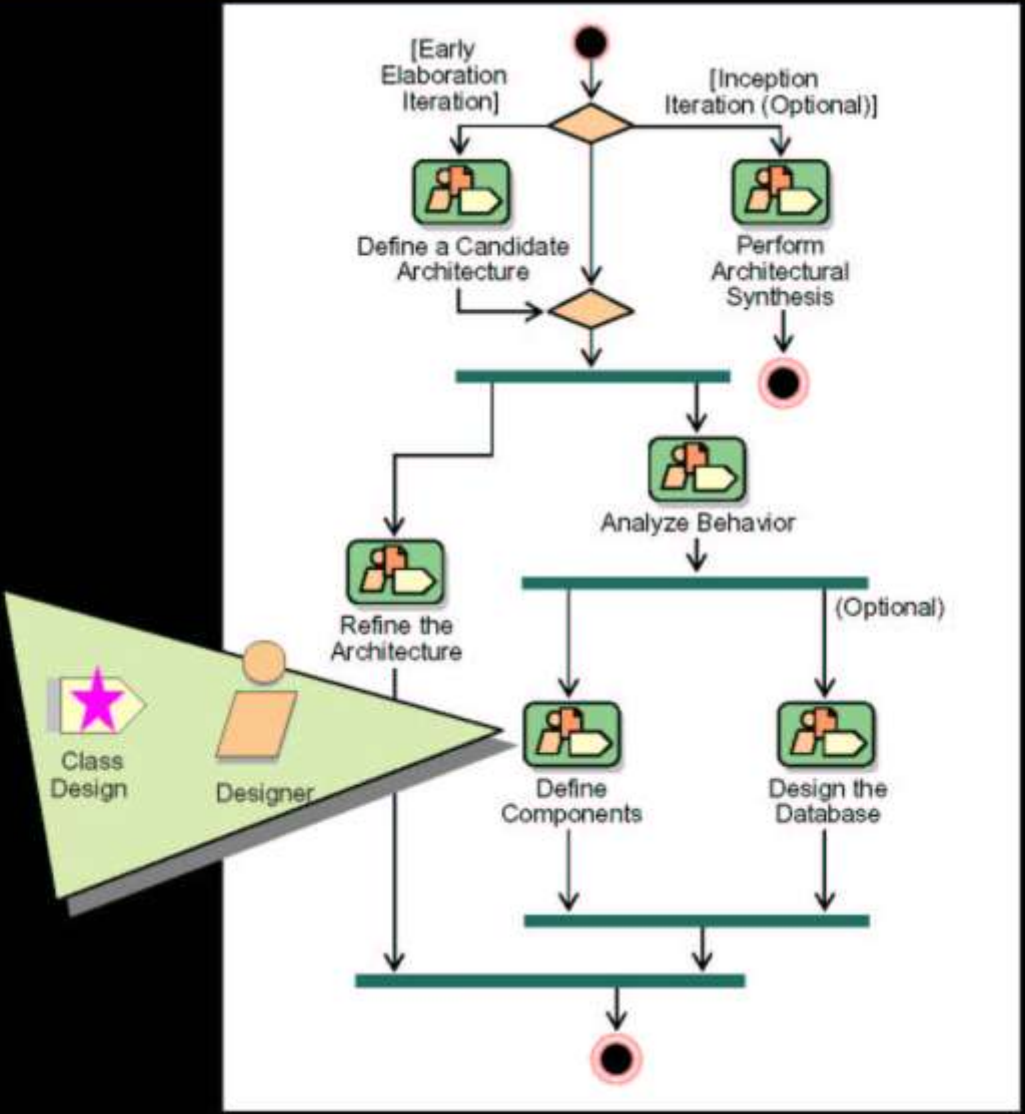
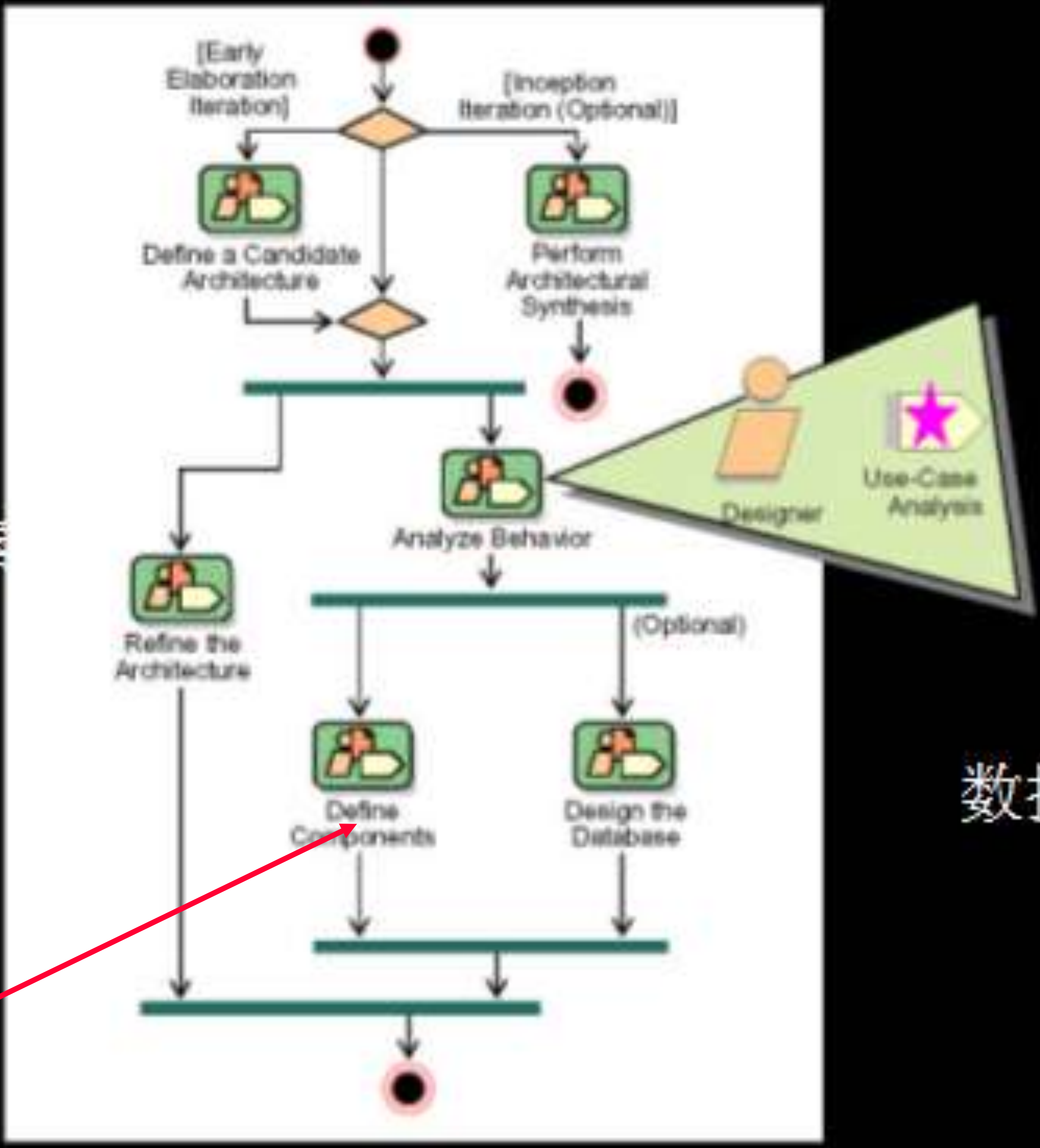
9 UC Design

