

MACIASZEK, L.A. (2005):
Requirements Analysis and System Design, 2nd ed.
Addison Wesley, Harlow England, 504p.
ISBN 0 321 20464 6

Chapter 6
System Architecture and Program Design

© Pearson Education Limited 2005

Topics

- *Distributed physical architecture*
- *Multi-layer logical architecture*
- *Architectural modeling*
- *Principles of program design and reuse*
- *Behavioral and structural collaboration*

What is design?

- *Once the technical details include software/hardware considerations, an **analysis** model becomes a **design** model*
 - ***architectural design** is the description of a system in terms of its modules*
 - *layered organization of classes and packages*
 - *the assignment of processes to computing facilities*
 - *reuse and component management*
 - ***detailed design** is the description of the internal workings of each module (use case)*
 - *algorithms and data structures*
 - *collaboration models required for the realization of use cases*

Distributed physical architecture

■ **Logical architecture**

- *the client and server are rather logical concepts*
 - **client** is a computing process that makes requests of the server process
 - **server** is a computing process that services the client requests

■ **Physical architecture**

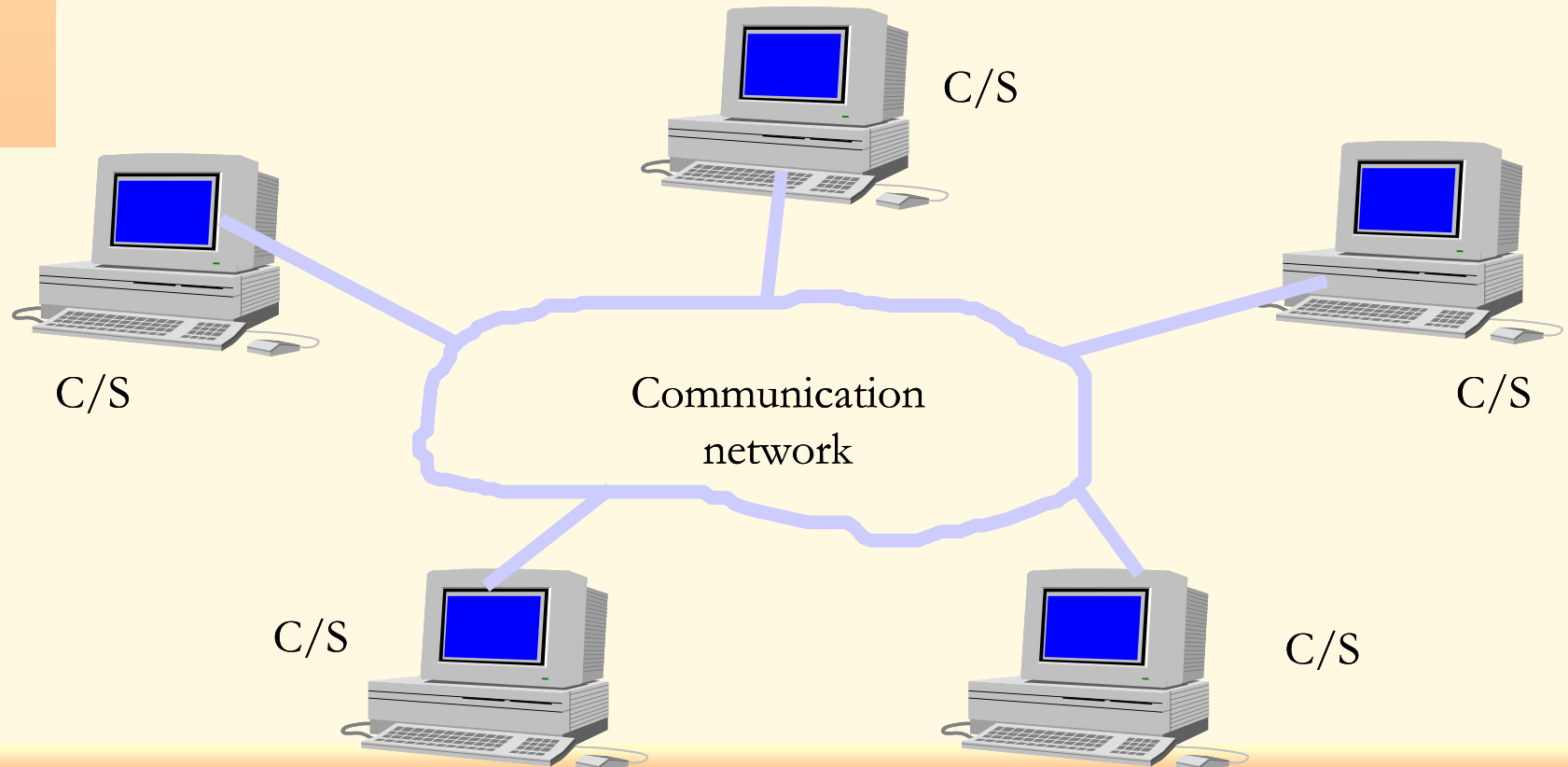
- *allocates processing components to computer nodes*
- **UML deployment diagrams**

■ **Distributed physical architecture:**

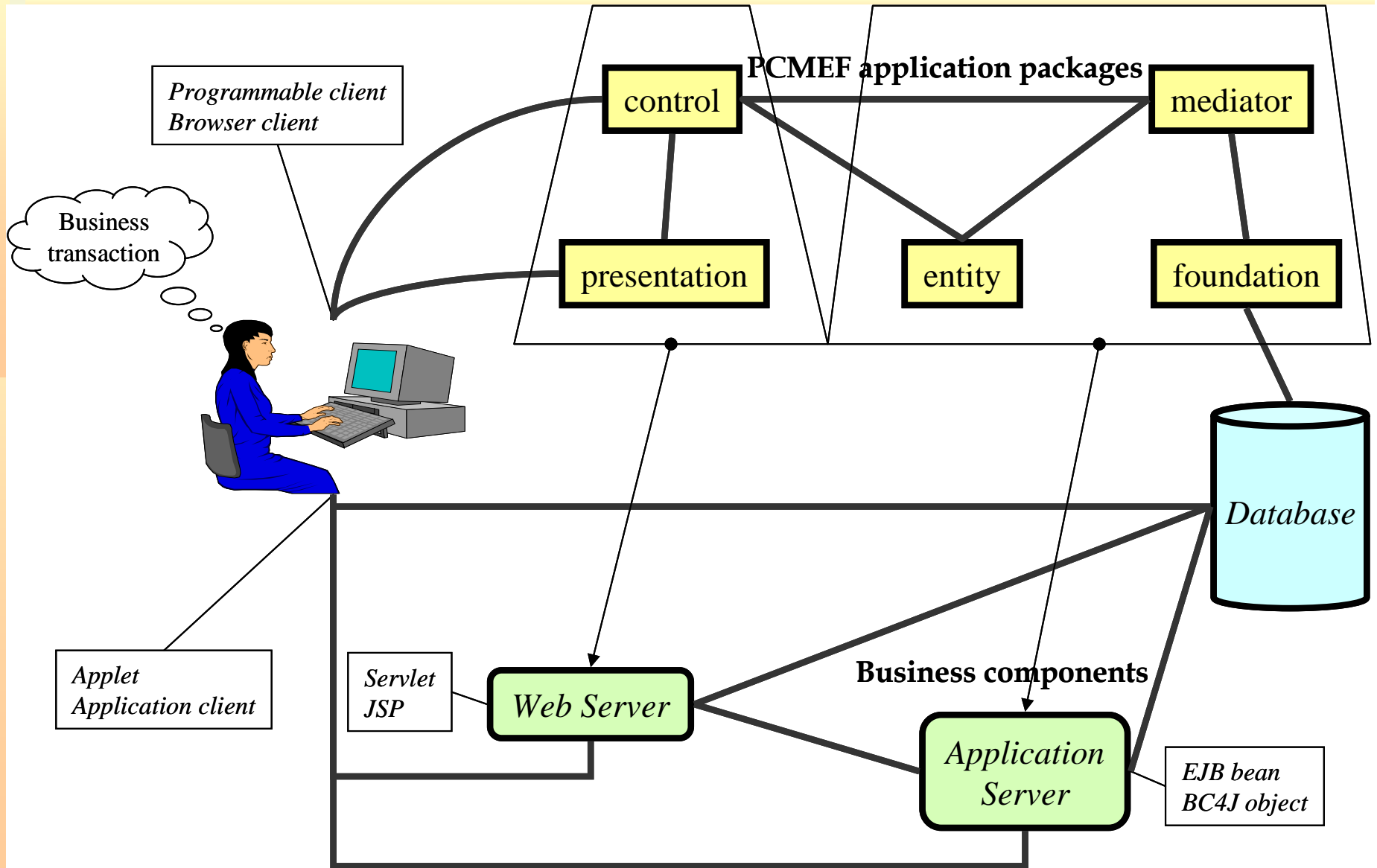
- *peer-to-peer architecture*
- *multi-tier architecture*
- *database-centric architecture*

Peer-to-peer architecture

- **distributed processing** system in which any process or node in the system may be both **client** and **server**
- special consideration - minimization of **network traffic** while maximizing the overall **system throughput**
- **distributed database** system – when it is possible in a single request to combine data from two or more database servers

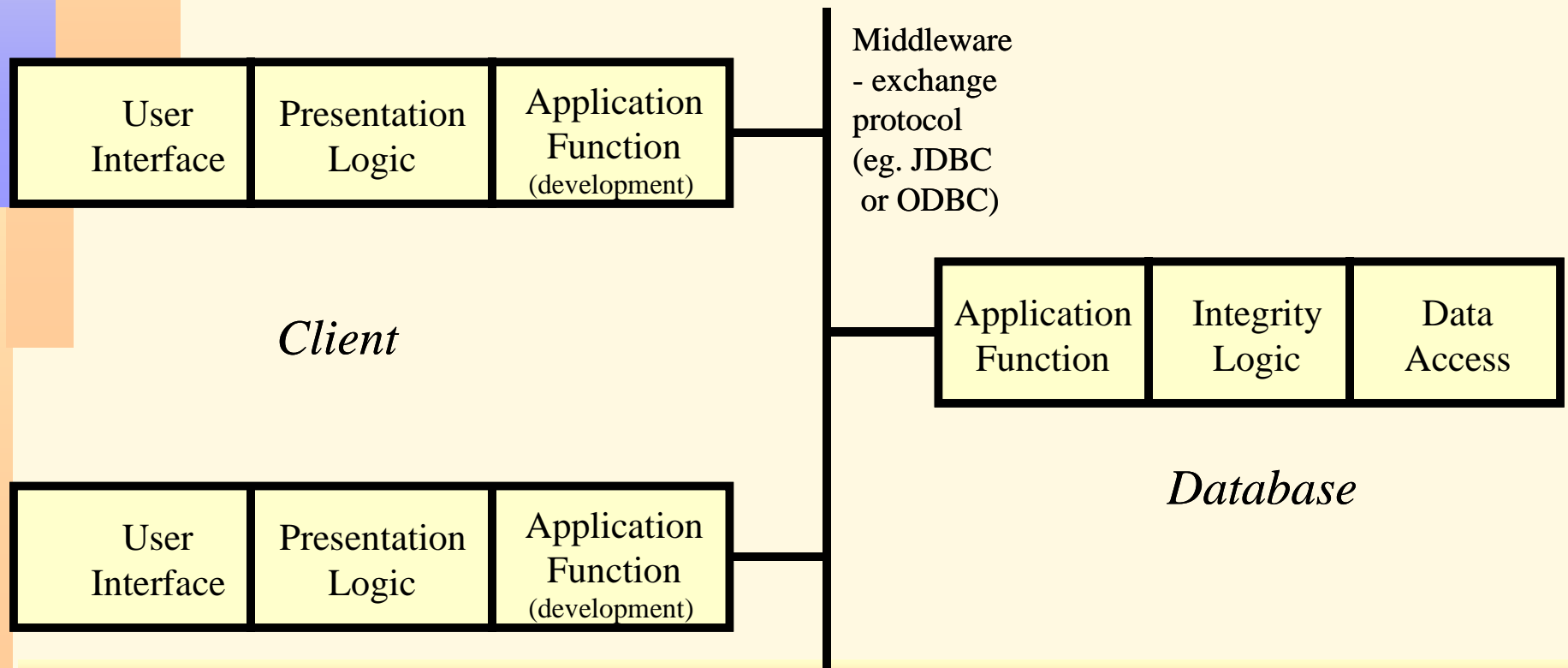


Multi-tier architecture



Database-centric architecture

- modern databases are **active** – they can be programmed
 - stored procedures
 - triggers

