Software Architectural Transaction Patterns

- A service often encapsulates data or provides access to data that need to be read or updated by clients
- Many services need to provide coordinated update operations
- A **transaction** is a request from a client to a service that consists of two or more operations that perform a single logical function and that must be completed in its entirety or not at all

Transaction properties

- Transactions have the following properties, sometimes referred to as ACID properties:
  - **Atomicity (A).** A transaction is an indivisible unit of work. It is either entirely completed (committed) or aborted (rolled back)
  - **Consistency (C).** After the transaction executes, the system must be in a consistent state
  - **Isolation (I).** A transaction's behavior must not be affected by other transactions
  - **Durability (D).** Changes are permanent after a transaction completes. These changes must survive system failures. This is also referred to as *persistence*

Example Banking Transaction (*Withdrawal*)

- A withdrawal transaction can be handled in one operation
- A semaphore is needed for synchronization to ensure that access to the customer account record is mutually exclusive
- The transaction processor locks the account record for this customer, performs the update, and then unlocks the record

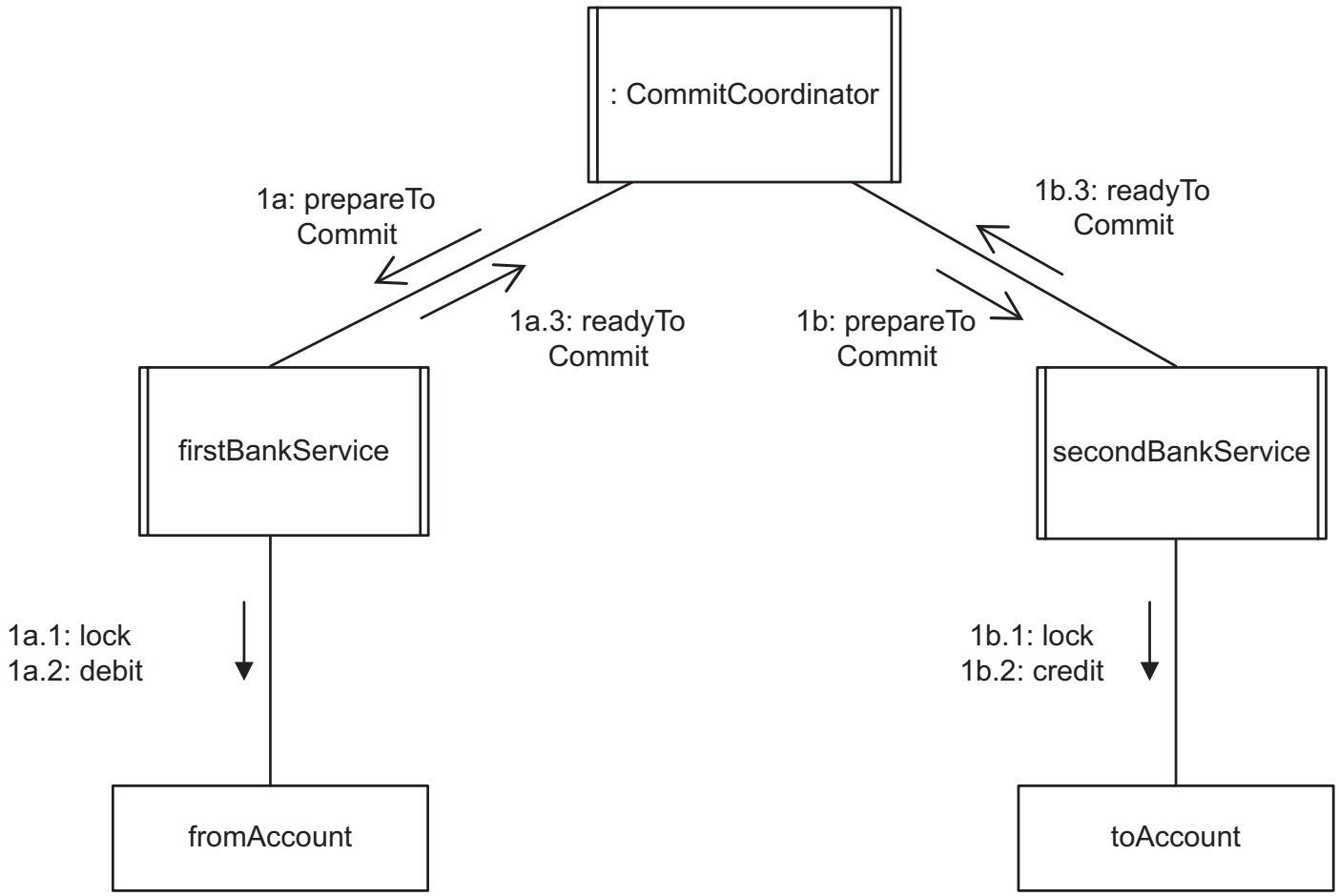
Example Banking Transaction (*Transfer*)

- Consider a transfer transaction between two accounts – for example, from a savings account to a checking account – in which the accounts are maintained at two separate banks (services)
- In this case, it is necessary to debit the savings account and credit the checking account
- Therefore, the transfer transaction consists of two operations that must be atomic – a debit operation and a credit operation – and the transfer transaction must bre either committed or aborted:
  - **Committed.** Both credit and debit operations occur
  - **Aborted.** Neither the credit nor the debit operation occurs

