Object Oriented Design - OOD

- La fase di OOD si suddivide in:
  - ***OOD preliminare*** (o **architetturale**, o **di sistema**): definisce la strategia ad alto livello per risolvere il problema e costruire una soluzione. Le decisioni riguardano l'organizzazione complessiva del sistema (*architettura di sistema*)
  - ***OOD dettagliato*** (o **degli oggetti**): definisce in modo completo le classi e le associazioni usate in fase di codifica, nonché le strutture dati e gli algoritmi dei metodi necessari per la codifica delle operazioni.
- Secondo l'approccio *iterativo* ed *incrementale* allo sviluppo del software, il modello di OOA si "*trasforma*" in modello di OOD quando si aggiungono dettagli tecnici relativi alle **soluzioni hardware/software** che si usano per definire ***come*** viene realizzato il sistema

Architettura di sistema

- Una **architettura di sistema** definisce la struttura delle componenti di un sistema software, le relazioni che intercorrono tra tali componenti ed i principi che ne governano il progetto e l'evoluzione nel tempo
- Evoluzione delle architetture di sistema:
  1. Architetture a mainframe
  2. Architetture a condivisione di file
  3. Architetture client/server (C/S):
     - 3.1 two-tier (*thin client, fat client*)
     - 3.2 three-tier (*upper layer, middle layer, bottom layer*)
  4. Architetture ad oggetti distribuiti
  5. Architetture *component-based*
- Da 3 a 5 si parla di **architetture distribuite**, ovvero architetture di *sistemi software distribuiti*

Sistemi software distribuiti

- In un **sistema software distribuito** l'elaborazione è distribuita su una collezione di calcolatori indipendenti connessi tramite una *infrastruttura di rete* (locale o geografica)
- La collezione di calcolatori indipendenti appare agli utenti del sistema come un unico calcolatore
- Nella transizione da architetture a mainframe ad architetture C/S hanno giocato un ruolo chiave le *tecnologie middleware*
- Con il termine **middleware** ci si riferisce allo strato software di connettività, interposto tra il sistema operativo e le applicazioni, che consiste di un insieme di servizi che permettono l'interazione di processi applicativi multipli eseguiti su uno o più calcolatori attraverso una infrastruttura di rete (es. *TP monitor*, *RPC*, *MOM*, *ORB*)

Caratteristiche dei sistemi distribuiti

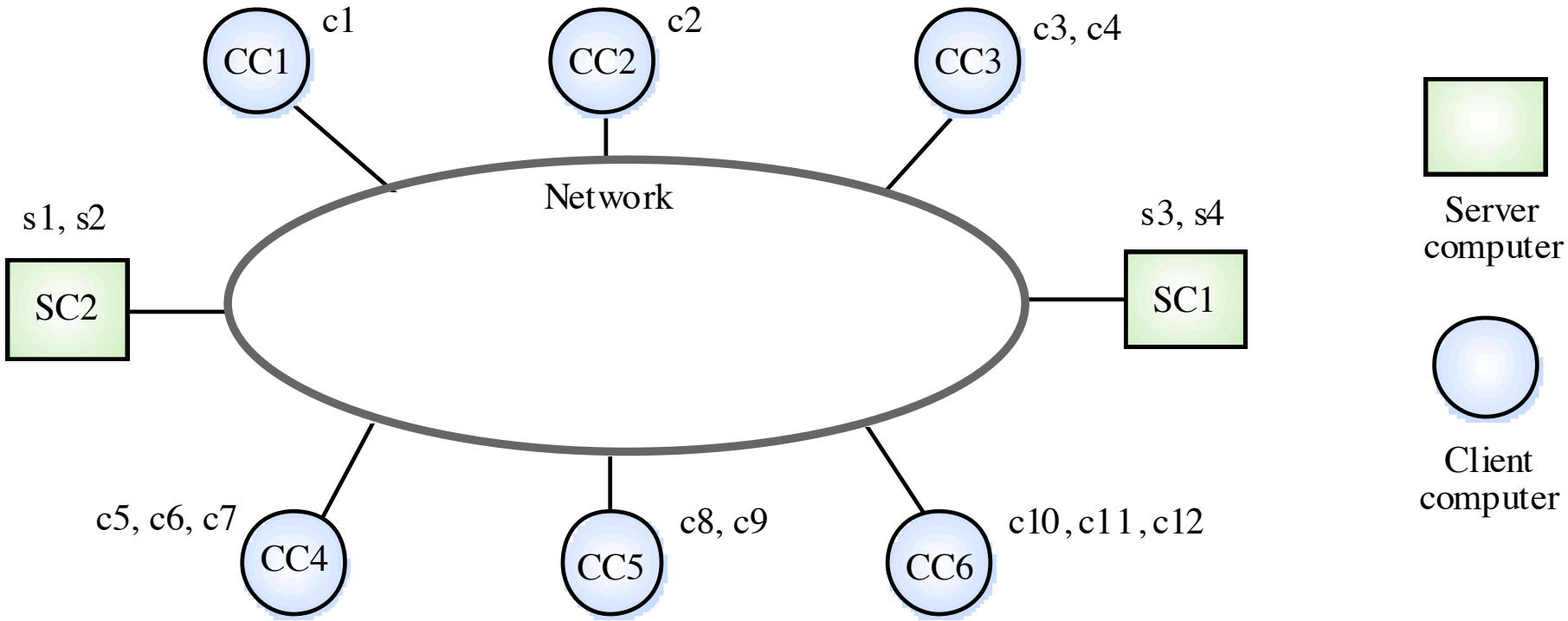
- Condivisione di dati e risorse
- *Openness* (gestione di risorse eterogenee)
- Concorrenza
- Scalabilità
- Bilanciamento del carico di lavoro
- *Fault-tolerance*
- Adattabilità agli scenari attuali per l'*enterprise computing*
- Trasparenza
- Fattori critici
  - qualità del servizio (*performance*, disponibilità, ecc.)
  - interoperabilità
  - sicurezza

Architetture client/server - C/S

- Il **client** è il processo che interagisce con l'utente, secondo le seguenti caratteristiche:
  - fornisce un'interfaccia per il colloquio con il sistema
  - sottopone uno o più richieste (in un linguaggio predefinito) al server
  - comunica con il server facendo uso di tecnologia *middleware*
  - presenta all'utente risultati restituiti dal server
- Il **server** è il processo (o l'insieme di processi residenti sullo stesso calcolatore) che fornisce servizi ai client con le seguenti caratteristiche:
  - fornisce servizi rispondendo alle richieste provenienti dal client (il server non inizia la conversazione con il client)
  - nasconde l'intero sistema C/S al client ed all'utente (un server può, ad un dato istante, diventare client di un altro server, senza che il client che l'ha invocato ne abbia conoscenza)

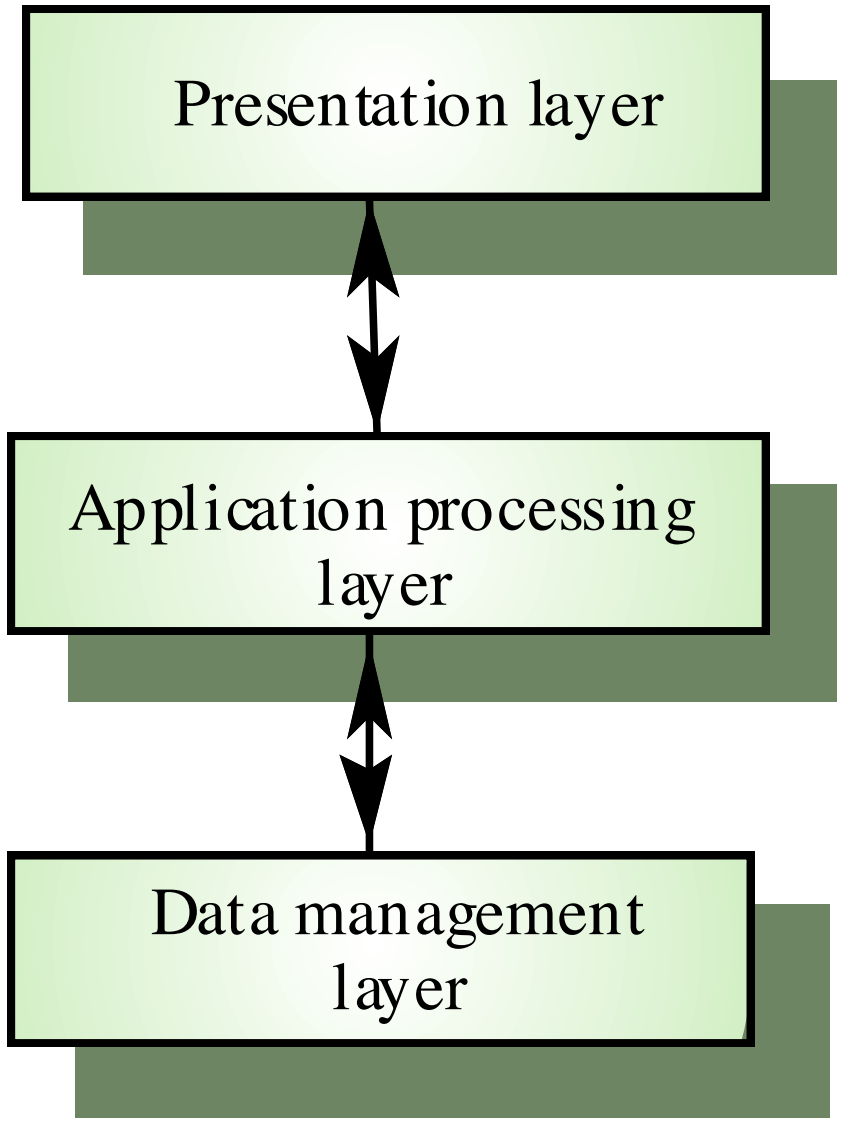
Architetture client/server - C/S (2)

Un'**architettura C/S** divide un'applicazione in processi separati operanti o sullo stesso calcolatore o su calcolatori distinti connessi mediante una infrastruttura di rete