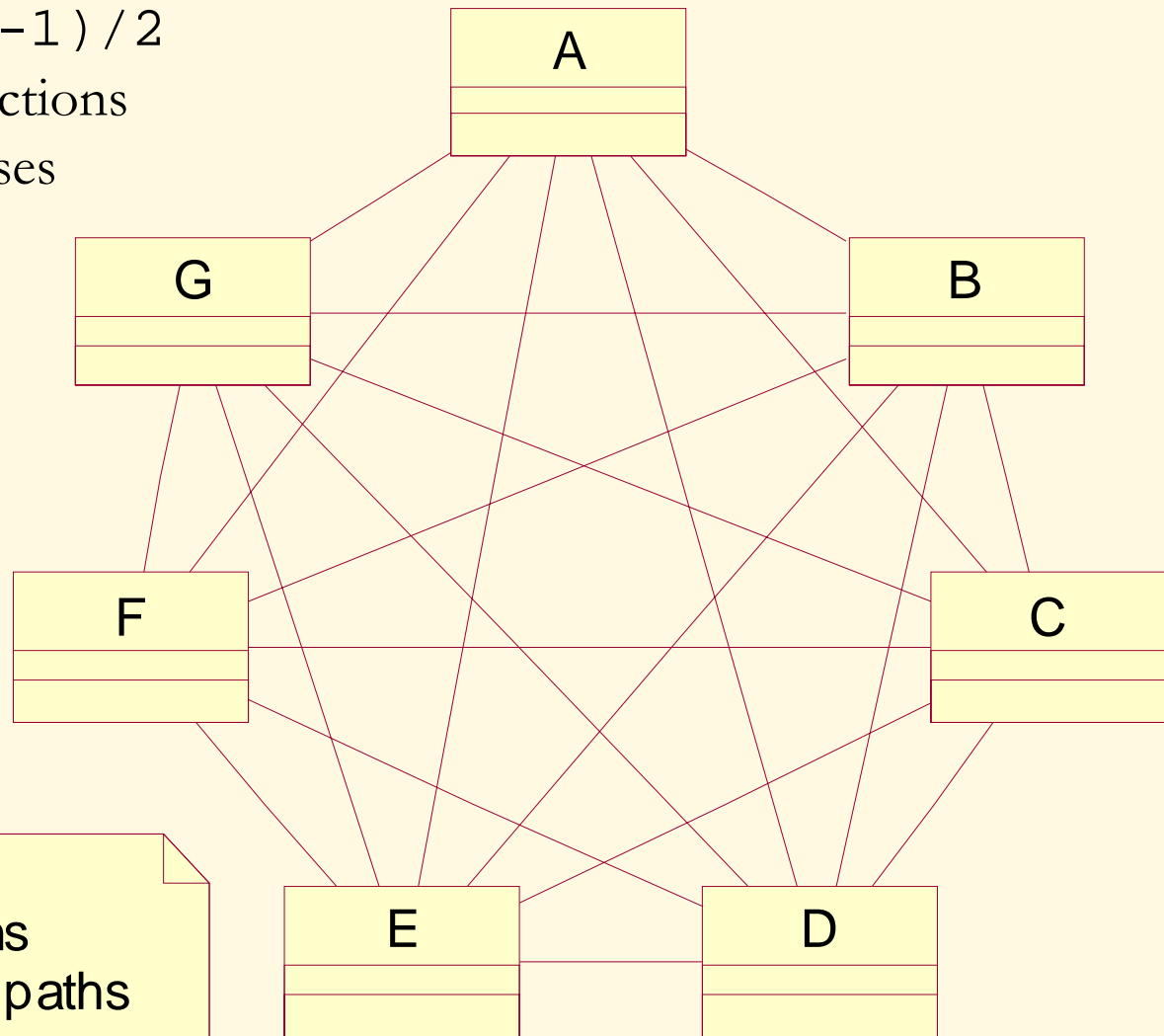


Multi-layer logical architecture

- *Unrestricted communication between objects cannot be permitted in system modeling*
 - ***Networks** of intercommunicating objects are bad news → in networks, the number of communication paths between objects grows exponentially with the addition of new objects*
- *Successful systems are organized in **hierarchies***
 - *Hierarchies reduce the complexity from exponential to polynomial → in a typical hierarchy, only objects in adjacent layers communicate directly*

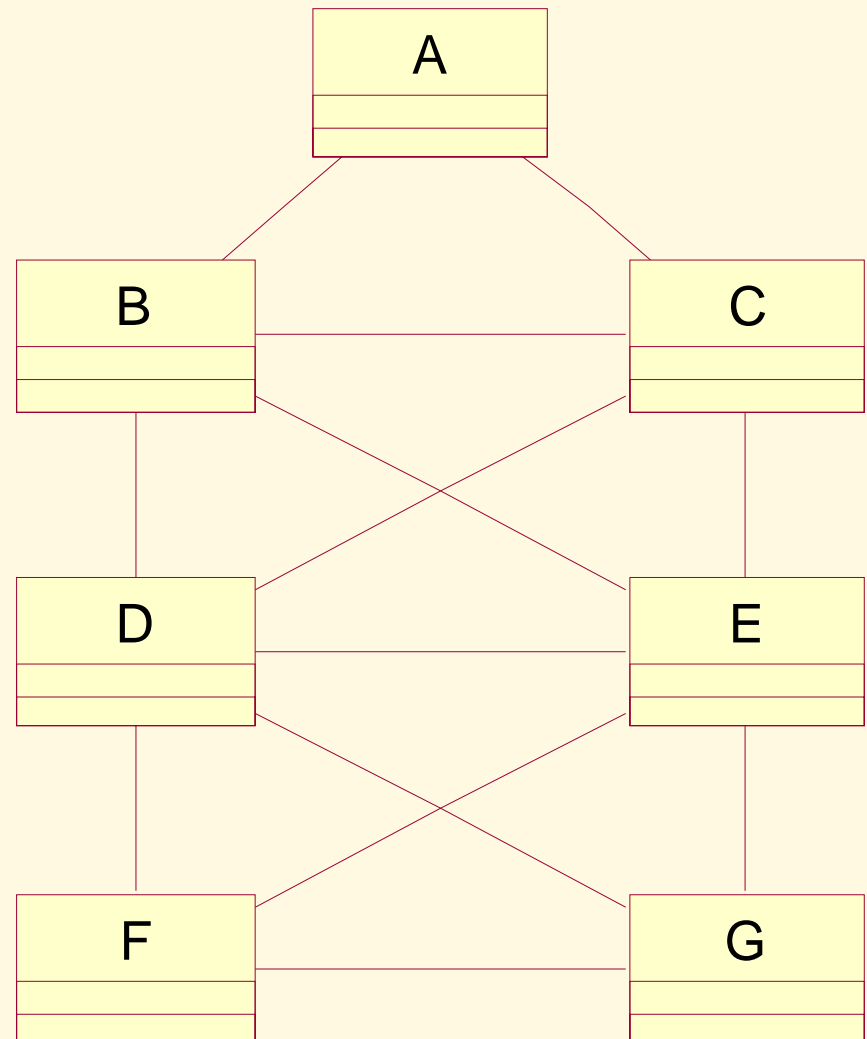
Complexity of networks

There are $n(n-1)/2$
possible connections
between n classes



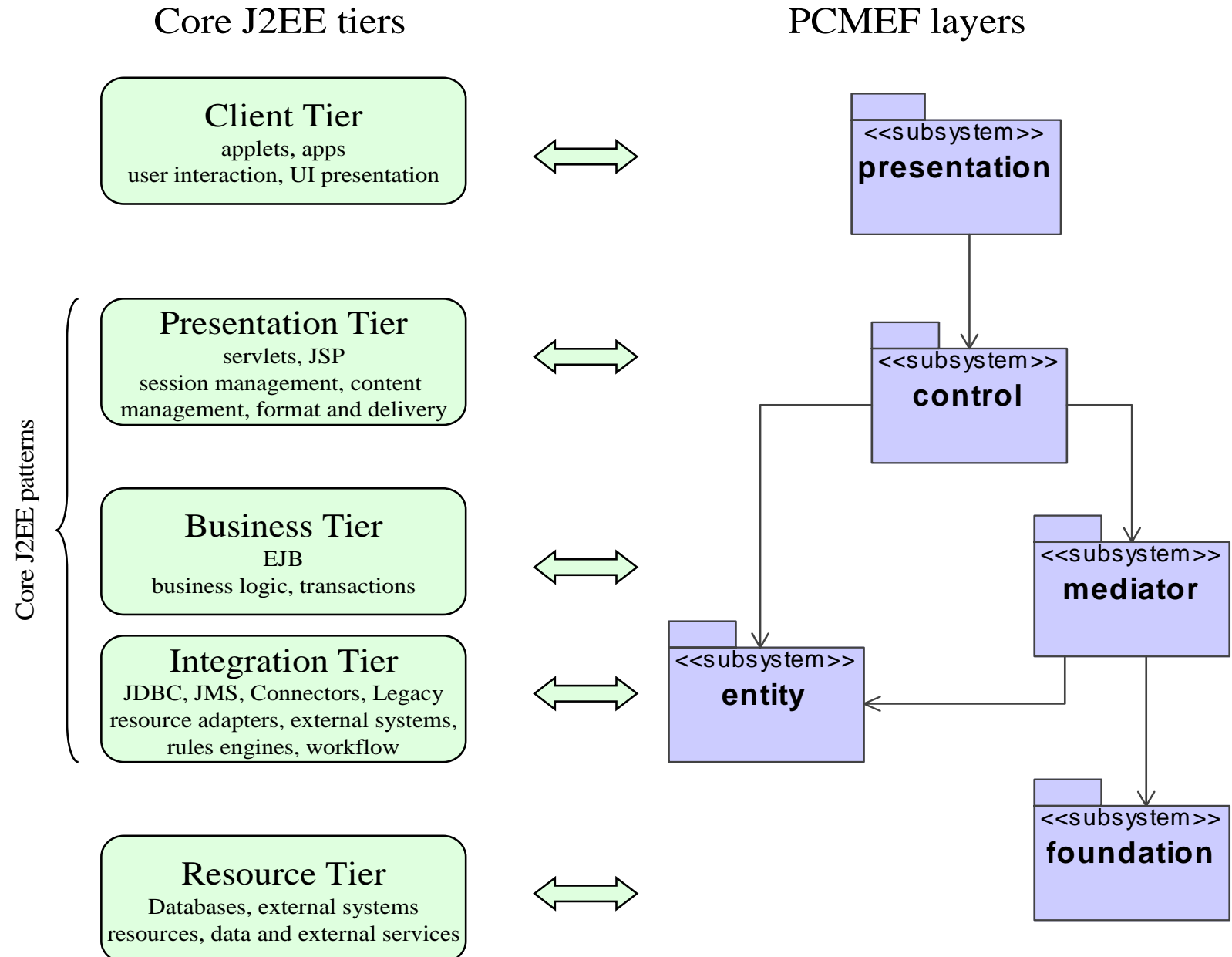
no layers
21 connections
42 interaction paths