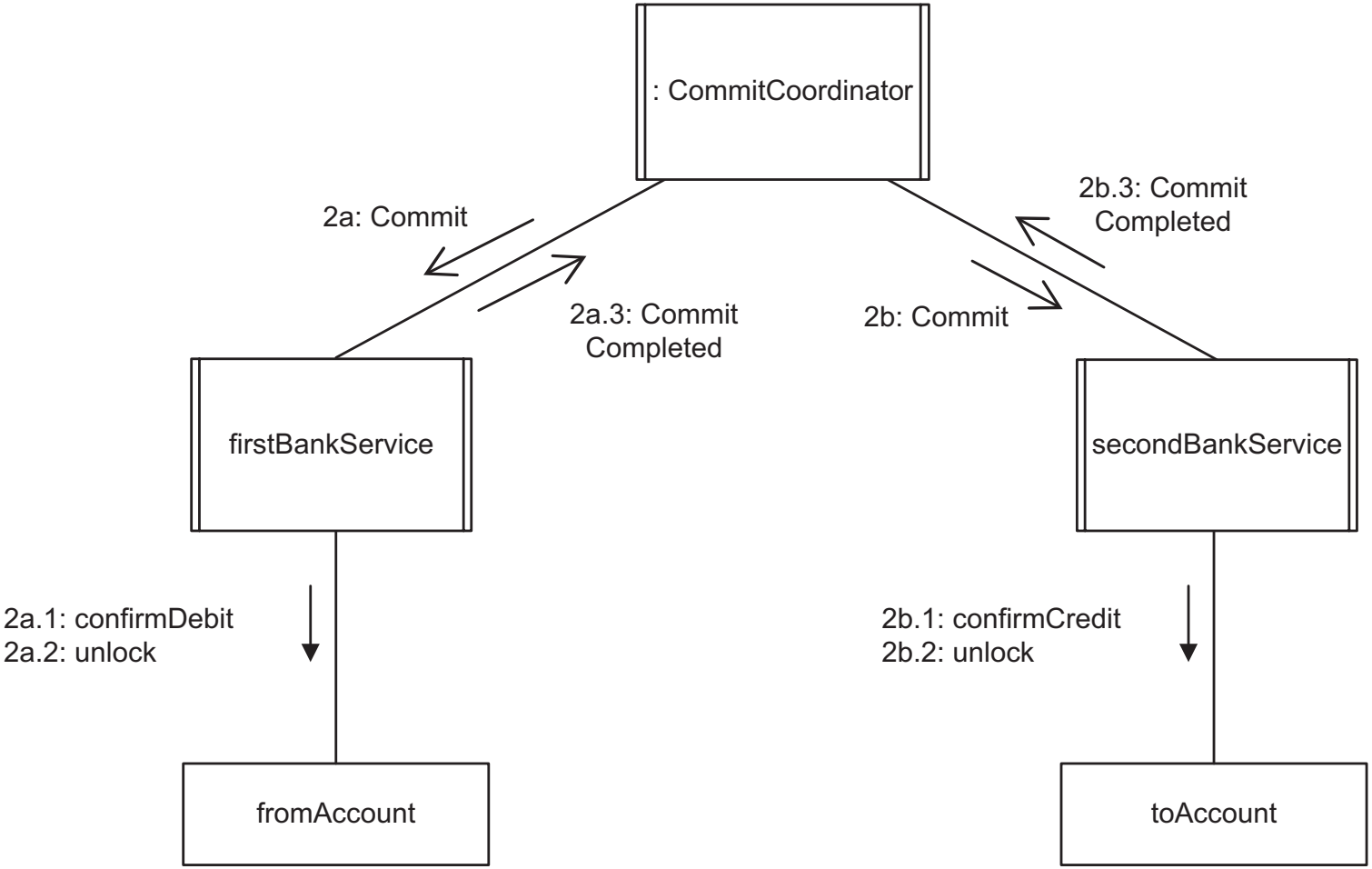
Two-Phase Commit Protocol

- The **Two-Phase Commit Protocol** pattern addresses the problem of managing atomic transactions in distributed systems, by synchronizing updates on different nodes
- Coordination of the transaction is provided by the **`CommitCoordinator`**
- There is one participant service for each node
- There are two participants in the bank transfer transaction:
  - **`firstBankService`**, which maintains the account *from* which money is being transferred (from Account), and
  - **`secondBankService`**, which maintains the account *to* which money is being transferred (to Account)

First Phase


: CommitCoordinator
1a: prepareTo Commit
1a.3: readyTo Commit
1b.3: readyTo Commit
1b: prepareTo Commit
firstBankService
secondBankService
1a.1: lock
1a.2: debit
1b.1: lock
1b.2: credit
fromAccount
toAccount

