Object Oriented Analysis and Design (OOA&D)

with the *Unified Modeling Language* (UML)

OOA&D

- The **OOA (Object Oriented Analysis) phase** makes use of an object-based approach to define WHAT the software product is required to do
- whereas the **OOD (Object Oriented Design) phase** deals with **HOW** the product does what specified in the OOA phase)
- OOA and OOD provides, from different viewpoints, a *correct*, *complete* and *consistent* representation of:
  - static and structural aspects of the product (*data view*)
  - functional aspects of the product (*behavioral view*)
  - control aspects of the product (*dynamic model*)

OOA methods

- An OOA method describes the set of procedures, techniques, notations and tools that defines a systematic approach to manage the OOA phase
- An OOA method takes as input the *user requirements* definition (described in the analysis requirements document)
- An OOA method yields as output the *system models* that define the system requirements specification (described in the analysis requirements document, as well)
- OOA methods make use of *visual notations* (*diagrams*) that can be integrated by textual specifications (e.g., in structured natural language)
- The development of OOA models is not a sequential process (e.g., data model first, then behavioral model and finally dynamic model)
- The model building activity is inherently *concurrent* and each model provides useful inputs to the remaining models
- OOA methods exploit *iterative* approaches that incrementally build the OOA models through step-wise refinement (*iterations*)

Some OOA (and OOD) methods

- Catalysis
  - specifically recommended for *component based distributed* software systems
- Objectory
  - created by Ivar Jacobson
  - founded on the identification of usage scenarios (*use case driven*).
- Shlaer/Mellor
  - specifically recommended for *real-time* software systems
- OMT (Object Modeling Technique)
  - created by James Rumbaugh
  - based on iterative modeling approaches
  - specifically focused on the *OOA phase*
- Booch
  - Created by Grady Booch
  - based on iterative modeling approaches
  - specifically focused on the *OOD phase*
- Fusion
  - developed in the middle 80's by HP
  - based on the merge of OMT and Booch methods
  - first attempt to standardize object-oriented software development processes

OOA (and OOD) notations

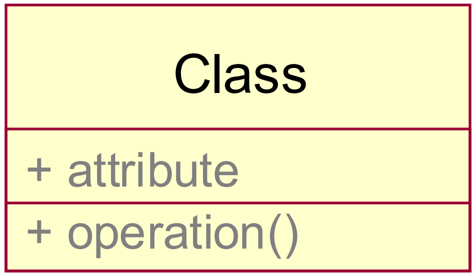
- Each of the various OOA (and OOD) methods has introduced its own modeling notation
- **UML** (**Unified Modeling Language**) has been introduced to unify the various notations and provide a standard language for modeling software systems
- In 1997 the OMG (Object Management Group) published the specification of *UML version 1.0*
- In July 2005 the OMG issued *UML version 2.0*, a major revision aligned with the OMG MDA (Model Driven Architecture)
- UML consists of a set of *diagram types* (each one with given semantics and notation) and a set of modeling extension (denoted as *profiles*)
- **UML is a modeling language, not a method!**

Data View

- Represents the logical organization of the product's data, from a *static and structural viewpoint*
- Is specified by use of *class diagrams*, which define:
  - classes
  - class attributes
  - class operations
  - associations between classes
- It is of paramount importance, since an object-oriented software systems basically consists of a set of (classified) objects that cooperate
- It is built by use of an *iterative* and *incremental* approach
- *Creativity* plays an important role, along with the experience, skills and domain knowledge of the software analyst

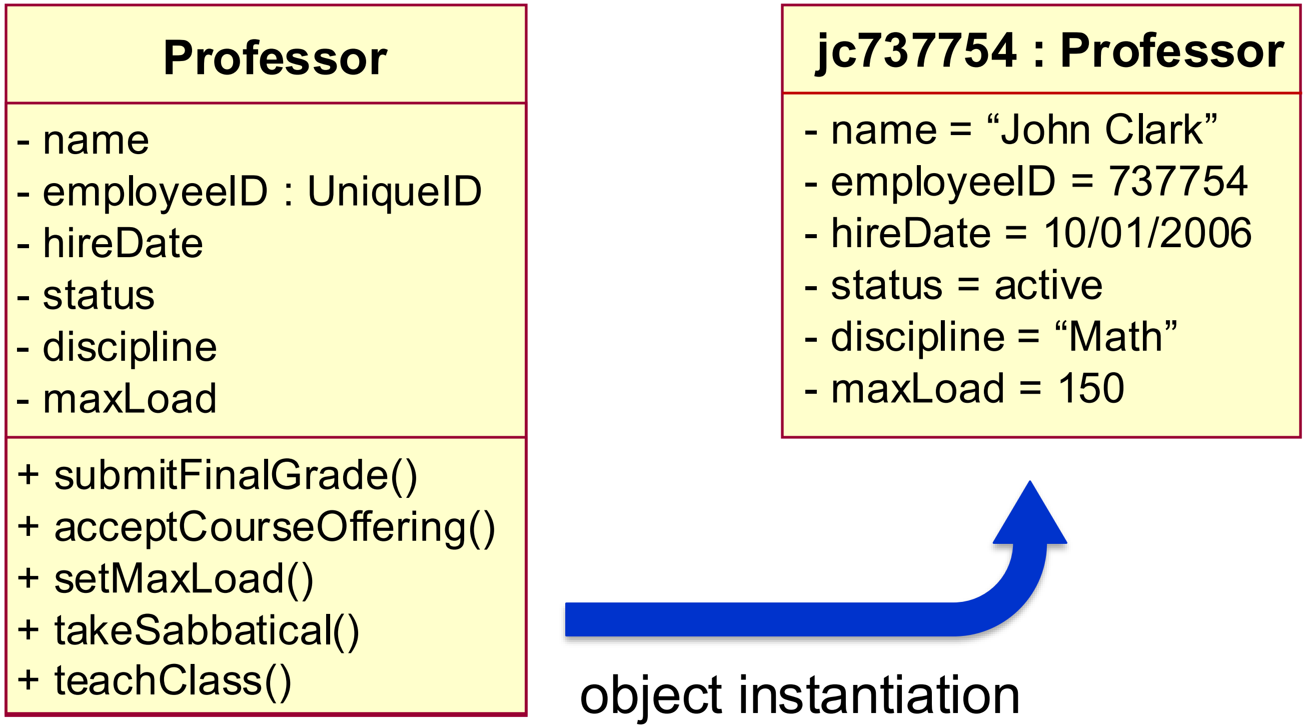
What Is a Class?

- A class is a description of a set of objects that share the same *attributes*, *operations*, *relationships*, and semantics
  - An object is an *instance of* a class
- A class is an abstraction that:
  - Emphasizes relevant characteristics
  - Suppresses other characteristics



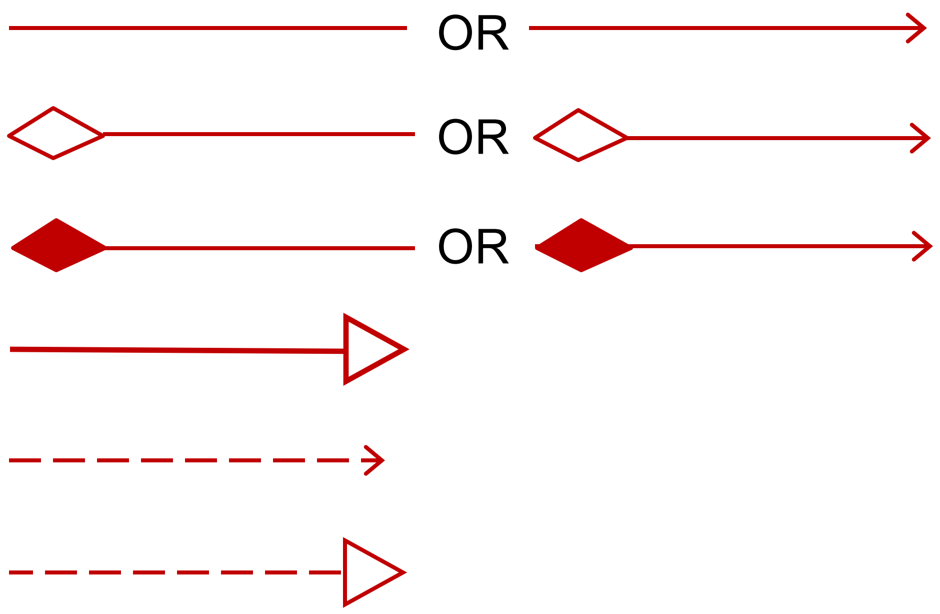
Representing Classes in the UML

- A class is represented using a rectangle with compartments



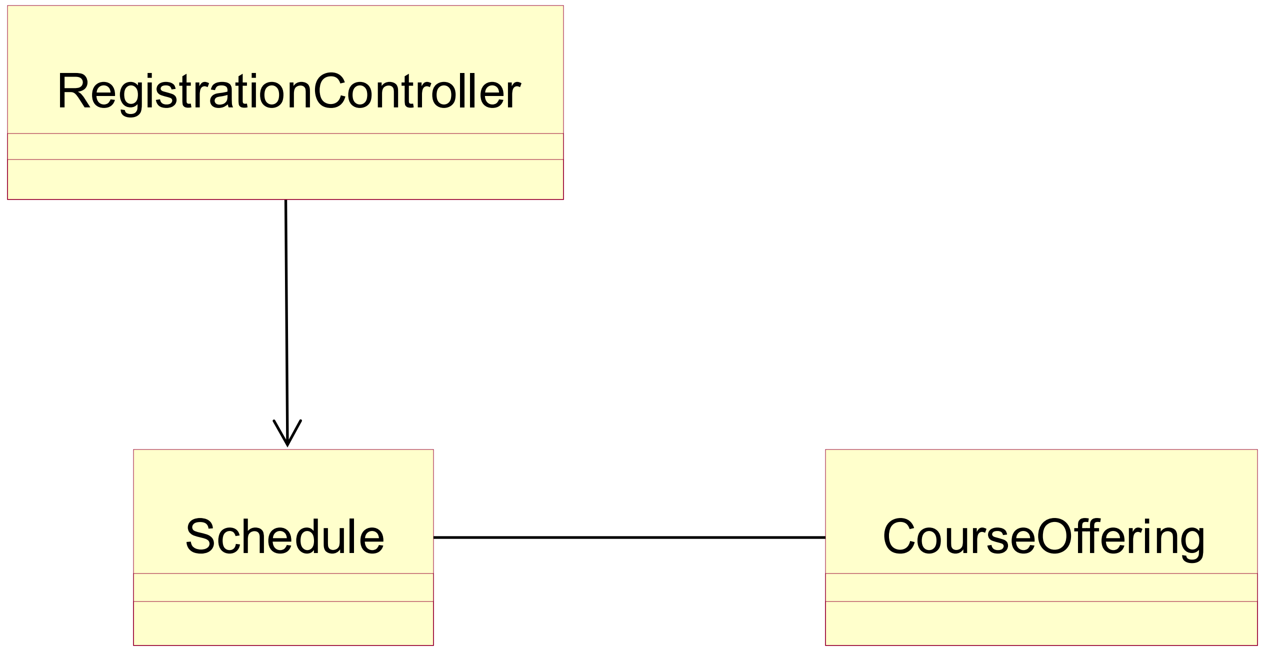
Class Relationships

- “The semantic connection between classes”
- Class diagrams may contain the following relationships:
  - Association
  - Shared
  - Composite
  - Generalization
  - Dependency
  - Realization



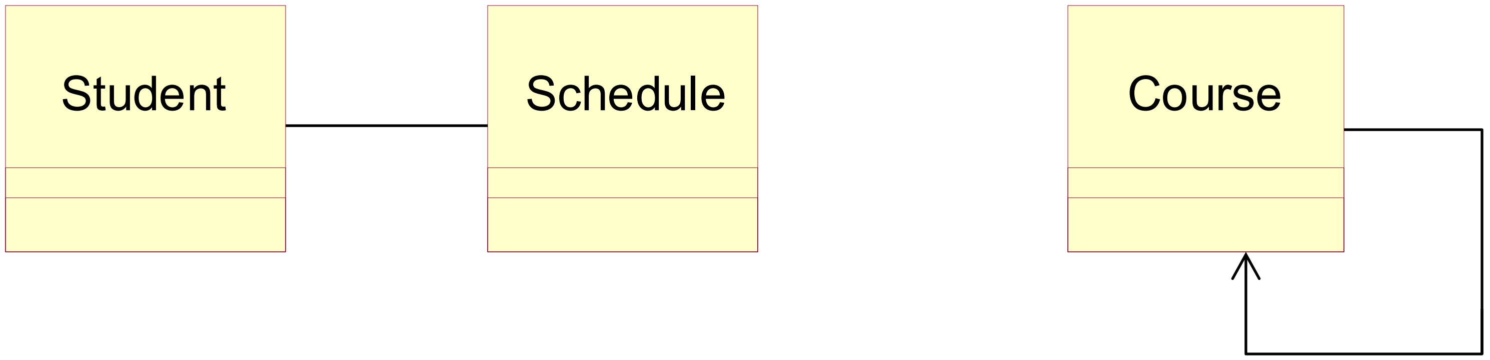
What Is Navigability?

Navigability indicates that it is possible to navigate from an associating class to the target class using the association



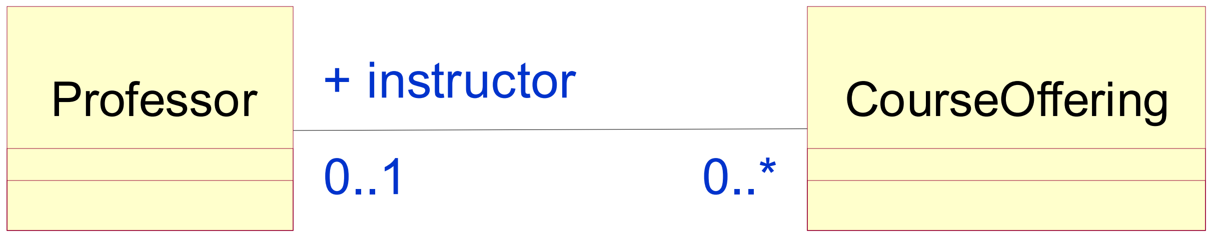
What Is an Association?

- The semantic relationship between two or more classifiers that specifies connections among their instances
  - A structural relationship, specifying that objects of one thing are connected to objects of another



What Is Multiplicity?

- Multiplicity is the number of instances one class relates to ONE instance of another class
- For each association, there are two multiplicity decisions to make, one for each end of the association
  - For each instance of Professor, many Course Offerings may be taught
  - For each instance of Course Offering, there may be either zero or one Professor as the instructor



Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One	0..1
Specified Range	2..4

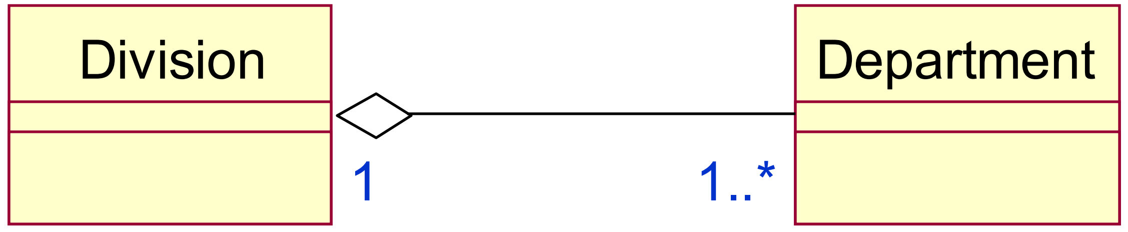
What Is Aggregation?