Complexity of hierarchies



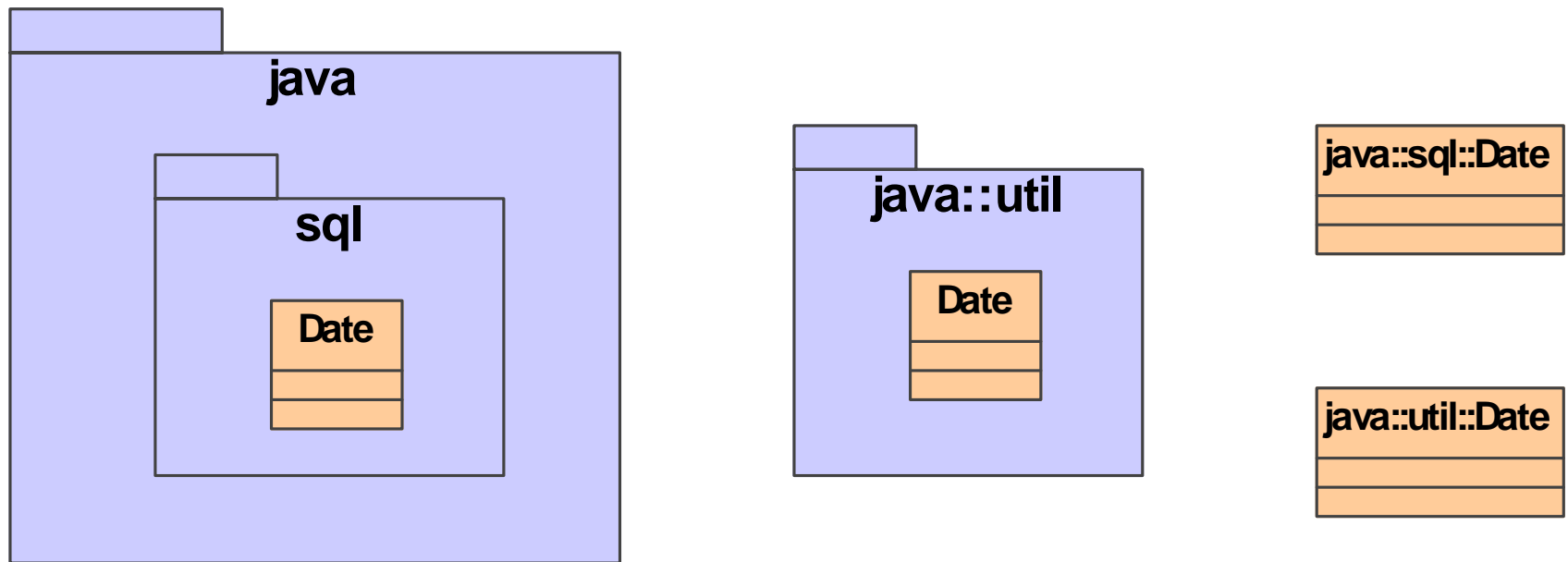
4 layers
13 connections
26 interaction paths

Multi-layer architectural frameworks

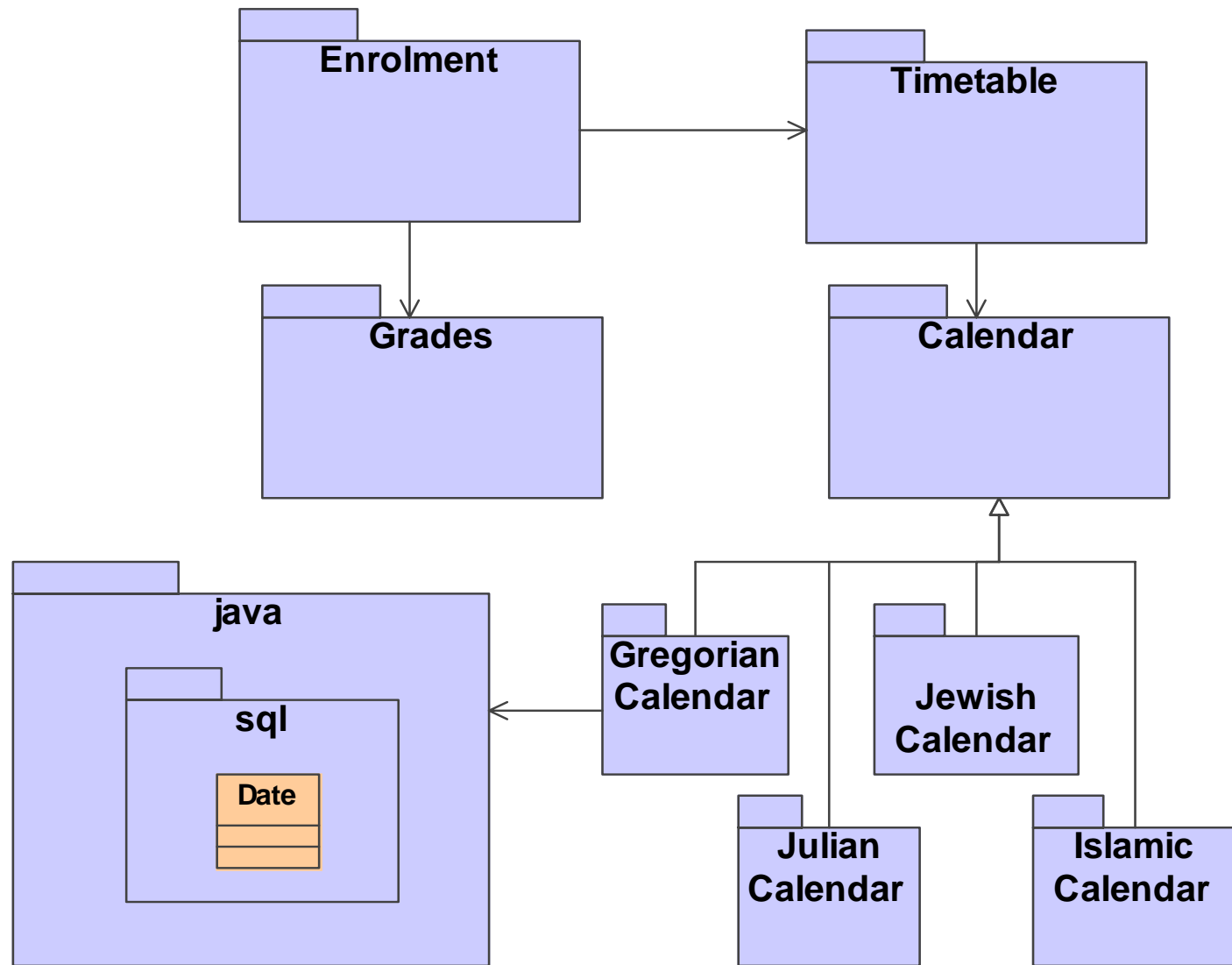


Packages

- *represent group of classes (or other modeling elements, e.g. use cases)*
- *can be nested*
- *can be related with two kinds of relationships: generalization and dependency*
- *class can only be owned by one package, but can “appear” in other packages*

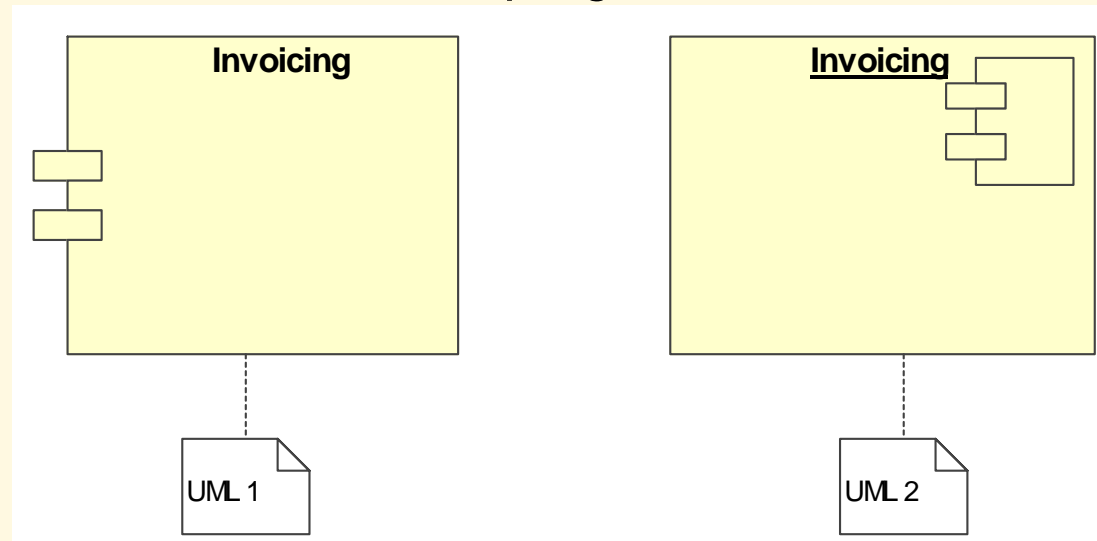


Example 6.1 (UE)



Components

- **Component** – a physical part of the system, a piece of implementation, a software program

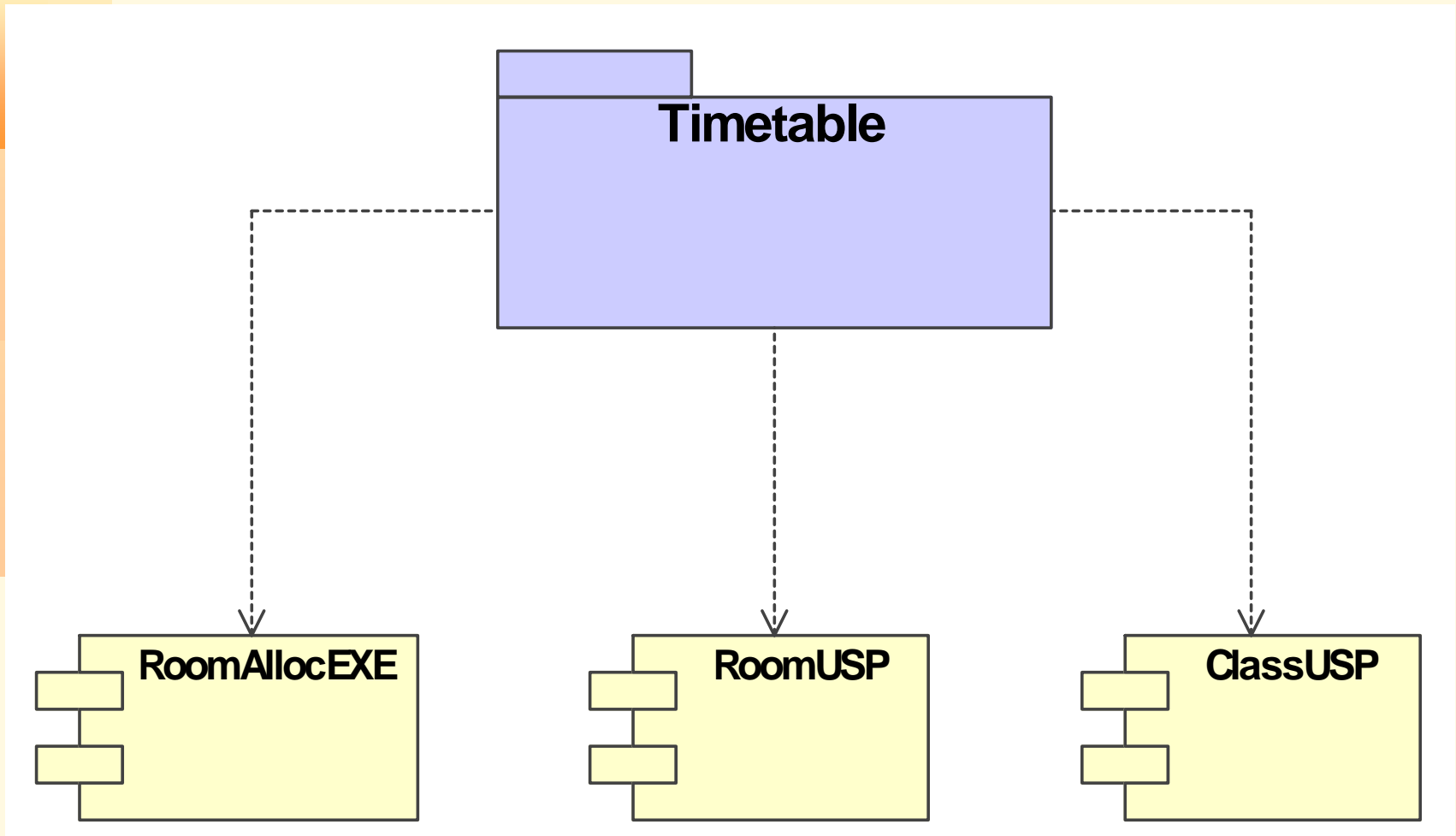


- A unit of independent deployment (never deployed partially).
- A unit of third-party composition (i.e. sufficiently documented and self-contained to be 'plugged into' other components by a third-party).
- It has no persistent state (i.e. cannot be distinguished from copies of its own).
- It is a replaceable part of a system, i.e. it can be replaced by another component that conforms to the same interface.
- It fulfills a clear function and is logically and physically cohesive.
- It may be nested in other components