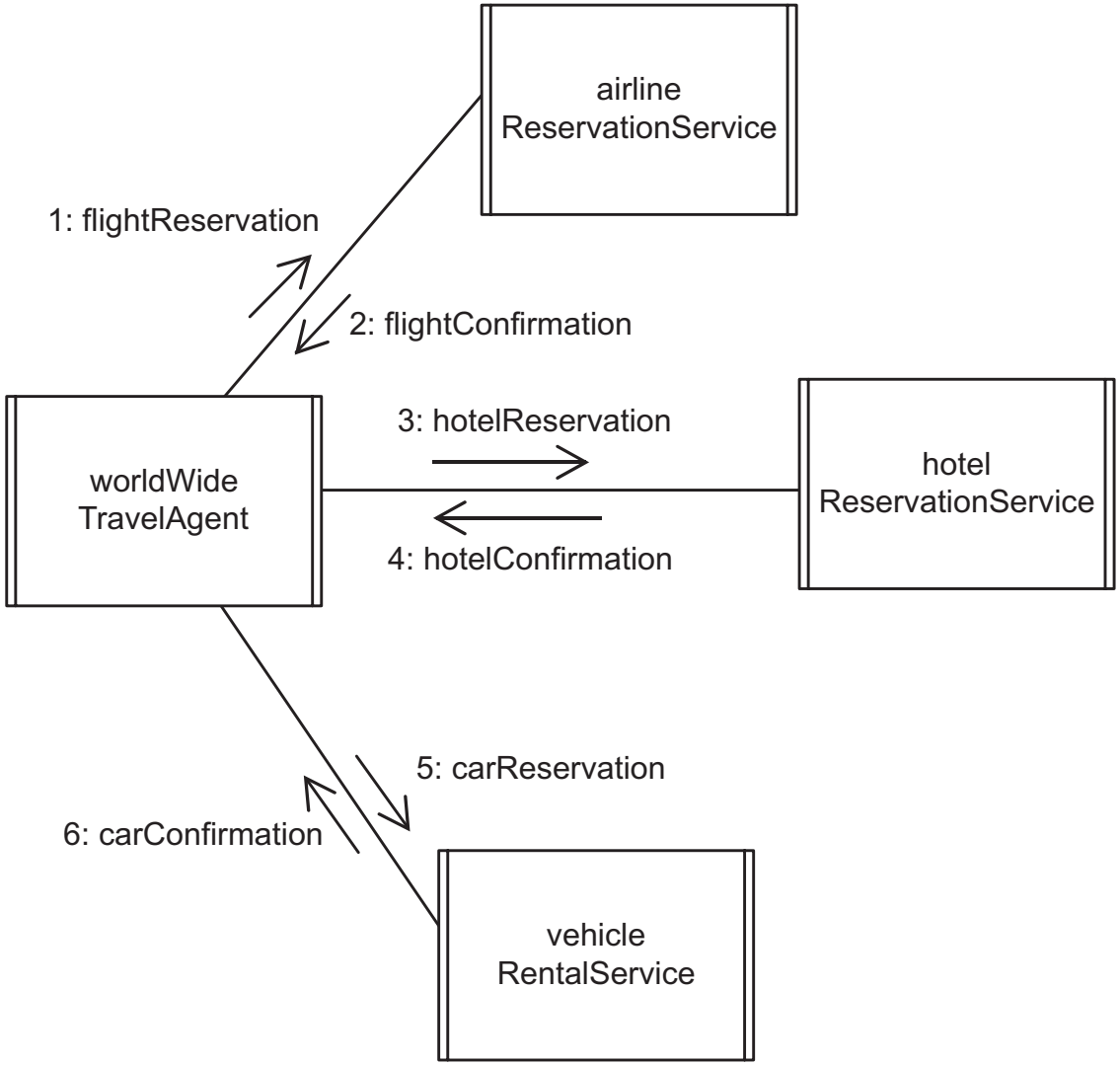
Second Phase



Compound Transaction Pattern

- The previous bank transfer transaction is an example of a flat transaction, which has an “all-or-nothing” characteristic
- A compound transaction, in contrast, might need only a partial rollback
- The **Compound Transaction** pattern can be used when the client’s transaction requirement can be broken down into smaller flat atomic transactions, in which each atomic transaction can be performed separately and rolled back separately
- For example, if a travel agent makes an airplane reservation, followed by a hotel reservation and a rental car reservation, it is more flexible to treat this reservation as consisting of three flat transactions. Treating the transaction as a compound transaction allows part of a reservation to be changed or canceled without the other parts of the reservation being affected.