- An aggregation is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts
- There are two kinds of aggregations:
  - **Shared** (weaker, *without existence-dependency*)
  - **Composite** (stronger, *with existence-dependency*)
- An aggregation is an "*is a part-of*" relationship
- Multiplicity is represented like other associations

Shared Aggregation

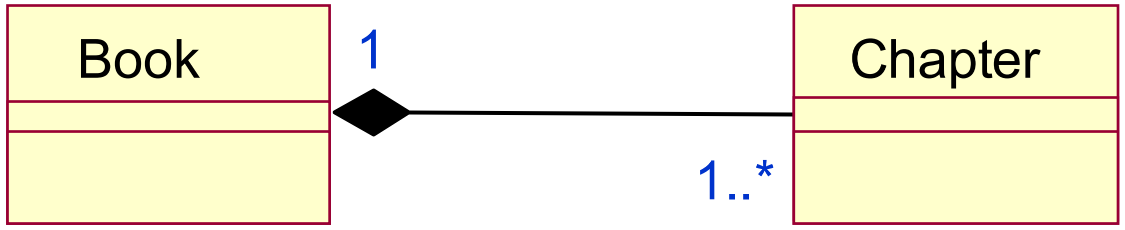
A ***shared aggregation***:

- Does not represent ownership of the part by the whole
- Is represented by a hollow diamond on the opposite end



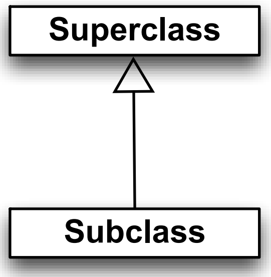
Composite Aggregation

- A ***composition*** is a stronger form of association in which the composite has sole responsibility for managing its parts, such as its allocation and deallocation
- It is represented by a diamond-filled adornment on the opposite end



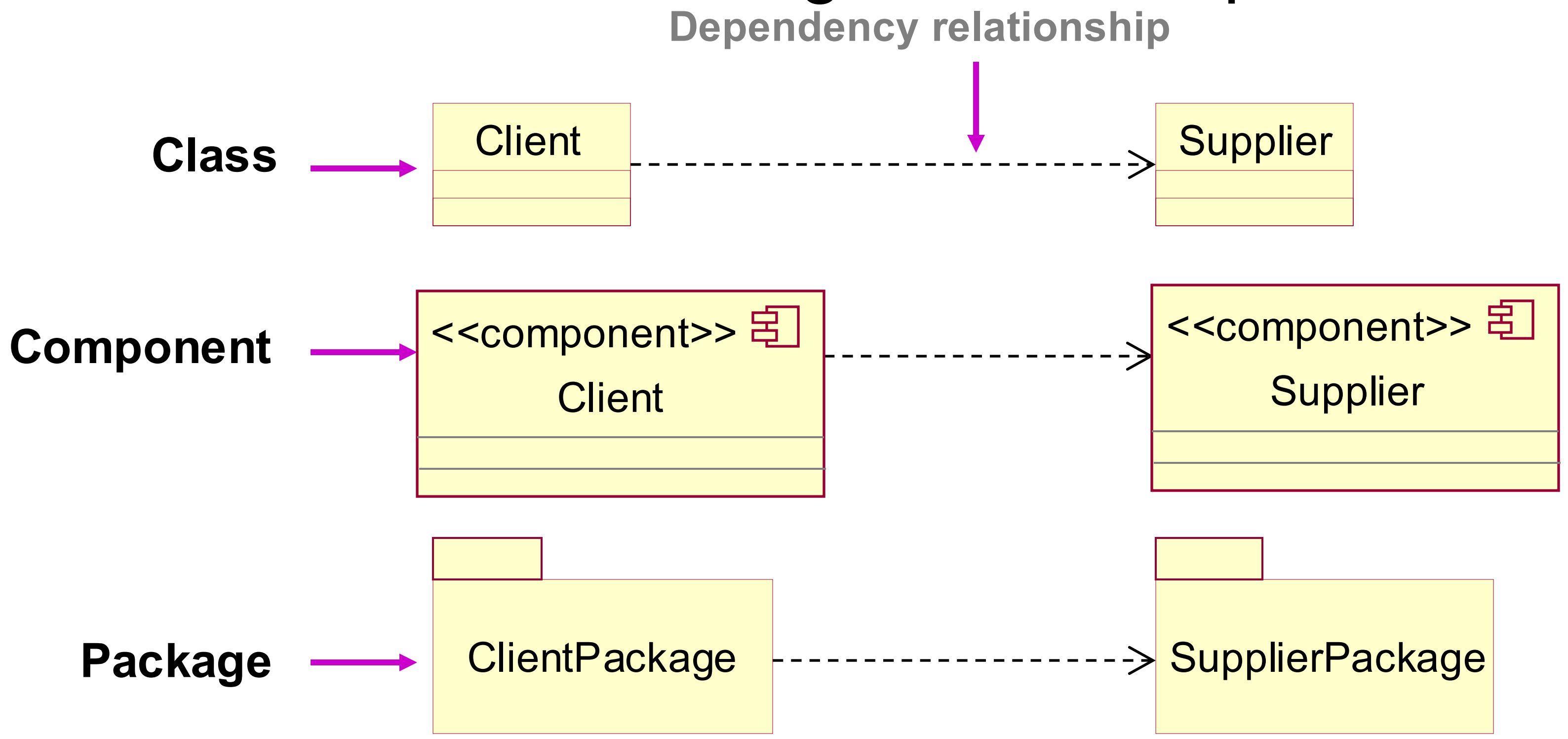
What Is Generalization?

- A relationship among classes where one class shares the structure, behavior, or both of one or more classes
- Used to represents relations of type:
  - "can-be" (to be read *from superclass to subclass*)
    - e.g., `Student` can be a `TeachingAssistant`
  - "is-a-kind-of" (to be read *from subclass to superclass*)
    - e.g., `TeachingAssistant` is a kind of `Student`
- Multiple inheritance
  - e.g., `TeachingAssistant` is also a kind of `Teacher`
- "Visualized" by use of a *solid line with an arrowhead* pointing to the superclass



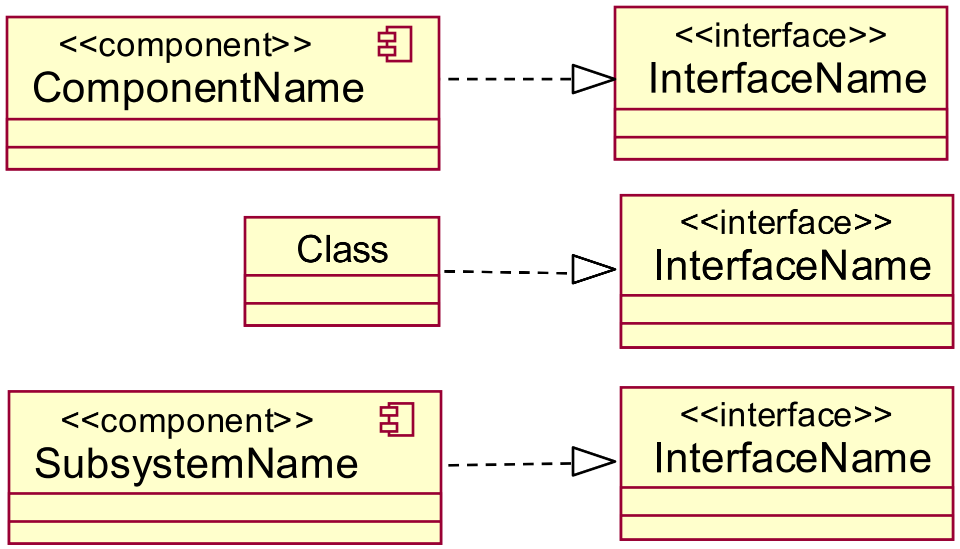
What Is Dependency?

- A relationship between two model elements where a change in one may cause a change in the other
- Non-structural, “using” relationship

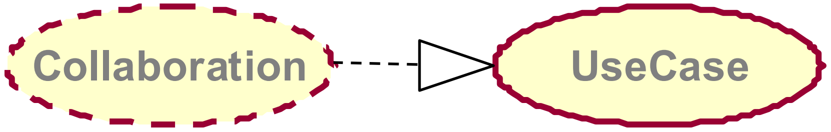


What Is Realization?

- One classifier serves as the contract that the other classifier agrees to carry out, found between:
  - Interfaces and the classifiers that realize them

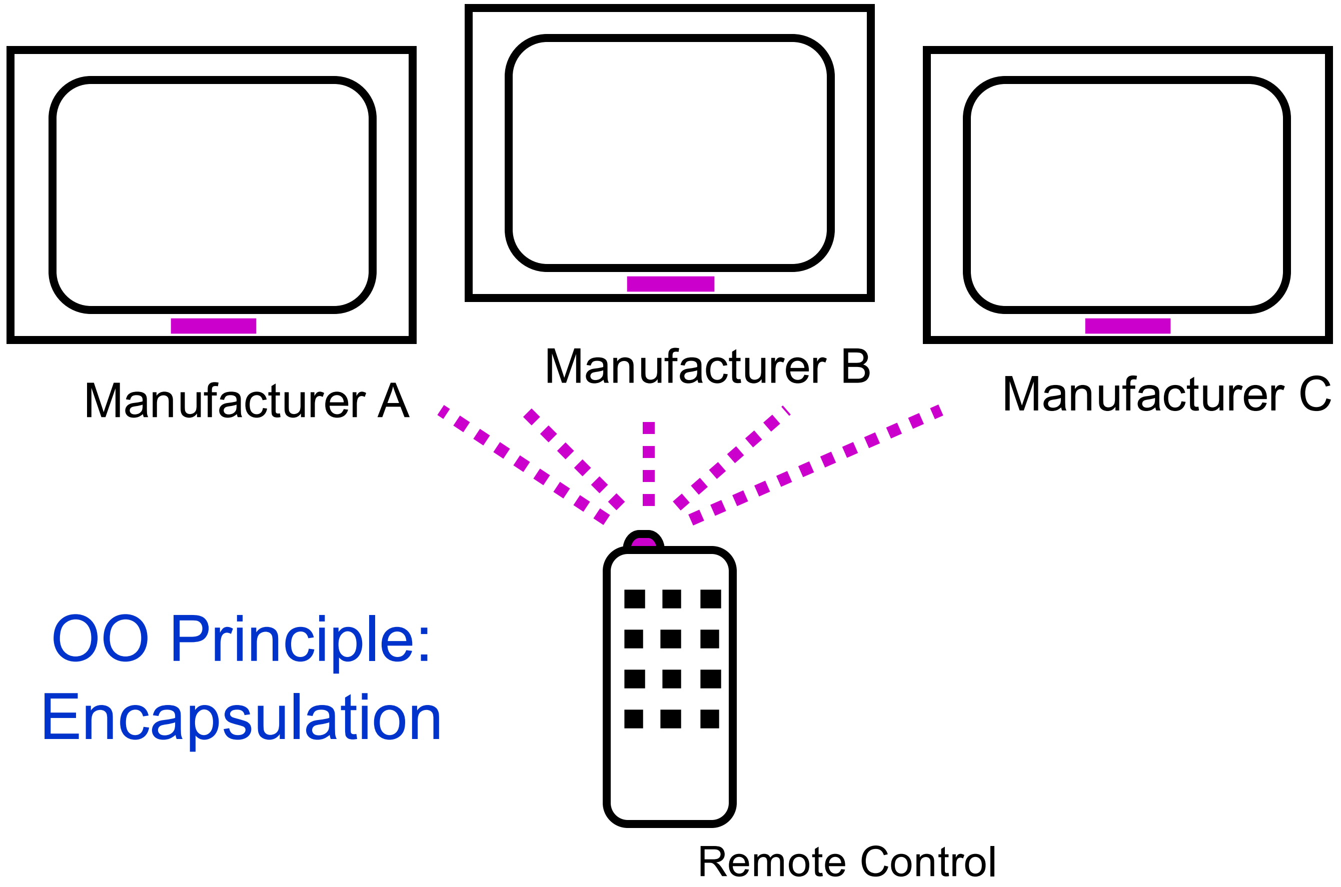


  - Use cases and the collaborations that realize them



What Is Polymorphism?

- The ability to hide many different implementations behind a single interface



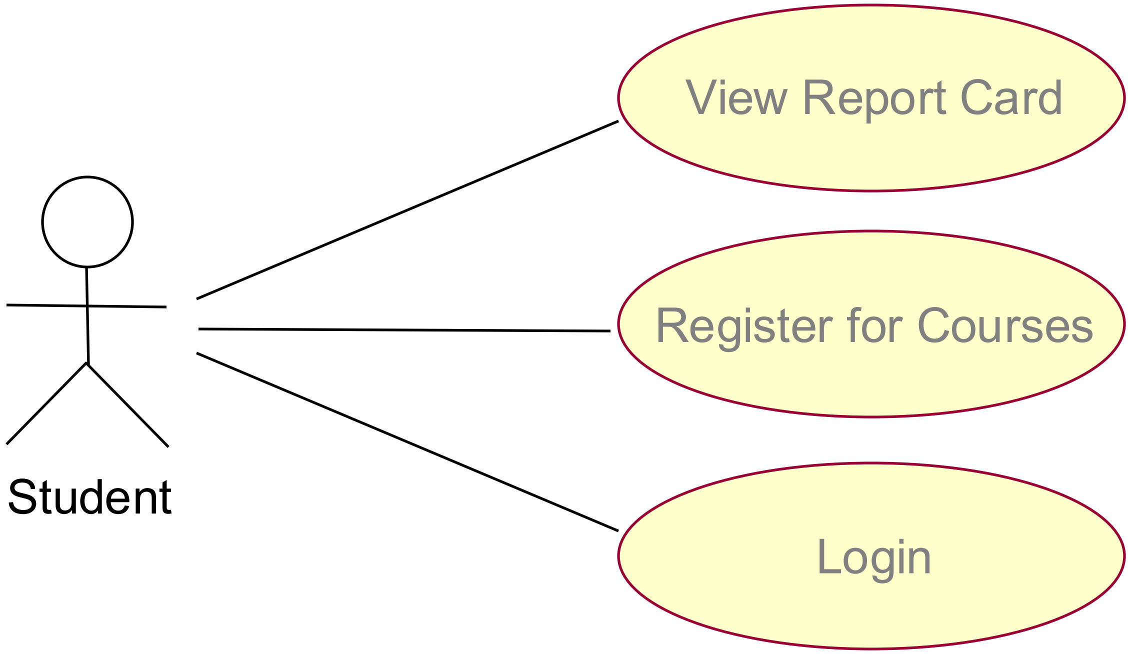
OO Principle:
Encapsulation

Behavioral View

- Specifies the functional aspects of the product from an operational point of view, by showing how objects collaborate and interact to provide the set of services offered by the product
- Consists of various diagrams:
  - *use case* diagram (to describe usage scenarios)
  - *activity* diagram (to specify execution flows)
  - *sequence* diagram (to specify objects interaction in time sequence)
  - *collaboration* diagram (to specify objects interaction)

What Is a Use-Case Diagram?

- A diagram that describes a system’s functional requirements in terms of use cases
- A diagram of the system’s intended functionality (use cases) and its environment (actors)



What Is an Activity Diagram?