Component versus package

- *At the logical level, every class belongs to a single package*
- *At the physical level, every class is implemented by at least one component*
 - *it is possible that a component implements only one class.*
 - *abstract classes defining interfaces are frequently implemented by more than one component*
- ***Packages** are typically larger architectural units than components*
 - *they tend to group classes in a horizontal way – by static proximity of classes in the application domain*
- ***Components** are vertical groups of classes with behavioral proximity*
 - *they may come from different domains but they contribute to a single piece of business activity, perhaps a use case*

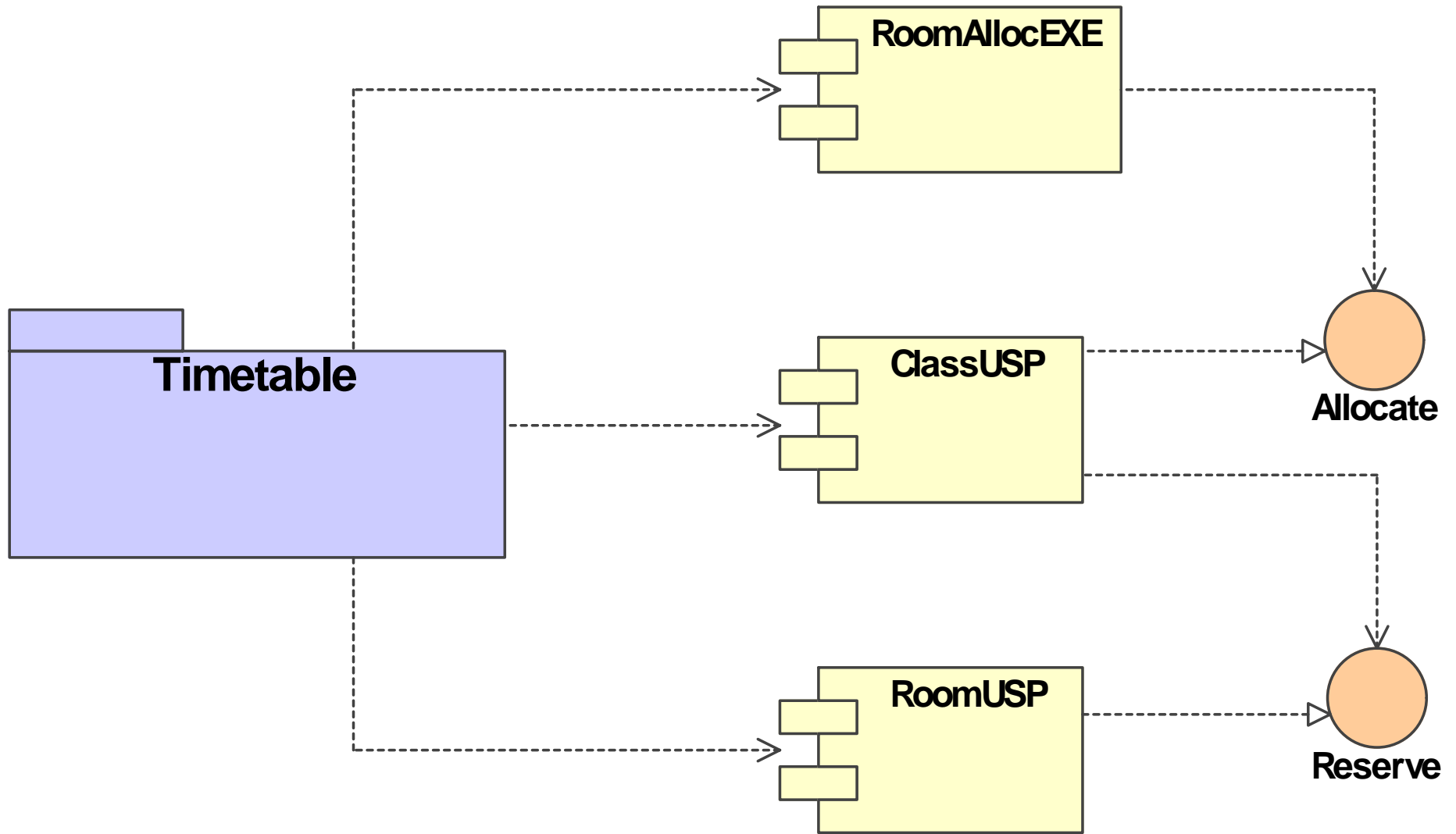
Example 6.2 (UE)



Component versus class and interface

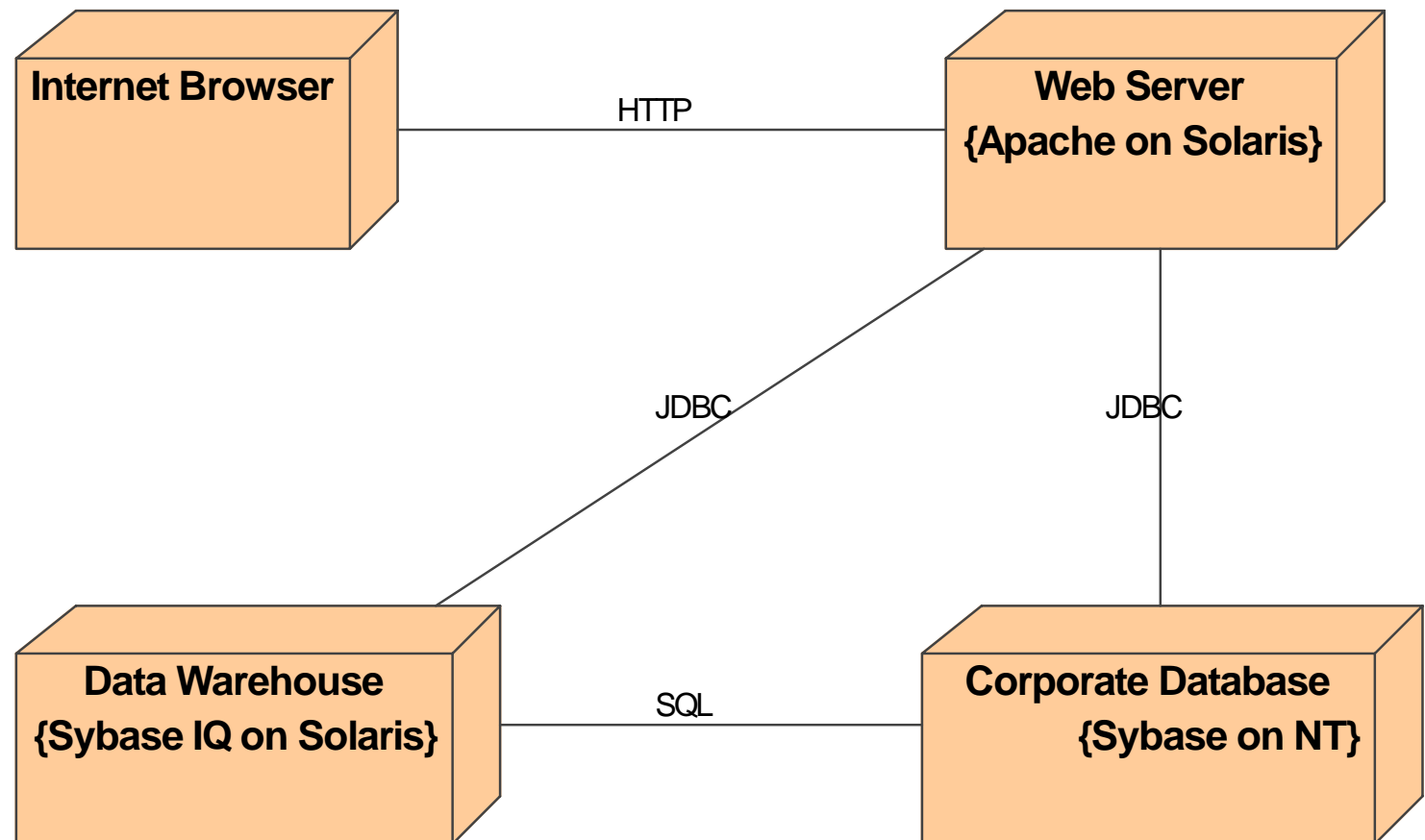
- Like classes, the components realize interfaces
- Difference 1:
 - **component** is a physical abstraction deployed on some computer node
 - **class** represents a logical thing that has to be implemented by a component to act as a physical abstraction
- Difference 2:
 - **component** reveals only some interfaces of the classes that it contains (other interfaces are encapsulated by the component and (if component is re-used) may not be even accessible to the developer, never mind the user)
 - the **interface** (a set of visible methods) that a component realizes may be implemented in a separate class → **dominant class** (dominant class represents the interface of the component)
 - **class** (as a logical abstraction) manifests all its interfaces to the developer (but not necessarily to the user)

Example 6.3 (UE) – component interfaces



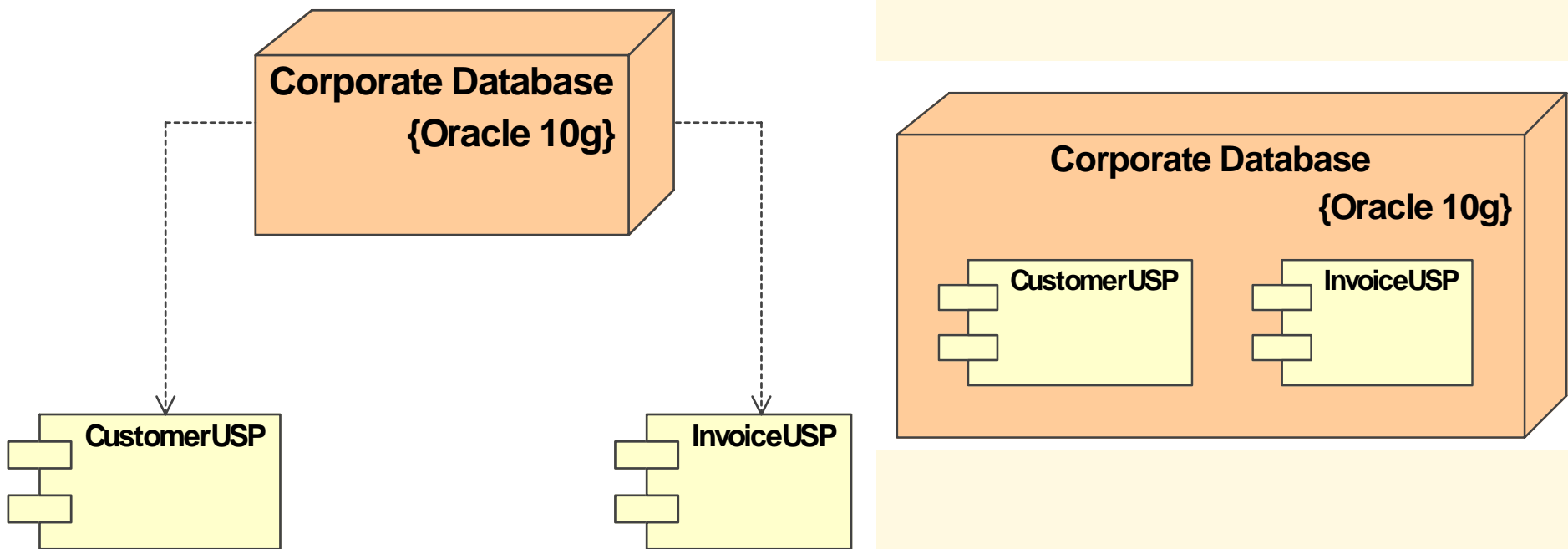
Nodes

- **Node** is a computational resource (a run-time physical object) in a **deployment diagram**
 - it has a memory and some computing capability
 - **connection relationships** link the nodes