Example Compound Transaction Pattern



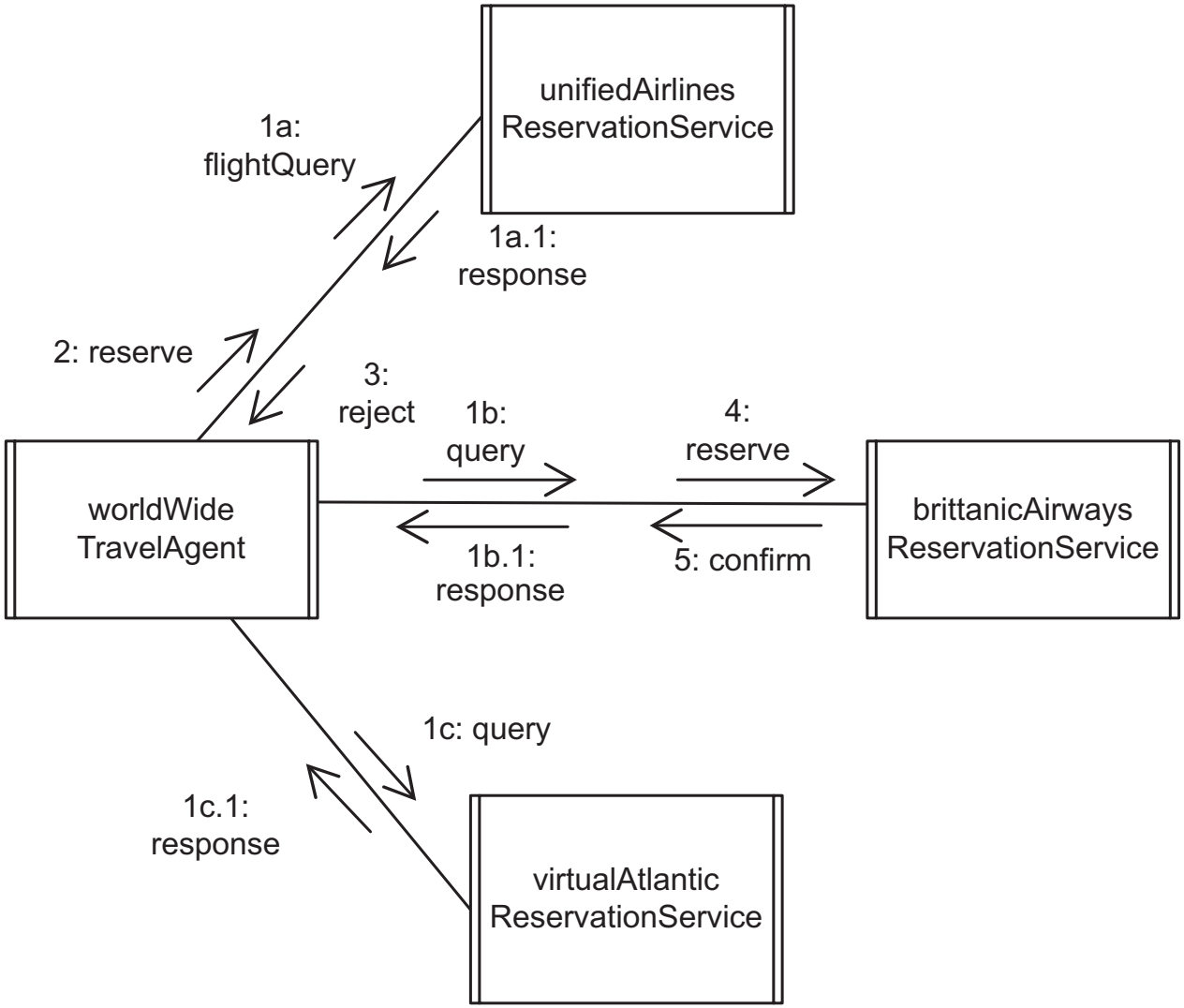
Long-Living Transaction Pattern

- A *long-living transaction* is a transaction that has a human in the loop and that could take a long and possibly indefinite time to execute, because individual human behavior is unpredictable
- The **Long-Living Transaction** pattern splits a long-living transaction into two or more separate transactions (usually two) so that human decision making takes place between the successive pairs (such as first and second) of transactions.

Example Long-Living Transaction

- Consider an airline reservation with human involvement in the transaction
- First a query transaction displays the available seats
- The query transaction is followed by a reserve transaction
- With this approach, it is necessary to recheck seat availability before the reservation is made
- A seat available at query time might no longer be available at reservation time because several agents might be querying the same flight at the same time
- If only one seat is available, the first agent will get the seat but not the others