Mastering Object-Oriented Analysis and Design with UML

Module 9: Use-Case Design

Class Design in Context





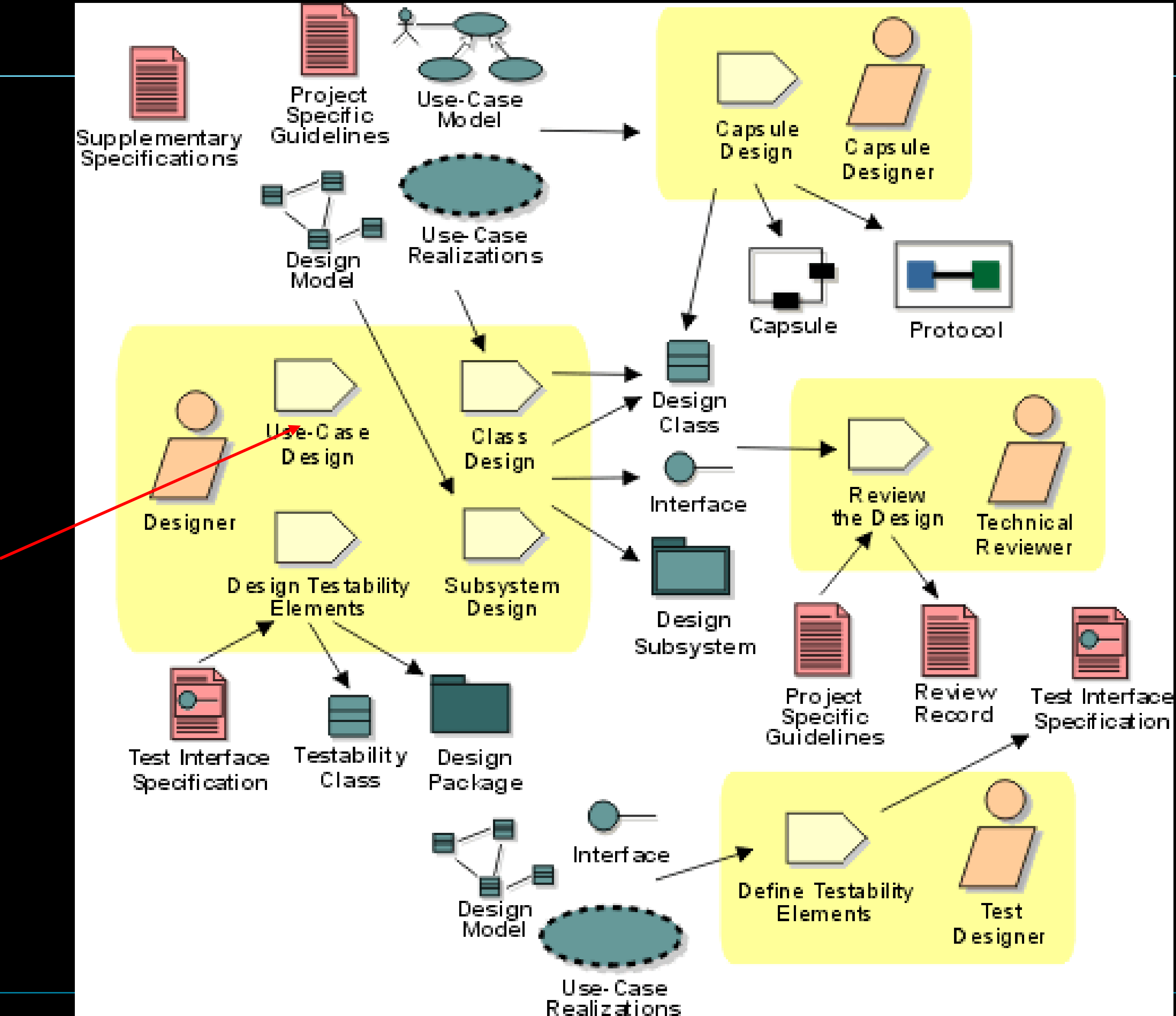
架构分析

用例分析

识别设计元素
运行时架构
描述分布

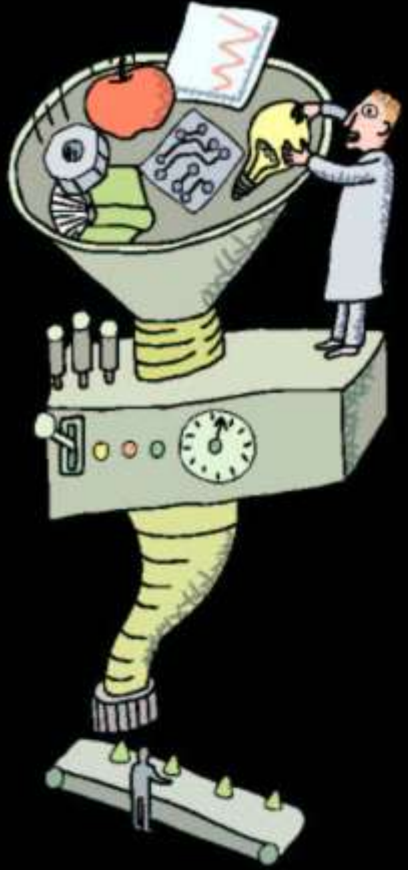
数据库设计

用例设计
子系统设计
类设计


Supplementary Specifications
Project Specific Guidelines
Use-Case Model
Capsule Design
Capsule Designer
Design Model
Use-Case Realizations
Capsule
Protocol
Use-Case Design
Class Design
Designer
Design Testability Elements
Subsystem Design
Design Class
Interface
Design Subsystem
Review the Design
Technical Reviewer
Project Specific Guidelines
Review Record
Test Interface Specification
Test Interface Specification
Testability Class
Design Package
Design Model
Interface
Use-Case Realizations
Define Testability Elements
Test Designer

Use-Case Design Steps

- Describe interaction among design objects
- Simplify sequence diagrams using subsystems
- Describe persistence-related behavior
- Refine the flow of events description
- Unify classes and subsystems

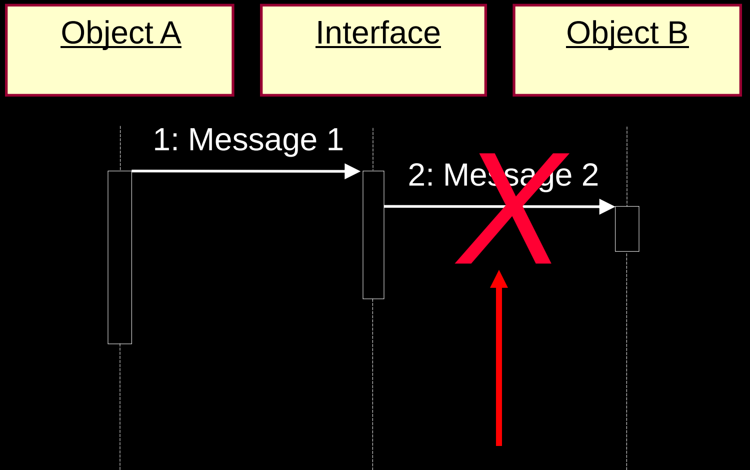


Use-Case Design Steps

- ★ Describe interaction among design objects
- Simplify sequence diagrams using subsystems
- Describe persistence-related behavior
- Refine the flow of events description
- Unify classes and subsystems

Representing Subsystems on a Sequence Diagram

- Interfaces
  - Represent any model element that realizes the interface
  - No message should be drawn from the interface
- Subsystem Component
  - Represents a specific subsystem
  - Messages can be drawn from the subsystem