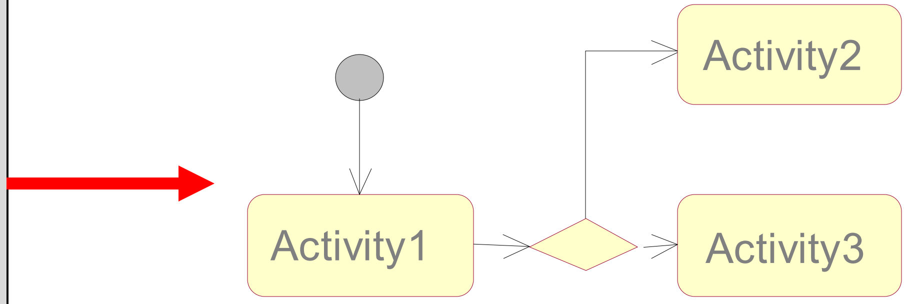
- An activity diagram can be used to specify the flow of activities in the use cases of the use case diagram
- It is essentially a flow chart, showing flow of control from one activity or action to another

Flow of Events

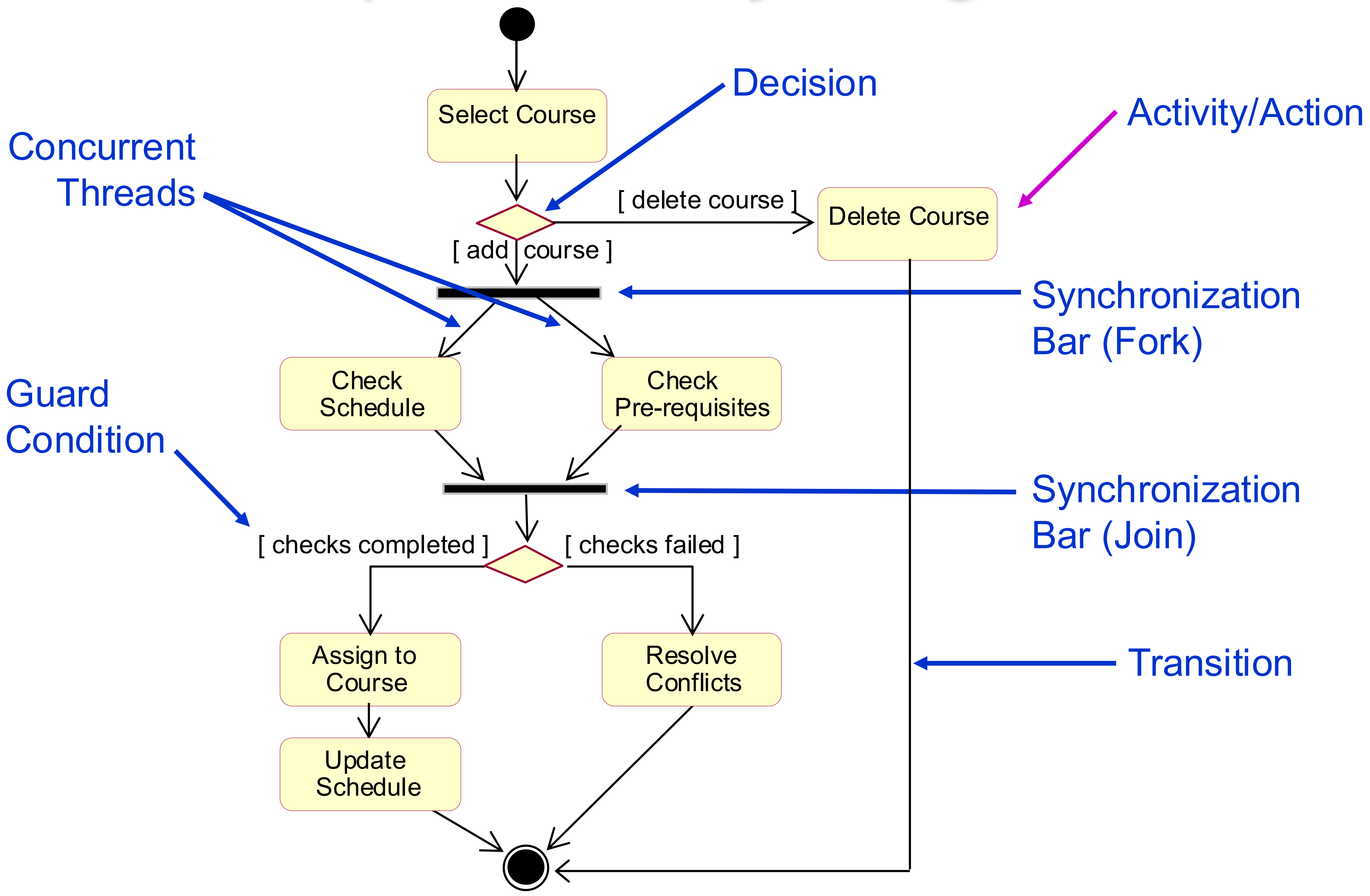
This use case starts when the Registrar requests that the system close registration.

1. The system checks to see if registration is in progress. If it is, then a message is displayed to the Registrar and the use case terminates. The Close Registration processing cannot be performed if registration is in progress.

2. For each course offering, the system checks if a professor has signed up to teach the course offering and at least three students have registered. If so, the system commits the course offering for each schedule that contains it.

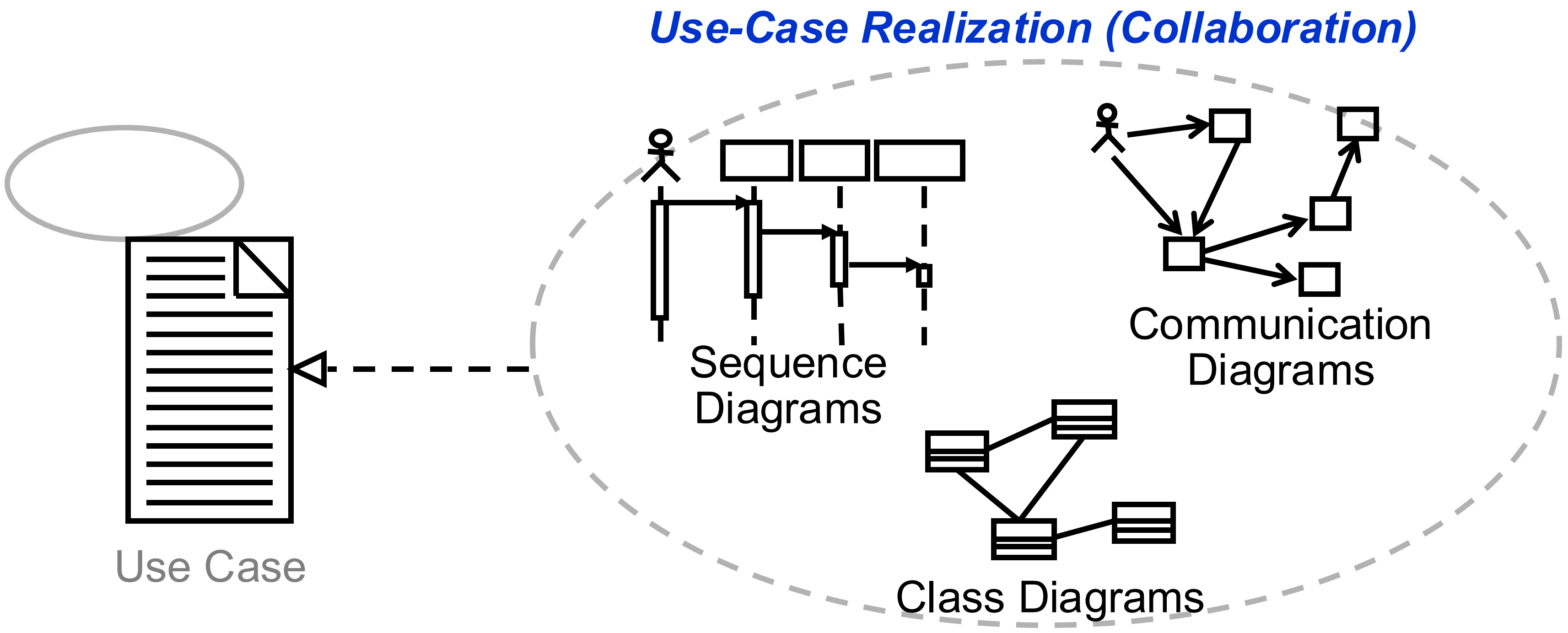


Example: Activity Diagram



Distribute Use-Case Behavior to Classes

- For each use-case flow of events:
  - Identify analysis classes
  - Allocate use-case responsibilities to analysis classes
  - Model analysis class interactions in Interaction diagrams



Allocating Responsibilities to Classes

- Use the BCE approach
  - ***Boundary Classes*** - Behavior that involves communication with an actor
  - ***Control Classes*** - Behavior specific to a use case or part of a very important flow of events
  - ***Entity Classes*** - Behavior that involves the data encapsulated within the abstraction

Allocating Responsibilities to Classes (cont.)

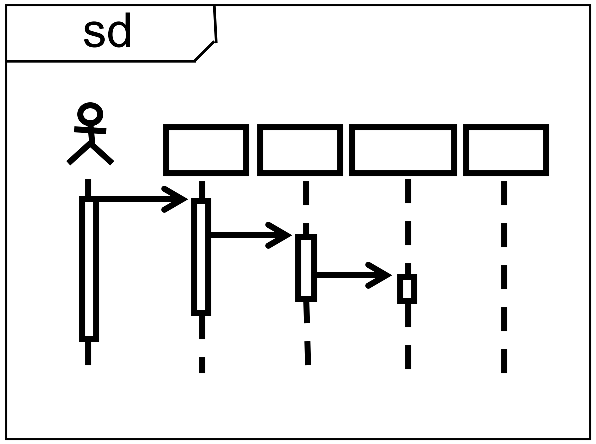
- Who has the data needed to perform the responsibility?
  - If one class has the data, put the responsibility with the data
  - If multiple classes have the data:
    - Put the responsibility with one class and add a relationship to the others
    - Create a new class, put the responsibility in the new class, and add relationships to classes needed to perform the responsibility
    - Put the responsibility in the *control class*, and add relationships to classes needed to perform the responsibility

What is an Interaction Diagram?

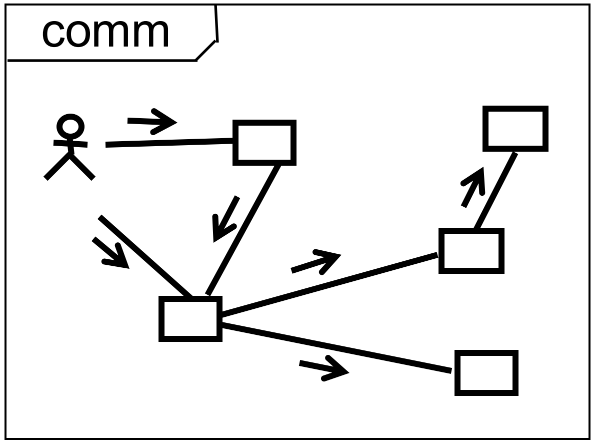
- Generic term that applies to several types of diagrams that emphasize object interactions, including the messages that may be dispatched among them
- Largely used interaction diagrams:
  - Sequence Diagram
  - Communication Diagram
- Additional interaction diagrams:
  - Timing Diagram
  - Interaction Overview Diagram

Interaction Diagrams

- Sequence Diagram
  - Time oriented view of interactions



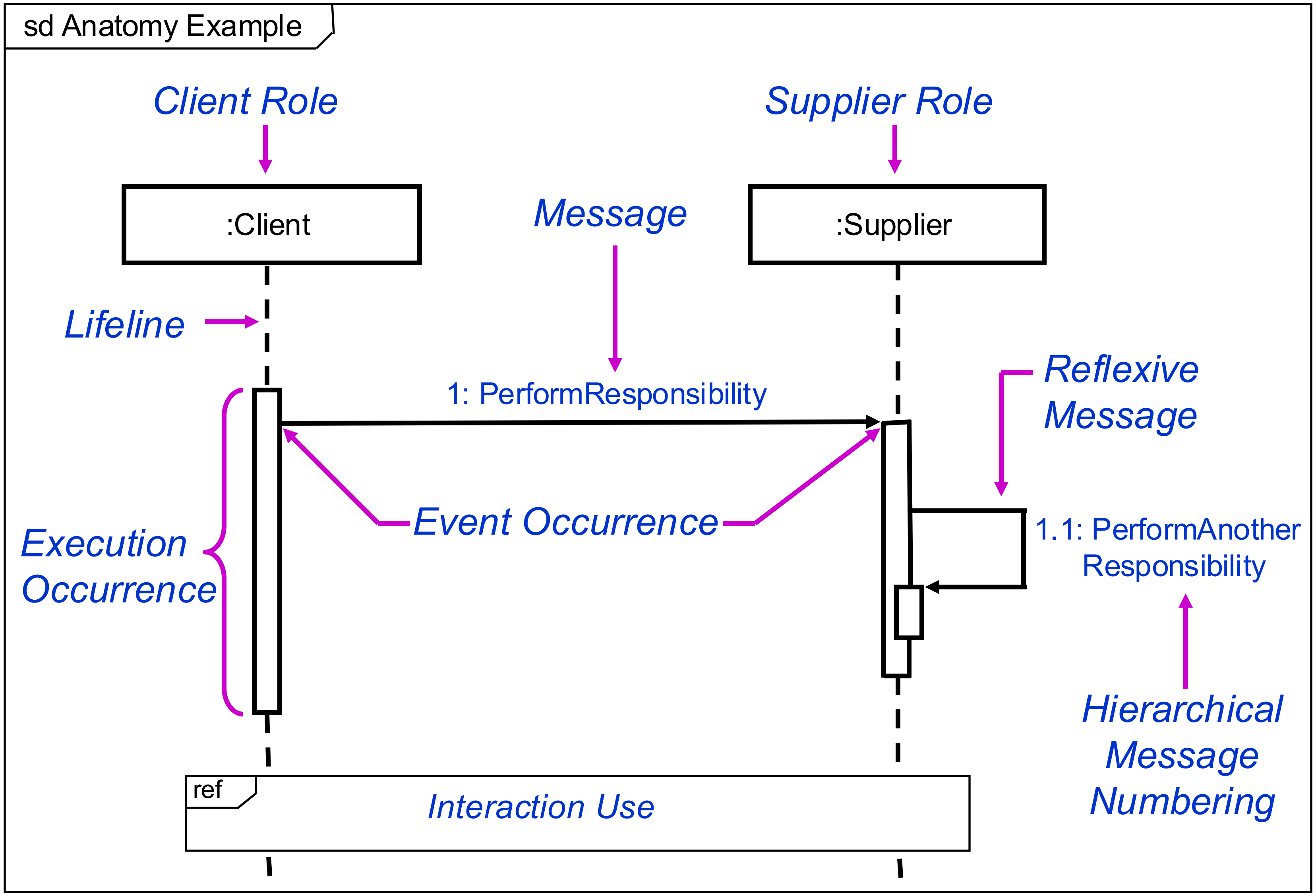
- Communication Diagram
  - Structural view of messaging roles or parts



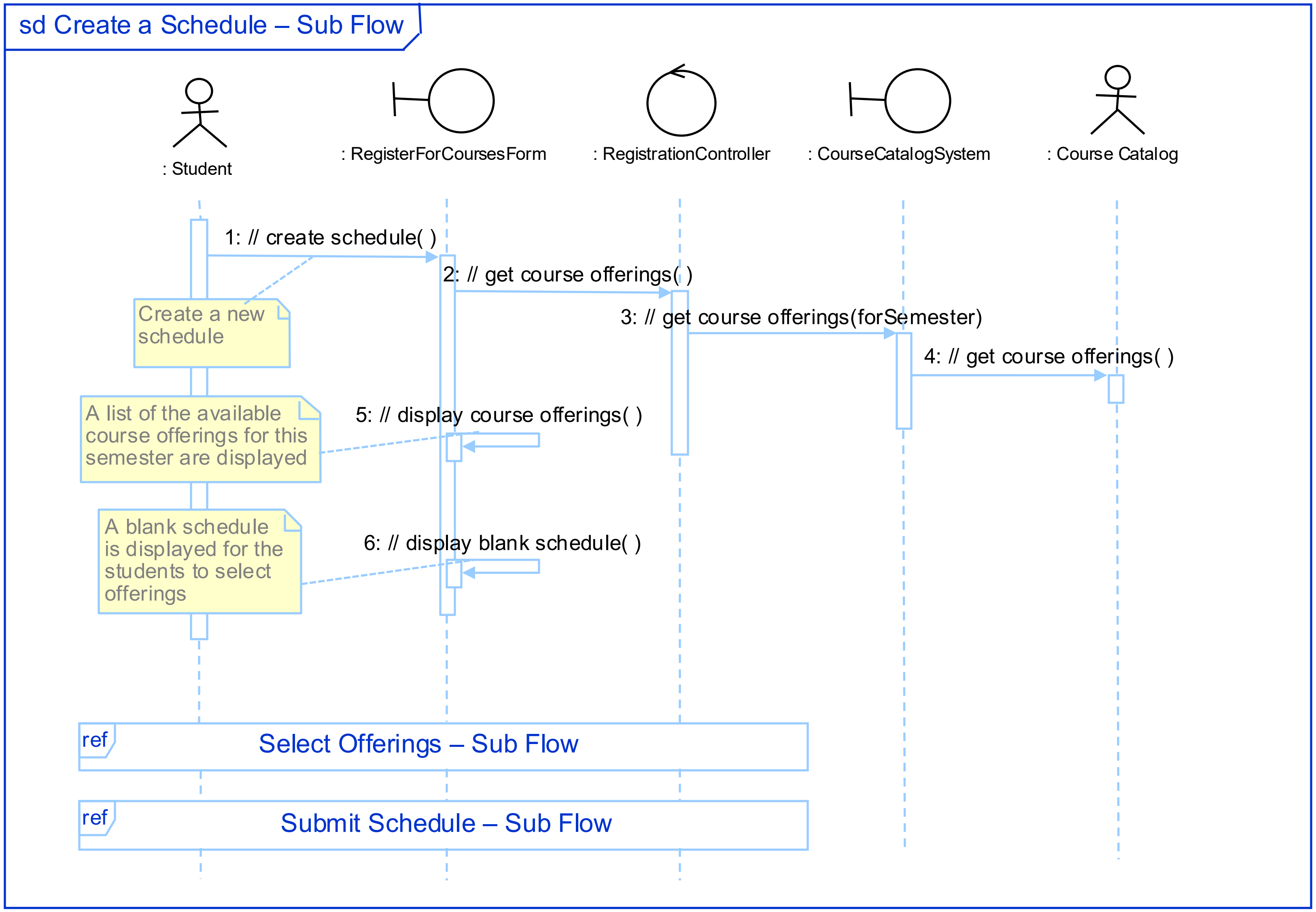
Sequence Diagrams vs. Communication Diagrams