Example Long-Living Transaction Pattern



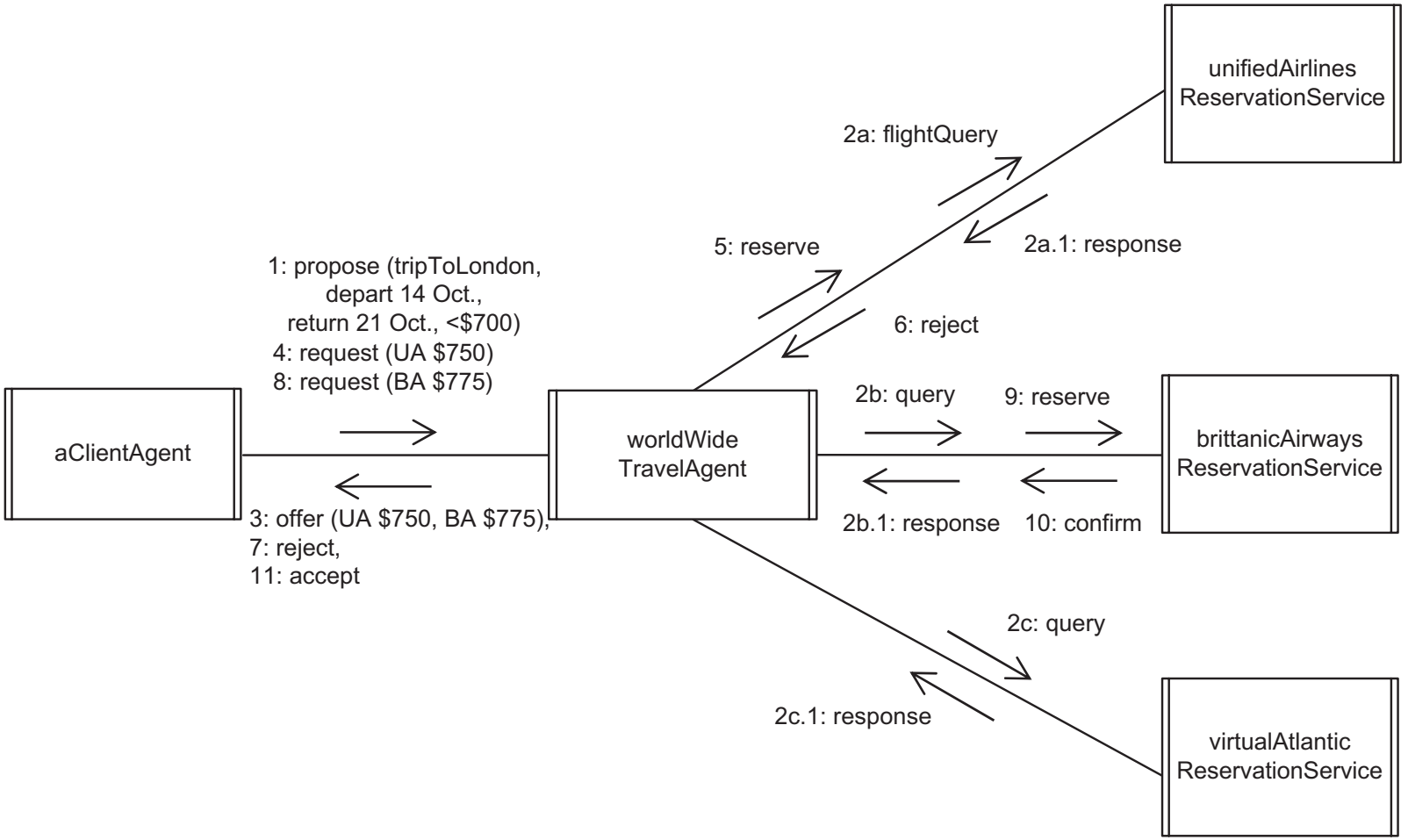
Negotiation Pattern

- In some SOAs, the coordination between services involves negotiations between software agents so that they can cooperatively make decisions
- In the **Negotiation** pattern (also known as the *Agent-Based Negotiation* or *Multi-Agent Negotiation* pattern), a client agent acts on behalf of the user and makes a proposal to a service agent
- The service agent attempts to satisfy the client's proposal, which might involve communication with other services
- Having determined the available options, the service agent then offers the client agent one or more options that come closest to matching the original client agent proposal
- The client agent may then request one of the options, propose further options, or reject the offer
- If the service agent can satisfy the client agent request, it accepts the request; otherwise, it rejects the request

Negotiation Services

- The client agent, who acts on behalf of the client, may do any of the following:
  - **Propose a service.** The client agent proposes a service to the service agent. This proposed service is *negotiable,* meaning that the client agent is willing to consider counteroffers
  - **Request a service.** The client agent requests a service from the service agent. This requested service is *nonnegotiable,* meaning that the client agent is not willing to consider counteroffers
  - **Reject a service offer.** The client agent rejects an offer made by the service agent
- The service agent, who acts on behalf of the service, may do any of the following:
  - **Offer a service.** In response to a client proposal, a service agent offers a counter- proposal
  - **Reject a client request/proposal.** The service agent rejects the client agent's proposed or requested service
  - **Accept a client request/proposal.** The service agent accepts the client agent's proposed or requested service

Example Negotiation Pattern



Service Interface Design in SOA

- New services are initially designed by using class structuring criteria
- During dynamic interaction modeling, the interaction between client objects and service objects is determined
- The approach taken for designing service operations is similar to that used in class interface design
- The messages arriving at a service form the basis for designing the service operations. The messages are analyzed to determine the name of the operation, as well as to determine the input and output parameters

Service Coordination in SOA

- In SOA applications that involve multiple services, coordination of these services is usually required
- To ensure loose coupling among the services, it is often better to separate the details of the coordination from the functionality of the individual services
- In SOA, different types of coordination are provided, including ***orchestration*** and ***choreography***

Orchestration and Choreography

- **Orchestration** consists of centrally controlled workflow coordination logic for coordinating multiple participant services
  - This allows the reuse of existing services by incorporating them into new service applications
- **Choreography** provides distributed coordination among services, and it can be used when coordination is needed between different business organizations
  - Thus, choreography can be used for collaboration between services from different service providers provided by different business organizations
  - Whereas orchestration is centrally controlled, choreography involves distributed control

Coordination

- Because the terms *orchestration* and *choreography* are often used interchangeably, the more general term **coordination** is used to describe the control and sequencing of different services as needed by a SOA application, whether they are centrally controlled or involve distributed control.
- Transaction patterns can also be used for service coordination

Detailed OOD

- *Preliminary OOD* provides the HW/SW execution platform to which the detailed design sub-phase must conform
- The main part of the detailed OOD sub-phase is devoted to refine what produced in the OOA phase
- The objective is to transform the OOA models, which have been defined in the problem domain, into *models defined in the solution domain*, which in turn will be used in the coding phase
- The detailed OOD sub-phase provides the detailed design of the architectural unites identified in the preliminary OOD sub-phase, through the *addition of technical details* (or by *producing additional models* at a decreased level of abstraction)
- The detailed OOD sub-phase defines the **collaboration between objects**, which is at the basis of each object-oriented program
- Such a collaboration is defined through the **realization of use cases and operations**
- Collaborations are to OOD what use cases are to OOA
  - if use cases drive OOA then collaborations drive OOD