Object A

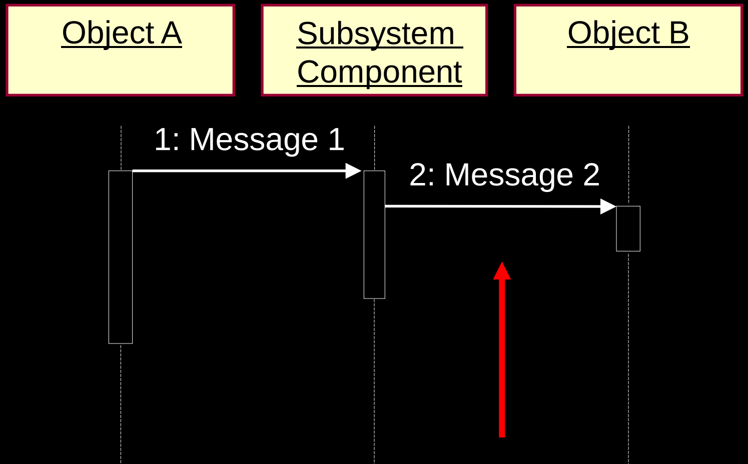
Interface

Object B

1: Message 1

2: Message 2

Invalid message



Object A

Subsystem Component

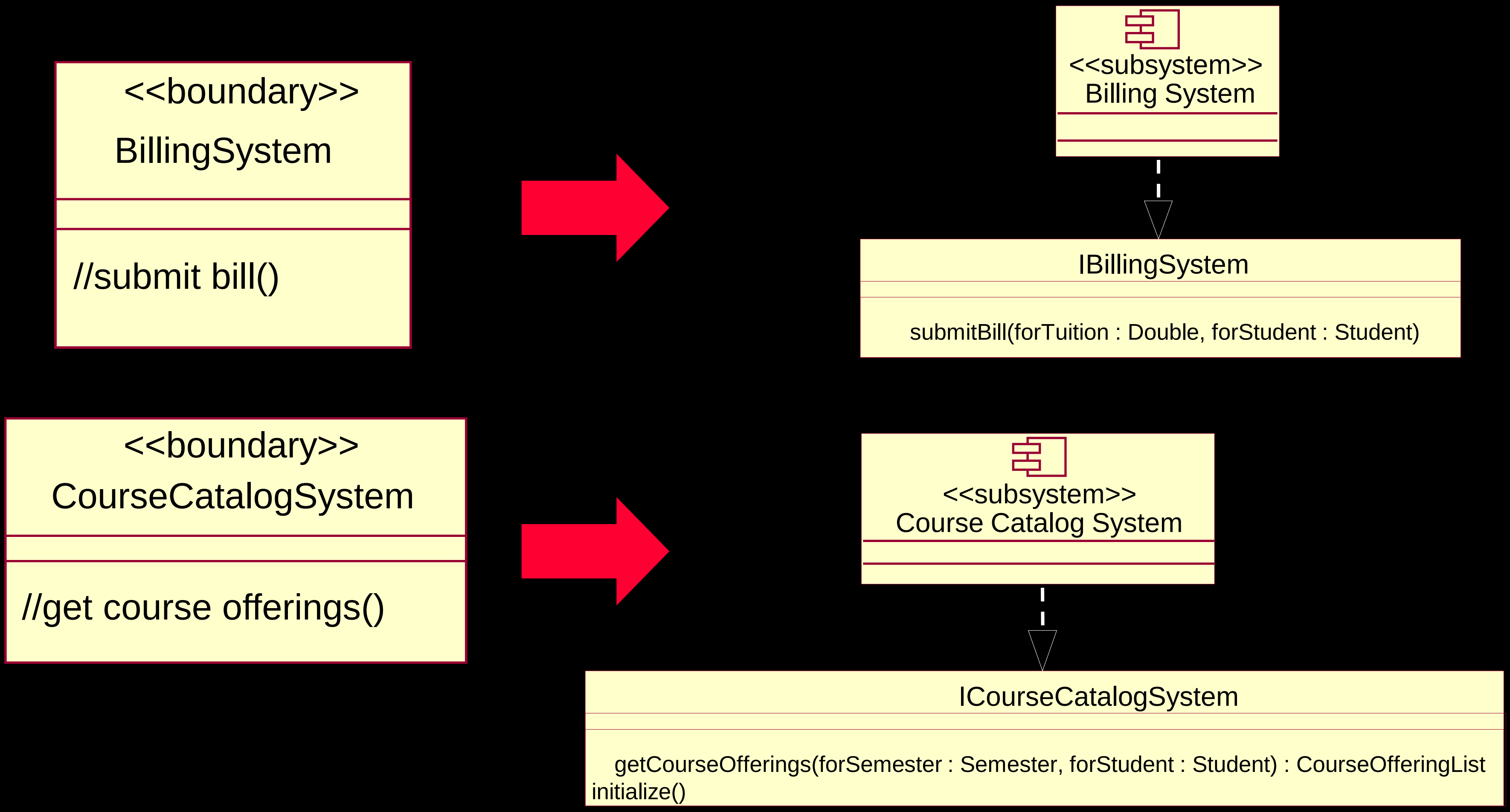
Object B

1: Message 1

2: Message 2

Valid message

Example: Incorporating Subsystem Interfaces



Analysis Classes

<<boundary>>
BillingSystem

//submit bill()

<<boundary>>
CourseCatalogSystem

//get course offerings()

Design Elements

<<subsystem>>
Billing System

IBillingSystem

submitBill(forTuition : Double, forStudent : Student)

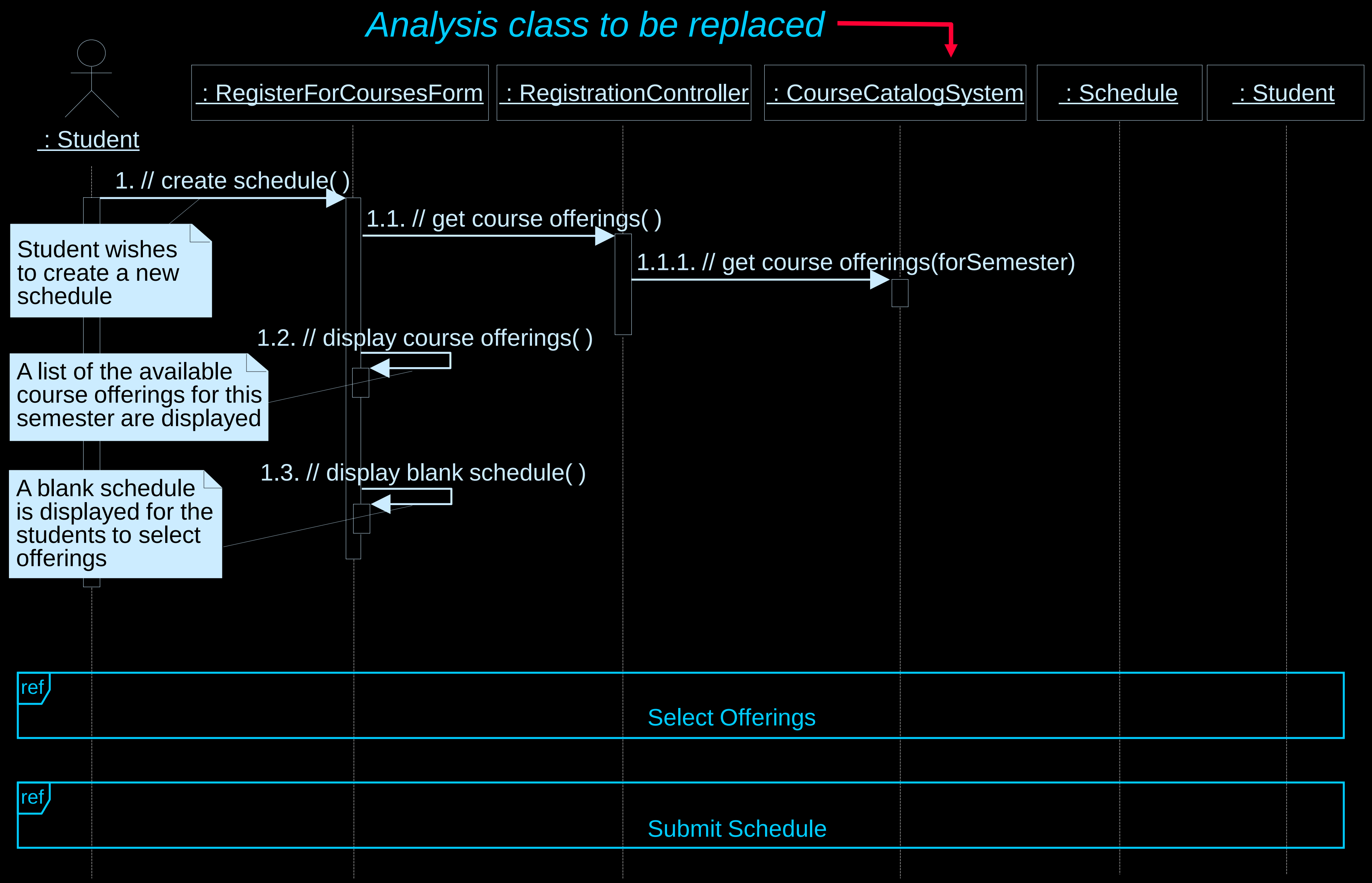
<<subsystem>>
Course Catalog System

ICourseCatalogSystem

getCourseOfferings(forSemester : Semester, forStudent : Student) : CourseOfferingList
initialize()

All other analysis classes are mapped directly to design classes.

Example: Incorporating Subsystems (Before)



Analysis class to be replaced

: Student

: RegisterForCoursesForm

: RegistrationController

: CourseCatalogSystem

: Schedule

: Student

1. // create schedule( )

Student wishes to create a new schedule

1.1. // get course offerings( )

1.1.1. // get course offerings(forSemester)

1.2. // display course offerings( )

A list of the available course offerings for this semester are displayed

1.3. // display blank schedule( )

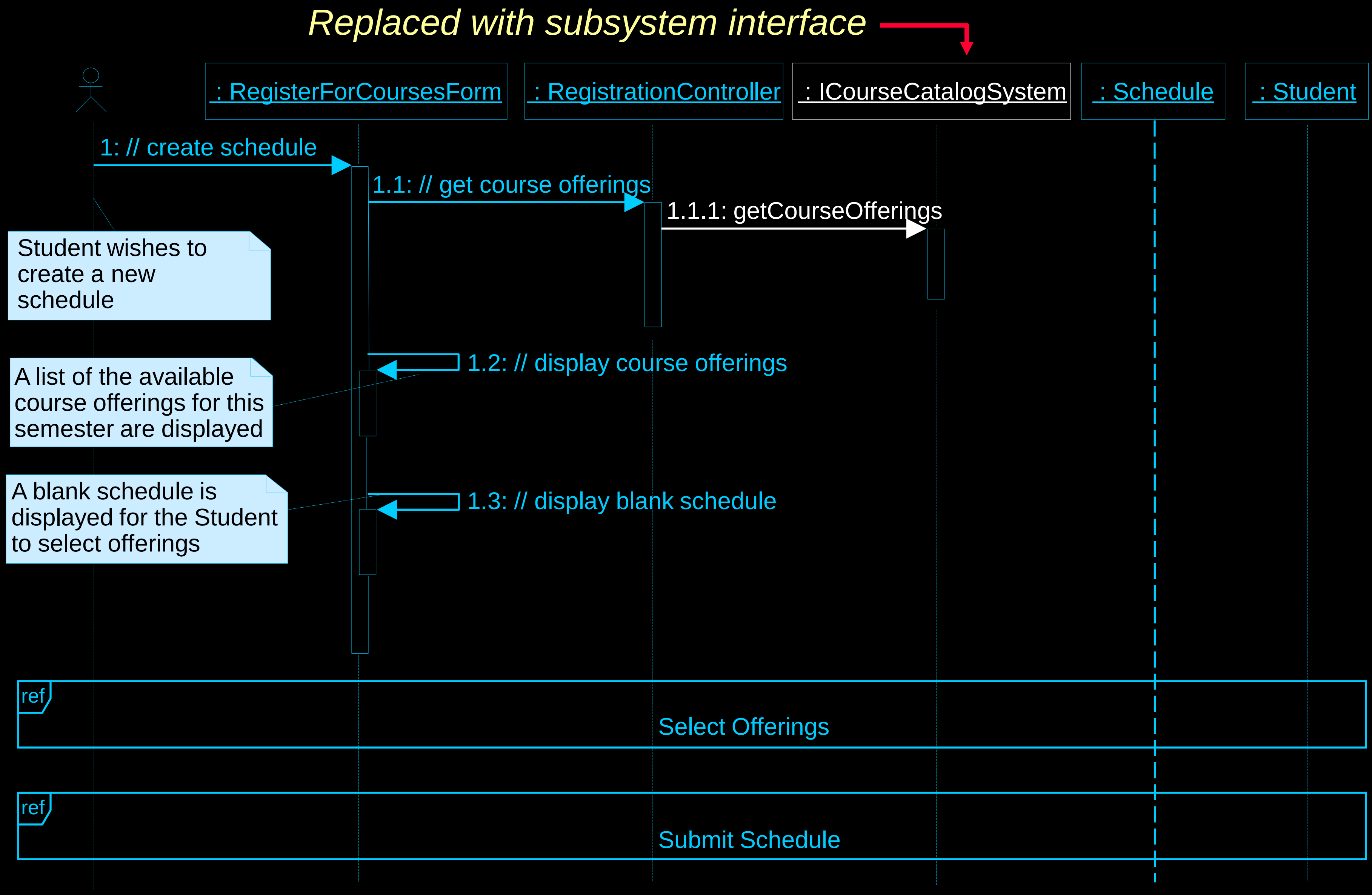
A blank schedule is displayed for the students to select offerings

ref Select Offerings

ref Submit Schedule

Example: Incorporating Subsystems (After)