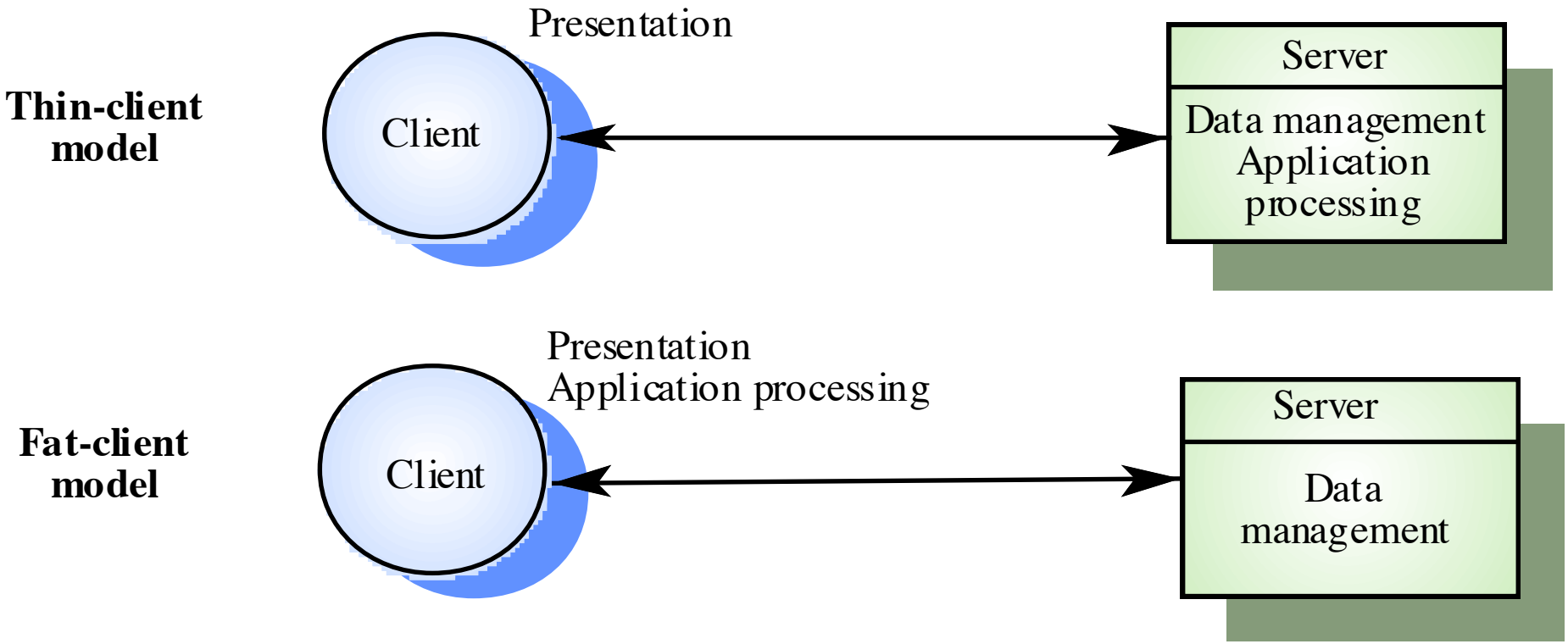
Application layers

- *Presentation layer*: gestisce la raccolta delle richieste utente e la presentazione dei risultati forniti dal sistema software
- *Application processing layer*: incapsula la logica mediante cui il sistema software realizza specifiche funzionalità
- *Data management layer*: gestisce l'accesso ai dati del sistema software



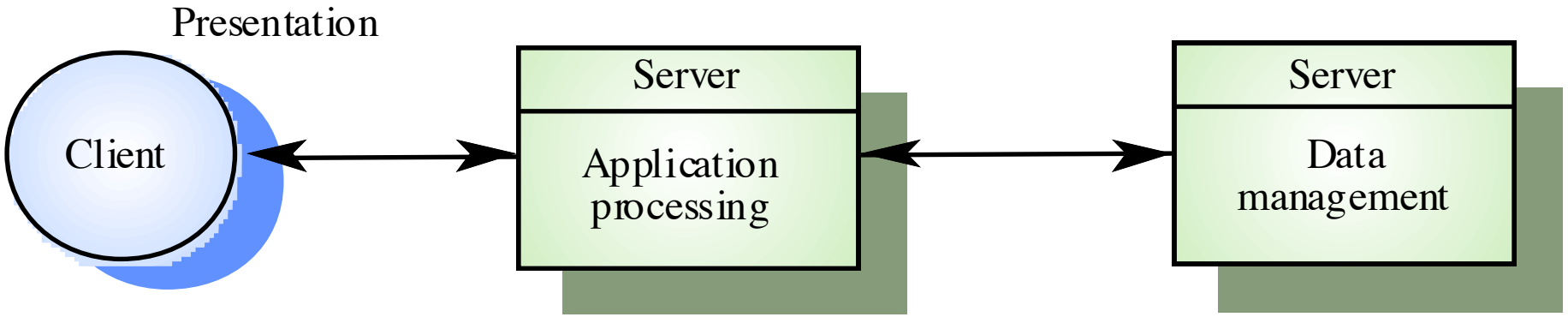
Architetture C/S *two-tier*

- modello **thin-client**: le funzioni di *application processing* e *data management* sono svolte esclusivamente dal server; il client viene usato solo per gestire l'interazione con l'utente
- modello **fat-client**: il server è responsabile del solo *data management*, mentre il software lato client realizza sia le funzioni di interazione con l'utente che le funzioni di *application processing*



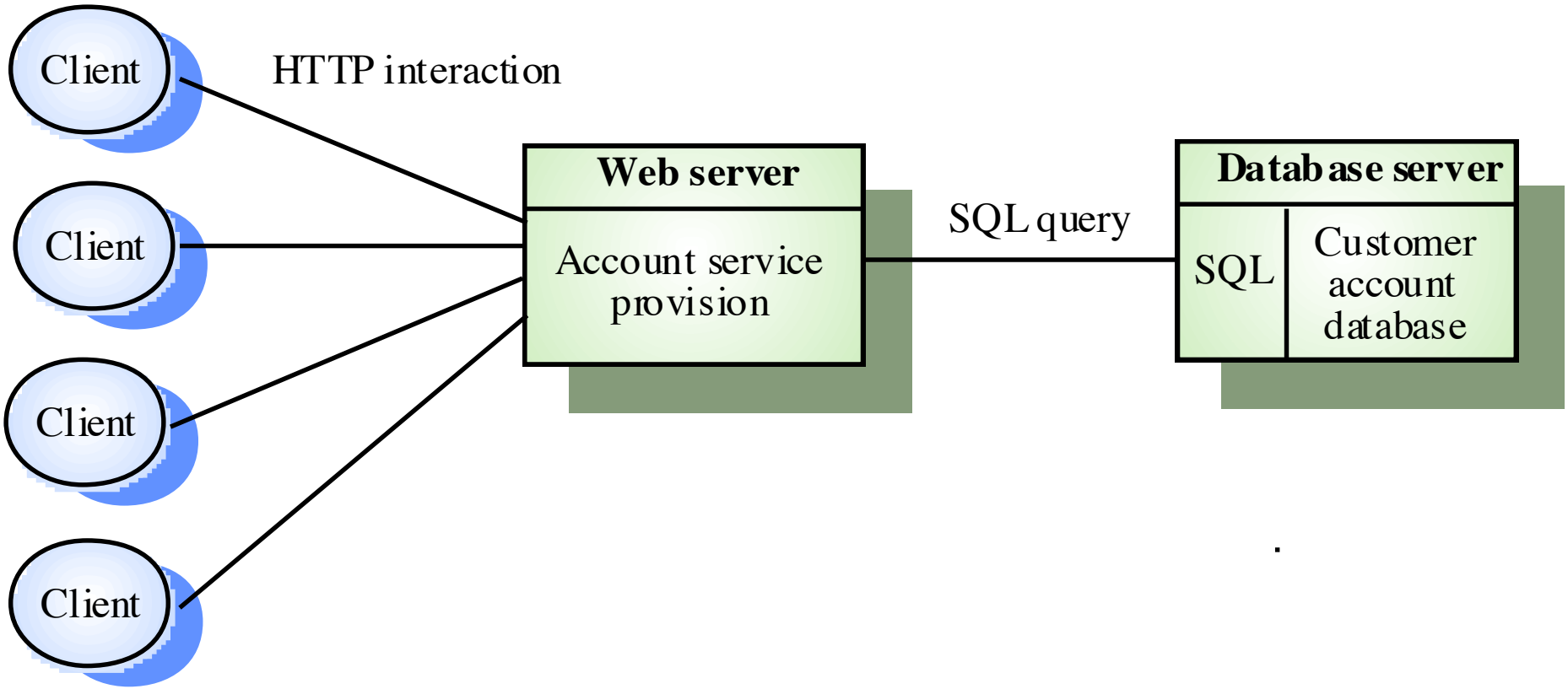
Architetture C/S *three-tier*

- In una architettura C/S *three-tier* i tre strati applicativi vengono eseguiti da processi distinti
- Rispetto alle architetture C/S *two-tier* si ottengono miglioramenti in termini di:
  - performance
  - flessibilità
  - manutenibilità
  - riusabilità
  - scalabilità