Realization of use cases

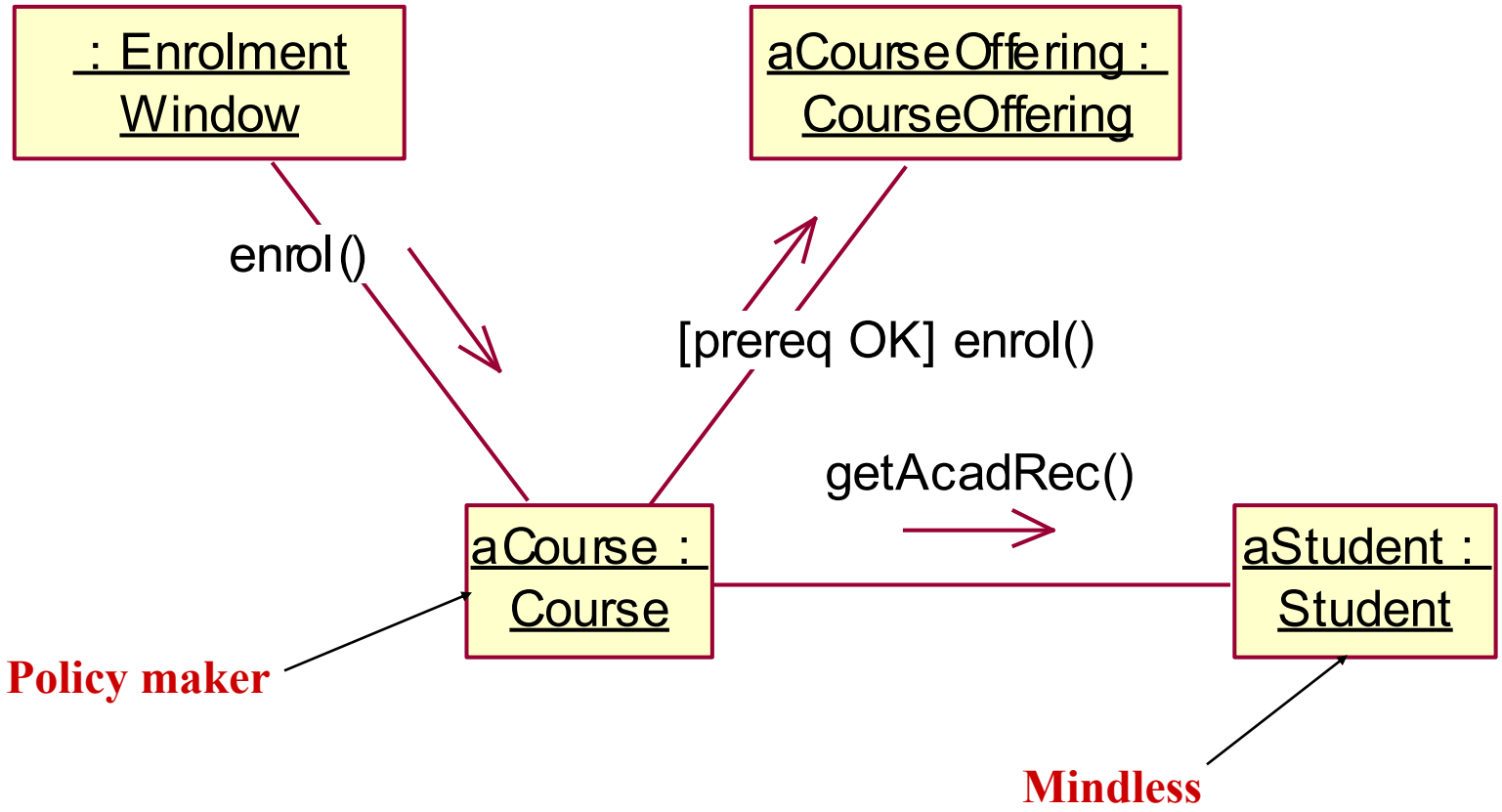
- *Use cases* introduced at OOA time are realized through *collaborations* at OOD time
- A single use case is typically realized by a set of collaborations, due to the different level of abstraction
- A collaboration has:
  - a **behavioral part**
    - represents the *dynamics* that shows how the static elements collaborate
    - defined by use of *communication diagrams*
  - a **structural part**
    - represents *static aspects of collaboration*
    - defined by elaborating the *OOA class diagram* with the implementation details, leading to *composite structure diagrams*