Components and nodes

- **Nodes** are locations on which the components run (**components** are deployed on a node)
- A node together with its components is sometimes called a **distribution unit**



Program design - overview

- *System design:*
 - **architectural design** → framework
 - **detailed design** → framework's front-end (GUI) and back-end (database)
 - **program design** → extends GUI and DB design and concentrates on one application program at a time
- *Program's execution logic splits up between the client and the server processes.*

Class cohesion and coupling

- *A good program design ensures a well-balanced cohesion and coupling of classes*
- ***Class cohesion*** *is the degree of inner self-determination of the class*
 - *it measures class independence*
 - *the stronger the cohesion the better*
- ***Class coupling*** *is the degree of connections between classes*
 - *it measures the class interdependence*
 - *the weaker the coupling the better (however, the classes have to be 'coupled' to cooperate!)*

Cohesion and coupling heuristics

- *Two classes should either be not dependent on one another or one class should be only dependent on the public interface of another class.*
- *Attributes and the related methods should be kept in one class (this heuristic is frequently violated by classes who have many accessor (*get*, *set*) methods defined in their public interface).*
- *A class should capture one and only one abstraction. Unrelated information, when a subset of methods operates on a proper subset of attributes, should be moved to another class.*
- *The system intelligence should be distributed as uniformly as possible (so that classes share the work uniformly).*

Kinds of class coupling

- 1. X inherits from Y*
- 2. X has an attribute of class Y*
- 3. X has a template attribute with a parameter of class Y*
- 4. X has a method with an input argument of class Y*
- 5. X has a method with an output argument of class Y*
- 6. X knows of a global variable of class Y*
- 7. X knows of a method containing a local variable of class Y*
- 8. X is a friend of Y*

The Law of Demeter

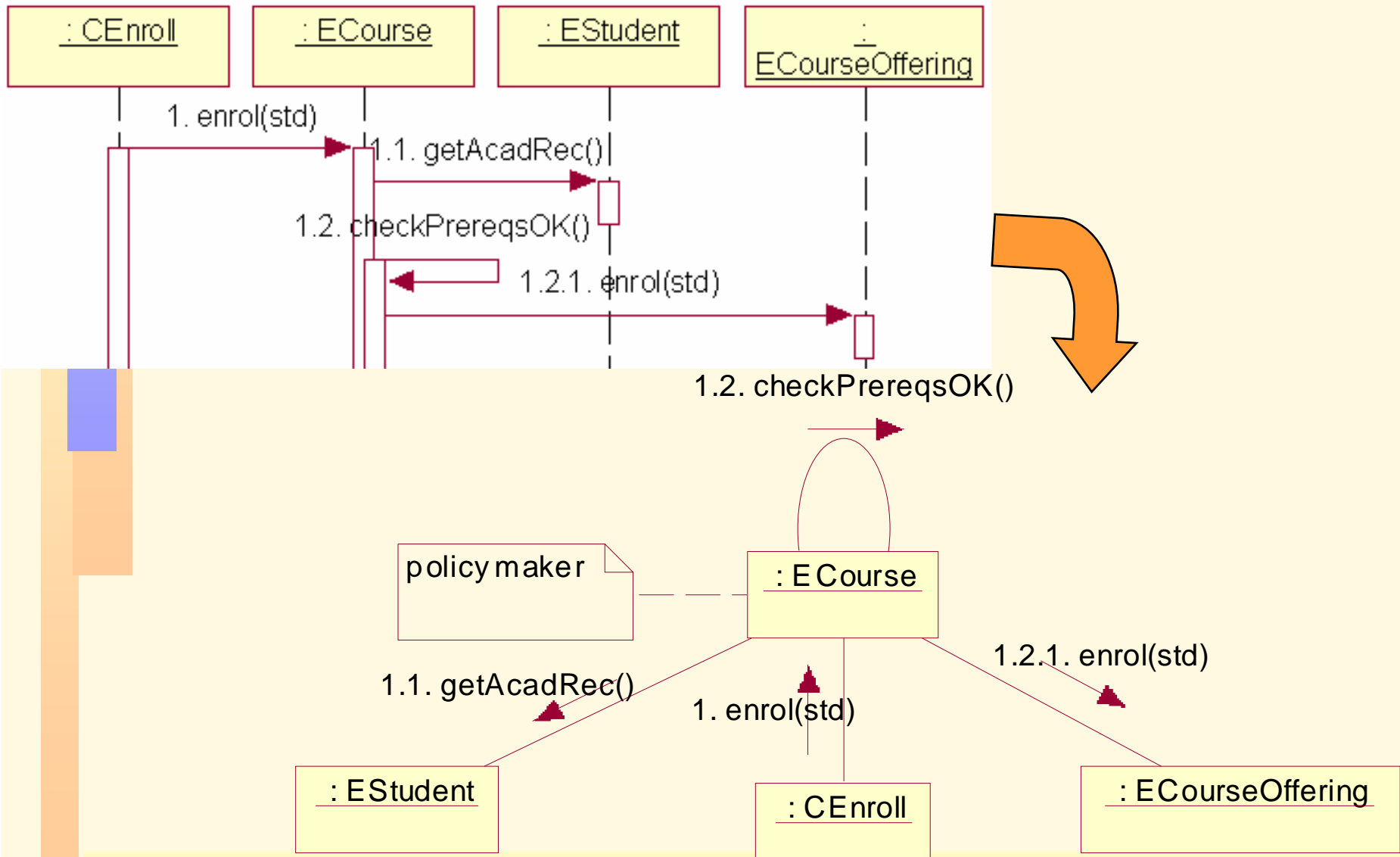
- **A target of a message** can only be one of the following objects:
 1. *The method's object itself (i.e. this in C++ and Java, self and super in Smalltalk).*
 2. *An object that is an argument in the method's signature.*
 3. *An object referred to by the object's attribute (including an object referred to within a collection of attributes).*
 4. *An object created by the method.*
 5. *An object referred to by a global variable.*

- **The Strong Law of Demeter** restricts the coupling induced by inheritance by limiting the third rule to the attributes defined in the class itself

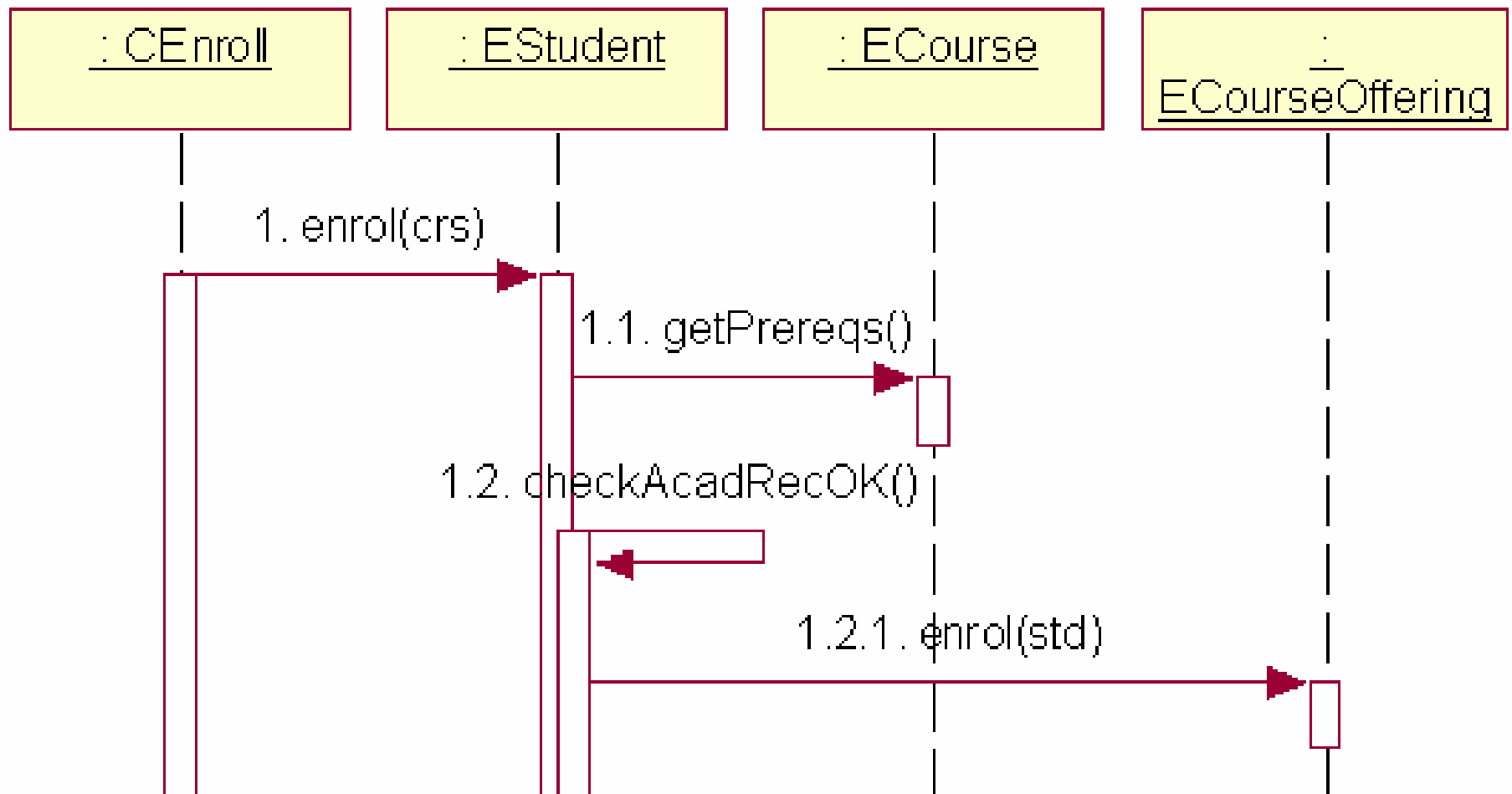
Accessor methods and mindless classes

- *A class should decide its own destiny*
 - *class can restrict other classes from accessing its own state by limiting the accessor methods in its interface*
 - ***accessor methods** define the observer (*get*) or mutator (*set*) operations*
- *A class with many accessor methods risks becoming **mindless***
- *There are situations when a class has to open up to other classes → whenever there is a need to implement a policy between two or more classes*

