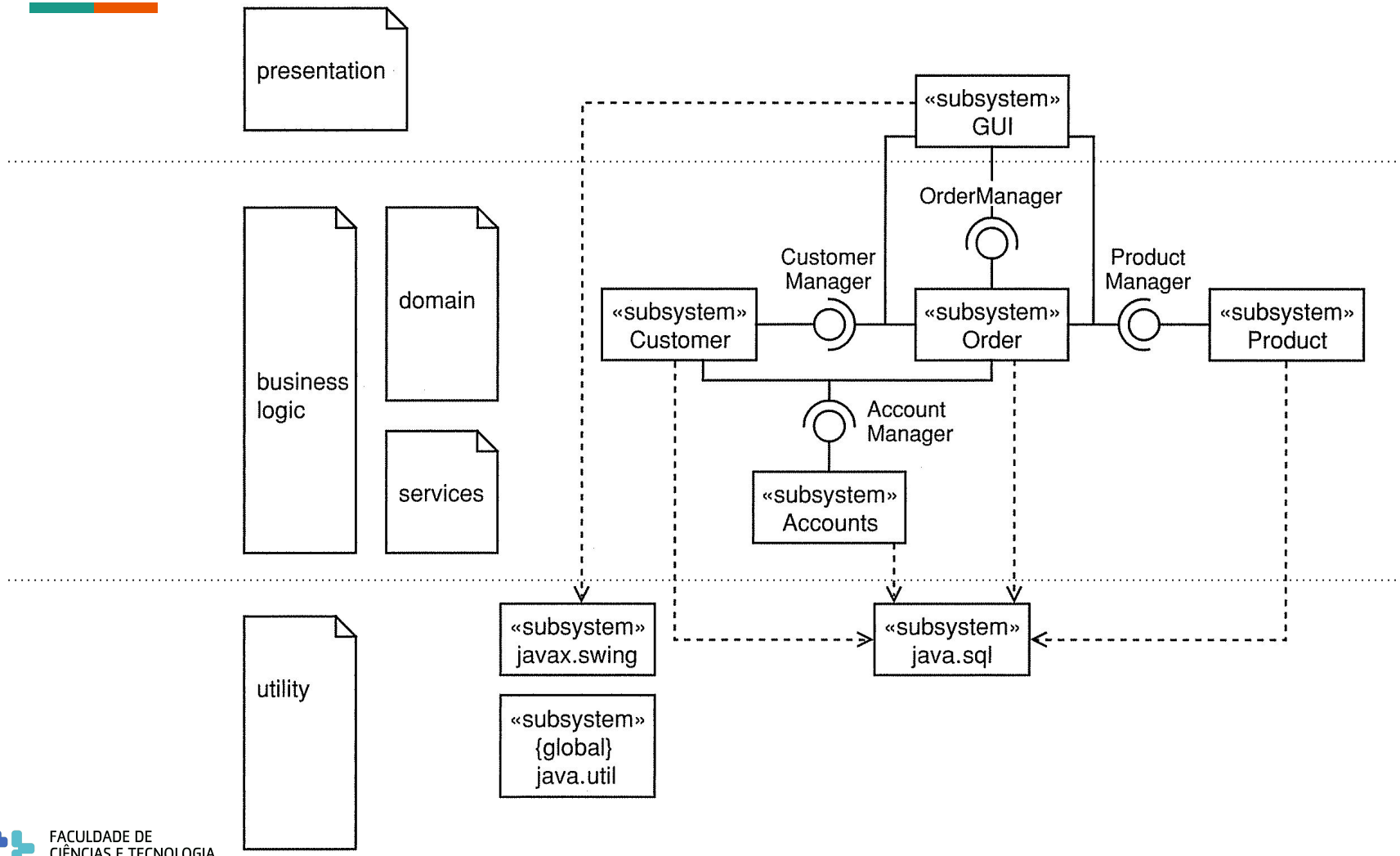


The layering pattern

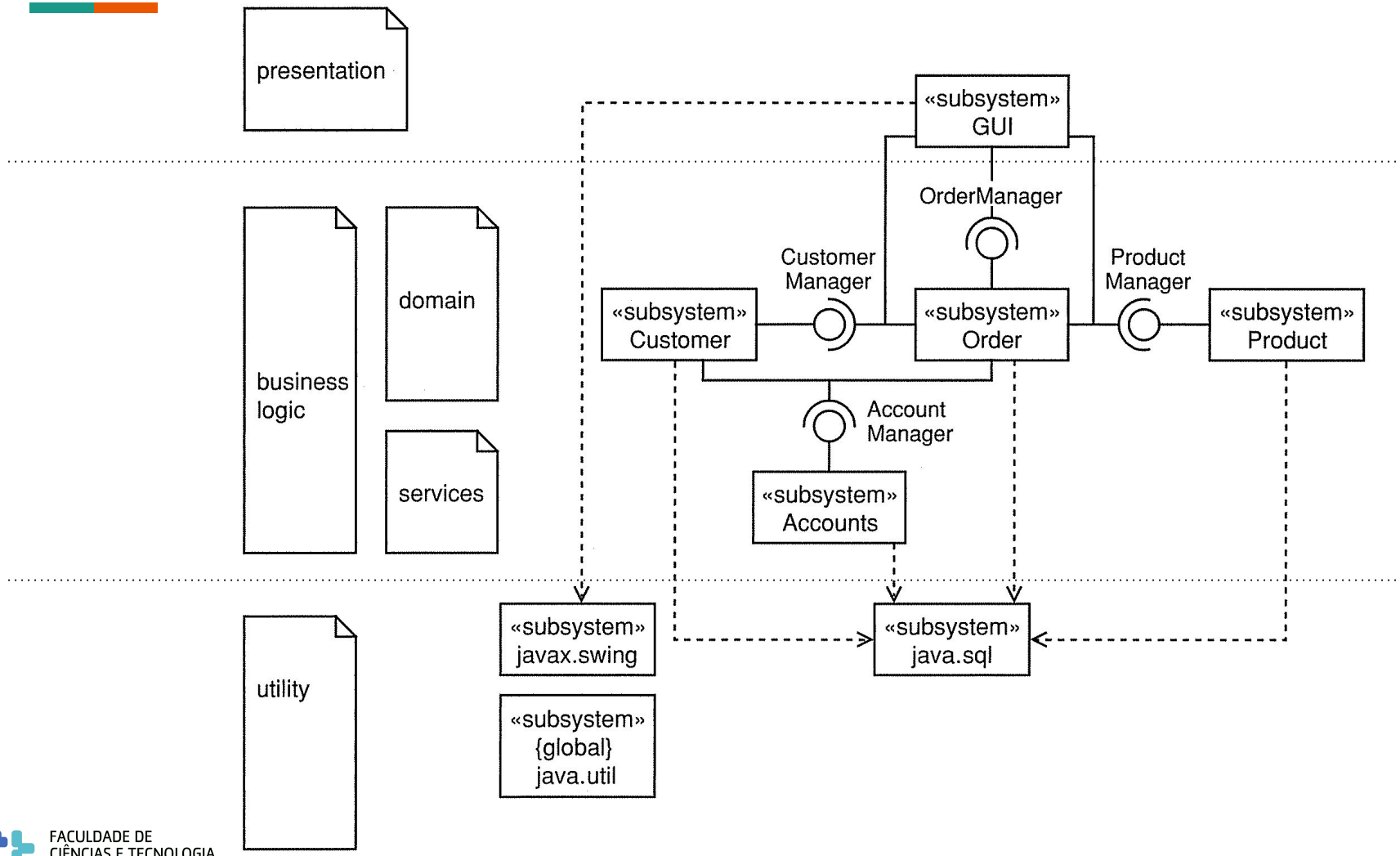
- Organizes subsystems into semantically cohesive layers
- Manages coupling between subsystems by
  - introducing new interfaces where needed
  - repackaging classes into new subsystems in a way that reduces coupling between subsystems
- Layers should be as decoupled as possible
  - dependencies only go one way
  - all dependencies are mediated by interfaces
- A common application of this pattern has three layers, with **presentation**, **business logic** and **utility**



Layered architecture example



The dependencies to the Java packages are shown through simple use, or by declaring them as global



Advantages and shortcomings of interfaces

- Designing to a contract is more flexible than designing to an implementation
- The added flexibility will help dealing with more volatile parts

- The added flexibility can lead to more complexity
- Some flexibility may be sacrificed for the more stable parts of the system
  - If they are not likely to change, the added benefit of the extra flexibility may not compensate the added complexity
- There may be a small performance penalty involved in using an interface (but this is fairly negligible, in general)

Hints and tips for building component diagrams

Top-down strategy

- Adequate to give a first overview of the project
- Helps to distribute work among collaborators in early phases of the project
- “Dangerous” because it promotes over-architecture and overdoing in the design
  - We might end-up developing components we don’t need (not complex enough)

Bottom-up strategy

- Interesting when we depart from a collection of classes and we try to evolve to a collection of reusable components
- Interesting to allow for the access to reusable functionalities in a given application
- Interesting while distributing the work among sub-teams

Guidelines for building components (⅓)