Esempio di architettura C/S *three-tier*

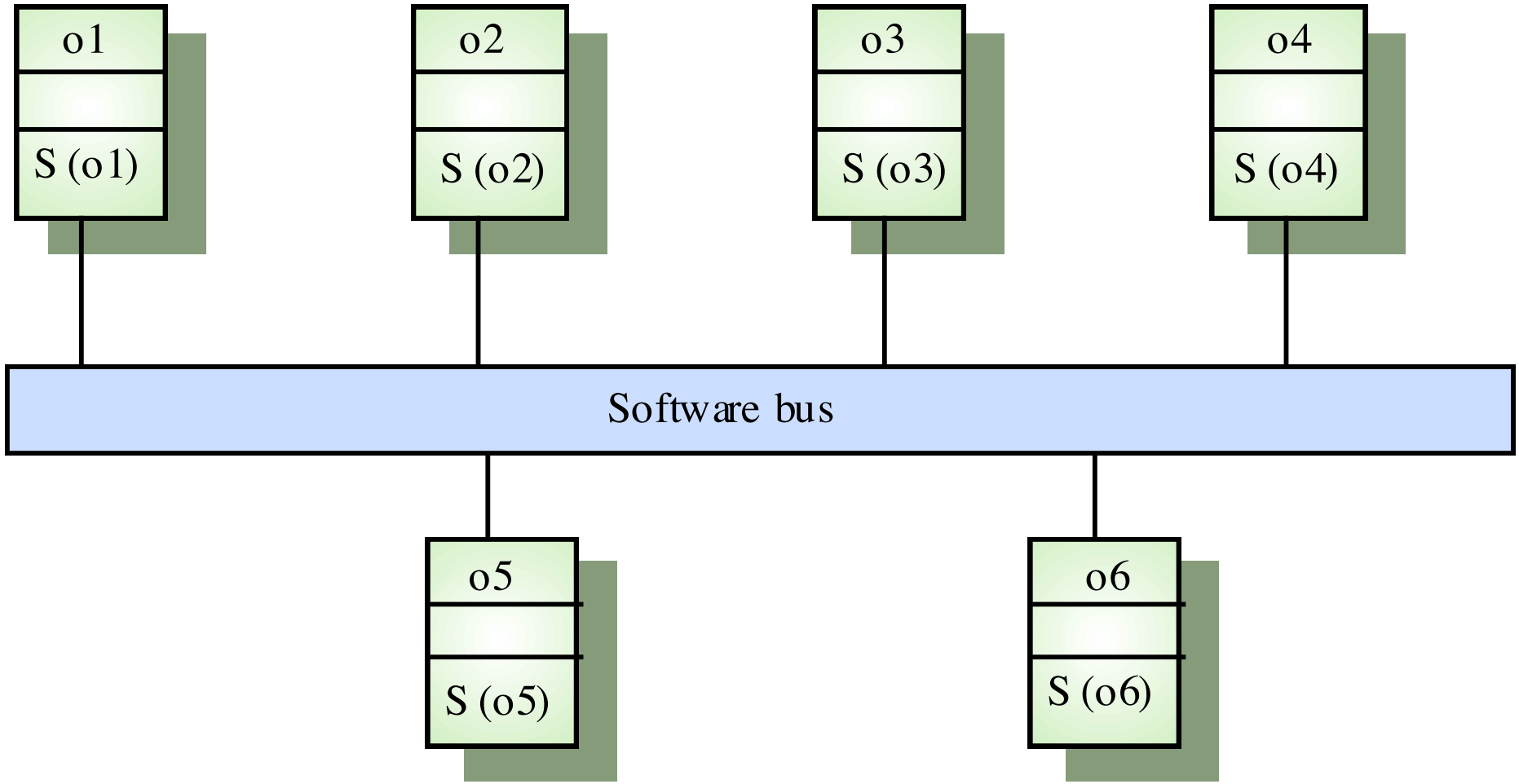
Internet Banking System



Architetture ad oggetti distribuiti

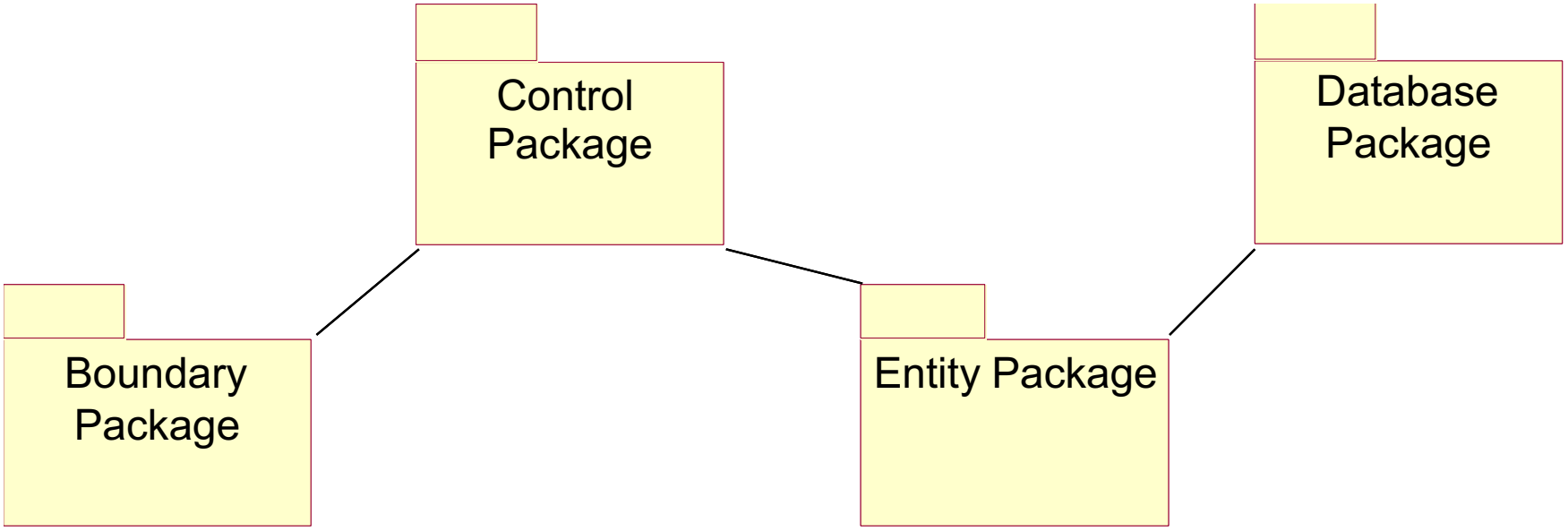
- In una architettura ad oggetti distribuiti non c'è distinzione tra client e server
- Ogni oggetto distribuito si comporta sia come *client* che come *server* (richiede e fornisce servizi agli altri oggetti del sistema software)
- La comunicazione remota tra gli oggetti distribuiti viene resa trasparente mediante l'utilizzo di tecnologia *middleware* a *software bus* (detto *object request broker*):
  - ***abstract bus**:* specifica dell'interfaccia mediante cui vengono forniti i servizi di comunicazione e scambio dati (modello dei tipi dei valori scambiati e modello di trasferimento del controllo)
  - ***bus implementation**:* realizzazione dell'abstract bus su una specifica piattaforma HW/SW ($\Rightarrow$ *separazione tra interfaccia ed implementazione*)
- Le applicazioni sono costituite da oggetti che risiedono su piattaforme distribuite ed eterogenee e comunicano mediante invocazione remota di metodi

Es. di architettura ad oggetti distribuiti



Approccio BCED

- Estensione dell'approccio BCE
- Introduce il **Database Package**, che raggruppa le classi che gestiscono l'interfaccia verso il DB (*native database interface*, ODBC *driver*, JDBC *driver*) mettendo a disposizione le operazioni per gestire l'accesso ai dati del DB da parte degli oggetti appartenenti a classi dell'*Entity Package*



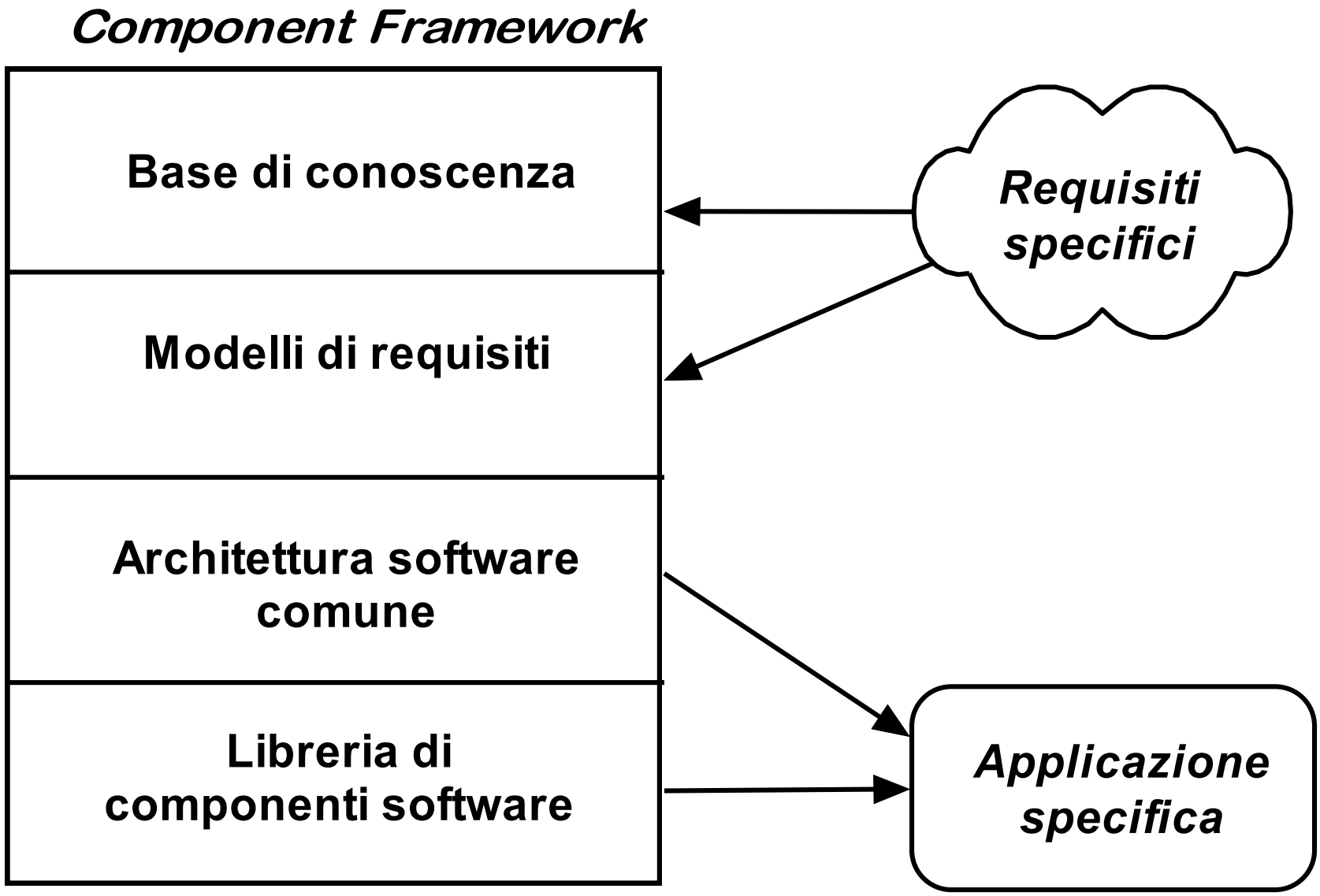
Architetture *component-based*

- Le **architetture *component-based*** definiscono sistemi software configurati a partire da *componenti software* progettati per lavorare insieme come facenti parte di un *component framework*
- Il *component framework* fa uso di una architettura software comune per formalizzare una determinata classe di applicazioni
- I sistemi software *component-based* nascono al fine di supportare lo sviluppo efficiente di sistemi software i cui requisiti abbiano elevate caratteristiche di *variabilità*
- E' quindi necessario identificare e realizzare *astrazioni software* che forniscano soluzioni efficienti ed affidabili atte a garantirne una cooperazione efficace
- Tali astrazioni, o *componenti*, possono essere usate per applicazioni differenti e adattate (*riconfigurate*) quando cambiano i requisiti

Architetture *component-based* (2)

- La chiave per costruire sistemi software basati su componenti è dunque il ***riuso black-box del software***
- Ogni componente mette a disposizione un insieme di parametri (o «plug») per:
  - effettuare il collegamento ad altri componenti, secondo regole fissate dette *regole di composizione*
  - ottenere il comportamento desiderato assegnando valori ai parametri del componente (*binding*), invece di adattare la struttura interna di un elemento software per modificarne le funzionalità (riuso *white-box*)
- Aspetti fondamentali nello sviluppo di sistemi software basati su componenti:
  - **incapsulamento** di strutture software come componenti astratti (caratteristica di *variabilità*)
  - **composizione** di componenti attraverso il *binding* dei parametri con valori specifici o verso altri componenti (caratteristica di *adattabilità*)

Component framework



Componenti in UML

- Un componente UML è definito come

  «*a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces*»