Collaboration – *control management*

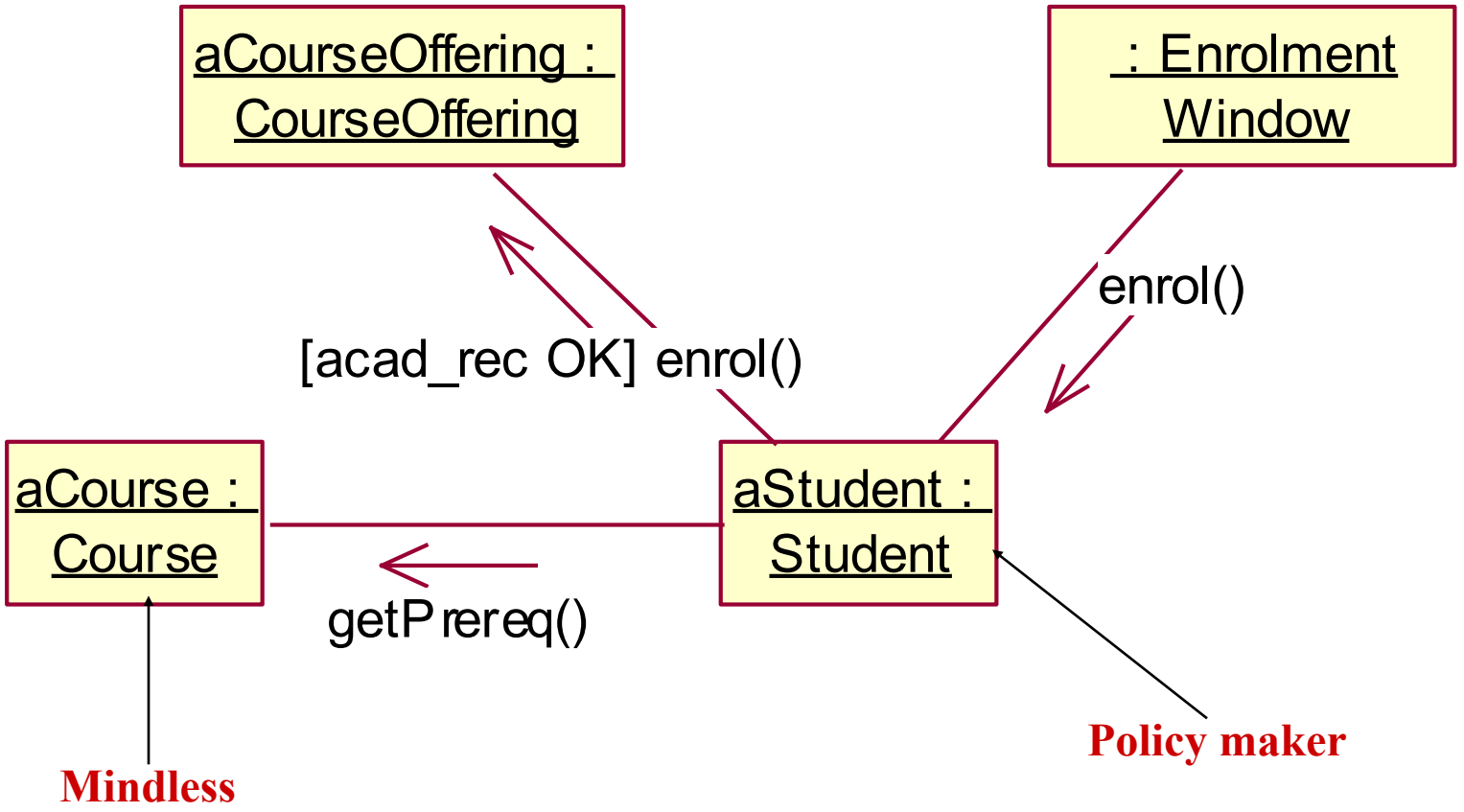
University Enrolment system

- Example
  - add a student (`Student`) to a course offering (`CourseOffering`)
- Actions to be executed
  1. identify the prerequisite courses for the course offering
  2. check if the student satisfies the prerequisites
- Consider that:
  - the `enrol()` message is sent by the boundary object `:EnrolmentWindow`
  - three entity classes are involved – `CourseOffering`, `Course` and `Student`
- At least four solutions possible (with different *class coupling* characteristics)