- Keep components cohesive
- Interface/Boundary classes should be in application components
- Technical classes are infrastructure components
- Define contracts between classes
- Hierarchies should be part of the same component

Guidelines for building components (2⁄3)

- Identify domain components
- Identify collaboration between business classes
- Server classes should be in the same component
- If a component has only one client unify with it

Guidelines for building components (3/3)

- Pure classes can not be used in the domain class
- Highly coupled Classes should be in the same component
- Minimize message exchange between components
- Define contracts between components

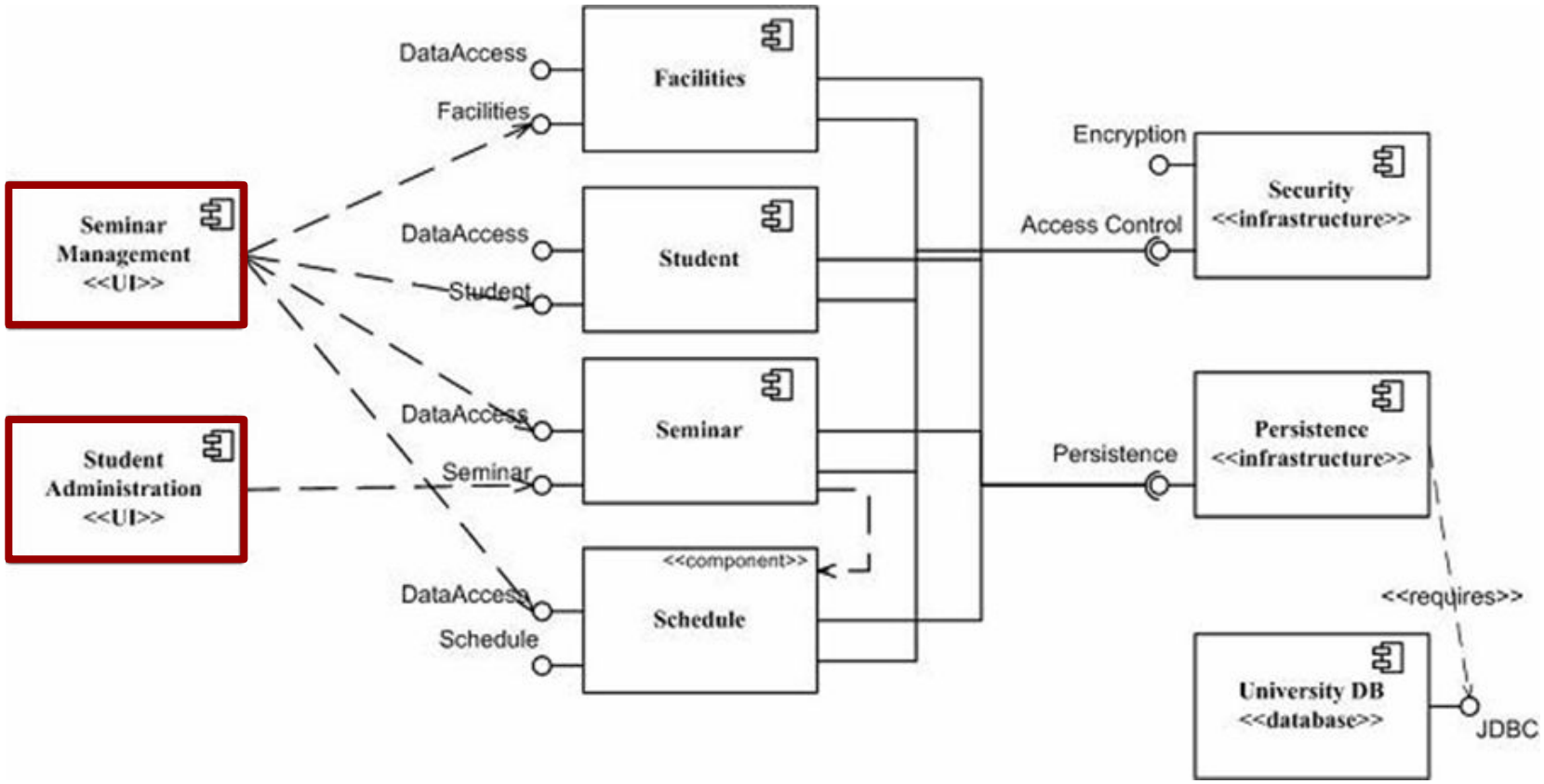
Keep component's cohesion

- A component must implement only a set of functionalities tightly related
- Examples:
  - The logic interface of a single-user application
  - A set of classes that represent a large scale domain concept
  - Classes, of technical nature, that together represent a common concept of infrastructure