Replaced with subsystem interface



: RegisterForCoursesForm

: RegistrationController

: ICourseCatalogSystem

: Schedule

: Student

1: // create schedule

1.1: // get course offerings

1.1.1: getCourseOfferings

1.2: // display course offerings

1.3: // display blank schedule

Student wishes to create a new schedule

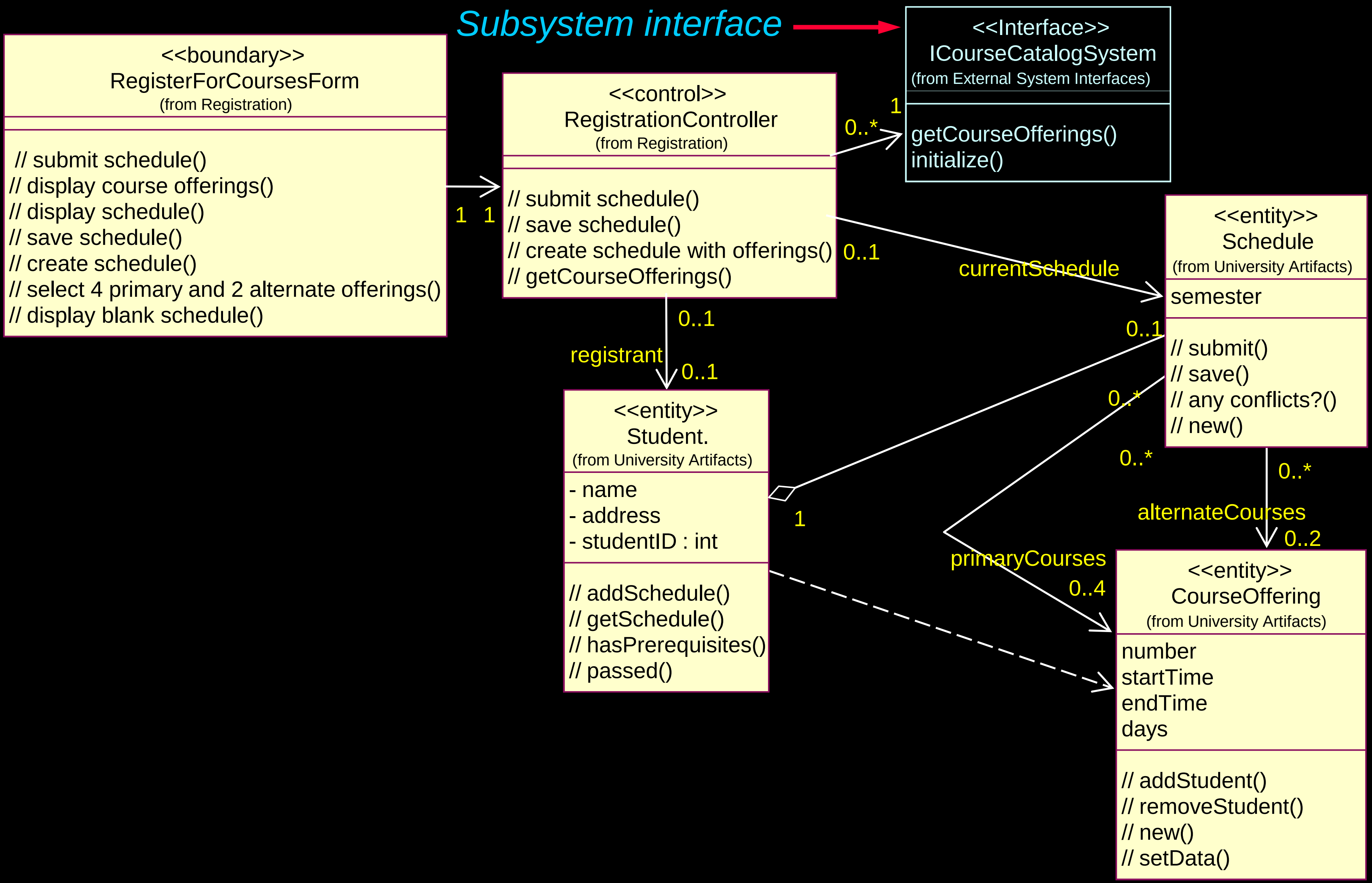
A list of the available course offerings for this semester are displayed

A blank schedule is displayed for the Student to select offerings

ref Select Offerings

ref Submit Schedule

Example: Incorporating Subsystem Interfaces (VOPC)



Incorporating Architectural Mechanisms: Security

- Analysis Class to Architectural-Mechanism Map from Use-Case Analysis

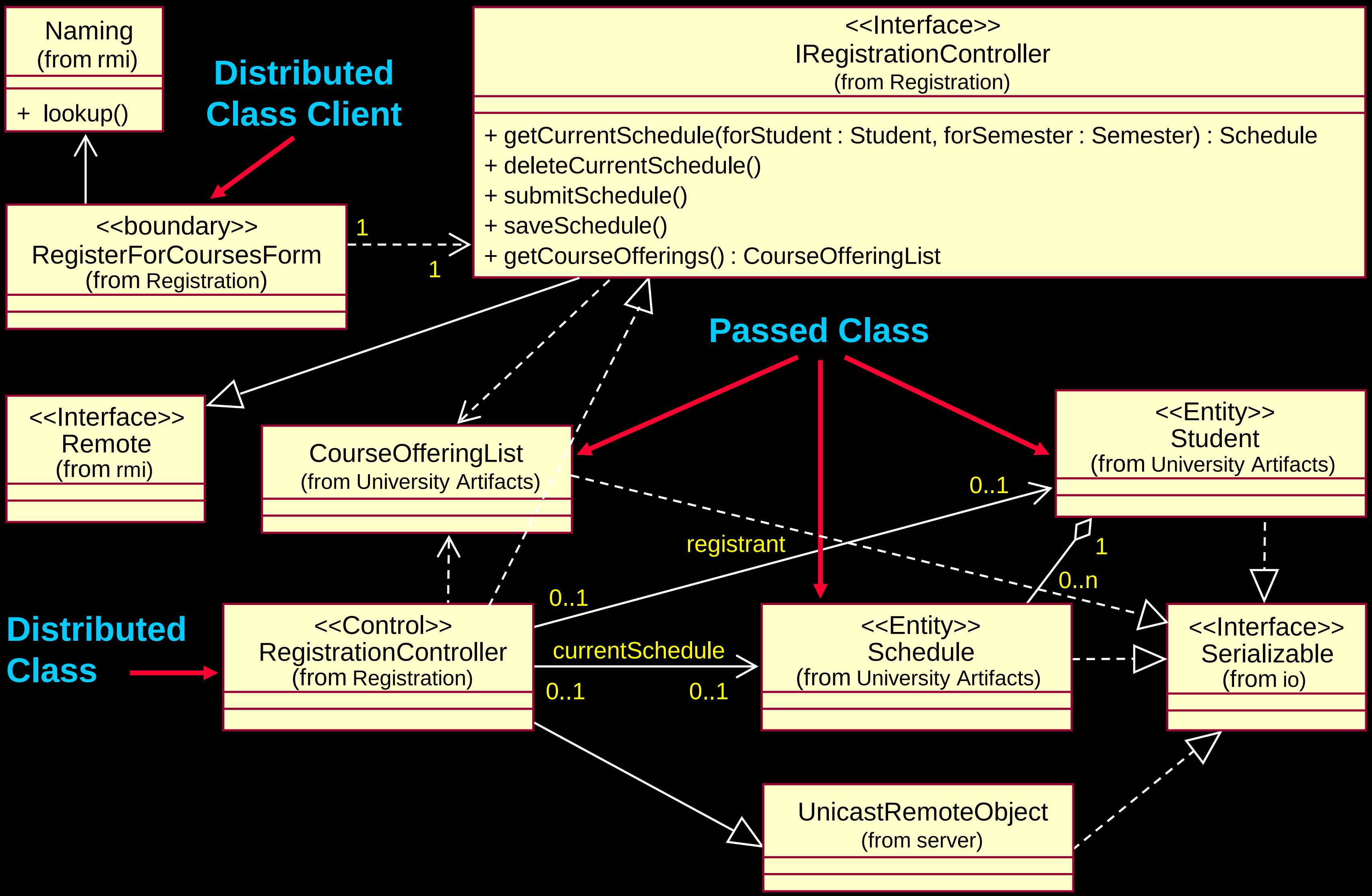
Analysis Class	**Analysis Mechanism(s)**
Student	Persistency, *Security*
Schedule	Persistency, *Security*
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution