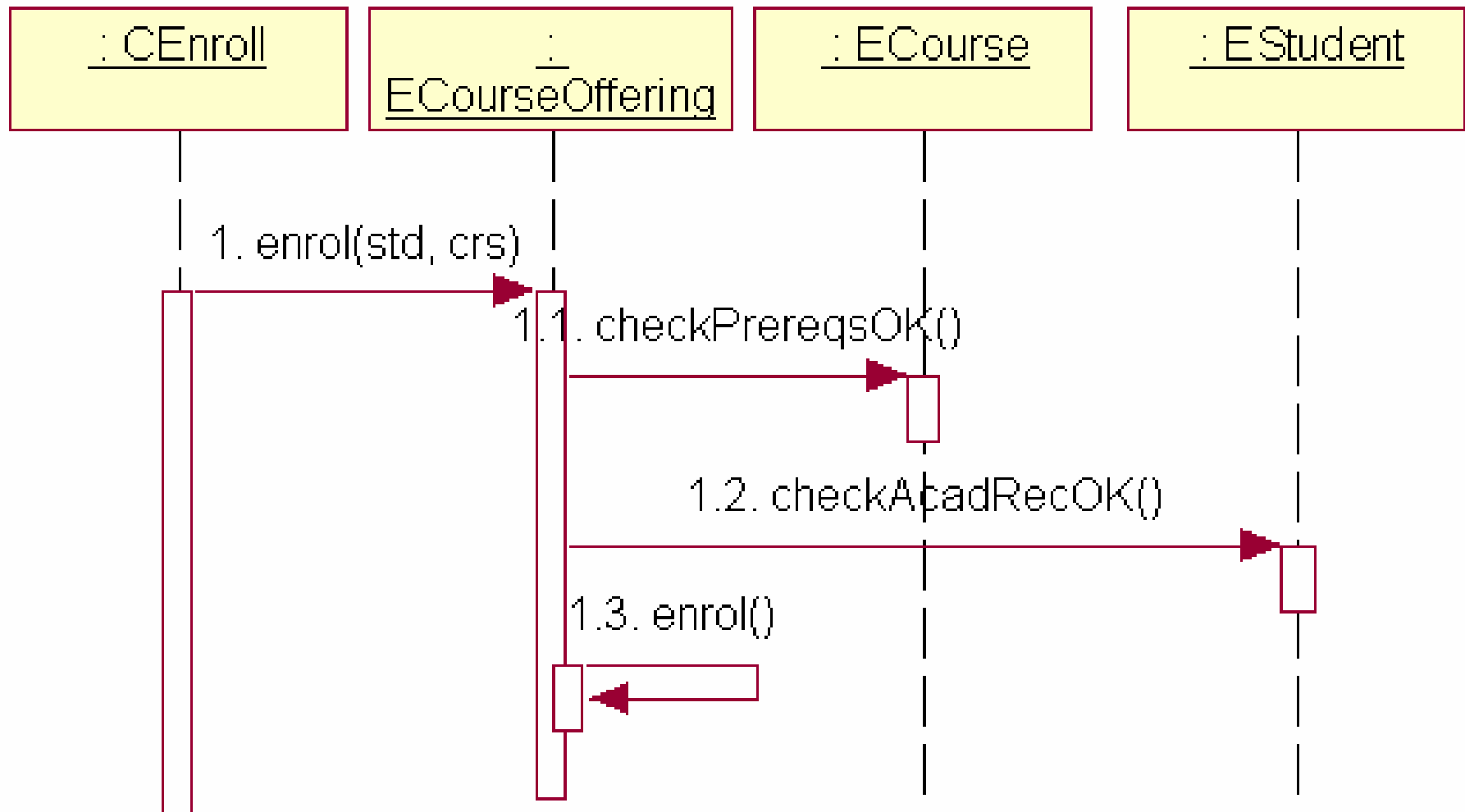
Example 6.4 (UE) – ECourse as policy maker



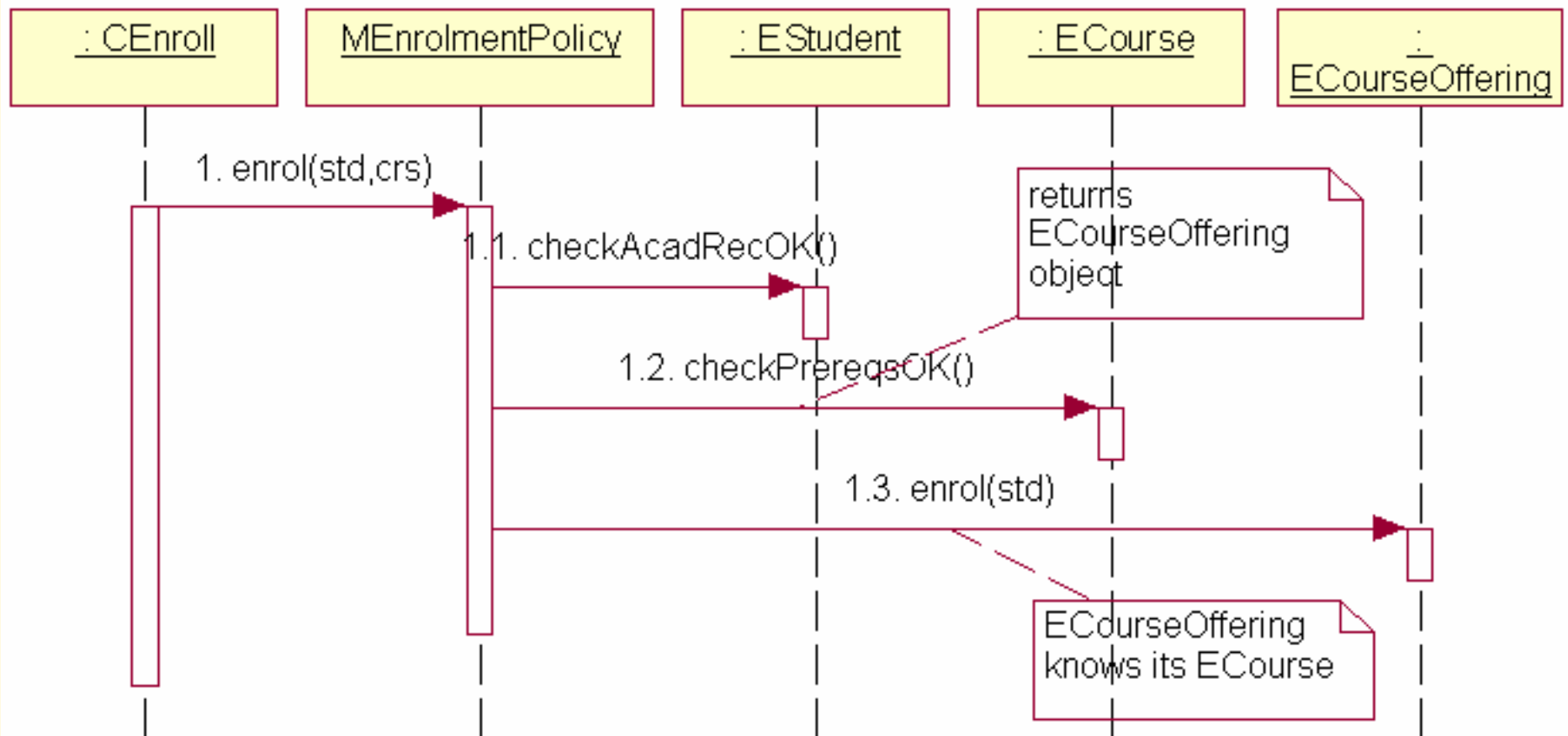
Example 6.4 (UE) – EStudent as policy maker



Example 6.4 (UE) – ECourseOffering as policy maker

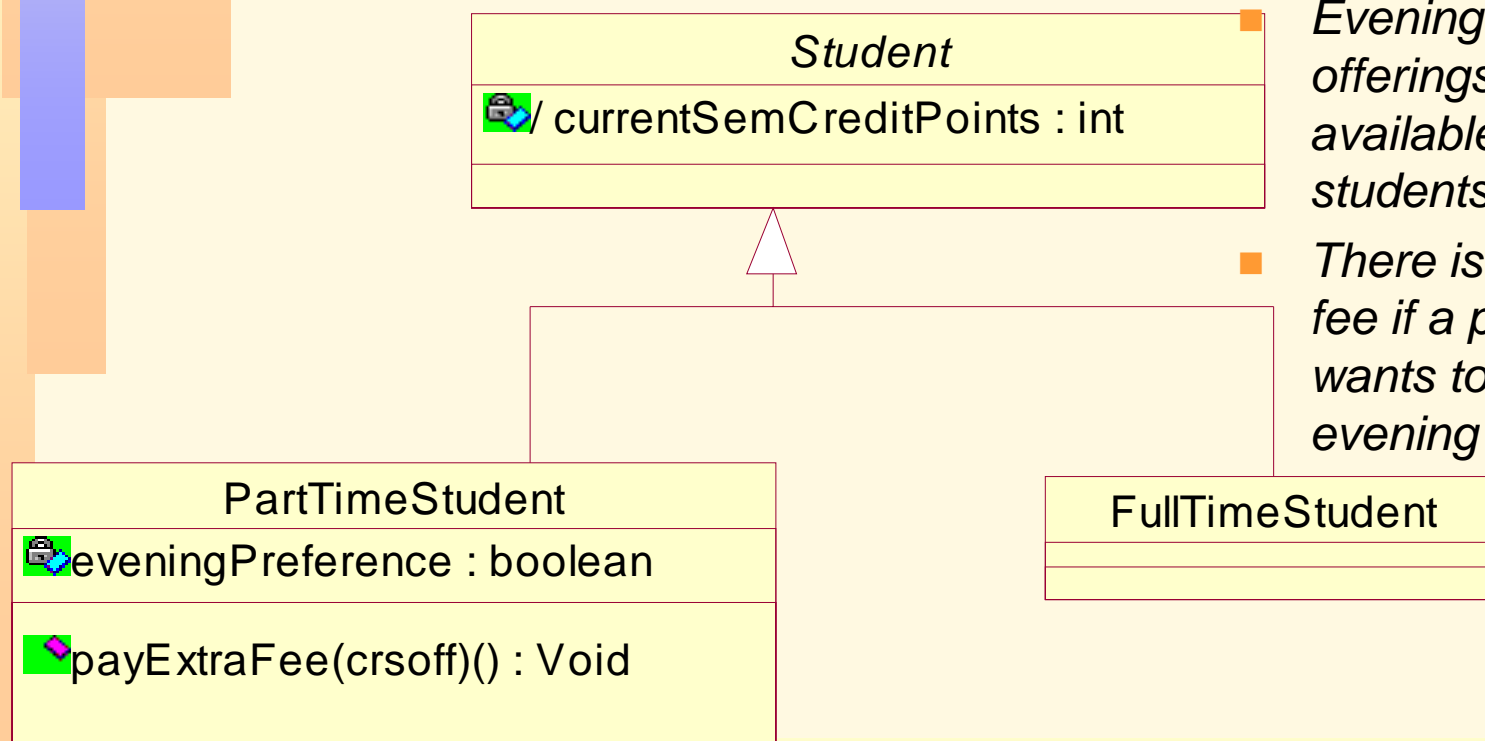


Example 6.4 (UE) – MEnrolmentPolicy as policy maker



Dynamic classification and mixed-instance cohesion

- **dynamic classification** (if object can dynamically change its class) not supported → **mixed-instance cohesion**
 - a class with mixed-instance cohesion has some features that are undefined for some objects of the class
 - partial solution is to extend generalization hierarchy by identifying new subclasses