Sequence Diagrams	Communication Diagrams
▪ Show the explicit sequence of messages ▪ Better for visualizing overall flow ▪ Better for OOA	▪ Show relationships in addition to interactions ▪ Better for visualizing patterns of collaboration ▪ Easier to use for OOD

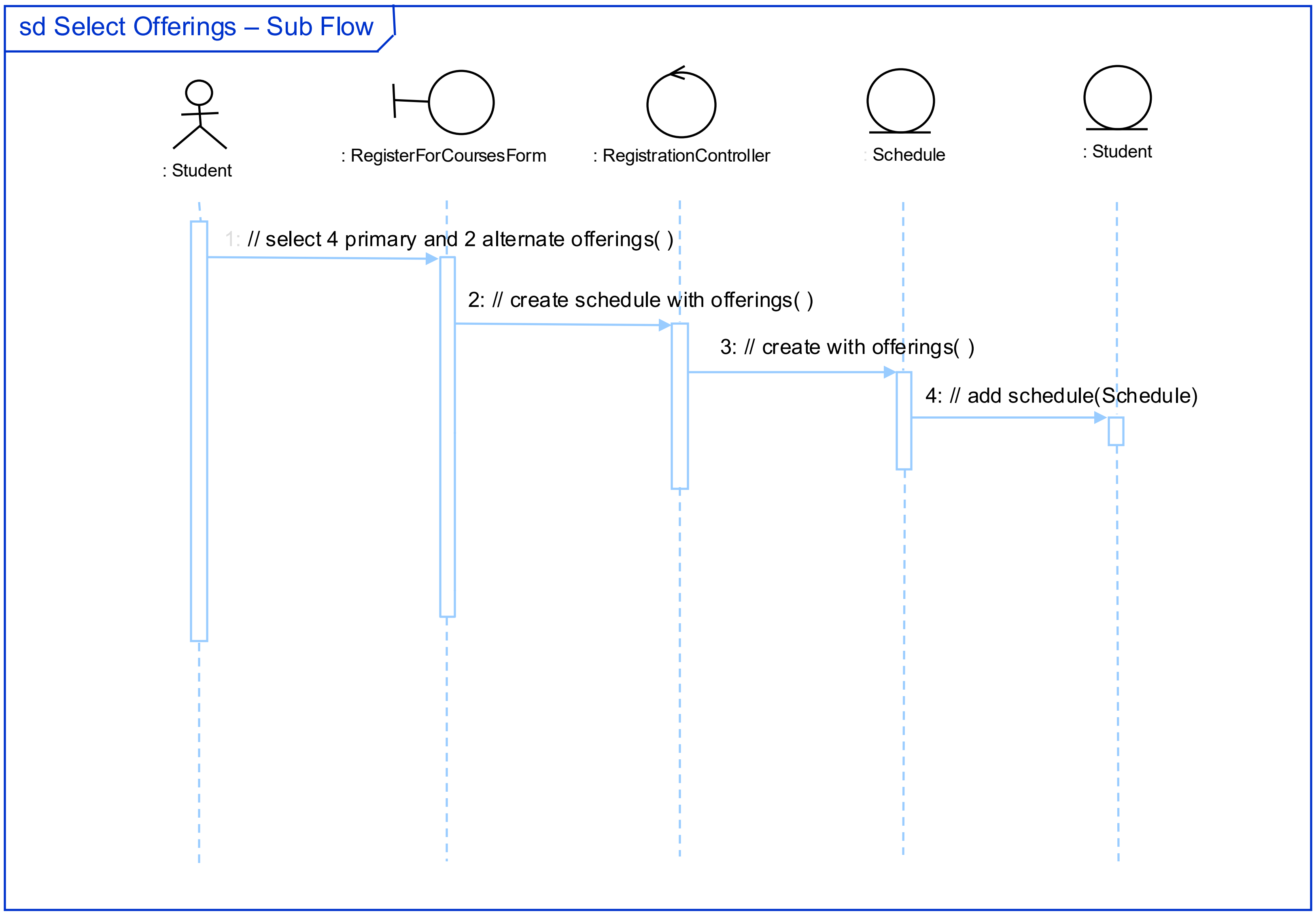
The Anatomy of Sequence Diagrams



Example: Sequence Diagram

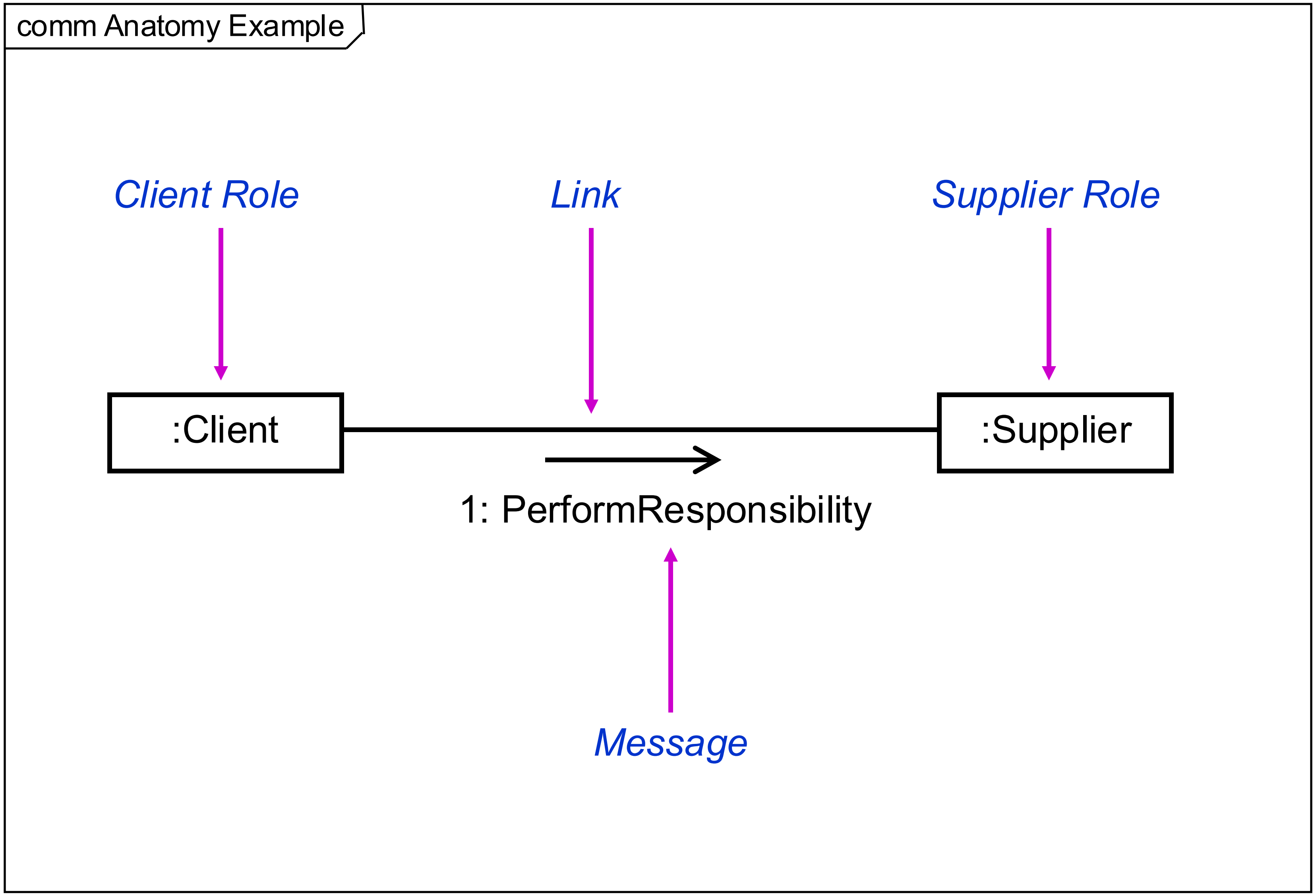

sd Create a Schedule – Sub Flow
: Student
: RegisterForCoursesForm
: RegistrationController
: CourseCatalogSystem
: Course Catalog
1: // create schedule( )
2: // get course offerings( )
3: // get course offerings(forSemester)
4: // get course offerings( )
5: // display course offerings( )
6: // display blank schedule( )
Create a new schedule
A list of the available course offerings for this semester are displayed
A blank schedule is displayed for the students to select offerings
ref
Select Offerings – Sub Flow
ref
Submit Schedule – Sub Flow

Example: Sequence Diagram (cont.d)

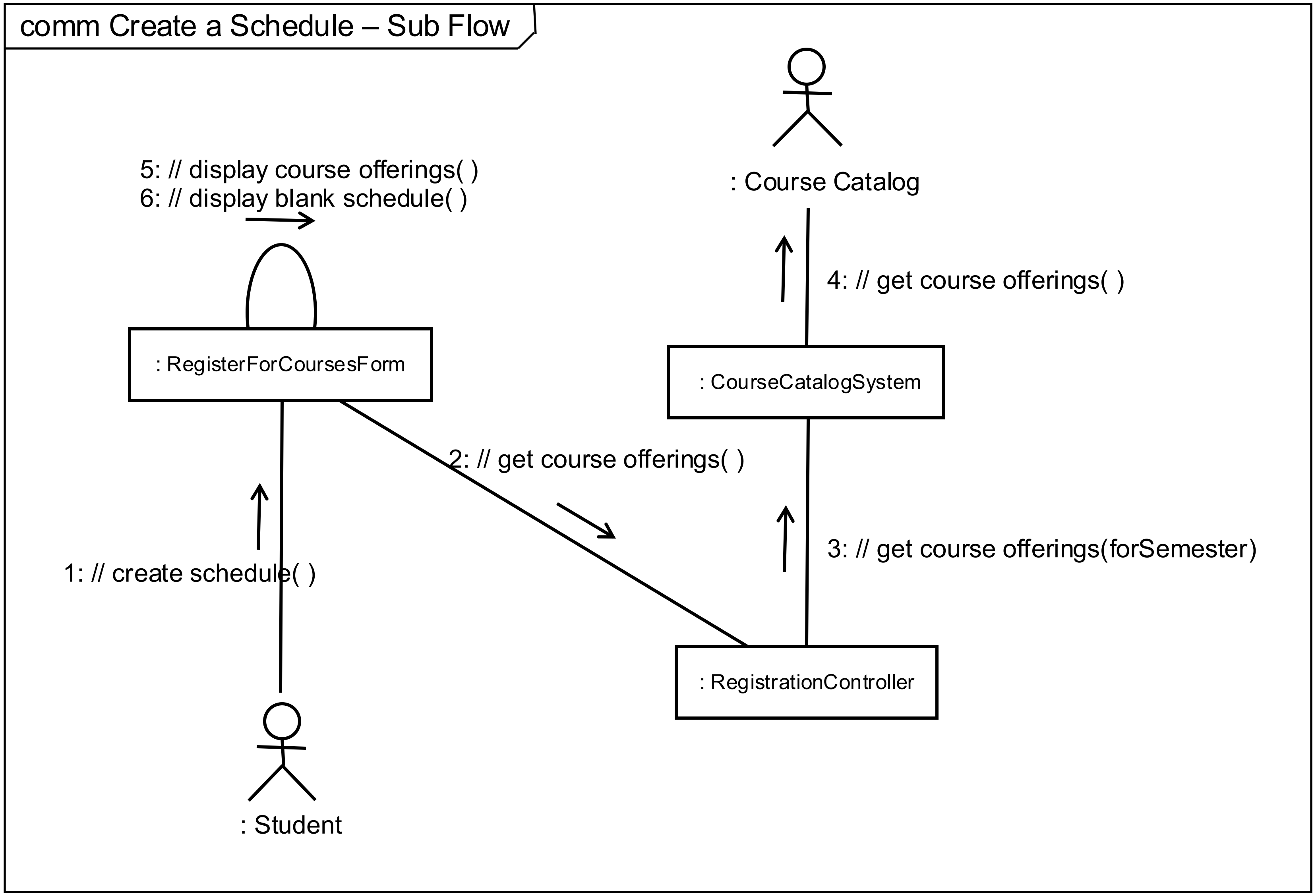


sd Select Offerings – Sub Flow
: Student
: RegisterForCoursesForm
: RegistrationController
: Schedule
: Student
1: // select 4 primary and 2 alternate offerings( )
2: // create schedule with offerings( )
3: // create with offerings( )
4: // add schedule(Schedule)

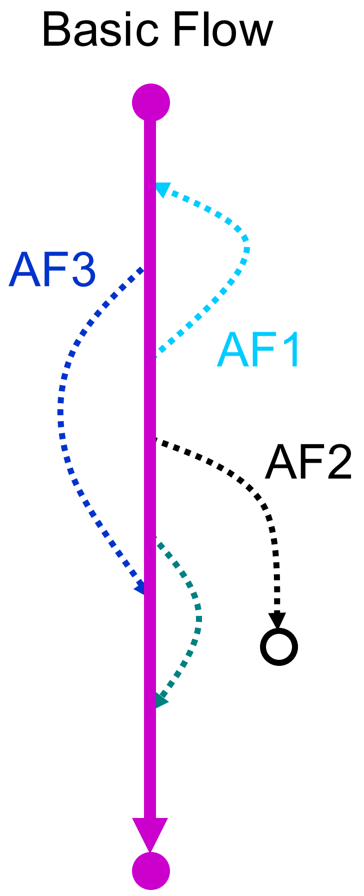
The Anatomy of Communication Diagrams



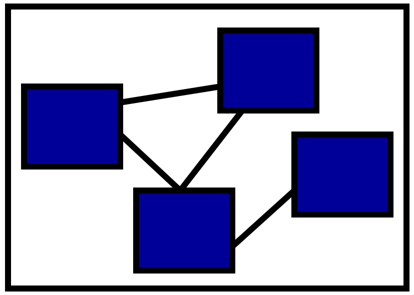
Example: Communication Diagram


comm Create a Schedule – Sub Flow
: Course Catalog
5: // display course offerings( )
6: // display blank schedule( )
4: // get course offerings( )
: RegisterForCoursesForm
: CourseCatalogSystem
2: // get course offerings( )
3: // get course offerings(forSemester)
1: // create schedule( )
: RegistrationController
: Student