Incorporating Architectural Mechanisms: Distribution

- Analysis Class to Architectural-Mechanism Map from Use-Case Analysis

Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	*Distribution*

Example: Incorporating RMI



Naming
(from rmi)
\+ lookup()

Distributed Class Client

<<boundary>>
RegisterForCoursesForm
(from Registration)

<<Interface>>
IRegistrationController
(from Registration)
\+ getCurrentSchedule(forStudent : Student, forSemester : Semester) : Schedule
\+ deleteCurrentSchedule()
\+ submitSchedule()
\+ saveSchedule()
\+ getCourseOfferings() : CourseOfferingList

Passed Class

<<Interface>>
Remote
(from rmi)

CourseOfferingList
(from University Artifacts)

<<Entity>>
Student
(from University Artifacts)

Distributed Class

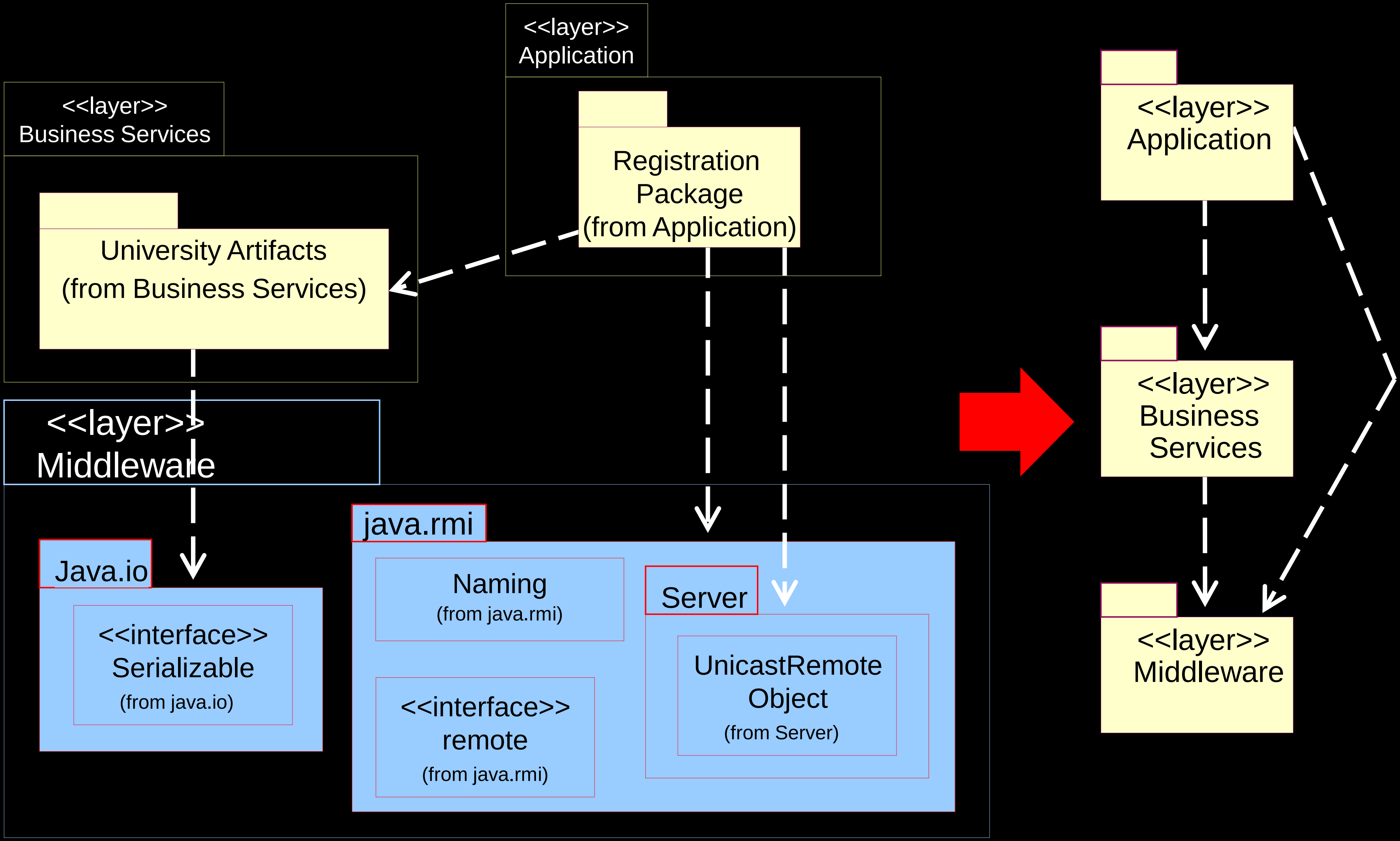
<<Control>>
RegistrationController
(from Registration)

<<Entity>>
Schedule
(from University Artifacts)

<<Interface>>
Serializable
(from io)

UnicastRemoteObject
(from server)

Example: Incorporating RMI (continued)

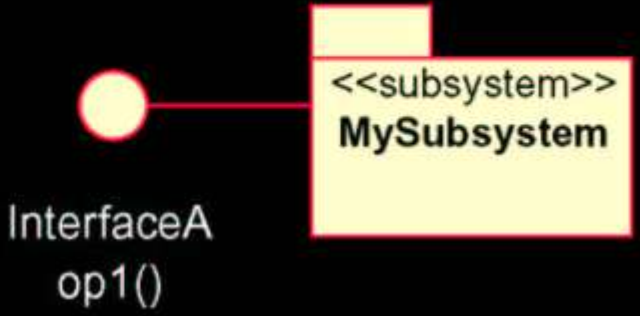


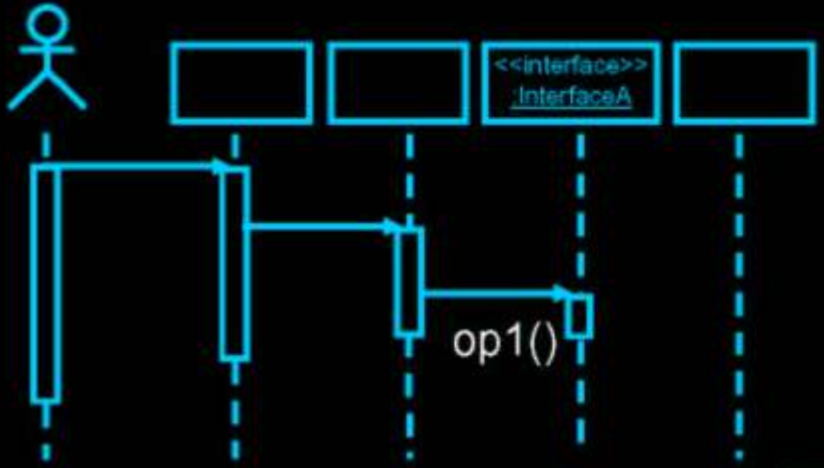
Use-Case Design Steps

- Describe interaction among design objects
- ★ Simplify sequence diagrams using subsystems
- Describe persistence-related behavior
- Refine the flow of events description
- Unify classes and subsystems

Guidelines: Encapsulating Subsystem Interactions

- Subsystems should be represented by their interfaces on interaction diagrams
- Messages to subsystems are modeled as messages to the subsystem interface
- Messages to subsystems correspond to operations of the subsystem interface
- Interactions within subsystems are modeled in Subsystem Design