User interface classes and <<boundary>> classes are part of the application

- The User Interface classes and System boundary classes should be in components with the stereotype <<application>>
- These classes can implement screens, pages, reports, or glue that allows for the option in between them
  - In Java this normally corresponds to Java Server Pages (JSPs), servlets, and classes that represent screens implemented over interface class libraries like Swing

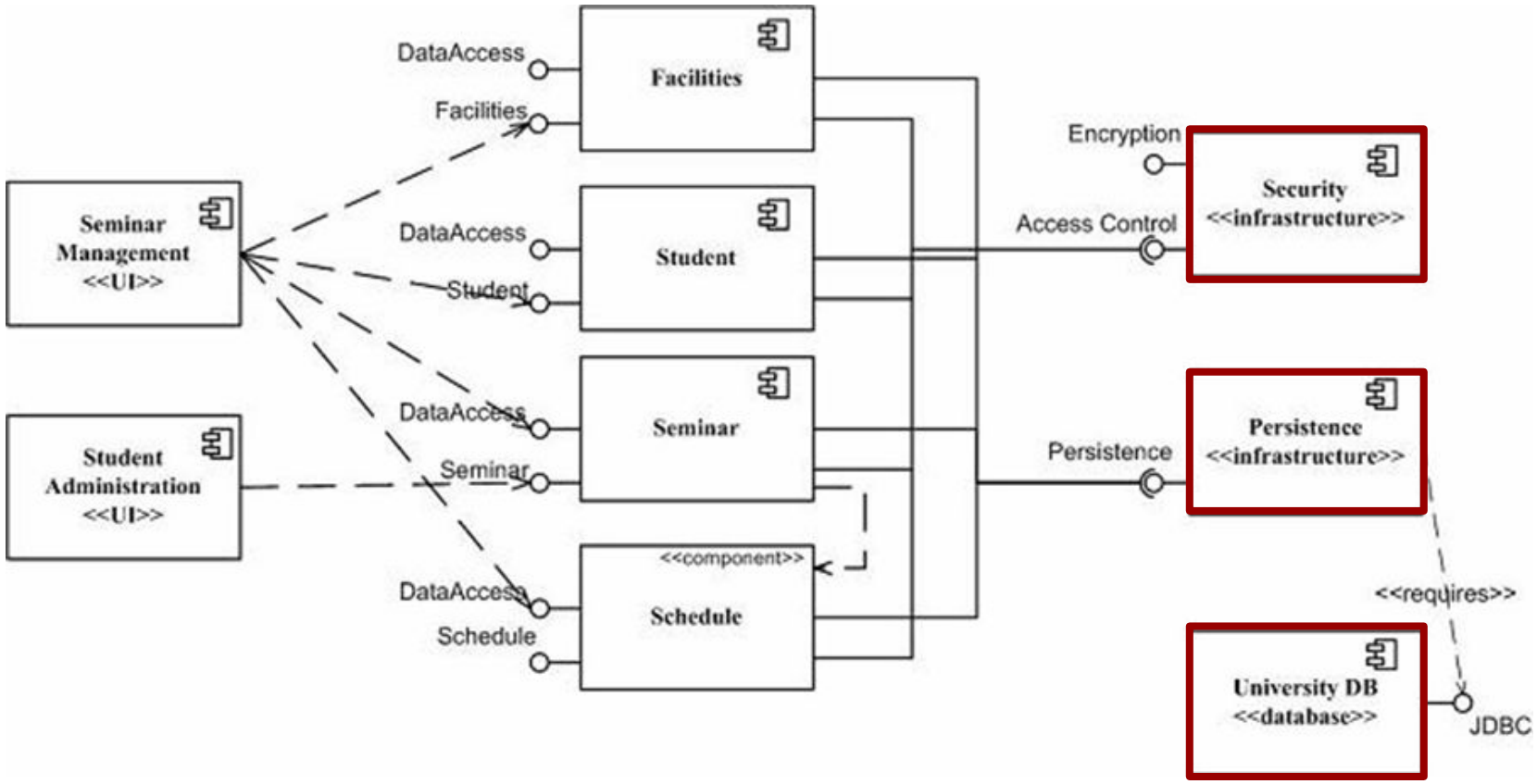
User interfaces attributed to application components



Technical classes are part of the infrastructure

- The technical classes should be distributed to components with the stereotype <<infrastructure>>
- The technical classes implement system services as security, persistency and middleware