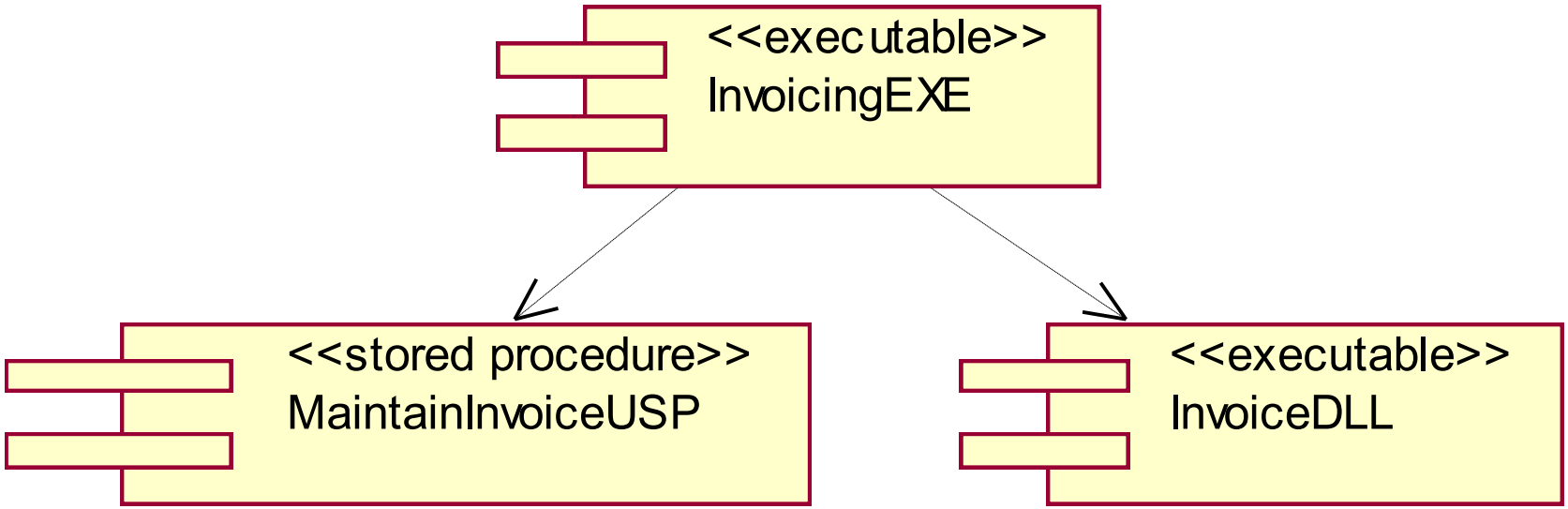
- UML definisce 5 stereotipi standard per i componenti:
  - Executable (es. un modulo eseguibile direttamente)
  - Library (es. un modulo statico o dinamico di libreria)
  - Table (es. una tabella di un DB)
  - File (es. codice sorgente o file dati)
  - Document (es. un documento in linguaggio naturale)

Caratteristiche di un componente

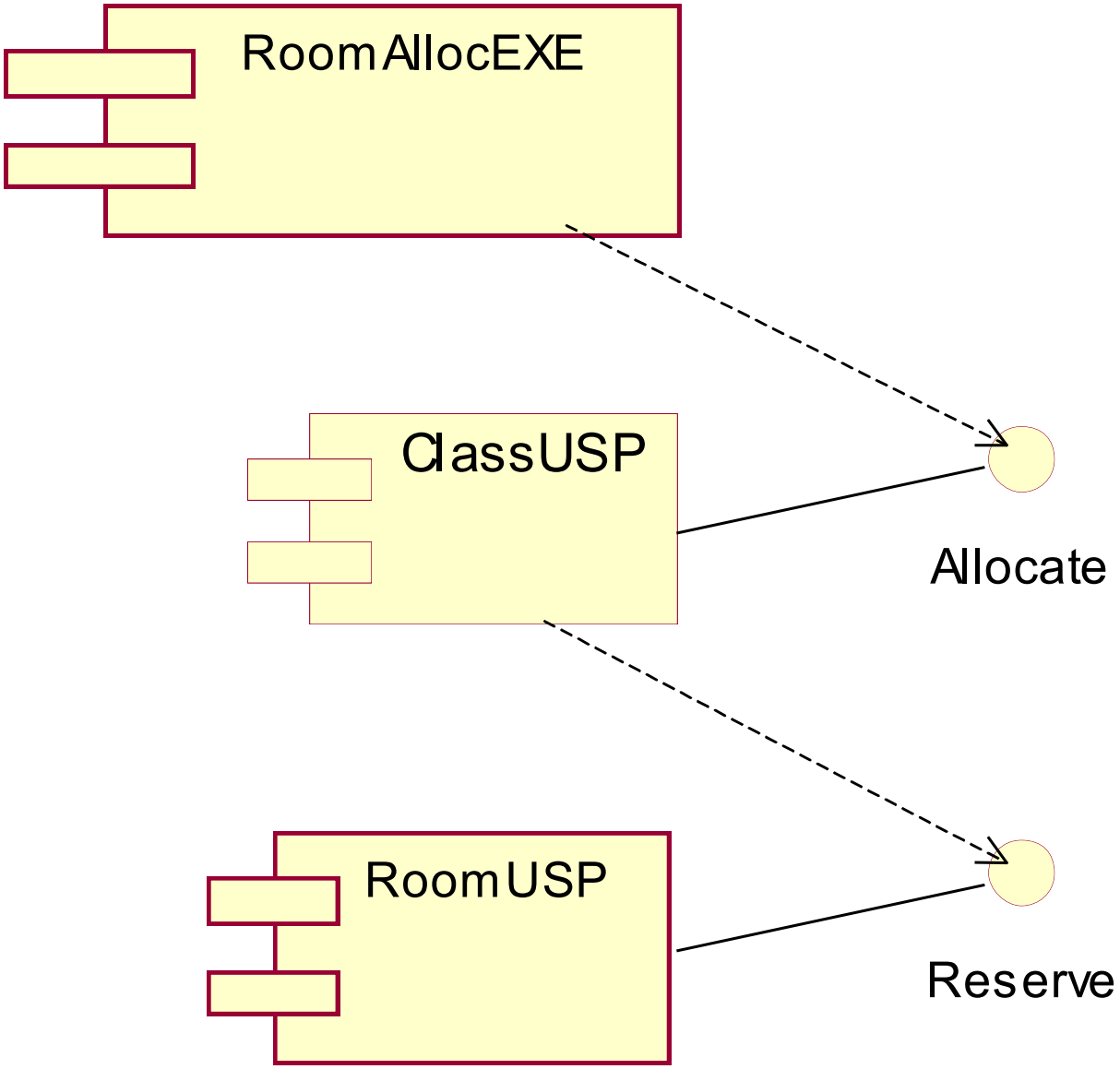
- Unità di ***independent deployment***
  - non è possibile il *deployment* parziale
- Unità di ***third-party composition***
  - sufficientemente documentato e *self-contained* per essere collegato (*plugged into*) ad altri componenti secondo regole fissate dette regole di composizione
- Non ha *persistent state*
  - in generale un componente non ha attributi ma «pubblica» un interfaccia che definisce l'insieme delle operazioni realizzate
  - copie differenti sono identiche (il componente rappresenta una unità di funzionalità)
- Definisce una parte **rimpiazzabile** di un sistema
  - ogni componente può essere sostituito con un altro componente conforme alla stessa interfaccia (netta separazione tra interfaccia ed implementazione)
- Svolge una funzione ben determinata e mostra elevate caratteristiche di coesione
- Può essere annidato in altri componenti

Component diagram

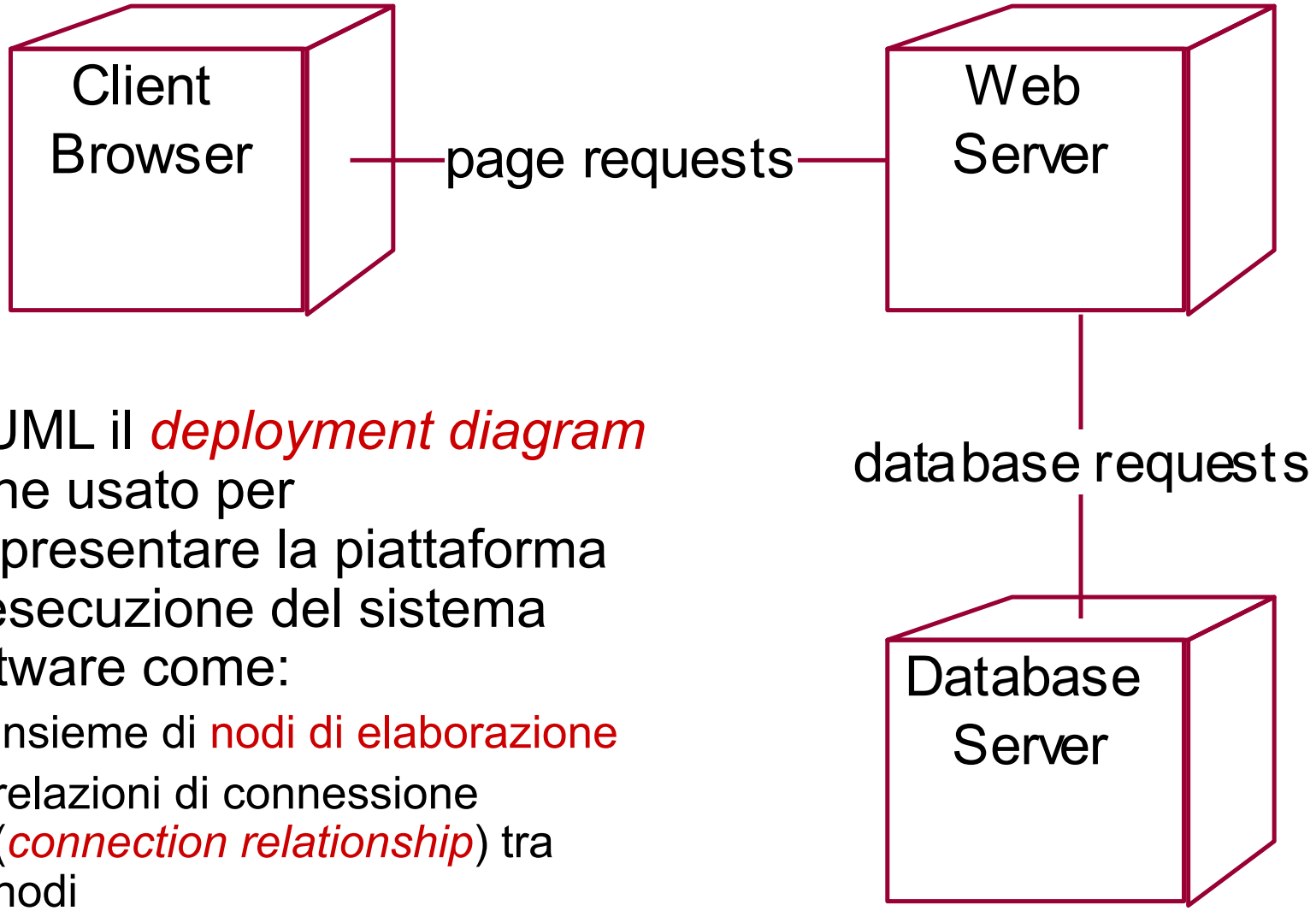
- Un component diagram UML mostra la struttura dei componenti e le relazioni che intercorrono tra i componenti
- Le relazioni tra componenti possono essere di due tipi:
  - dependency relationship
  - composition relationship