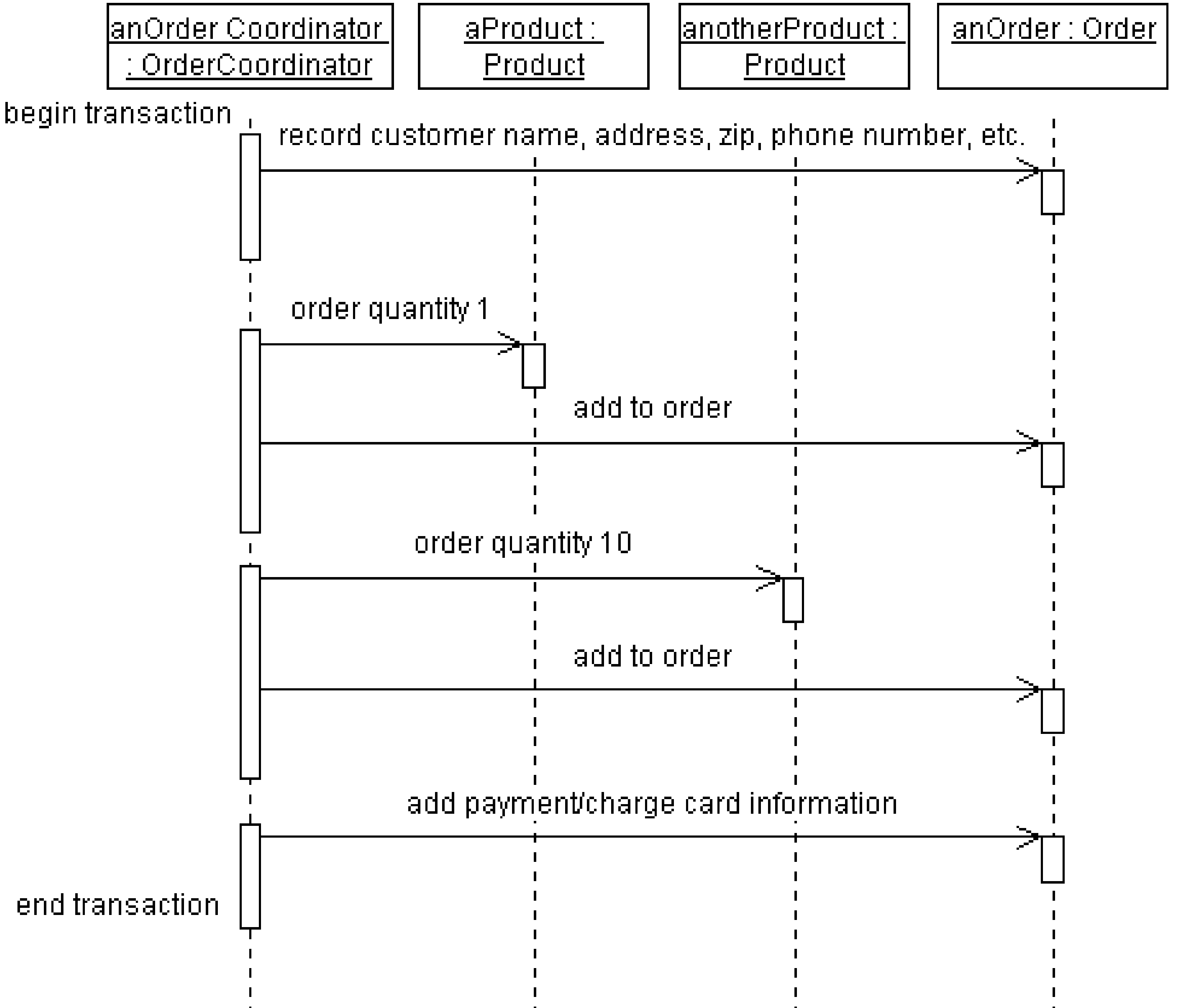
Advantages of Encapsulating Subsystem Interactions

Use-case realizations:

- Are less cluttered
- Can be created before the internal designs of subsystems are created (parallel development)
- Are more generic and easier to change (Subsystems can be substituted.)

Use-Case Design Steps

- Describe interaction among design objects
- Simplify sequence diagrams using subsystems
- ★ Describe persistence-related behavior
- Refine the flow of events description
- Unify classes and subsystems



anOrder Coordinator : OrderCoordinator

aProduct : Product

anotherProduct : Product

anOrder : Order

begin transaction

record customer name, address, zip, phone number, etc.

order quantity 1

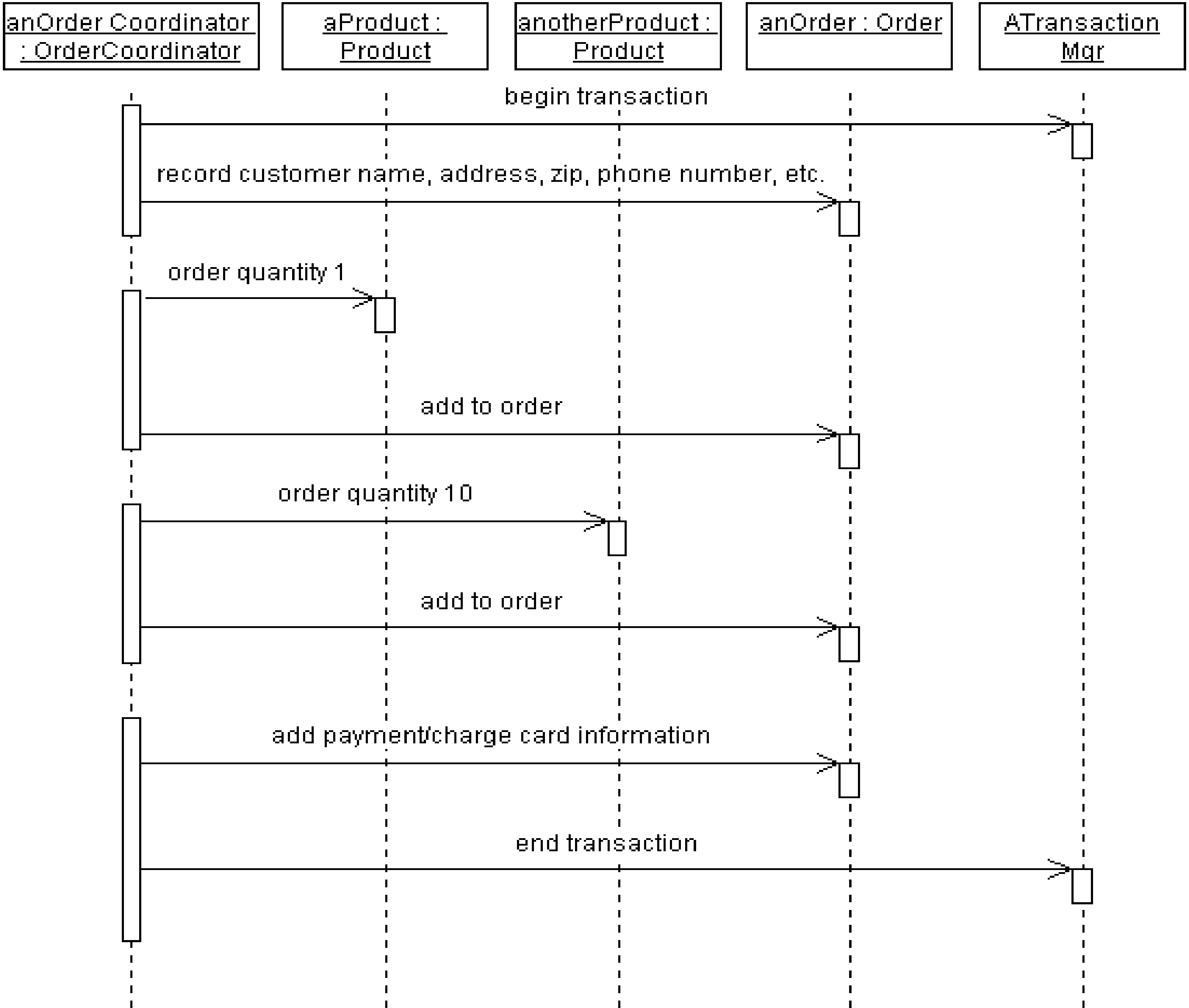
add to order

order quantity 10

add to order

add payment/charge card information

end transaction



anOrder Coordinator : OrderCoordinator

aProduct : Product

anotherProduct : Product

anOrder : Order

ATransaction Mgr

begin transaction

record customer name, address, zip, phone number, etc.