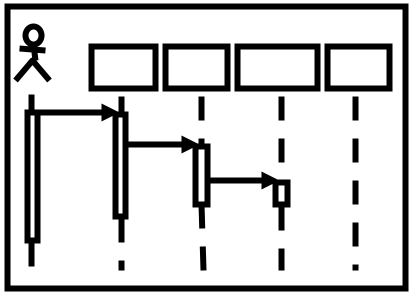
One Interaction Diagram is not enough



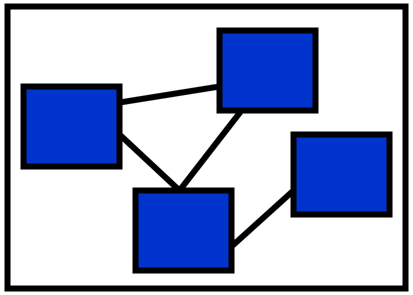
Alternate Flow 1



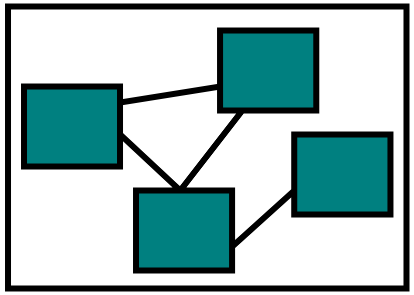
Alternate Flow 2



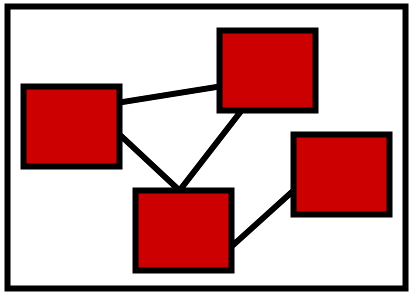
Alternate Flow 3



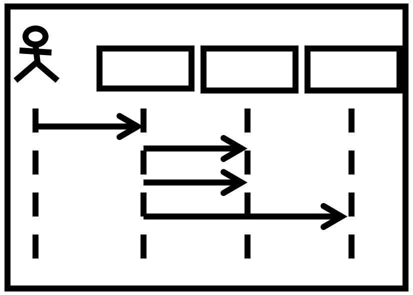
Alternate Flow 4



Alternate Flow 5



Alternate Flow n



Dynamic View

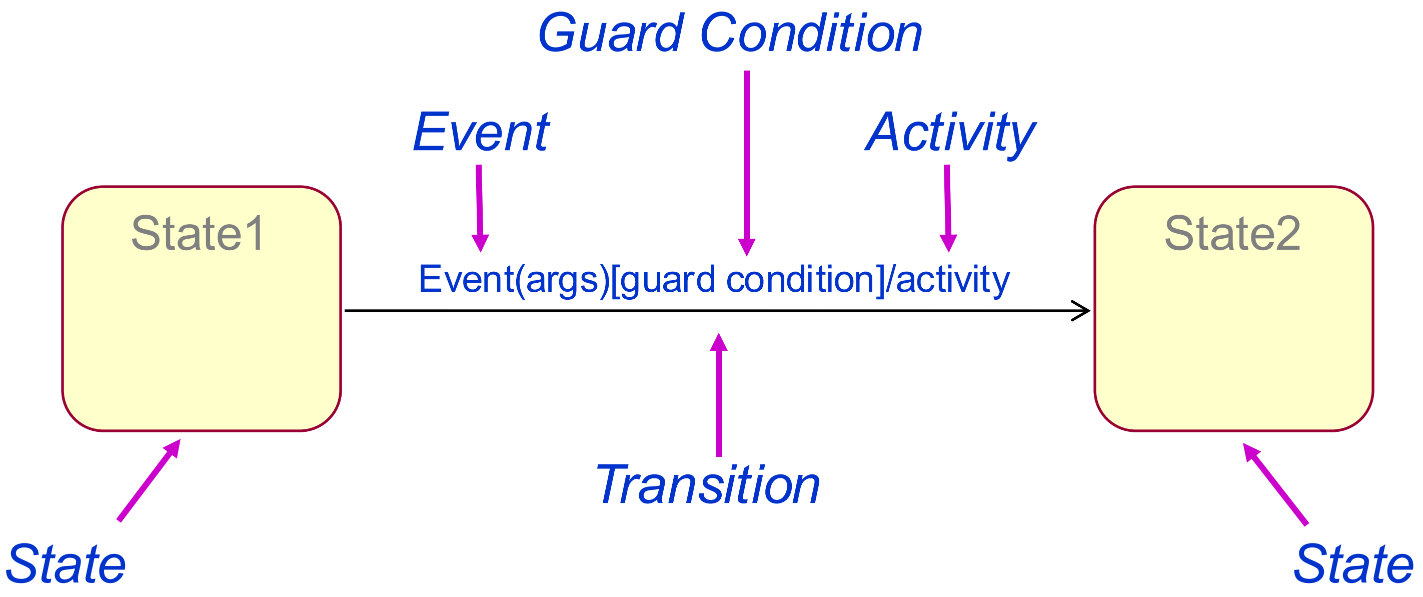
- Represents the dynamic behavior of class instances, in terms of possible **states**, along with **events** and conditions that originate state **transitions** (and actions to be executed at event occurrence)
- Makes use of *state machine diagrams*

Define States

- Purpose
  - Design how an object’s state affects its behavior
  - Develop state machines to model this behavior
- Things to consider:
  - Which objects have significant state?
  - How to determine an object’s possible states?
  - How do state machines map to the rest of the model?

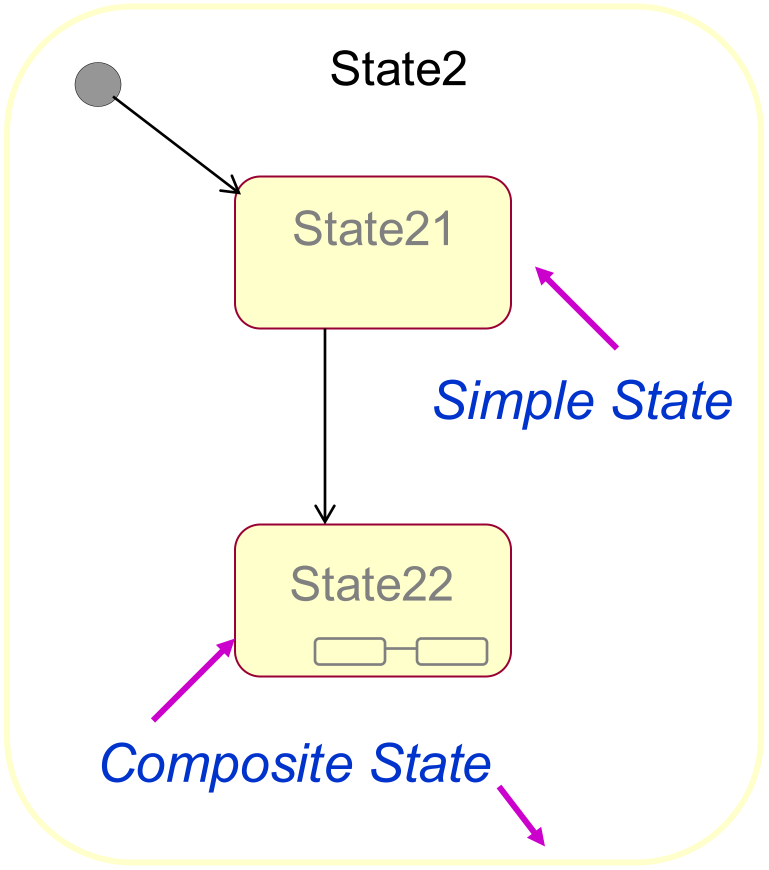
What is a State Machine?

- A directed graph of states (nodes) connected by transitions (directed arcs)
- Describes the life history of a reactive object



Types of States

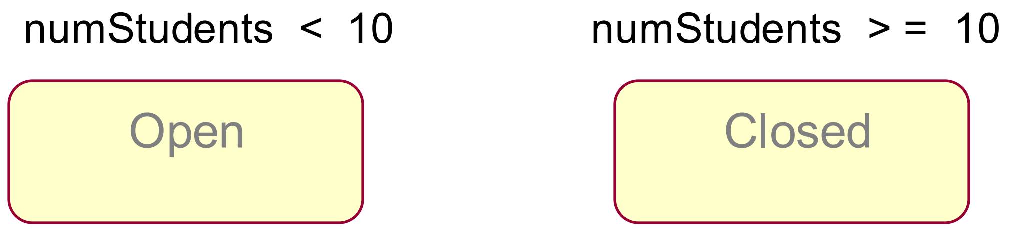
- *Simple* state - A state that has no nested states within it
- *Composite* state - A state that contains substates, the contents of which may be hidden in a particular view
  - Denoted with a small icon of two linked state symbols



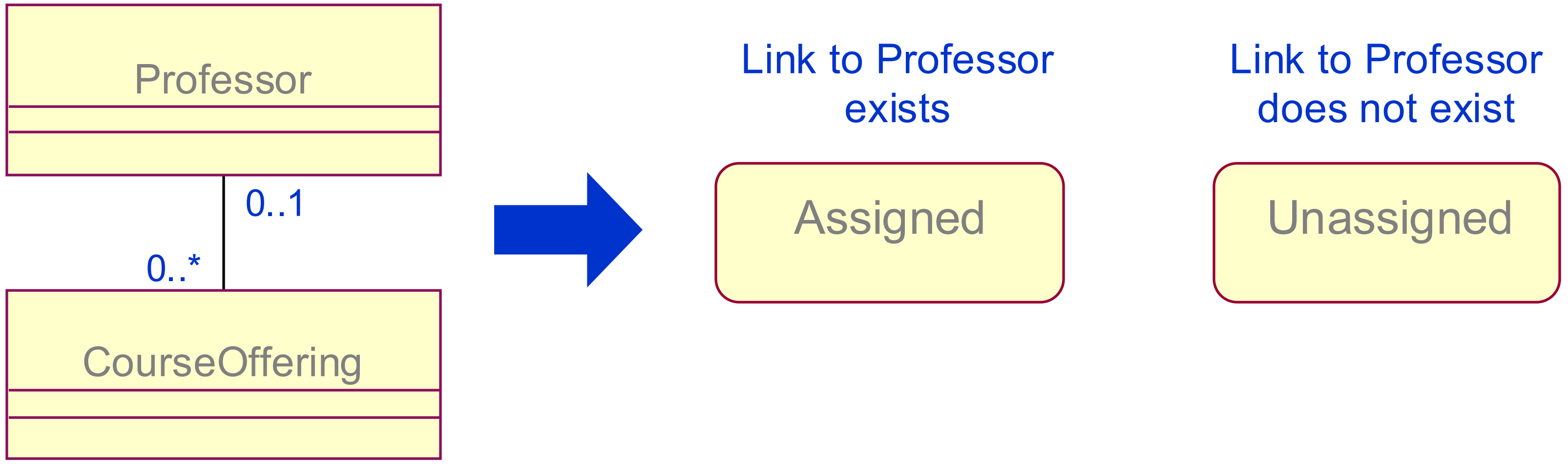
Identify and Define the States

- Significant, dynamic attributes

The maximum number of students per course offering is 10

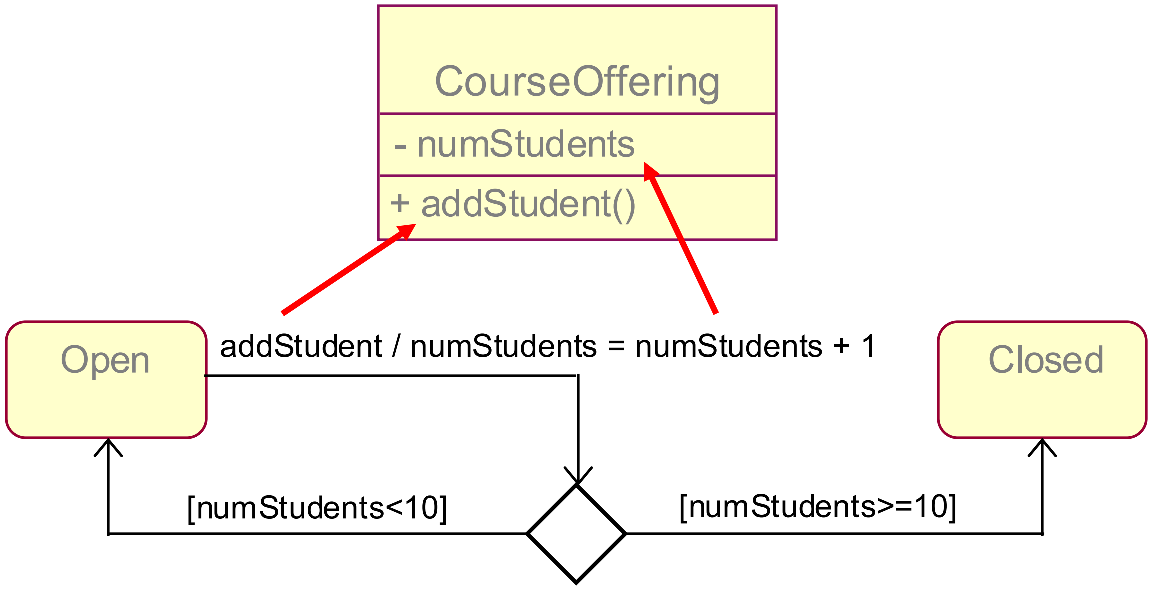


- Existence and non-existence of certain links

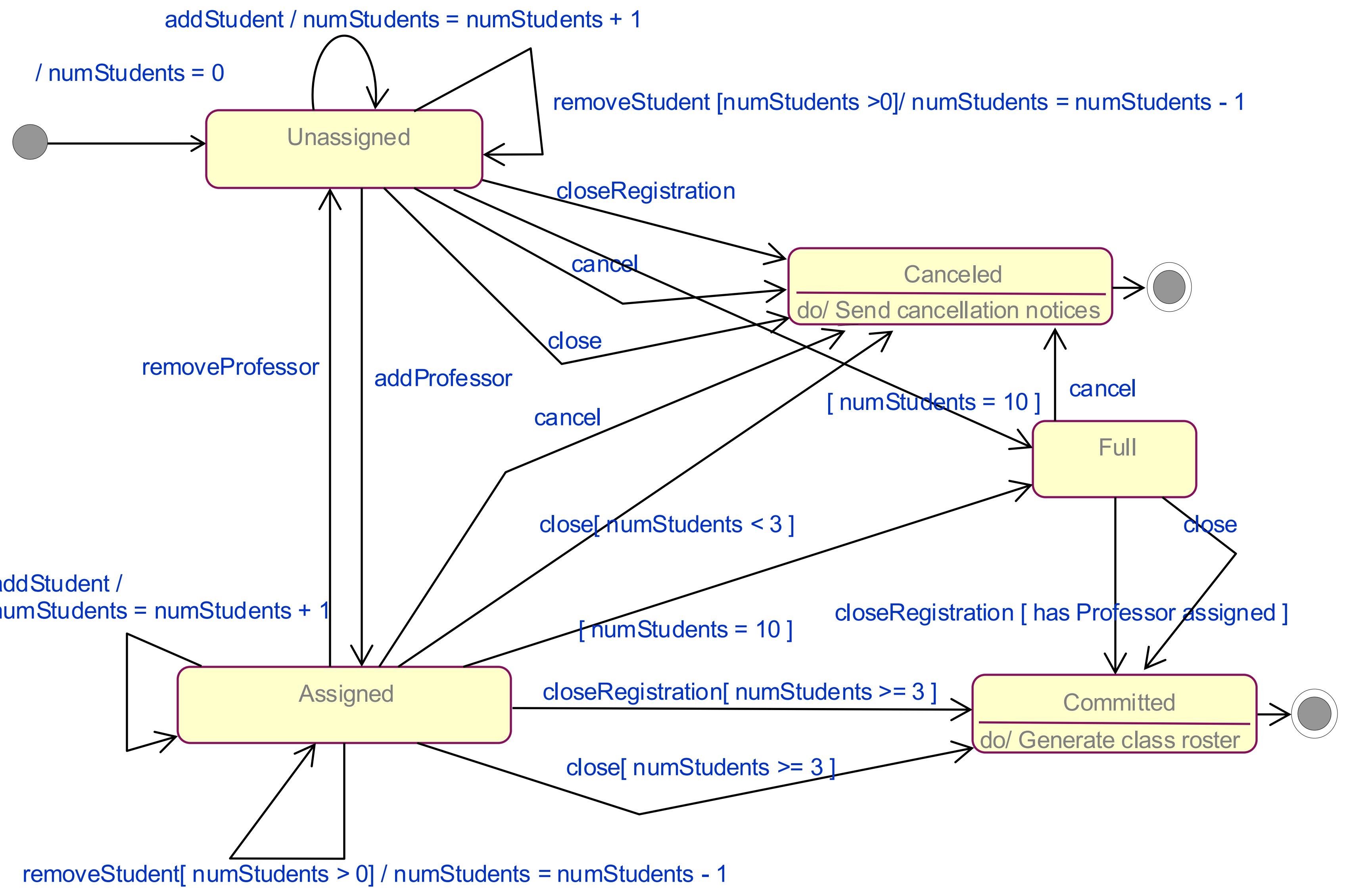


How Do State Machines Map to the Rest of the Model?