Rappresentazione di interfacce



Deployment diagram

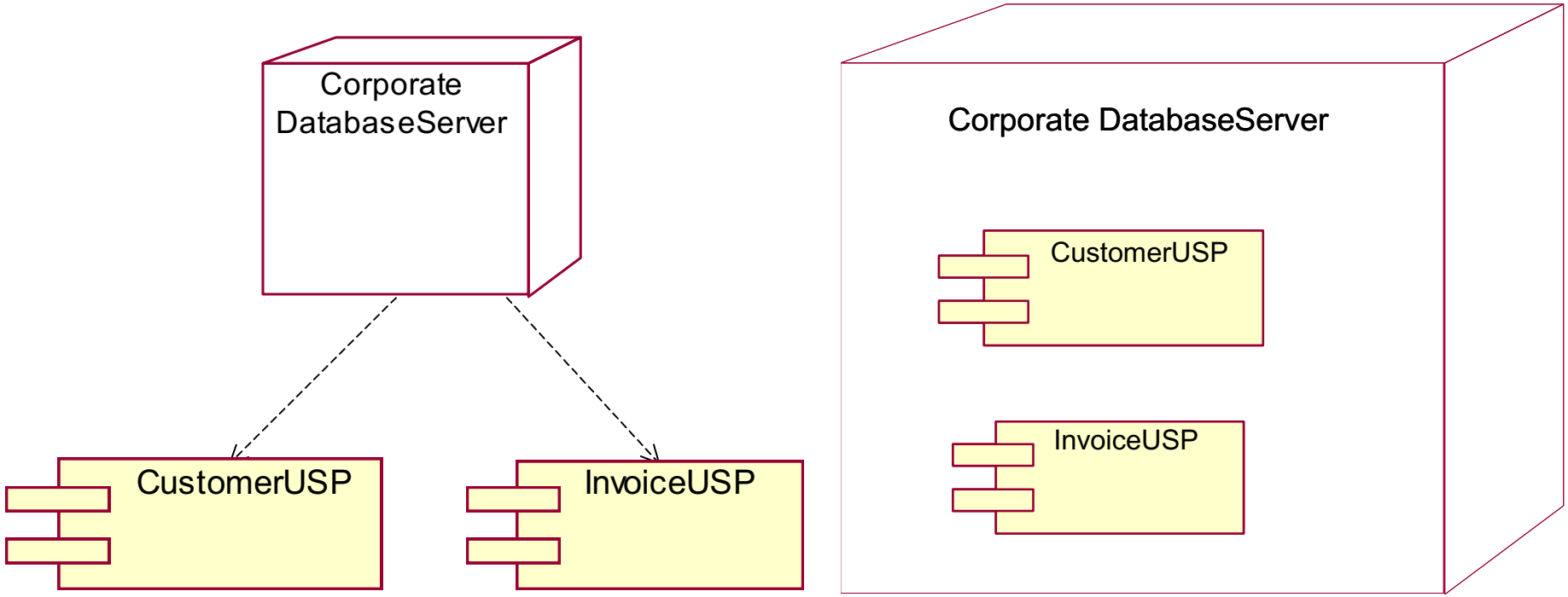


In UML il *deployment diagram* viene usato per rappresentare la piattaforma di esecuzione del sistema software come:

- insieme di nodi di elaborazione
- relazioni di connessione (*connection relationship*) tra nodi

Allocazione di componenti su nodi

- Un nodo di elaborazione del *deployment diagram* «esegue» i componenti che vengono allocati su di esso
- Un nodo con l'insieme dei componenti allocati definisce una cosiddetta *distribution unit*

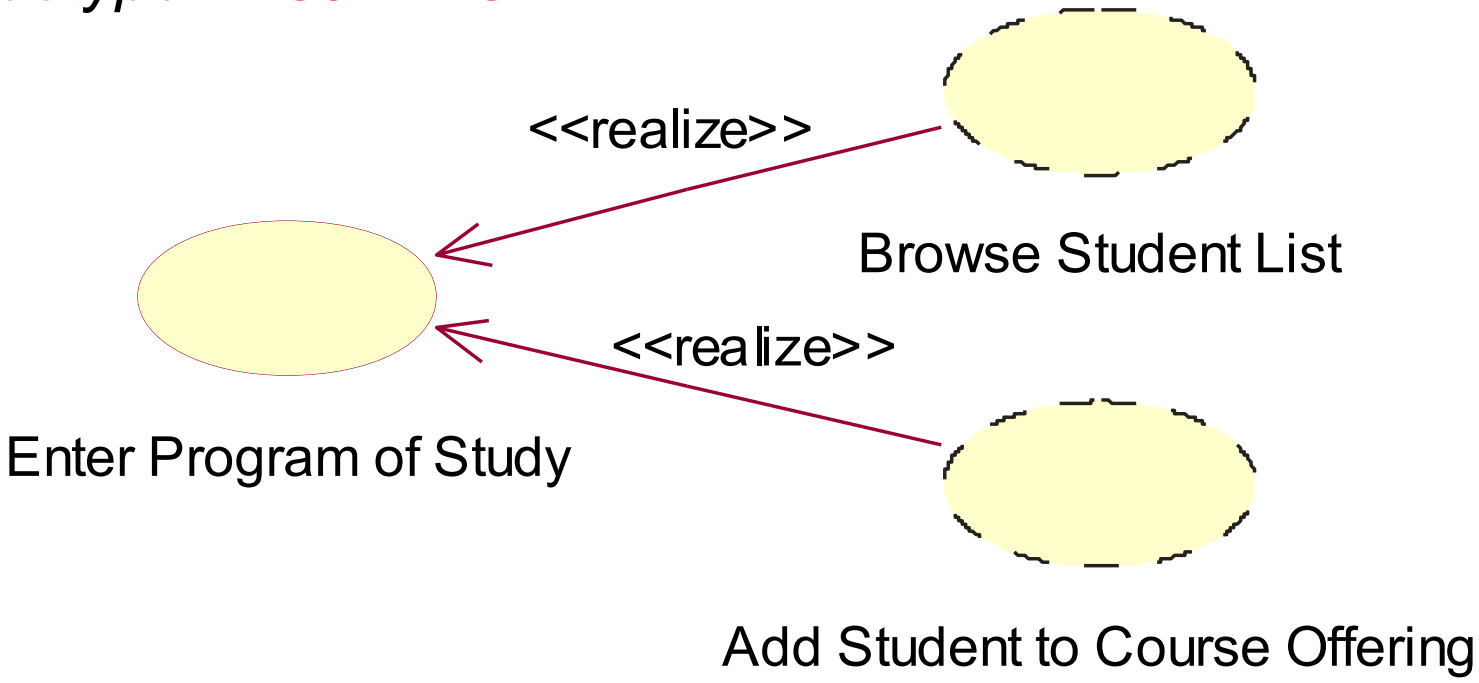


OOD dettagliato

- L'impatto della sotto-fase di OOD preliminare su quella di OOD dettagliato è determinato dalla definizione di una piattaforma HW/SW di esecuzione alla quale l'OOD dettagliato deve conformarsi
- Per il resto, la sotto-fase di OOD dettagliato rappresenta una diretta continuazione della fase di OOA
- L'obiettivo è trasformare i modelli di OOA, espressi nel dominio del problema, in modelli espressi nel dominio della soluzione, a partire dai quali effettuare la codifica del sistema
- Nella sotto-fase di OOD dettagliato si progettano le unità architetturali del sistema identificate nella sotto-fase di OOD preliminare mediante aggiunta di dettagli tecnici ai modelli esistenti (o mediante creazione di nuovi modelli ad un livello di astrazione inferiore)
- La sotto-fase di OOD dettagliato si occupa di definire **come** avviene la **collaborazione** tra oggetti che è alla base di ogni sistema software *object-oriented*
- Tale collaborazione viene definita in termini di **realizzazione di casi d'uso e di operazioni**

Collaborazione: notazione UML

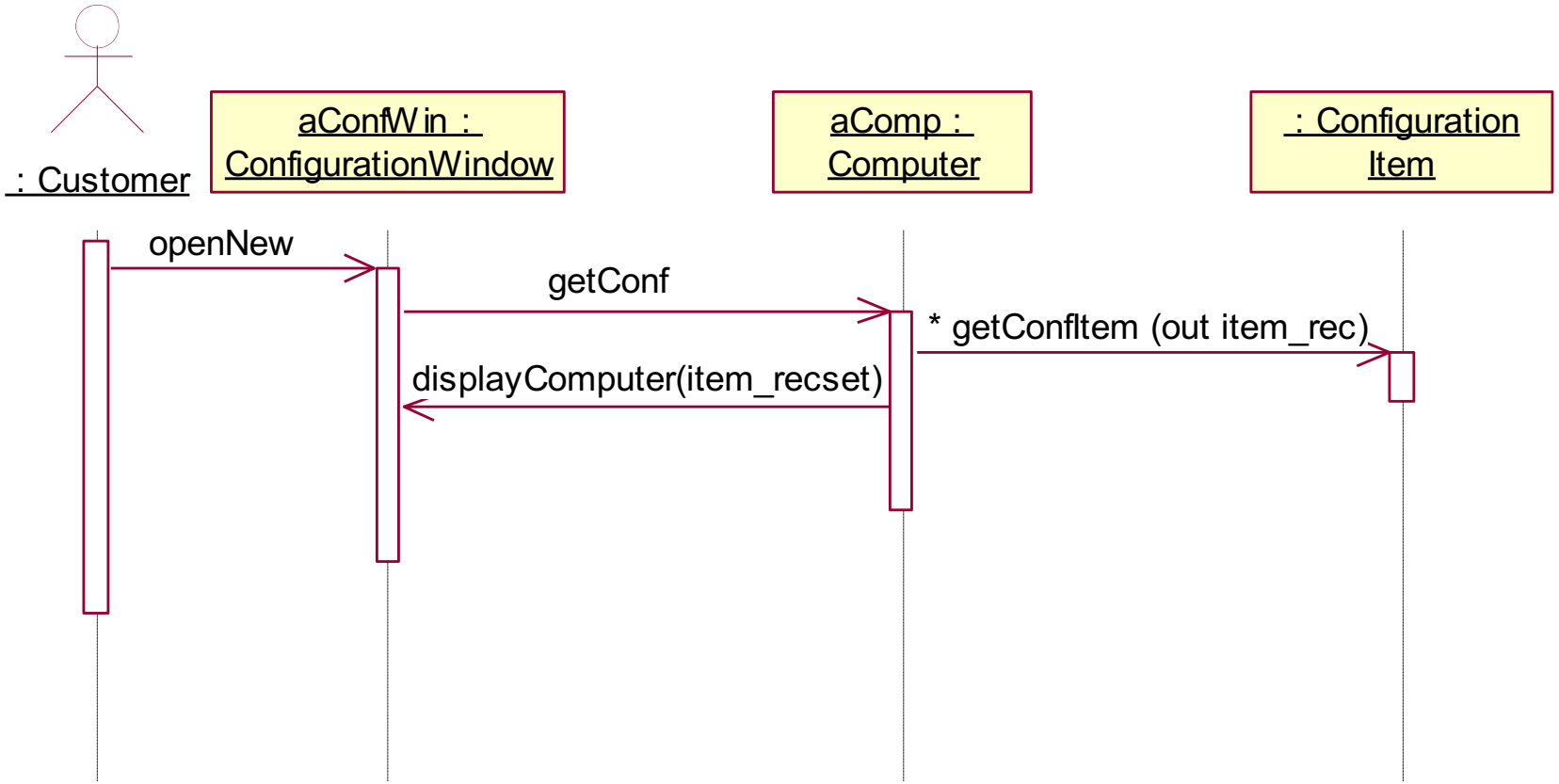
- La notazione per la **collaborazione** è simile a quella utilizzata per i casi d'uso, e si rappresenta mediante un'ellisse con il bordo tratteggiato
- La relazione (di realizzazione) con i casi d'uso si rappresenta mediante archi orientati etichettati con lo *stereotype* **«realize»**