order quantity 1

add to order

order quantity 10

add to order

add payment/charge card information

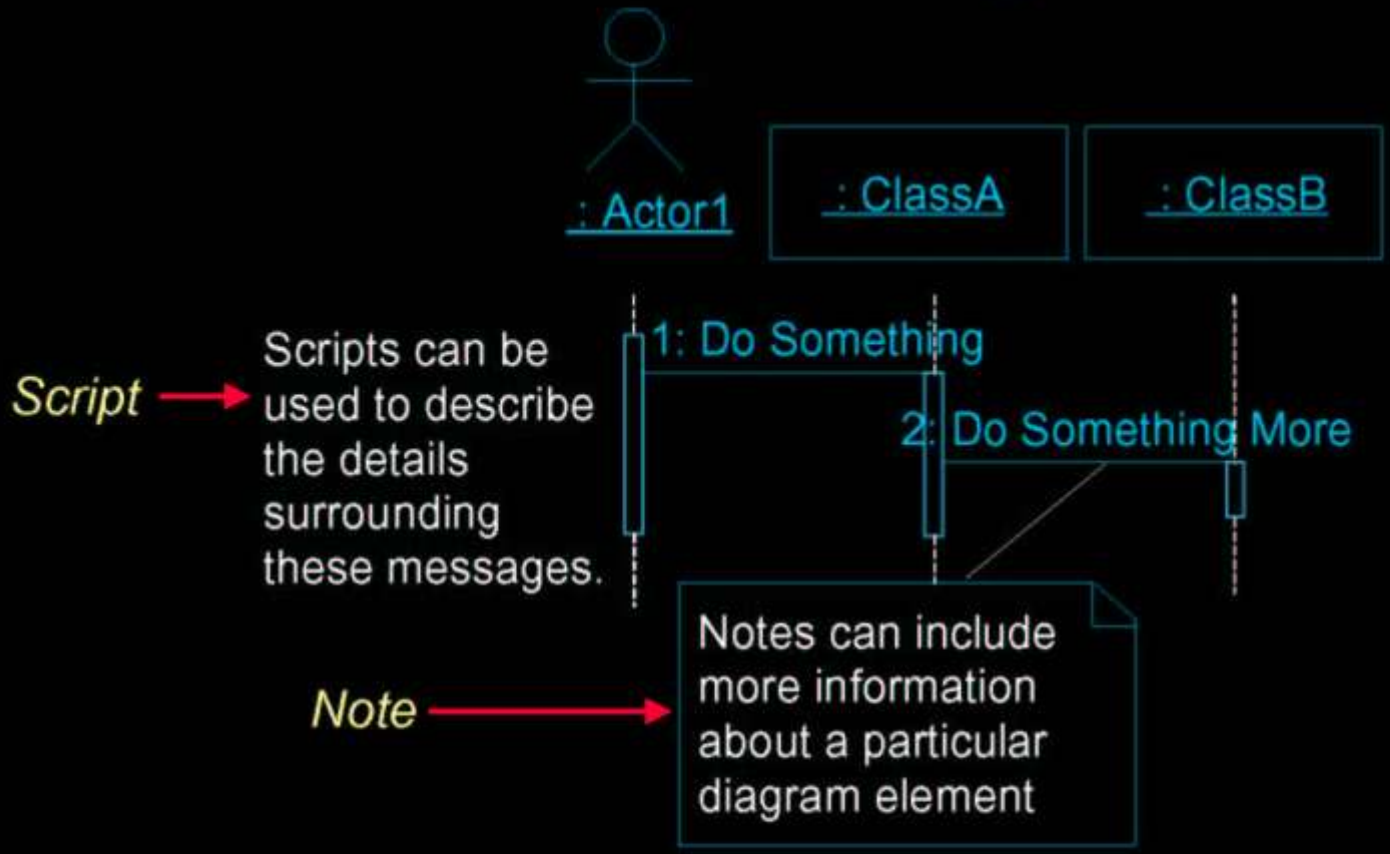
end transaction

Use-Case Design Steps

- Describe interaction among design objects
- Simplify sequence diagrams using subsystems
- Describe persistence-related behavior
- ★ Refine the flow of events description
- Unify classes and subsystems

Detailed Flow of Events Description Options

- Annotate the interaction diagrams



Script → Scripts can be used to describe the details surrounding these messages.

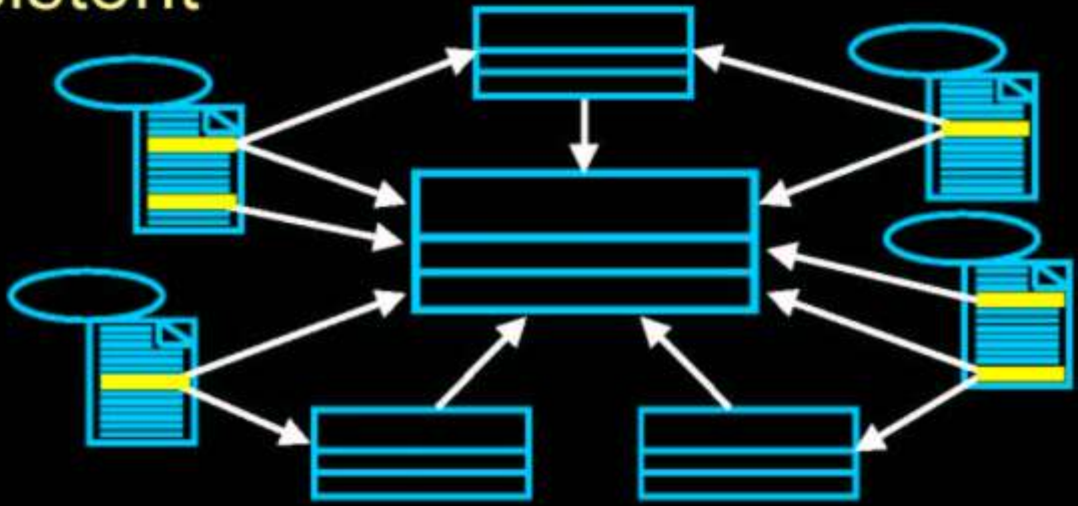
Note → Notes can include more information about a particular diagram element

Use-Case Design Steps

- Describe interaction among design objects
- Simplify sequence diagrams using subsystems
- Describe persistence-related behavior
- Refine the flow of events description
- ★ Unify classes and subsystems

Design Model Unification Considerations

- Model element names should describe their function
- Merge similar model elements
- Use inheritance to abstract model elements
- Keep model elements and flows of events consistent