- Events may map to (the start) of operations
- Methods description should be updated with state-specific information
- States are typically represented using attributes



Example: State Machine (*CourseOffering*)


addStudent / numStudents = numStudents + 1
/ numStudents = 0
removeStudent [numStudents >0]/ numStudents = numStudents - 1
Unassigned
closeRegistration
cancel
Canceled
do/ Send cancellation notices
close
removeProfessor
addProfessor
cancel
[numStudents = 10]
cancel
Full
close[numStudents < 3]
close
addStudent /
numStudents = numStudents + 1
closeRegistration [has Professor assigned]
[numStudents = 10]
Assigned
closeRegistration[numStudents >= 3]
Committed
do/ Generate class roster
close[numStudents >= 3]
removeStudent[numStudents > 0] / numStudents = numStudents - 1

Object Oriented Design - OOD

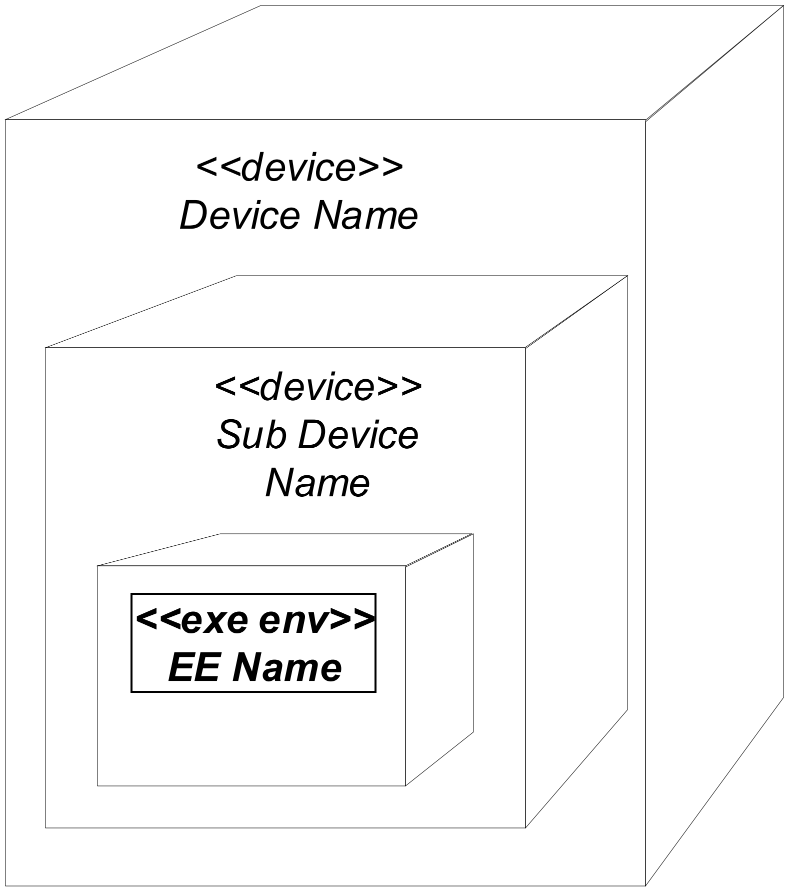
- The OOD phase consists of the following two sub-phases:
  - ***architectural*** (or ***system***) ***OOD***: defines the overall strategy to build a solution that solves the problem specified at OOA time. Decisions are taken that deal with the overall organization of the software (*system architecture*)
  - ***detailed OOD***: provides the complete definition of classes and associations to be implemented, as well as the data structures and the algorithm of methods that implement class operations
- According to an iterative and incremental development approach, the OOA model is “transformed” into the OOD model, which adds the technical details of the *hardware/software solution* that defines ***how*** the software must be implemented

Architectural OOD: Platform Configuration

- A platform configuration describes the hardware/software solution that defines how the functionality of the system can be distributed across physical nodes
  - Explain the relationship between model elements and their implementation, as well as their deployment
- It is obtained by:
  - defining the platform configuration by use of a *deployment diagram*
  - allocating system elements (artifacts) to nodes of the deployment diagrams

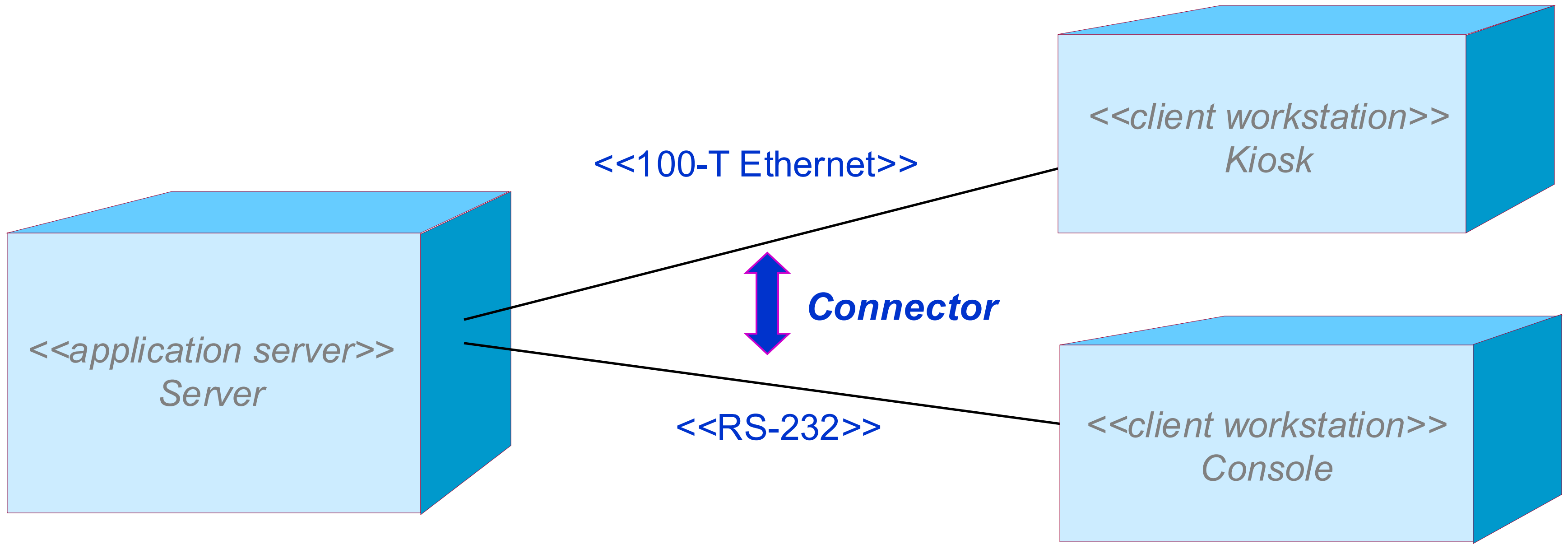
What Is a Node?

- Represents a run-time computational resource, and generally has at least memory and often processing capability
- Types:
  - Device - Physical computational resource with processing capability. Devices may be nested
  - Execution Environment - Represents particular execution platforms

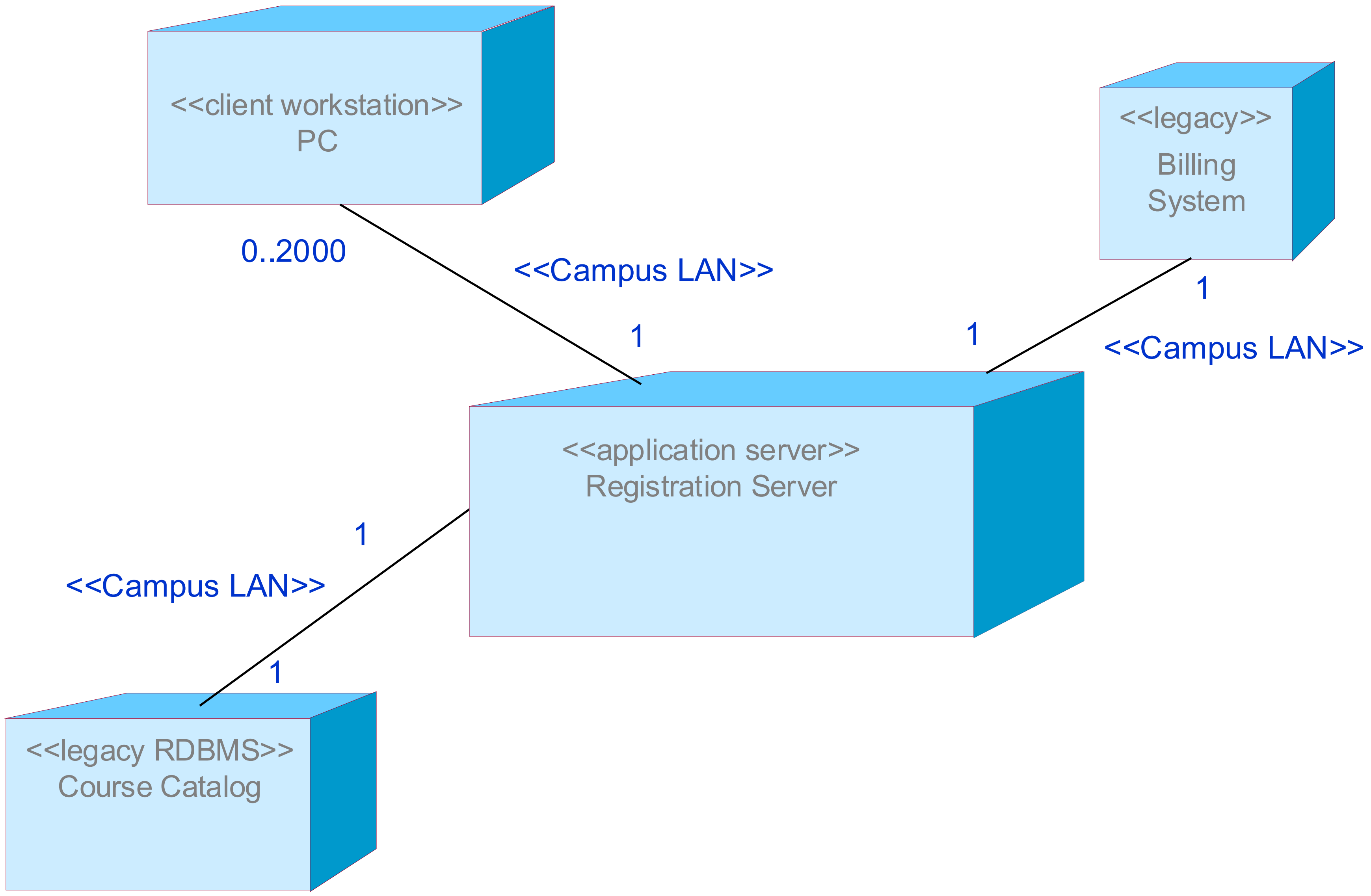


What Is a Connector?

- A connector represents a communication mechanism described by:
  - Physical medium
  - Software protocol



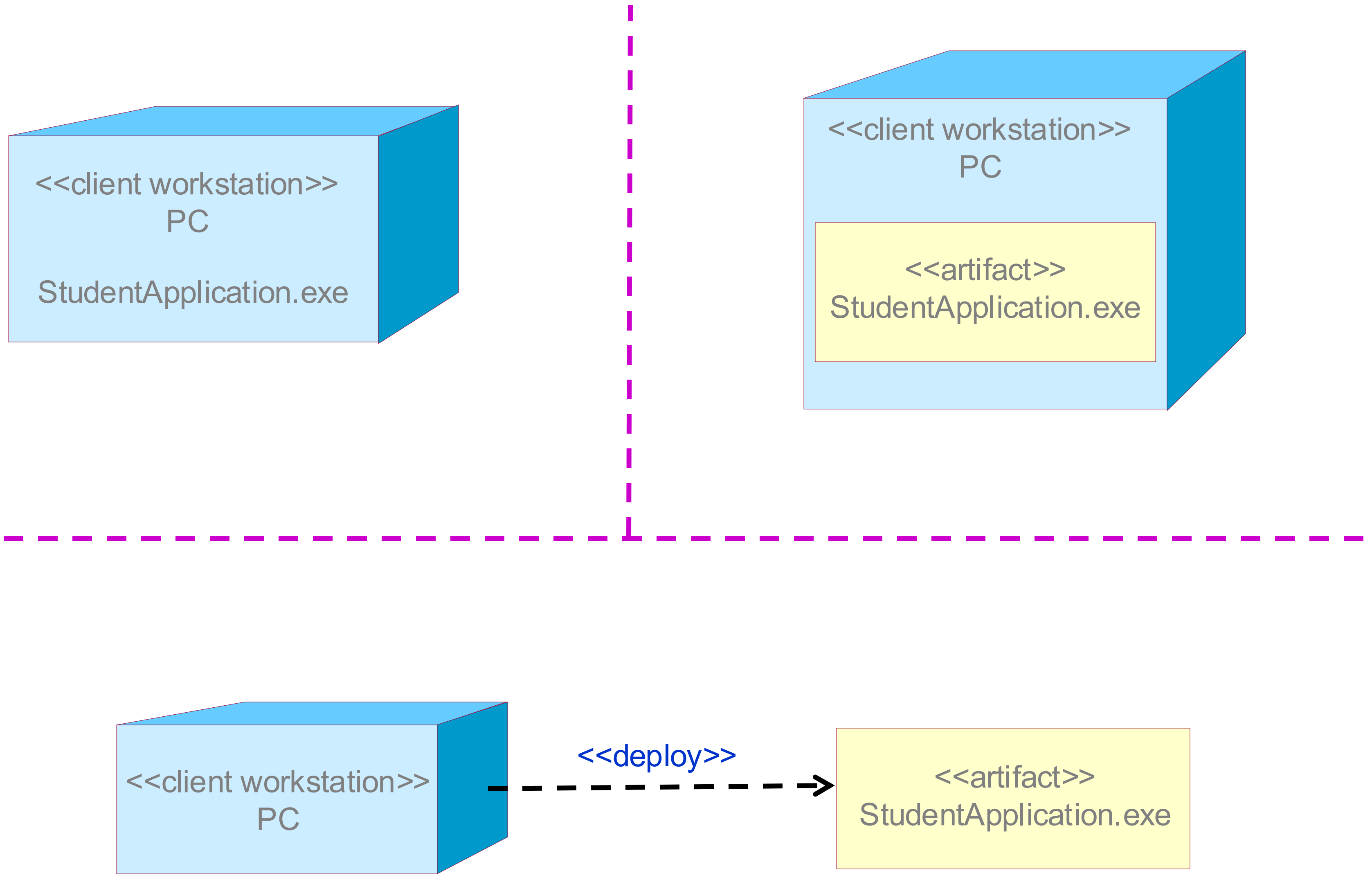
Example: Deployment Diagram



What is Deployment?