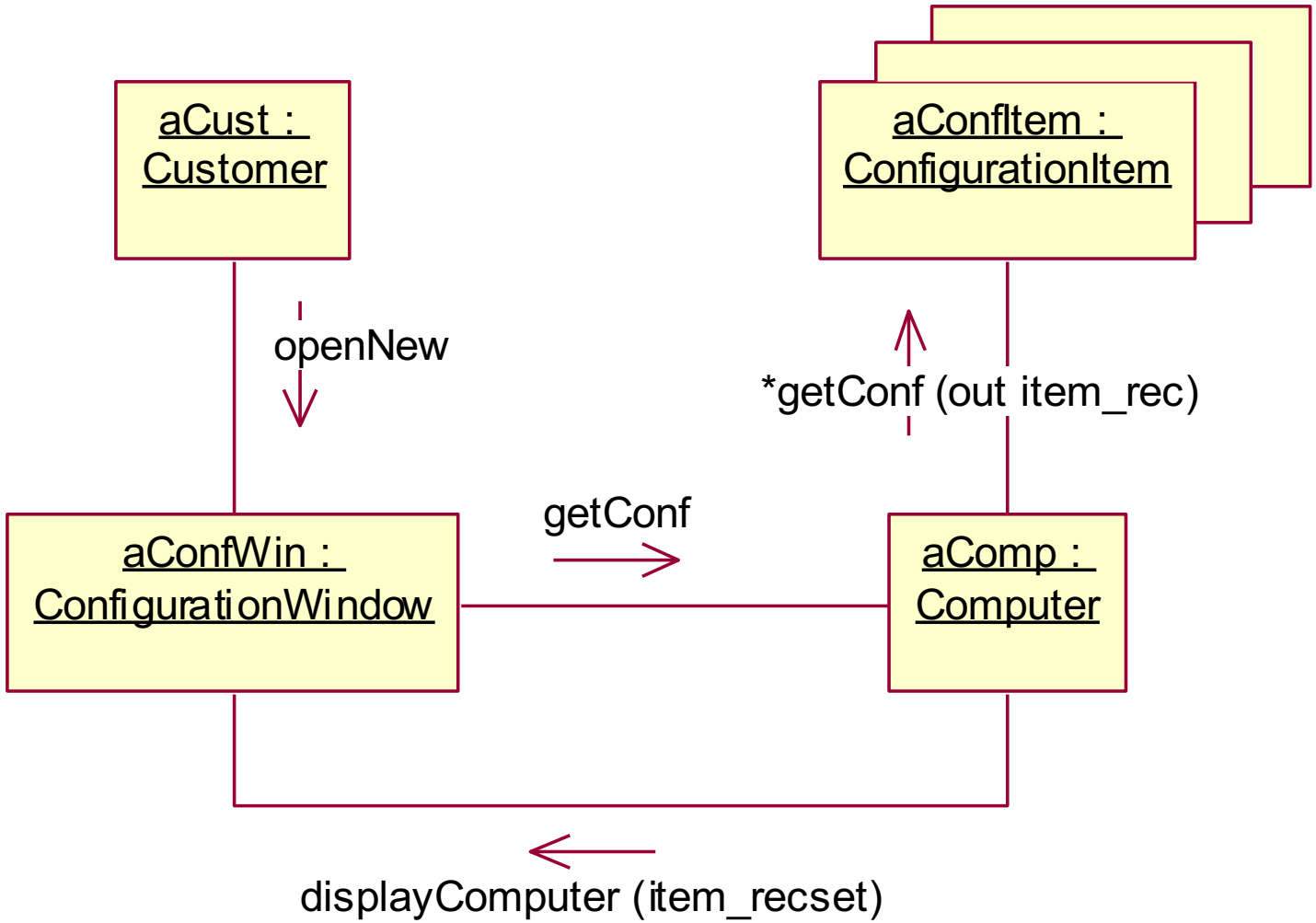
Collaboration diagram

- Il **collaboration diagram** è uno dei due formalismi UML per rappresentare l'interazione tra gli oggetti del sistema
- E' una rappresentazione equivalente al *sequence diagram*, ma preferibile in fase di OOD
- Il sequence diagram, usato in fase di OOA, pone l'enfasi sulla sequenza temporale di messaggi che gli oggetti del sistema si scambiano, ma presenta i seguenti svantaggi per l'utilizzo in fase di OOD:
  - imprecisione nella rappresentazione di percorsi alternativi
  - ingombro eccessivo al crescere del numero di oggetti
- Il collaboration diagram mostra in modo esplicito le relazioni statiche tra gli oggetti del sistema con il flusso dei messaggi che gli oggetti si inviano, con i seguenti vantaggi per l'utilizzo in fase di OOD:
  - migliore leggibilità nella rappresentazione delle interazioni tra un numero elevato di oggetti
  - possibilità di specificare pienamente e annotare ogni messaggio

Da sequence diagram.....



.....a collaboration diagram



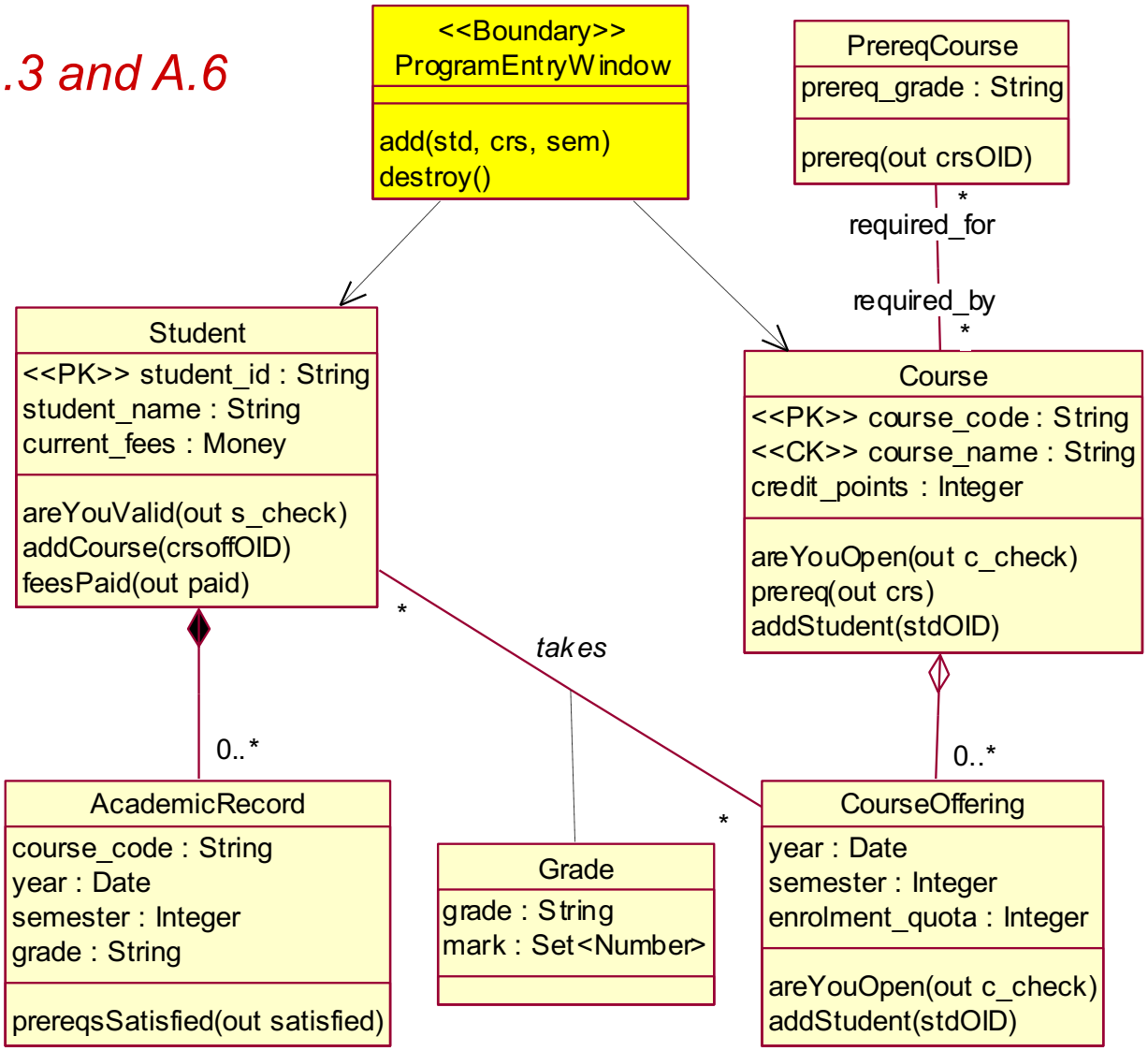
La sequenza temporale dei messaggi può essere catturata con l'introduzione dei cosiddetti *sequence number*, che ordinano in modo progressivo i messaggi che gli oggetti si scambiano (i *sequence number* sono opzionali)