Technical parts attributed to infrastructure components



Define contracts among components

- A class contract is a method that corresponds to a message sent by other objects
  - A class Seminar contracts could include operations like enrollStudent() and dropStudent()
- When identifying components, we can ignore any operation that is not a contract
  - Those operations do not contribute to the communication between distributed objects

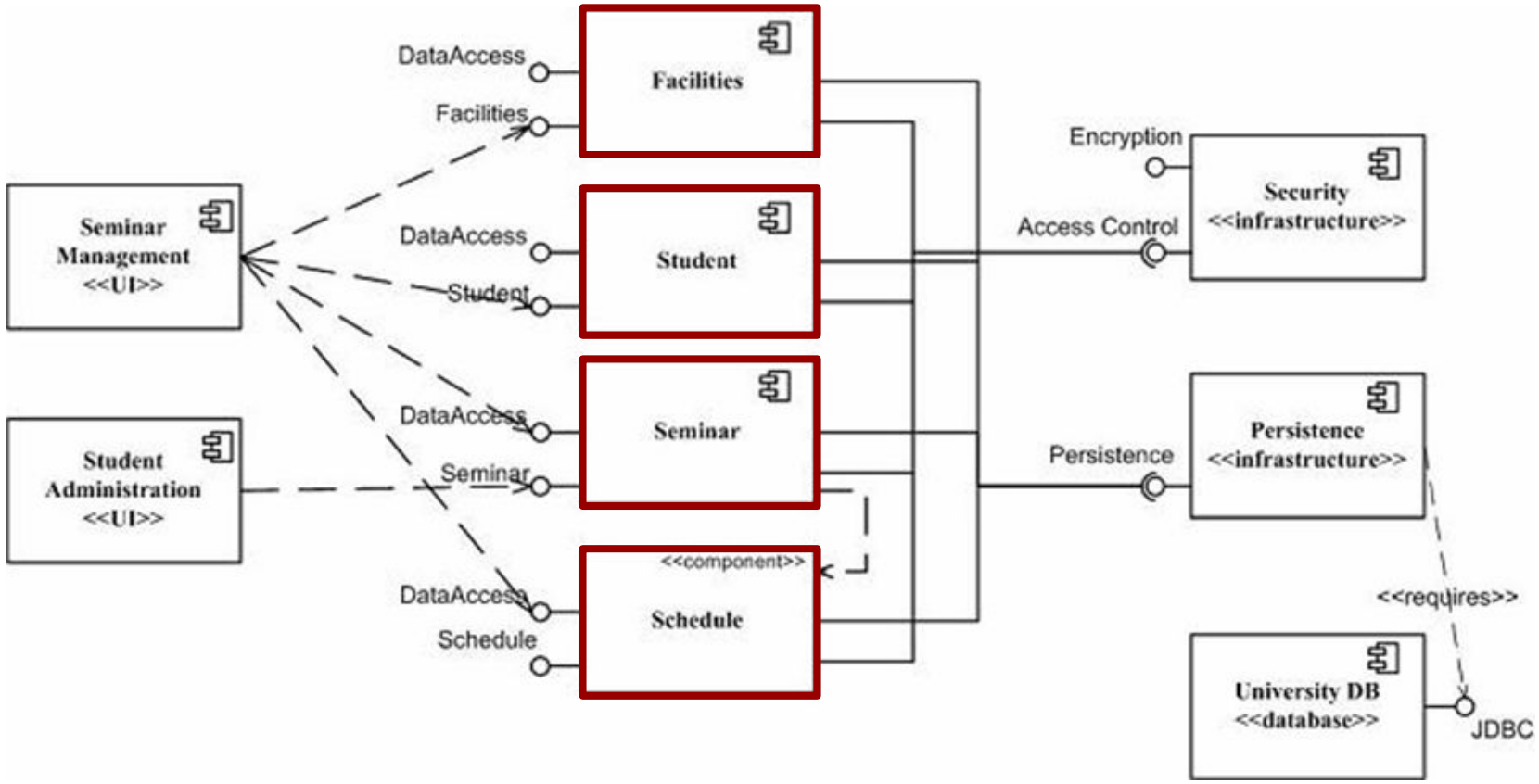
Place class hierarchies inside the same component

- In general, all classes from the same hierarchy, be it inheritance or composition, are part of the same component
  - This leads to a highly cohesive component, while minimizing coupling between components

Identify domain components

- Domain components (business) are composed by a set of classes that collaborate between each other to support a cohesive set of contracts (**high cohesion**)
- To minimize the network traffic, to reduce the reactive time of our application, we try to design our components in such a way that the information flow happens mostly inside a component and not among distinct components (**low coupling**)

Also identify domain components



Bibliography

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”, Second Edition, Addison-Wesley 2006

- Chapter 19