- Deployment is the assignment, or mapping, of software artifacts to physical nodes during execution
  - Artifacts are the entities that are deployed onto physical nodes
    - Processes are assigned to computers
- Artifacts model physical entities
  - Files, executables, database tables, web pages, and so on
- Nodes model computational resources
  - Computers, storage units

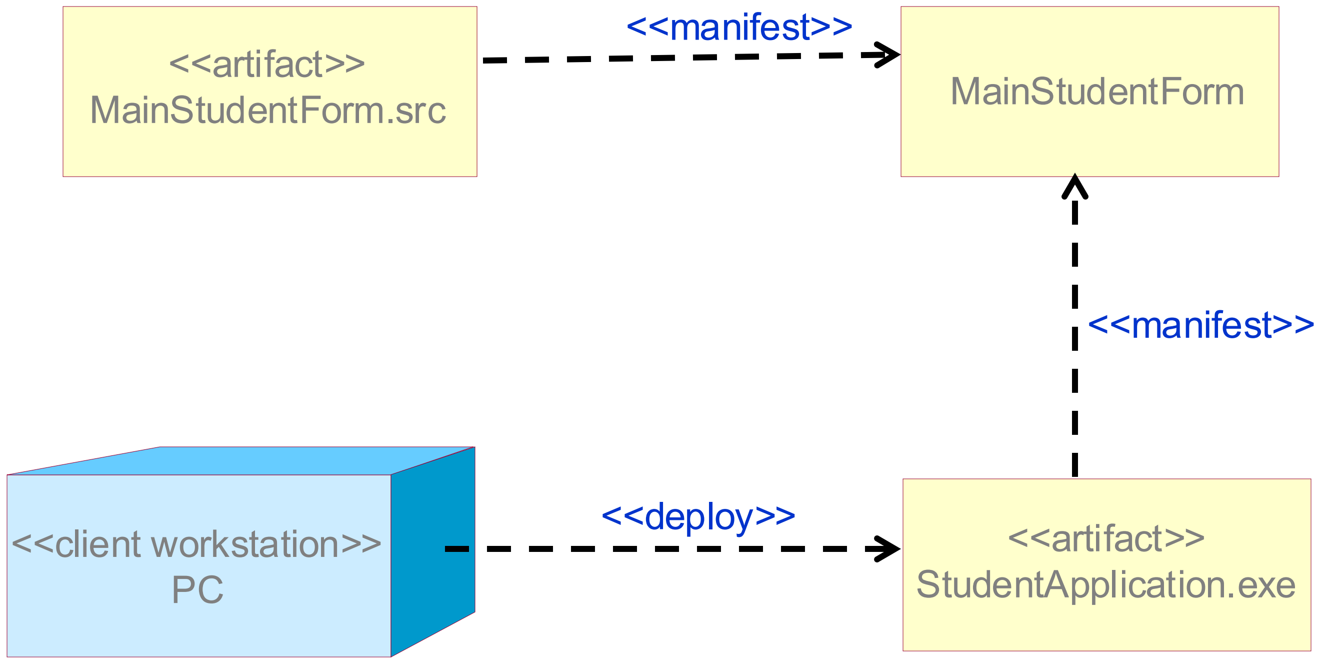
Example: Deploying Artifacts to Nodes



What is Manifestation?

- The physical implementation of a model element as an artifact
  - A relationship between the model element and the artifact that implements it
  - Model elements are typically implemented as a set of artifacts
  - Examples of model elements are *source files*, *executable files*, *documentation files*

Example: Manifestation



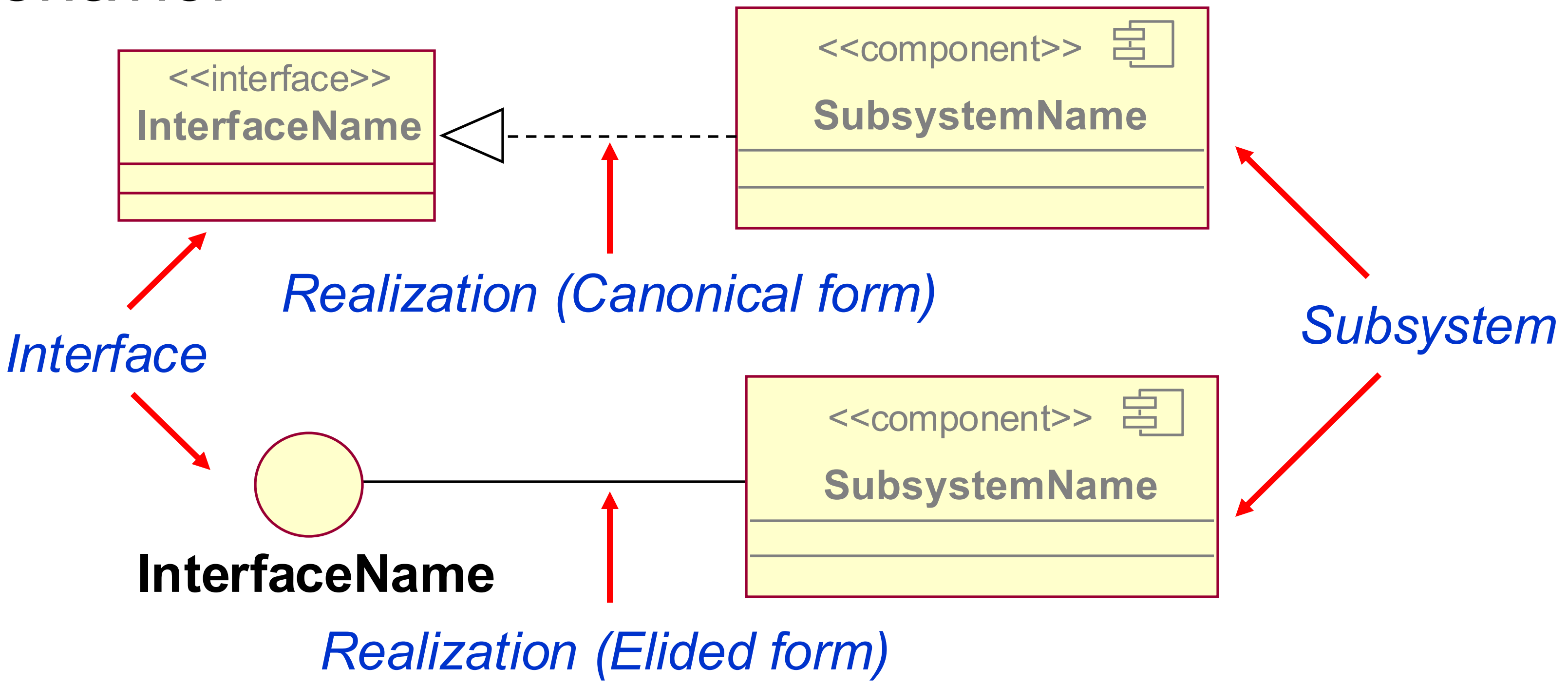
Detailed OOD: Subsystem Design

- Describe how the subsystem’s behaviors are distributed to internal elements
- Explain how to document and model the internal structure of a subsystem
- Define relationships to external elements, upon which the subsystem might be dependent

Subsystems and Interfaces

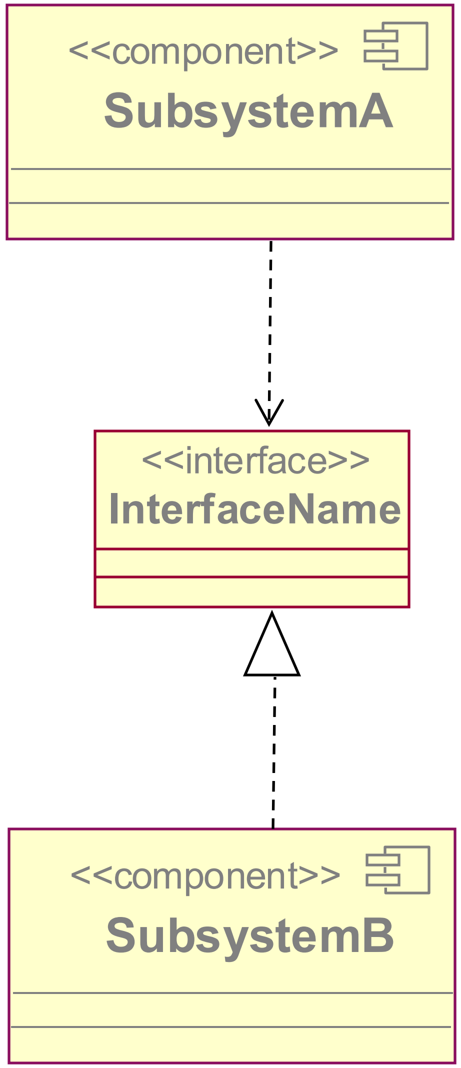
A ***subsystem***:

- Is a first class element modeled as a component
- Realizes one or more interfaces that define its behavior



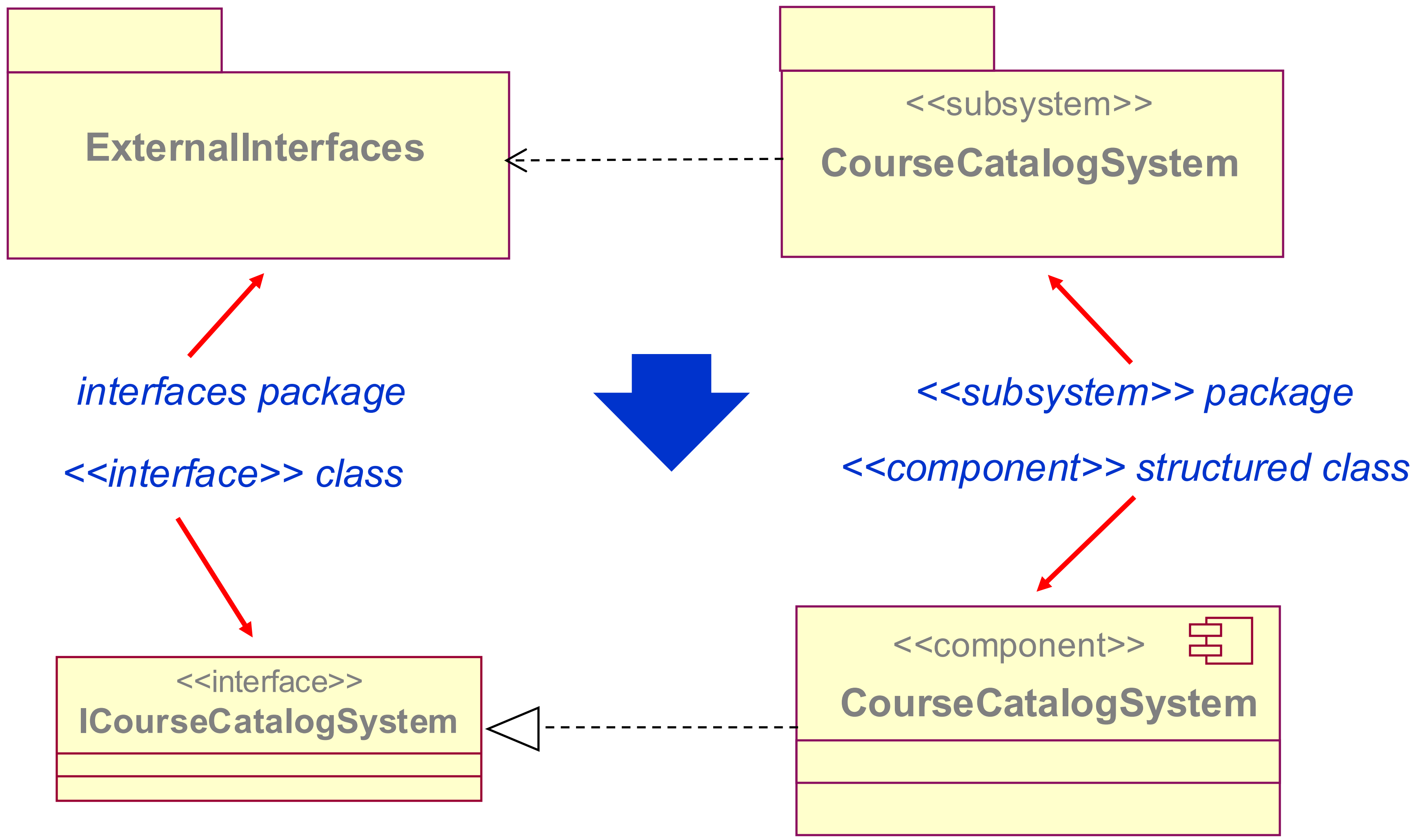
Subsystem Guidelines

- Goals
  - Loose coupling
  - Portability, plug-and-play compatibility
  - Insulation from change
  - Independent evolution
- Strong Suggestions
  - Do not expose details, only interfaces
  - Depend only on other interfaces



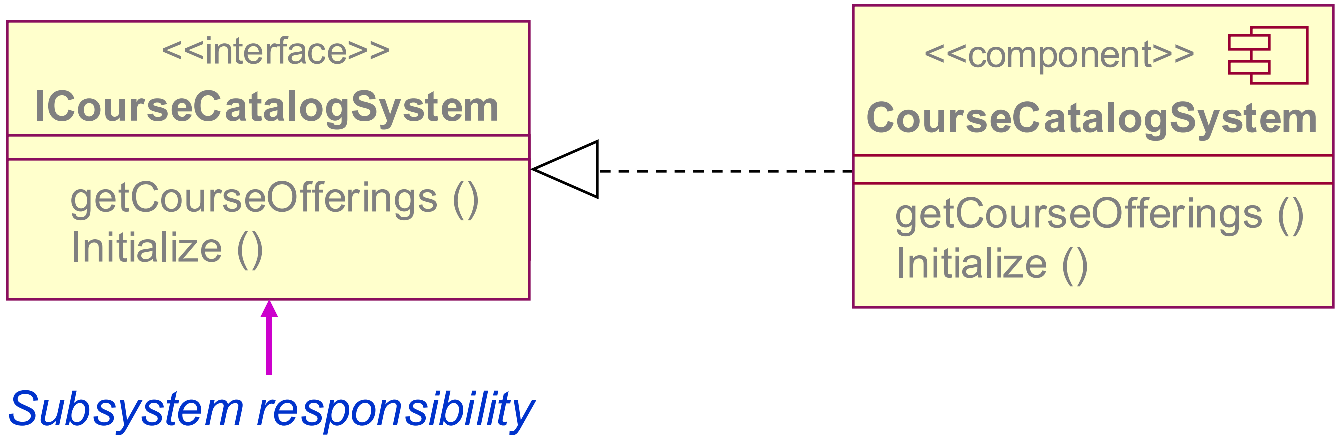
Key is abstraction and encapsulation

Modeling Convention for Subsystems and Interfaces



Subsystem Responsibilities

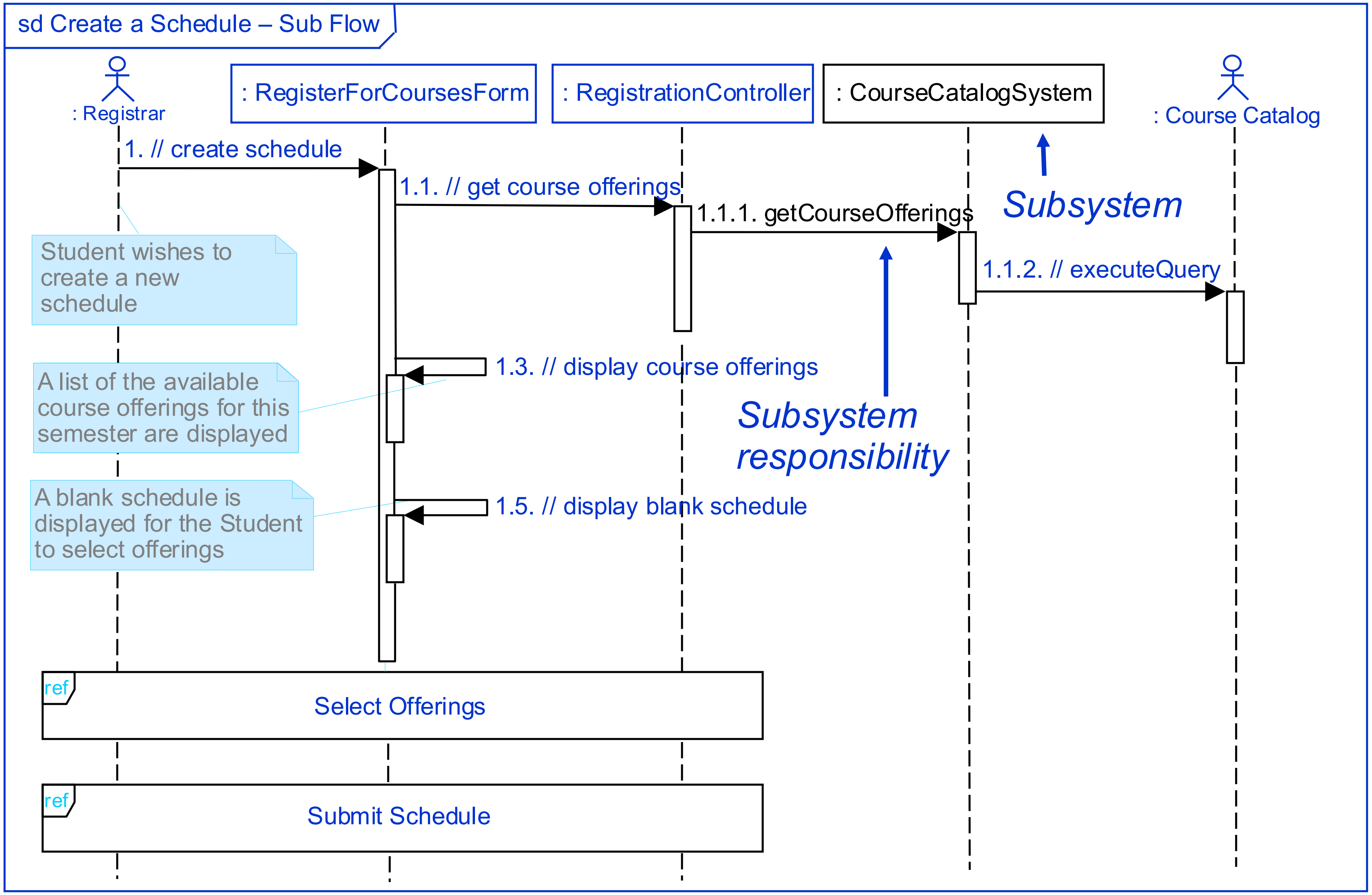
- Subsystem responsibilities defined by interface operations
  - Model interface realizations
- Interface operations may be realized by
  - Internal class behavior
  - Subsystem behavior