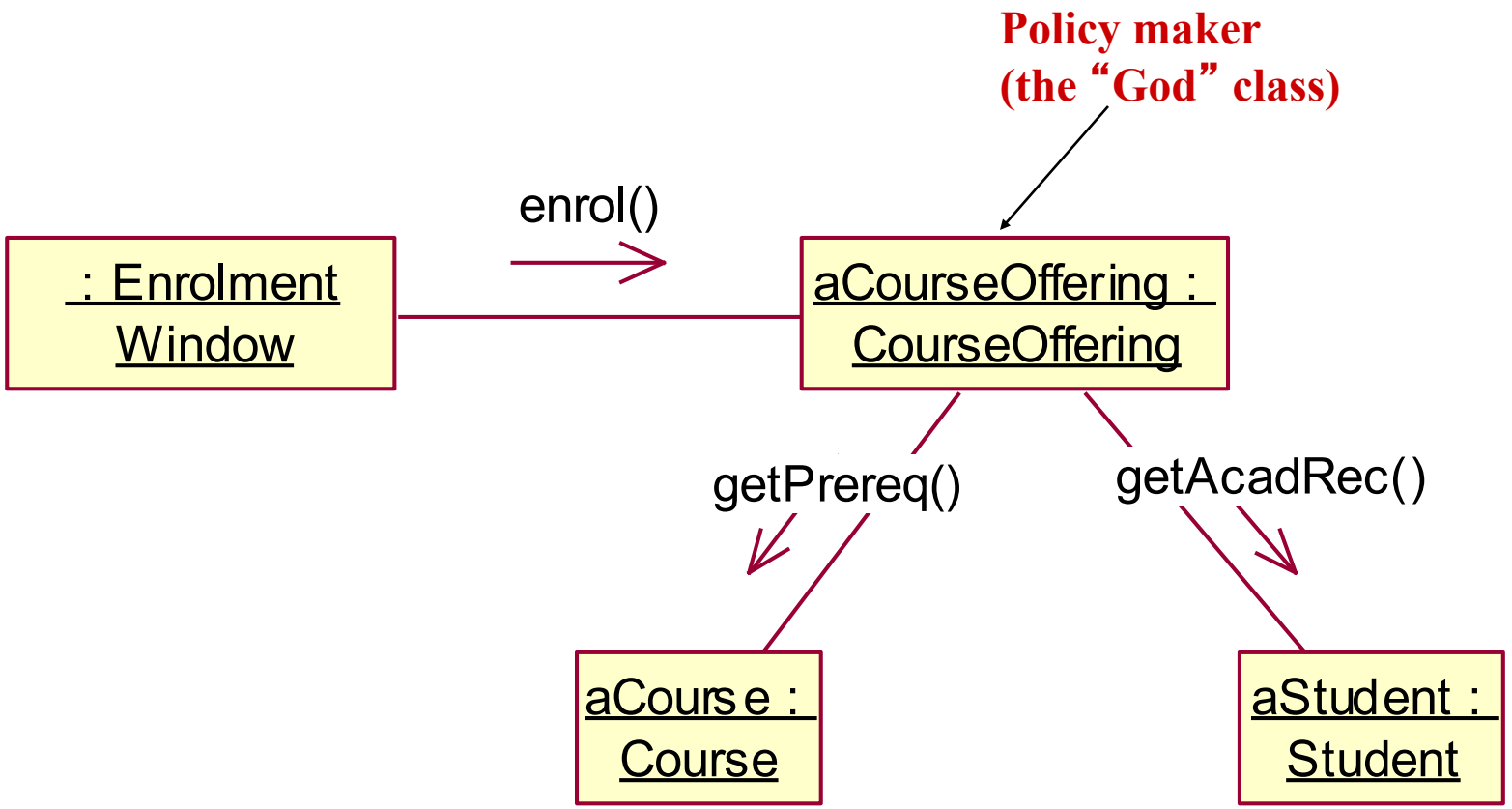
Control management - *solution 1*



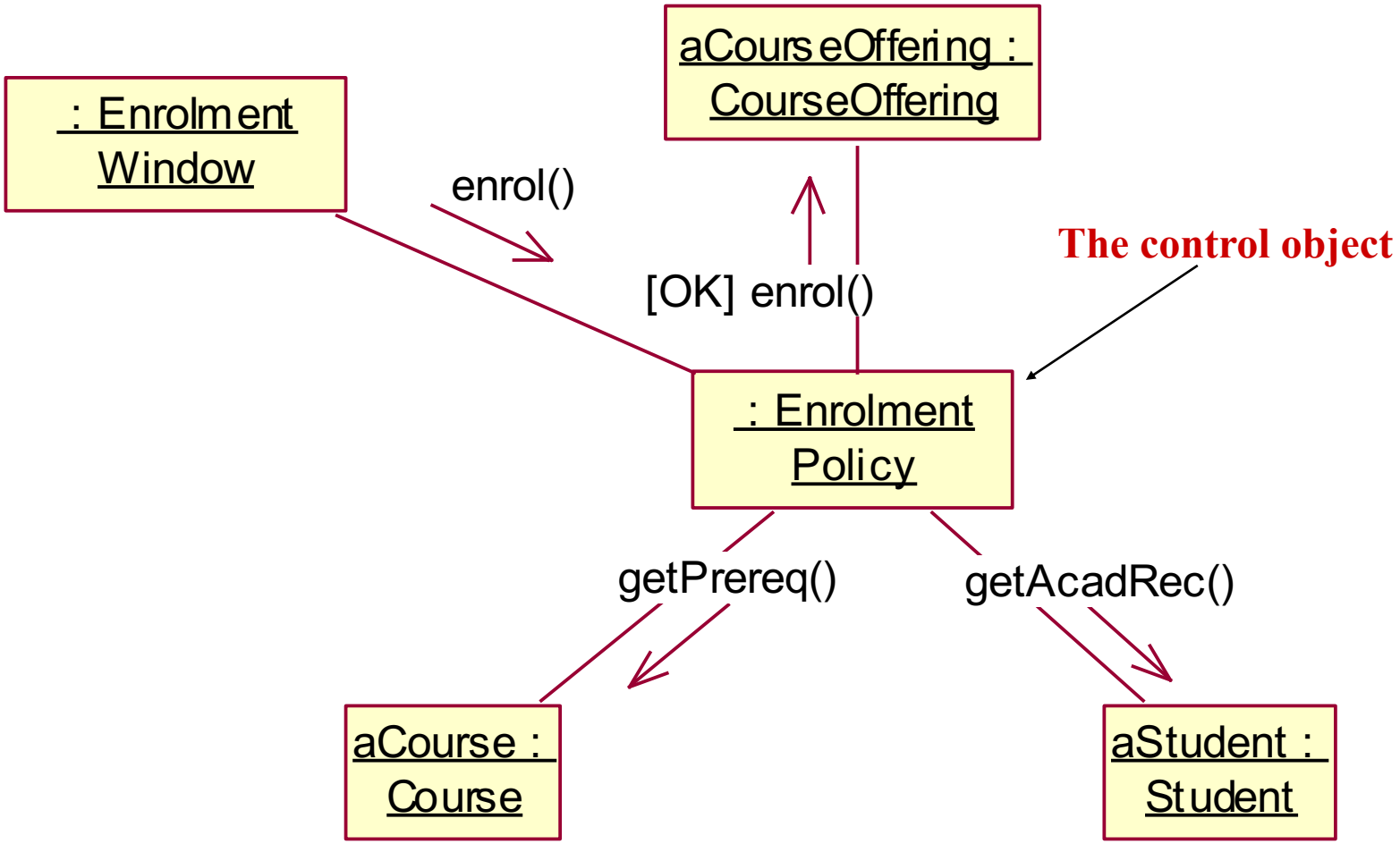
Control management – *solution 2*



Control management – *solution 3*



Control management – *solution 4*



Boundary-Control-Entity (BCE) Approach!

Coupling issues

- The BCE approach specifies three layers of classes
- Objects communicate within a layer and between the adjacent layers
- *Intra-layer coupling*
  - desirable
  - localizes software maintenance and evolution to individual layers
- *Inter-layer coupling*
  - to be minimized
  - communication interfaces to be carefully defined
- The *Law of Demeter* can be used to reduce the inter-layer class coupling

Law of Demeter

- The target of a message (in class methods) 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 (*strong law* ➔ inherited attributes cannot be used to identify the target object)
  4. An object created by the method
  5. An object referred to by a global variable
- Also known as the "*don't talk to strangers*"