Realizzazione di casi d'uso

- I casi d'uso introdotti in fase di OOA sono realizzati mediante collaborazioni
- A causa del differente livello di astrazione, un caso d'uso è realizzato da un insieme di collaborazioni
- Ogni collaborazione ha:
  - una **parte strutturale**
    - rappresenta gli *aspetti statici della collaborazione*
    - descritta attraverso l'elaborazione (rispetto alla versione prodotta in fase di OOA) della porzione di *class diagram* corrispondente
  - una **parte comportamentale**
    - rappresenta gli *aspetti dinamici della collaborazione* tra gli elementi della parte strutturale
    - descritta mediante *collaboration diagram*

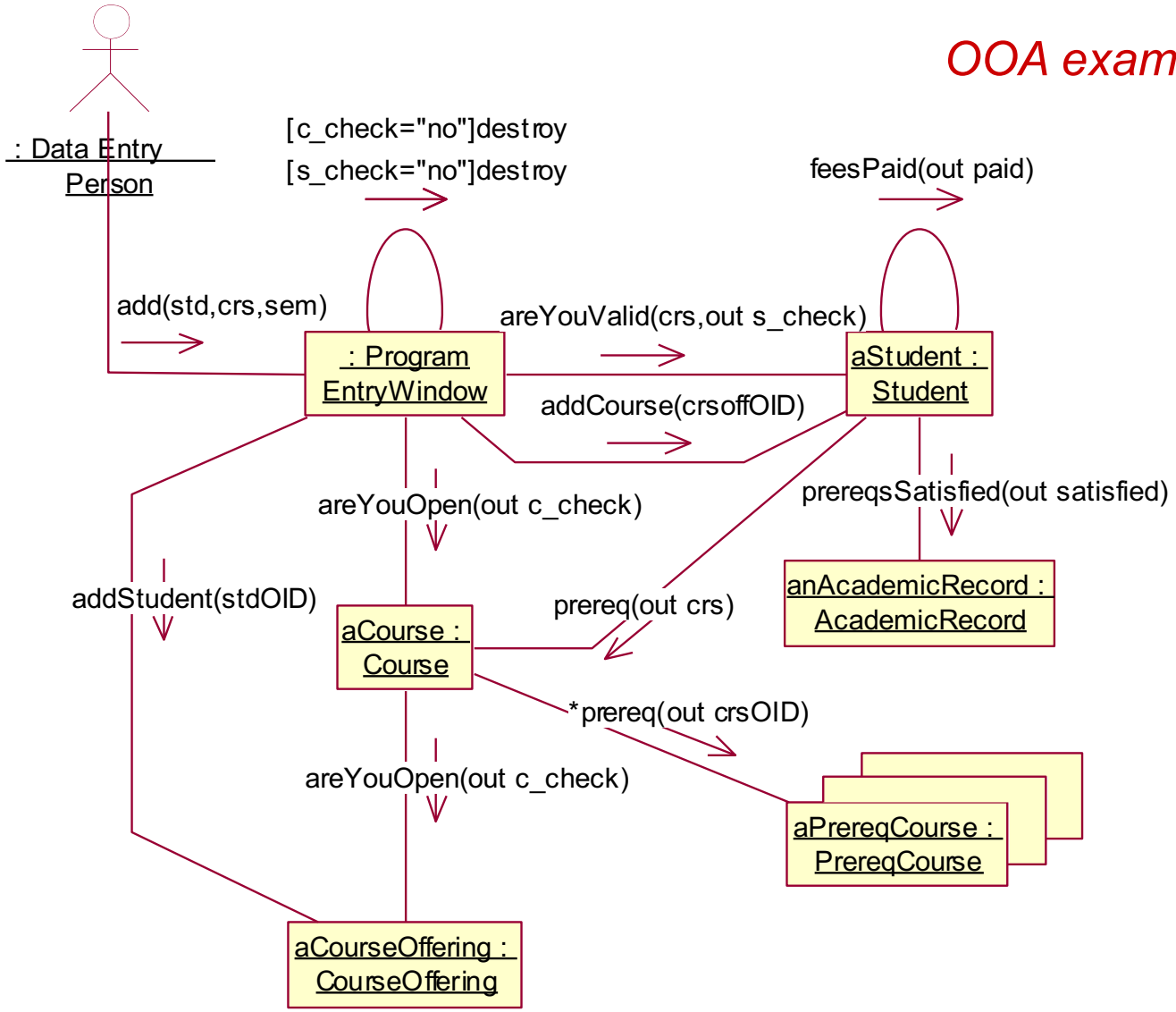
Collaborazione – parte strutturale

Refer to
OOA examples A.3 and A.6



Collaborazione – parte comportamentale

Refer to
OOA examples A.3 and A.6



Collaborazione – gestione del controllo

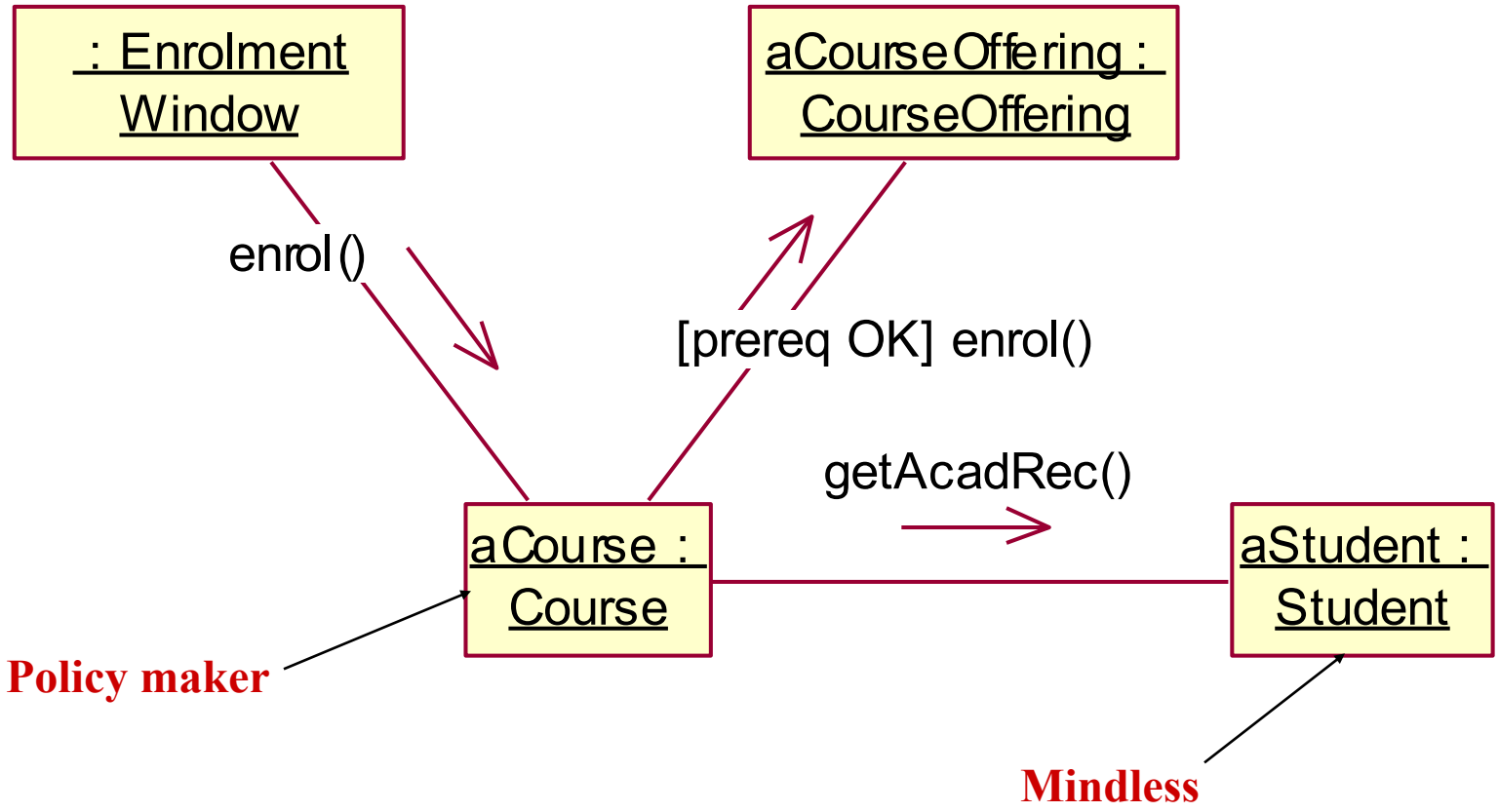
University Enrolment system

- Esempio:

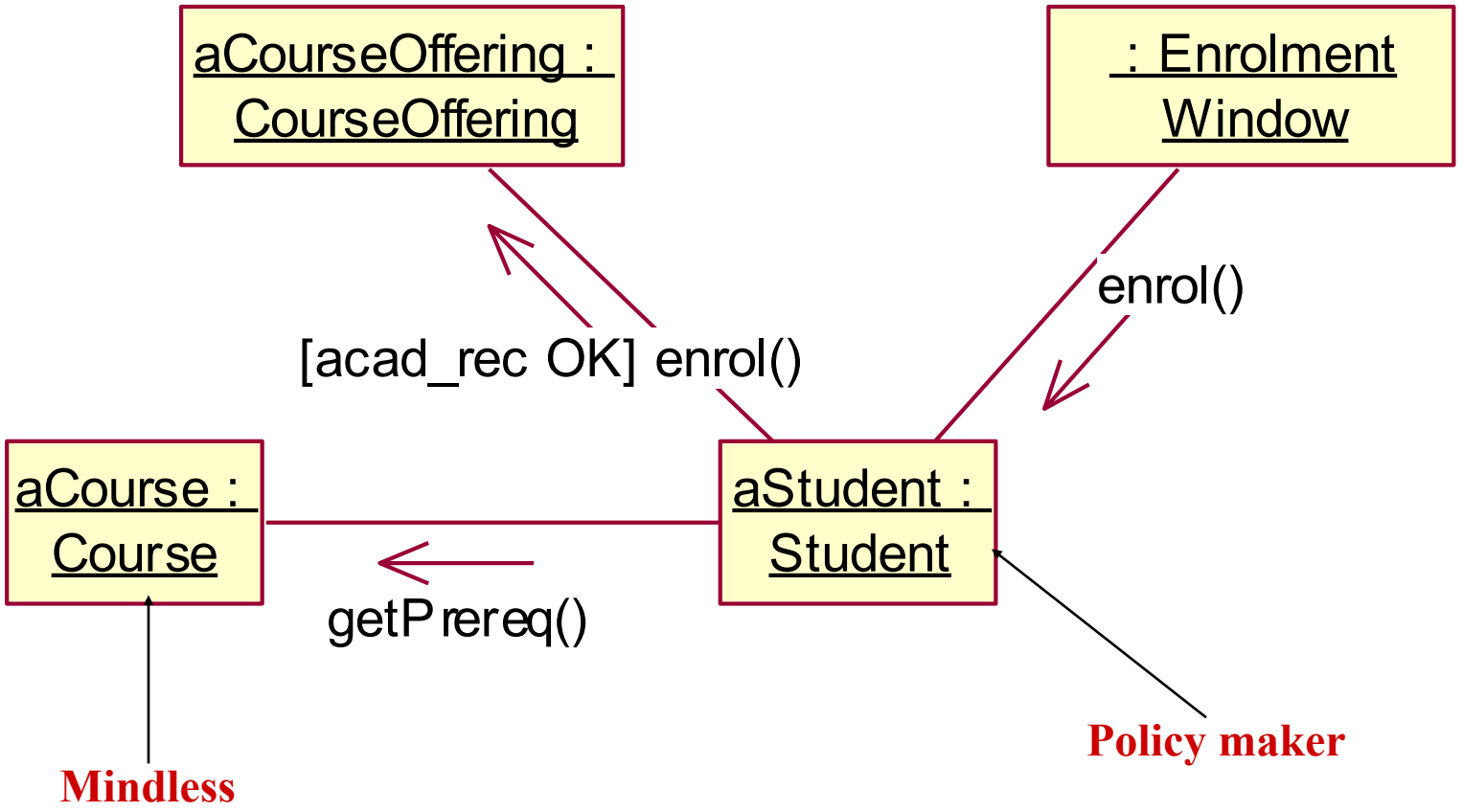
  aggiunta di uno studente (`Student`) ad un modulo (`CourseOffering`).