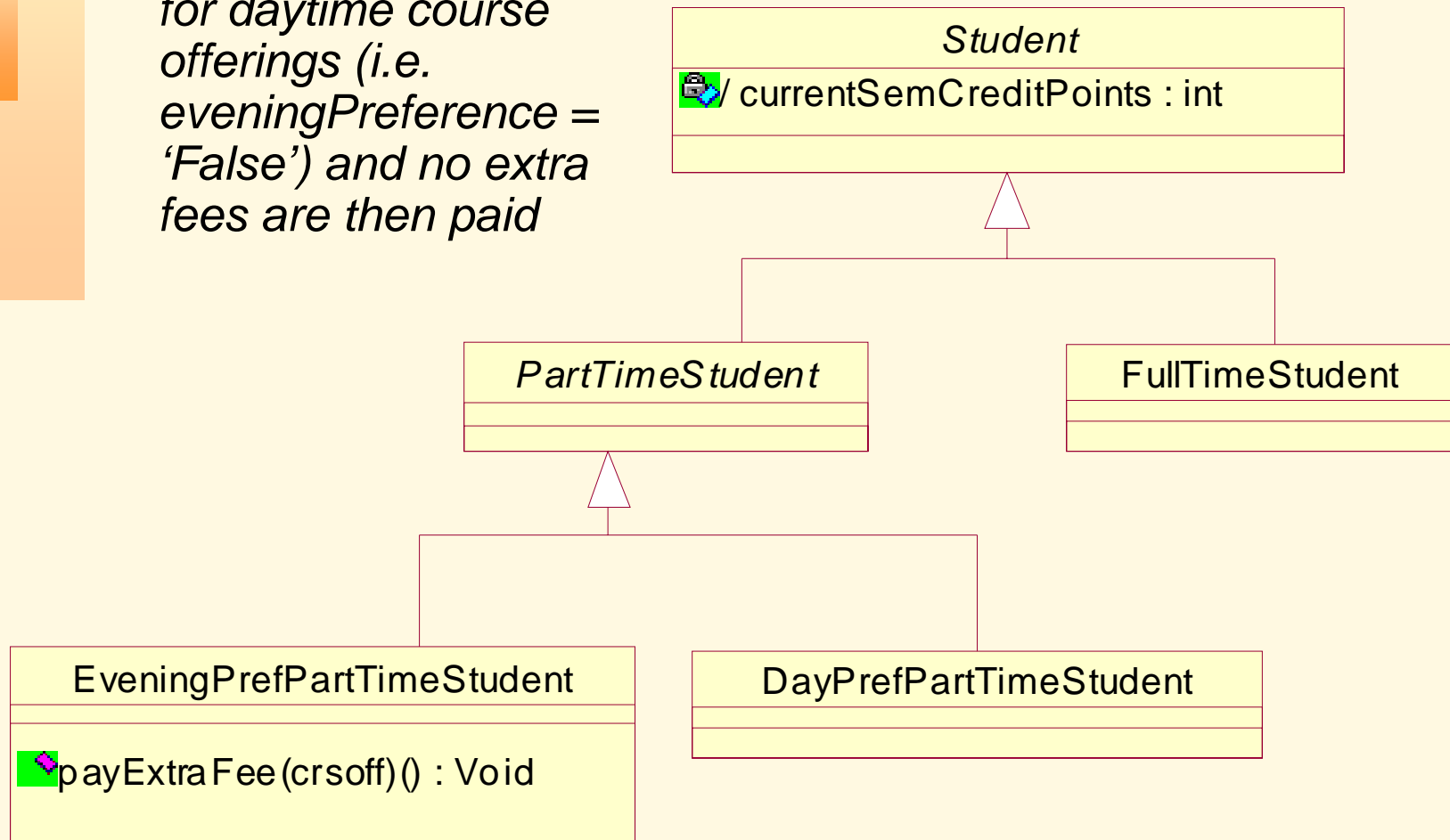


Evening course offerings are only available to part-time students.

- There is a small extra fee if a part-time student wants to enroll in an evening course offering.

Further reduction of mixed-instance cohesion

- A part-time student may have a preference for daytime course offerings (i.e. `eveningPreference = 'False'`) and no extra fees are then paid



Reuse strategy

- *Reuse granularity:*
 - *the class*
 - *the component*
 - *the solution idea*

- *Reuse strategies*
 - *Toolkits (class libraries).*
 - *Frameworks.*
 - *Analysis and design patterns.*

Toolkit reuse

- A toolkit emphasizes code reuse at a class level
 - the programmer 'fills the gaps' in the program by making calls to concrete classes in some library of classes
 - the main body of the program is not reused – it is written by the programmer
- Two kinds (levels) of toolkit:
 - **Foundation toolkits**
 - the foundation classes provided by object programming environments (e.g. collections (such as Set, List or Index))
 - **Architecture toolkits**
 - available as part of a system software (e.g. object database system)

Framework reuse

- A framework emphasizes design reuse at a component level
 - framework provides the skeleton of the program
 - the programmer then ‘fills the gaps’ in this skeleton (customizes it) by writing the code that the framework needs to call
- A framework is a customizable application software (e.g. ERPS (Enterprise Resource Planning Systems) such as SAP, PeopleSoft, Baan, or J.D. Edwards)
- Object-oriented frameworks for IS development (“business objects”) are proposed within distributed component technologies such as CORBA, DCOM and EJB

Pattern reuse

- *A **pattern** is a documented solution that has been shown to work well in a number of situations*
- *Patterns emphasize application of reuse to the development approach*
 - *represent good development practices*
 - *can apply to the analysis or design phase (hence, analysis patterns and design patterns).*

Behavioral and structural collaboration

- *In passing, we used the term **collaboration** freely, and exchangeably with the term **interaction**, to refer to sets of objects collaborating to perform a task*
- **Collaboration** – *a specification for the realization of use cases and operations:*
 - ***structural collaboration** represented by a class model and*
 - ***behavioral collaboration** represented by an interaction model (sequence or collaboration diagram)*
- *To specify the realization of a use case (or operation), we need to have prior detailed description of that use case (or operation) → **use case document***

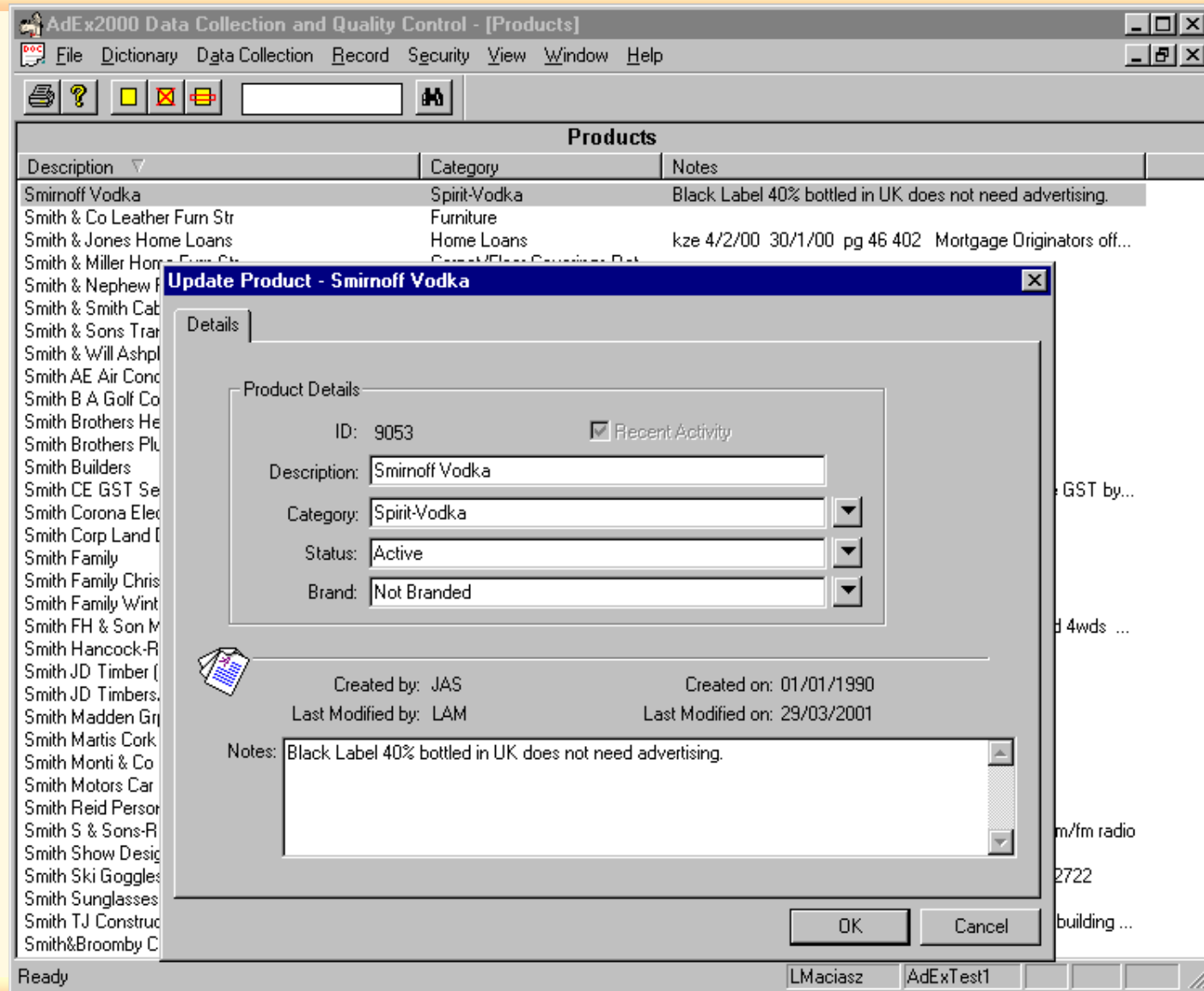
Use case document - example

Use Case	<i>CRUD Product</i>
Brief Description	This use case enables the user of the AE system to maintain information about products. It includes facilities to view (read) a list of products, create a new product, delete a product and update the information about a product.
Actors	Data Collection Employee, Data Verification Employee, Valorization Employee, Reporting Employee
Preconditions	The actor possesses system privileges to maintain products. Any Employee can view a list of products. Only Data Collection and Data Verification Employees can create, update or delete products.
Main Flow	<i>1.Basic Flow</i> This use case starts when an Employee chooses to work with Products by selecting the Maintain Products option of the AE system. The system retrieves and displays the following information for all products in a browse window. The <u>Read Products subflow</u> is performed.