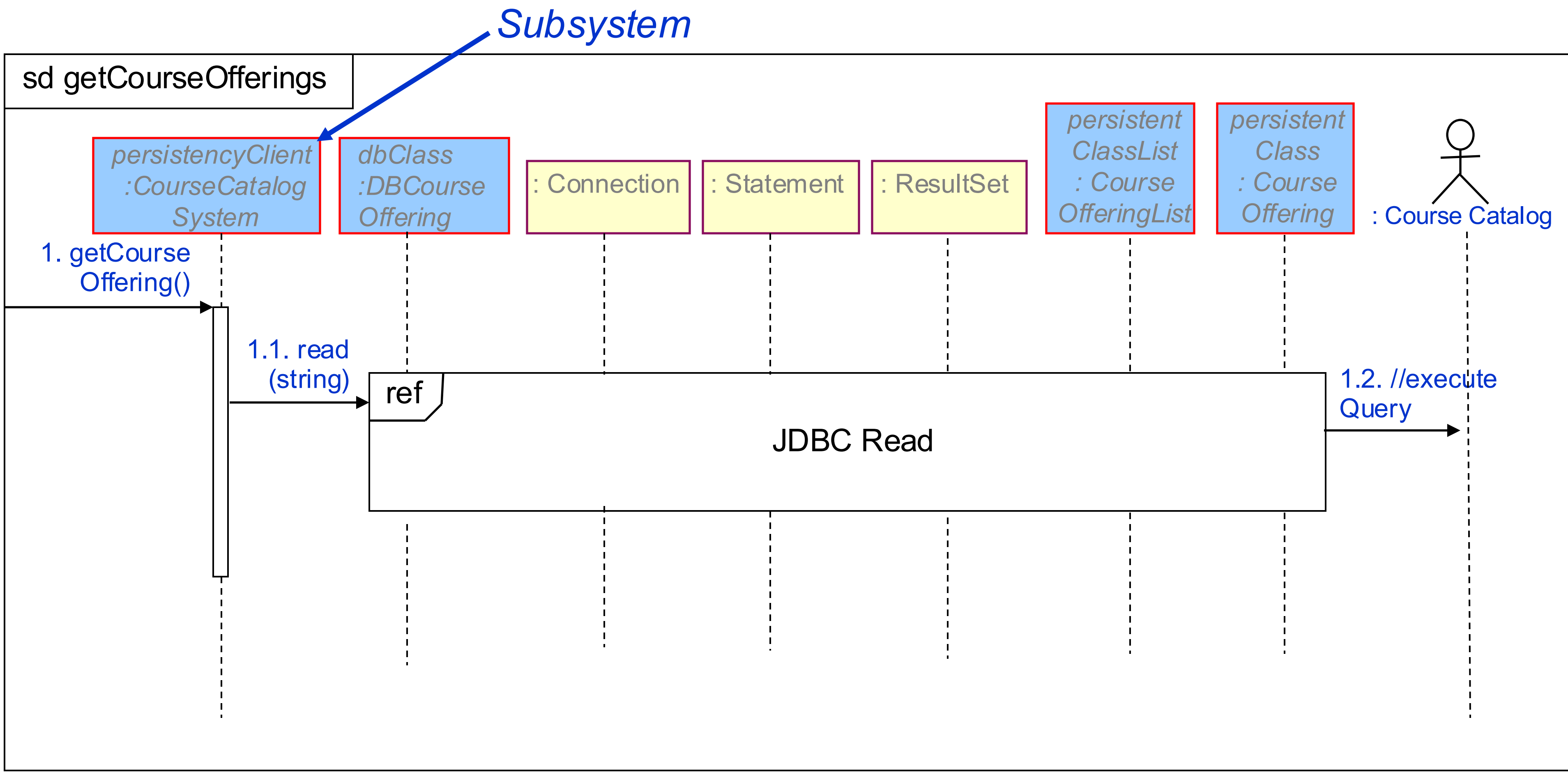
Distributing Subsystem Responsibilities

- Identify new, or reuse existing, design elements (for example, classes and subsystems)
- Allocate subsystem responsibilities to design elements
- Incorporate applicable mechanisms (for example, persistence, distribution)
- Document design element collaborations in “interface realizations”
  - One or more interaction diagrams per interface operation
  - Class diagrams containing the required design element relationships

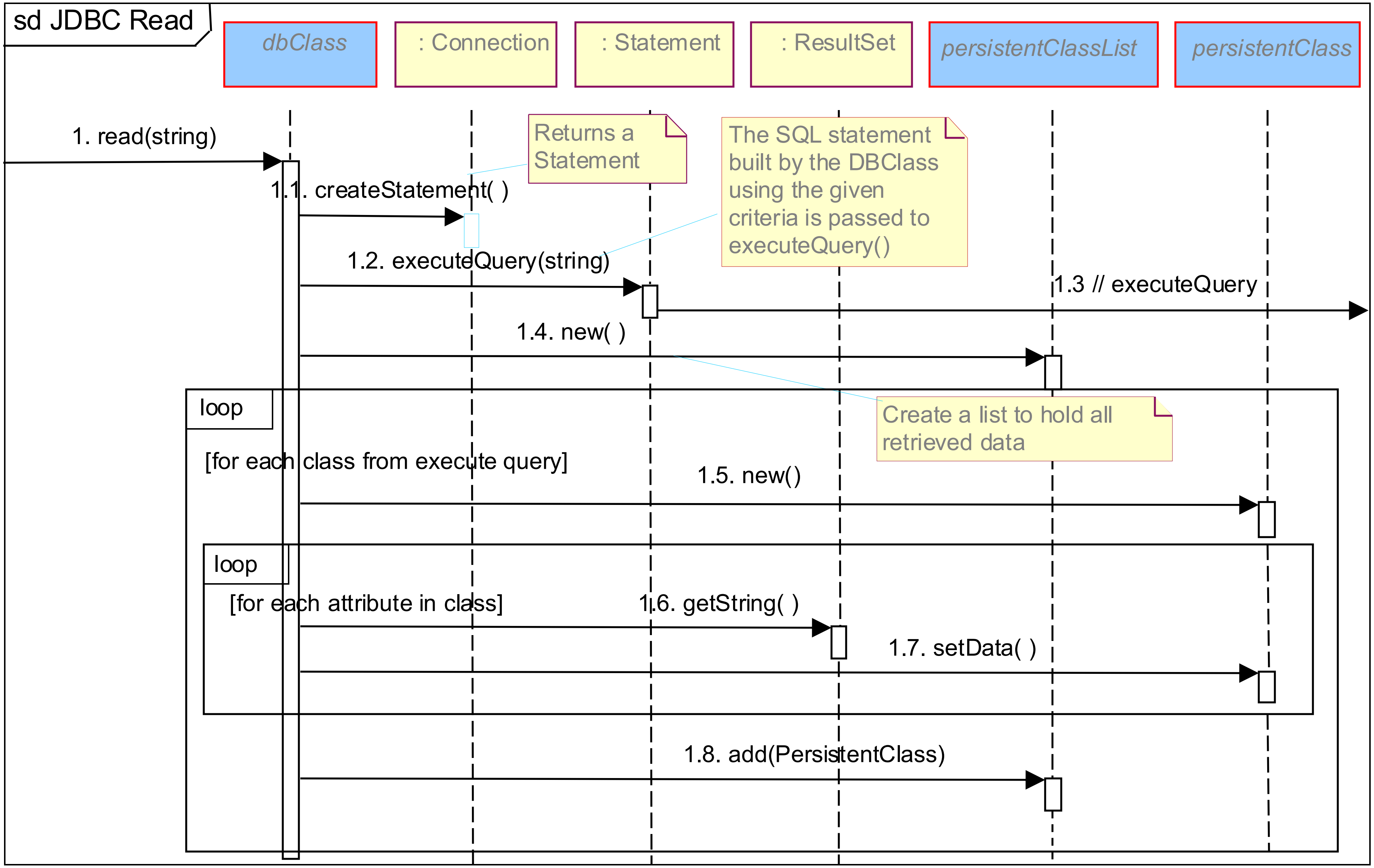
Example: CourseCatalogSystem Subsystem


sd Create a Schedule – Sub Flow
: Registrar
: RegisterForCoursesForm
: RegistrationController
: CourseCatalogSystem
: Course Catalog
1. // create schedule
1.1. // get course offerings
1.1.1. getCourseOfferings
1.1.2. // executeQuery
Subsystem
Student wishes to create a new schedule
1.3. // display course offerings
A list of the available course offerings for this semester are displayed
Subsystem responsibility
1.5. // display blank schedule
A blank schedule is displayed for the Student to select offerings
ref
Select Offerings
ref
Submit Schedule

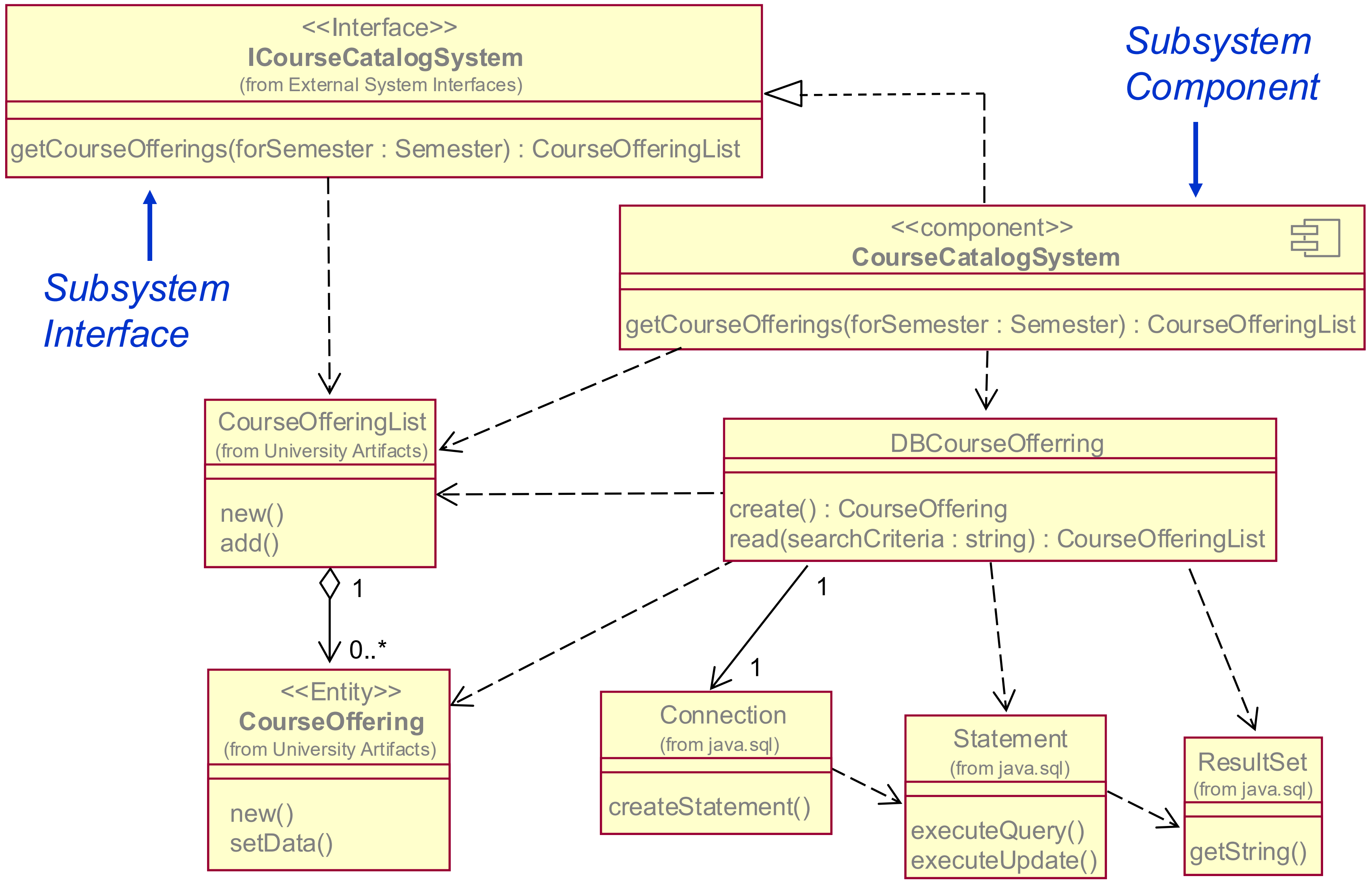
Example: Local CourseCatalogSystem Subsystem Interaction



Example: Persistency: RDBMS: JDBC: Read


sd JDBC Read
dbClass
: Connection
: Statement
: ResultSet
persistentClassList
persistentClass
1. read(string)
1.1. createStatement( )
Returns a Statement
The SQL statement built by the DBClass using the given criteria is passed to executeQuery()
1.2. executeQuery(string)
1.3 // executeQuery
1.4. new( )
Create a list to hold all retrieved data
loop
[for each class from execute query]
1.5. new()
loop
[for each attribute in class]
1.6. getString( )
1.7. setData( )
1.8. add(PersistentClass)

Example: CourseCatalogSystem Subsystem Elements



<<Interface>>
ICourseCatalogSystem
(from External System Interfaces)
getCourseOfferings(forSemester : Semester) : CourseOfferingList
Subsystem Component
Subsystem Interface
<<component>>
CourseCatalogSystem
getCourseOfferings(forSemester : Semester) : CourseOfferingList
CourseOfferingList
(from University Artifacts)
new()
add()
DBCourseOfferring
create() : CourseOffering
read(searchCriteria : string) : CourseOfferingList
1
0..*
1
1
<<Entity>>
CourseOffering
(from University Artifacts)
new()
setData()
Connection
(from java.sql)
createStatement()
Statement
(from java.sql)
executeQuery()
executeUpdate()
ResultSet
(from java.sql)
getString()