- Operazioni da effettuare:
  1. Ottenere l'elenco dei corsi che risultano come prerequisiti per il modulo considerato
  2. Verificare che lo studente sia in possesso dei prerequisiti
- Si consideri che:
  - Il messaggio `enrol()` viene inviato dal *boundary object* `:EnrolmentWindow`
  - Tre *entity classes* vengono coinvolte: `CourseOffering`, `Course`, and `Student`
- Almeno 4 soluzioni possibili (con differenti caratteristiche di *class coupling*)

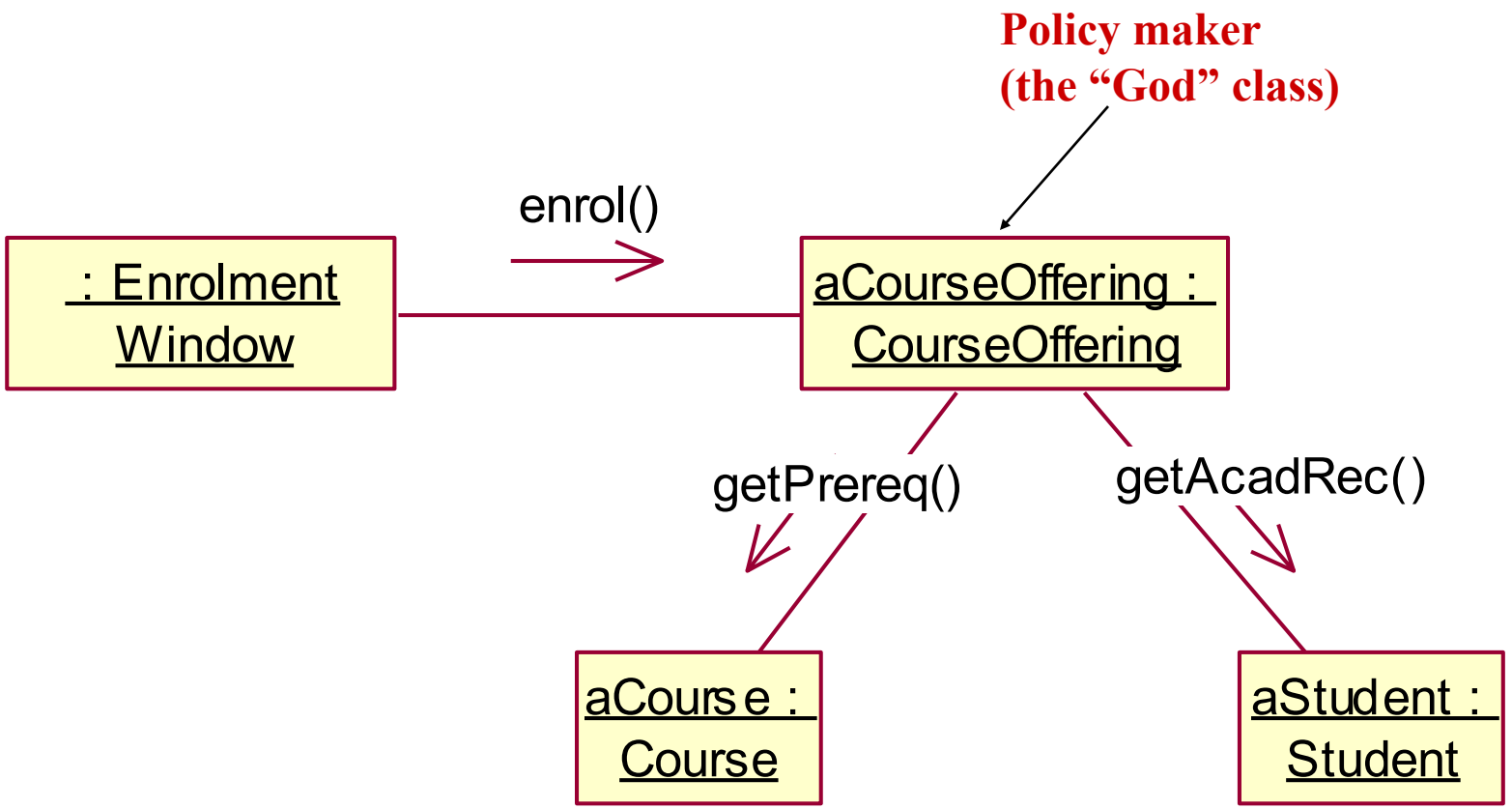
Gestione del controllo - *soluzione 1*



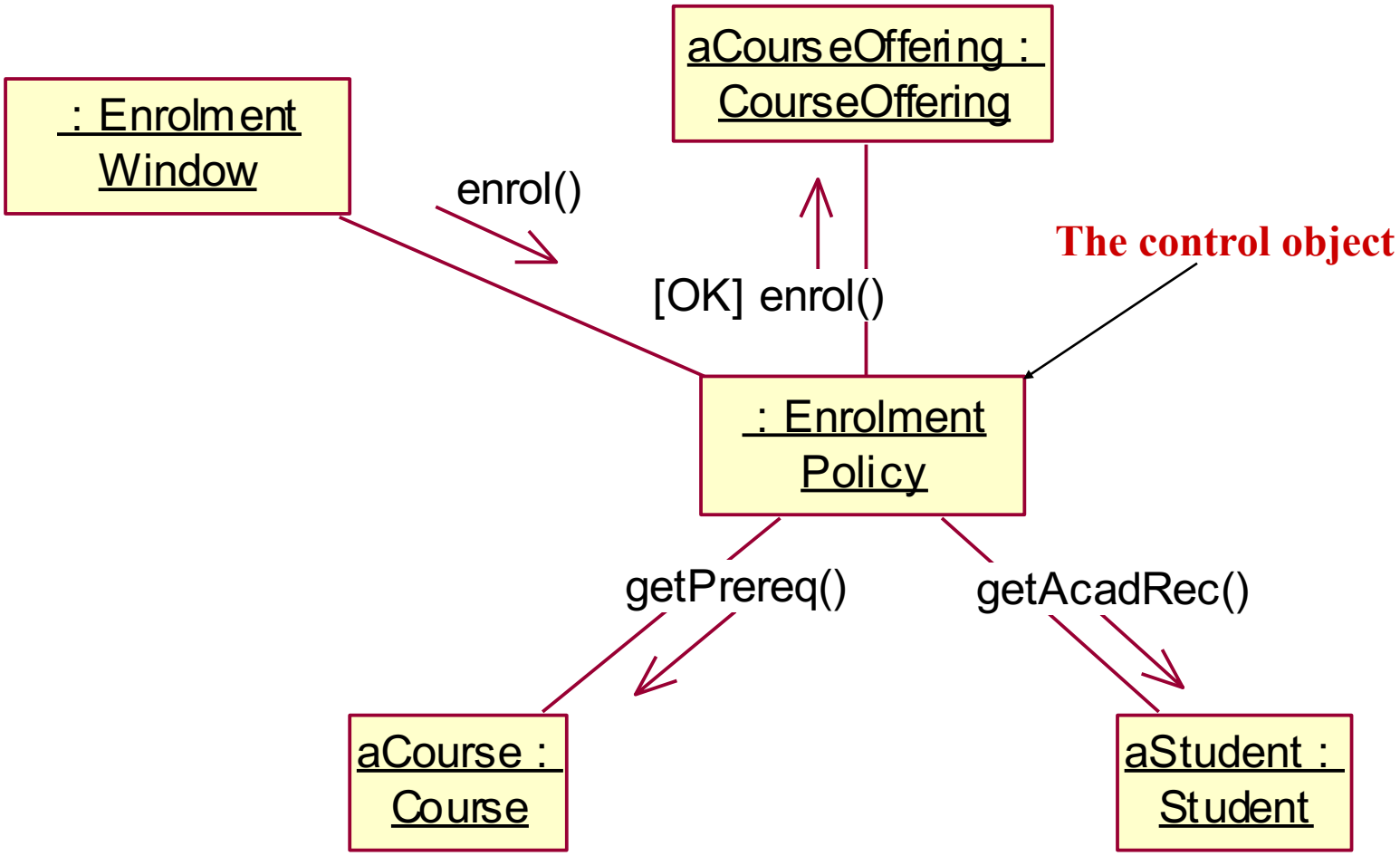
Gestione del controllo - *soluzione 2*



Gestione del controllo - *soluzione 3*



Gestione del controllo - *soluzione 4*



Legge di Demeter
(per la riduzione del *class coupling*)

- Per qualsiasi metodo **`m`** di un oggetto **`x`** di una classe, il destinatario dei messaggi nel corpo di **`m`** deve essere:
  - **`x`** stesso (ovvero **`this`**)
  - un oggetto presente come argomento di **`m`**
  - un oggetto riferito da un attributo di **`x`** (nell'interpretazione debole si estende anche ad attributi ereditati)
  - un oggetto istanziato da **`m`**
  - un oggetto riferito da una variabile globale
- La legge di Demeter limita i riferimenti diretti e le dipendenze che attraversano i confini di una classe

Realizzazione di operazioni

- Le collaborazioni permettono di modellare operazioni complesse che richiedono l'interazione tra più oggetti
- Operazioni che si possono confinare all'interno di una o due classi sono meglio modellate facendo uso di ***activity diagram***
- Lo stesso principio si applica alle operazioni che vengono identificate come componenti di operazioni più complesse modellate come collaborazioni
- Infine, le operazioni elementari possono essere progettate facendo uso di una notazione di tipo pseudo-codice, o PDL (program design language)

Realizzazione di operazioni (2)

Ex. CourseOffering.addStudent (stdOID)