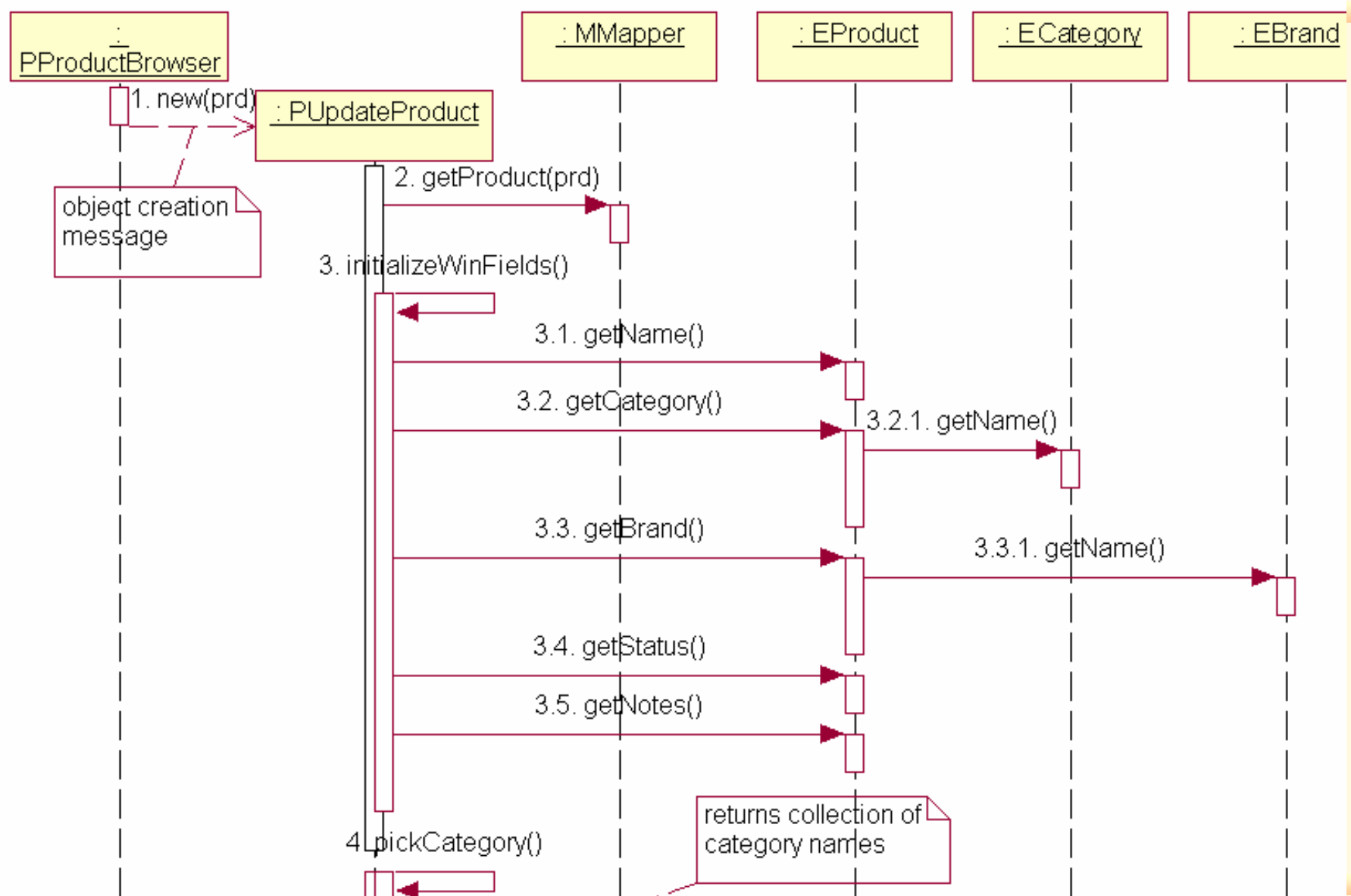
Use case document – example (cont.)

Use Case	<i>CRUD Product</i>
Main Flow	<p><i>1.Basic Flow</i></p> <p><i>2.Subflows</i></p> <p><i>2.1 Read Products</i></p> <p>The following information is displayed in the AE Row Browser window: <code>product_name</code>, <code>category_name</code>, <code>notes</code>, <code>created_by</code>, <code>last_modified_by</code>, <code>created_on</code>, <code>last_modified_on</code>.</p> <p>The information is displayed in a tabular (columns and rows) view, with vertical and horizontal scroll bars, if necessary.</p> <p>The display is named <code>Products</code> and all columns are named.</p> <p>...</p>
Alternative Flows	<p><i>AF1</i> The system will not allow creating/updating a product with <code>product_name</code> that already exists in the database.</p> <p><i>AF4</i> The system will not allow opening any two update/delete dialog box windows for the same product by more than one user.</p>
Postconditions	<p>After a product is successfully created/updated, the browser window highlights the row with that product information.</p> <p>After a product is successfully deleted, the browser window is refreshed and highlights the first visible row.</p>

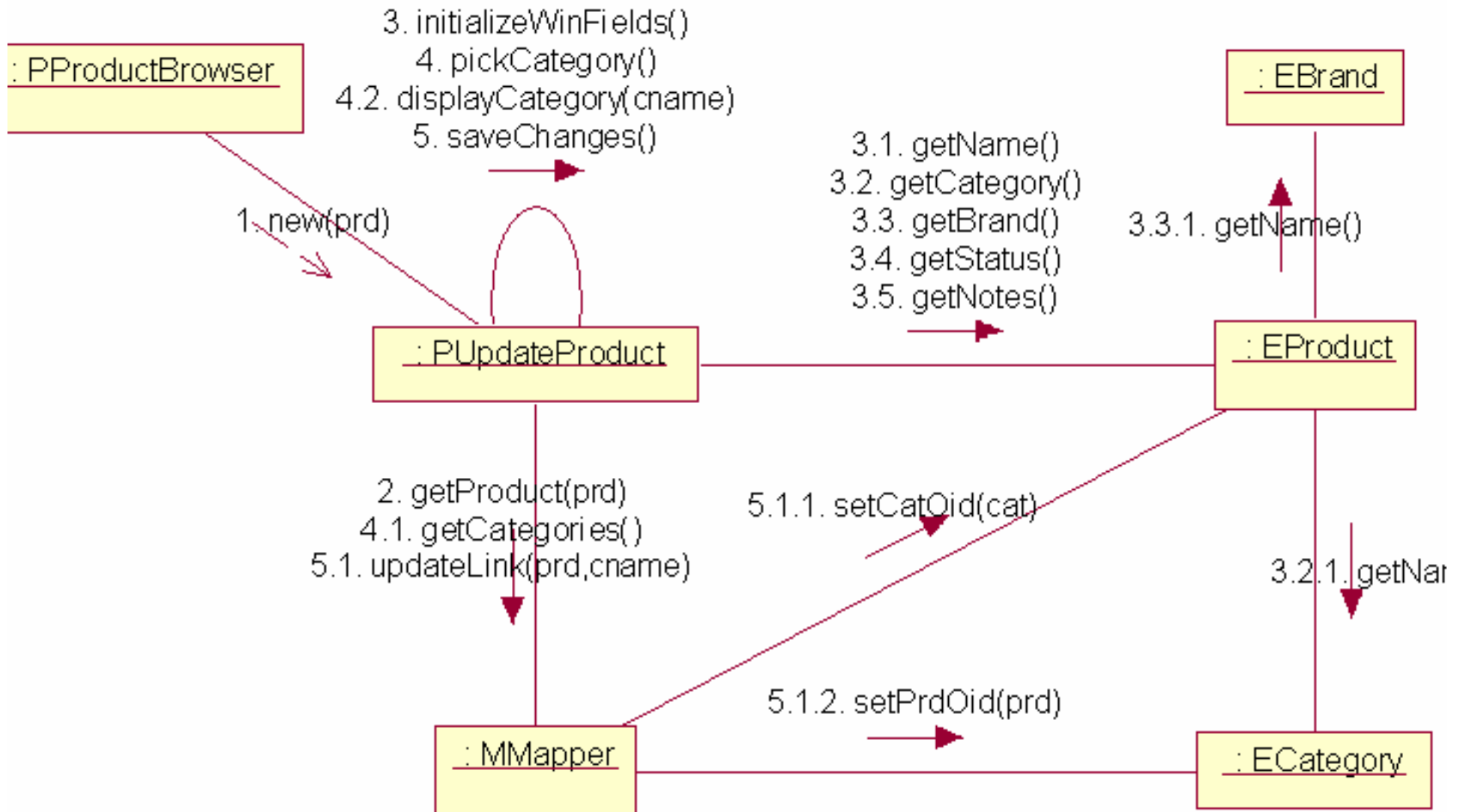
Example 6.7 (AE) - behavioral collaboration



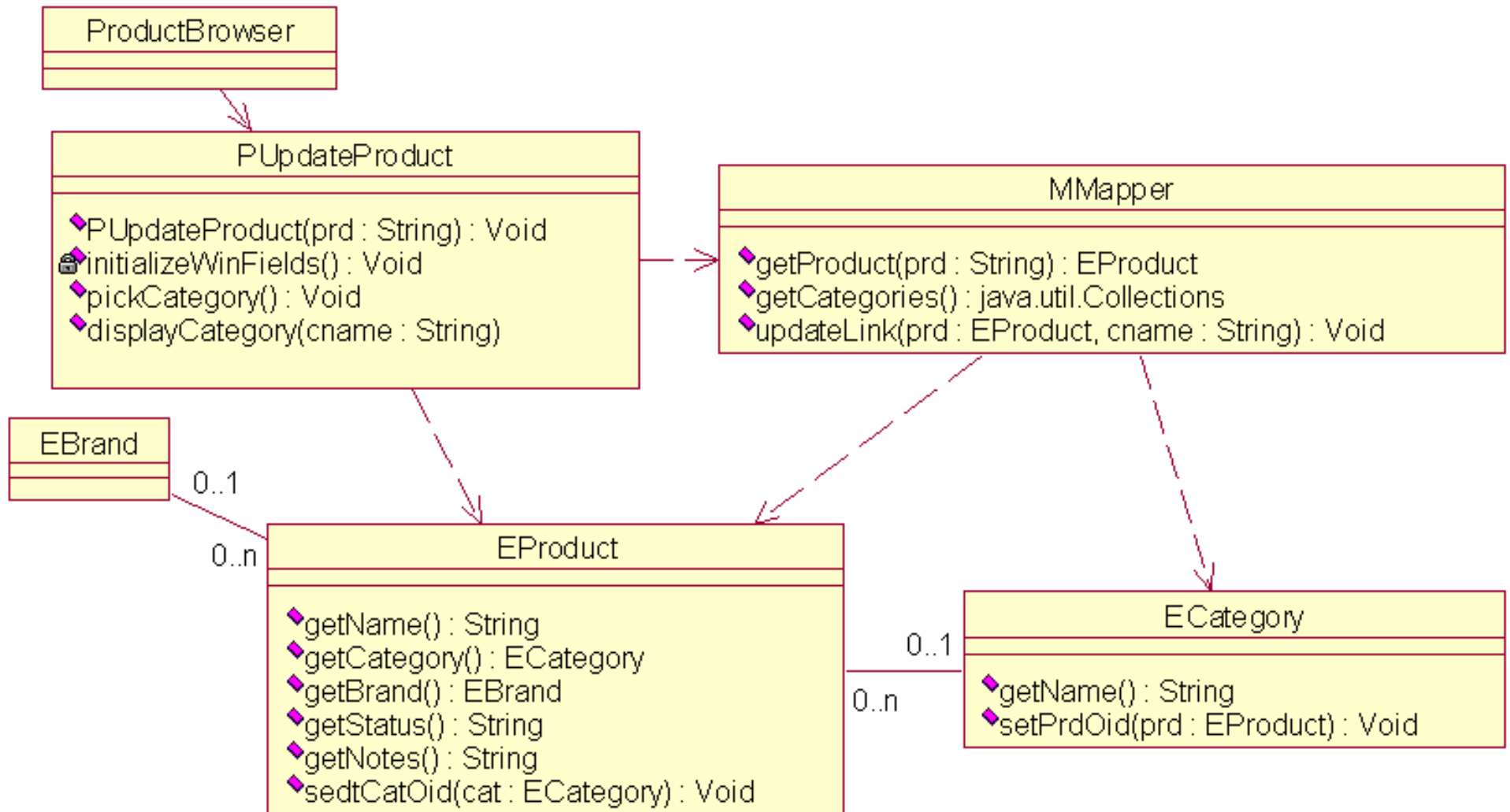
Example 6.7 (AE) - behavioral collaboration



Example 6.7 (AE) - behavioral collaboration



Example 6.8 (AE) - structural collaboration



Summary

- *Typical IS applications are based on the **client/server** architectural principle*
- *Proper structuring of classes in layers (packages, subsystems), organized according to the PCMEF or similar **framework**, is an important **architectural** objective*
- ***Architectural modeling** encompasses allocation of software elements (classes, interfaces, etc.) into packages, components and nodes*
- *A well-designed program maximizes the class **cohesion** while minimizing the class **coupling***
- ***Reuse** is a major design consideration affecting the architectural as well as the detailed design issues*
- *The detailed design concentrates on **collaborations***
 - ***Structural** aspects of collaboration are modeled in class diagrams*
 - ***Behavioral** aspects in sequence and/or collaboration diagrams*