10 Subsystem Design

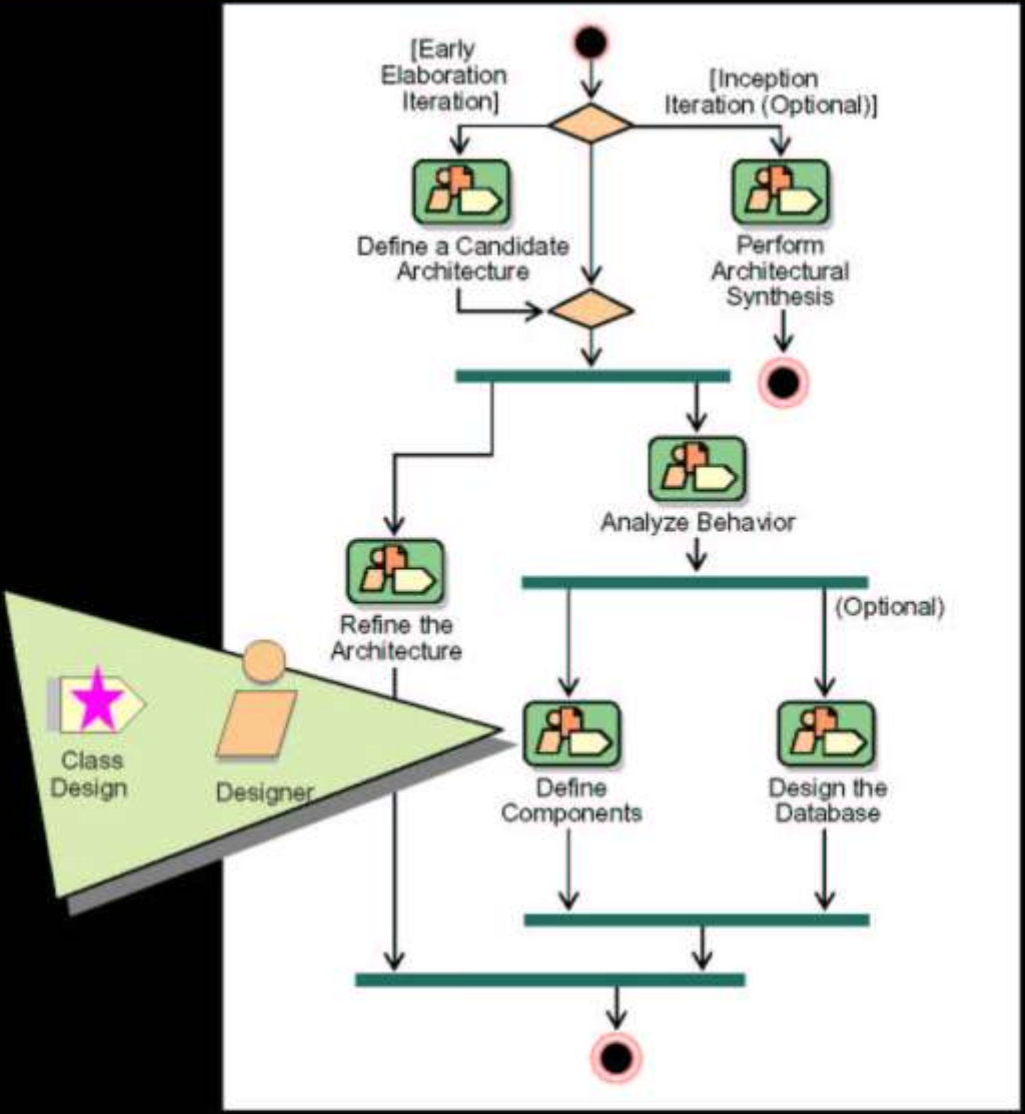
IBM Software Group

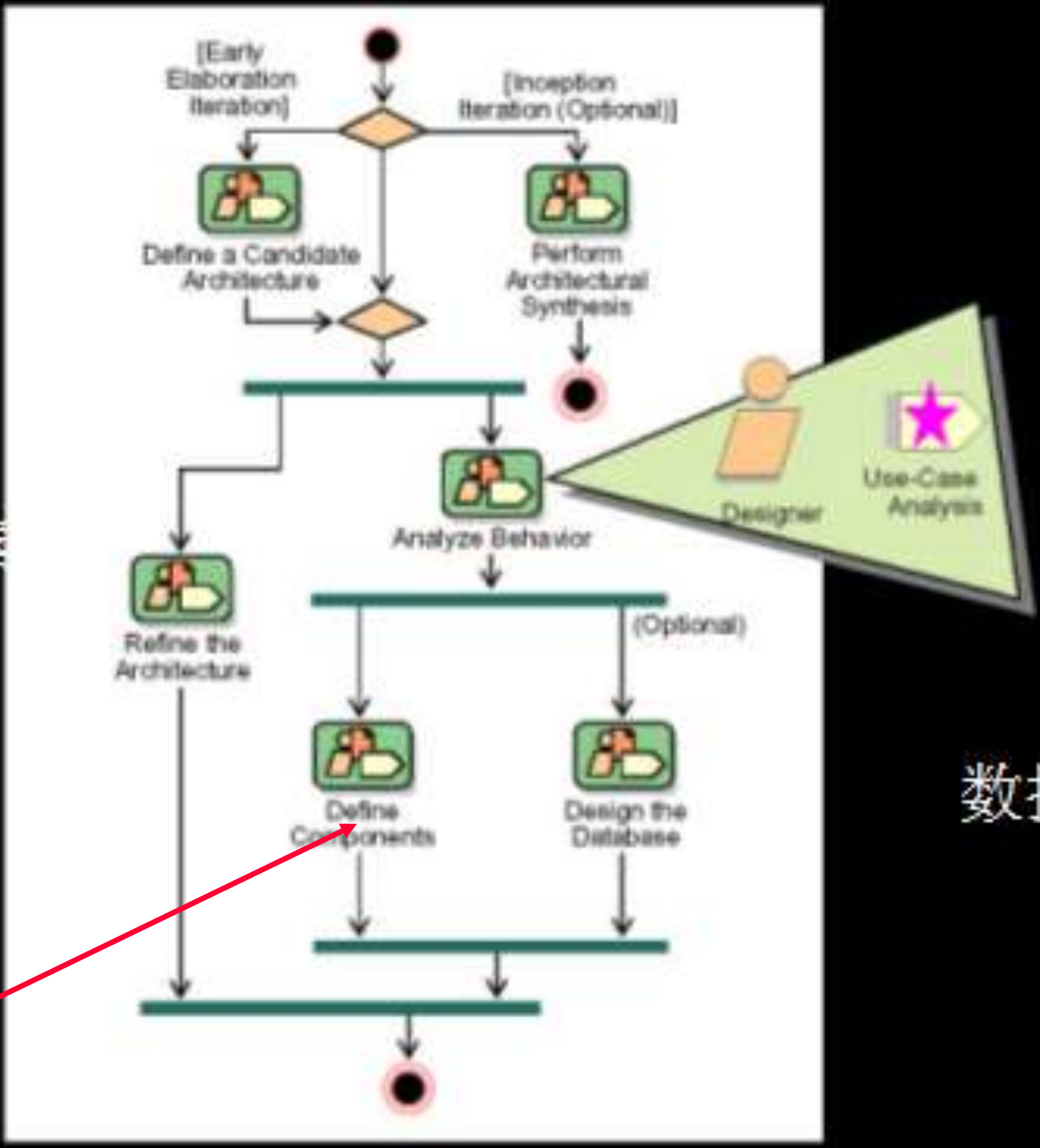
Mastering Object-Oriented Analysis and Design with UML

Module 10: Subsystem Design

Rational. software

Class Design in Context





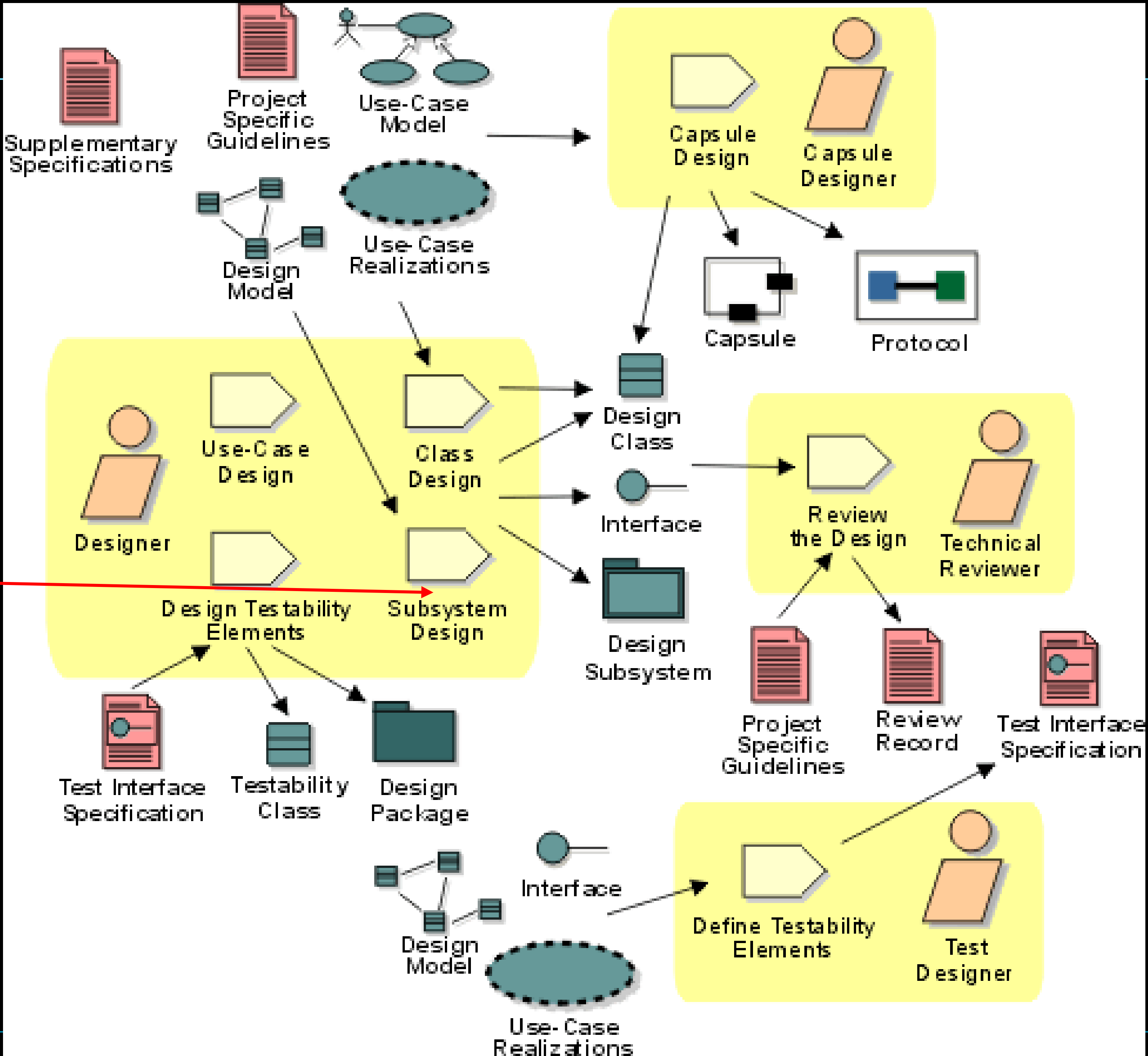
架构分析

用例分析

识别设计元素
运行时架构
描述分布

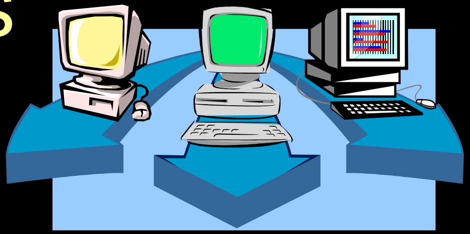
数据库设计

用例设计
子系统设计
类设计


Supplementary Specifications
Project Specific Guidelines
Use-Case Model
Capsule Design
Capsule Designer
Design Model
Use-Case Realizations
Capsule
Protocol
Designer
Use-Case Design
Class Design
Design Class
Interface
Design Testability Elements
Subsystem Design
Design Subsystem
Review the Design
Technical Reviewer
Project Specific Guidelines
Review Record
Test Interface Specification
Test Interface Specification
Testability Class
Design Package
Design Model
Interface
Use-Case Realizations
Define Testability Elements
Test Designer

Subsystem Design Steps

- Distribute subsystem behavior to subsystem elements



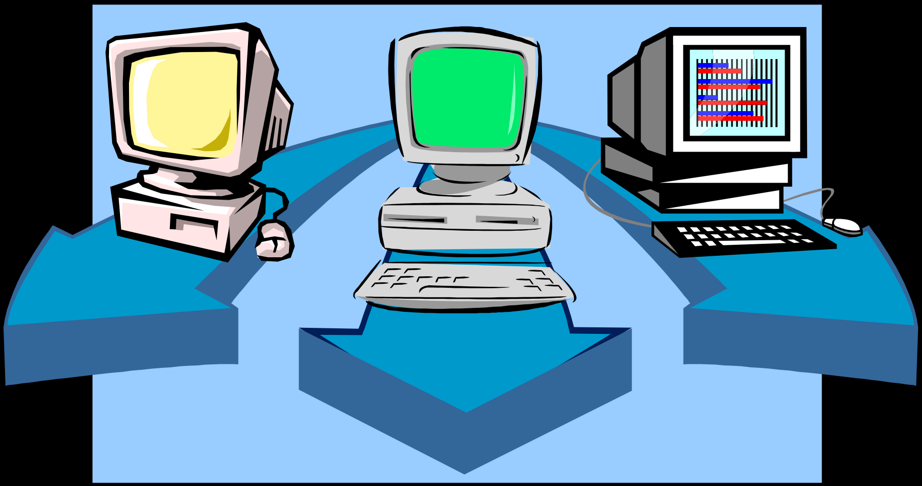
- Document subsystem elements

- Describe subsystem dependencies

- Checkpoints

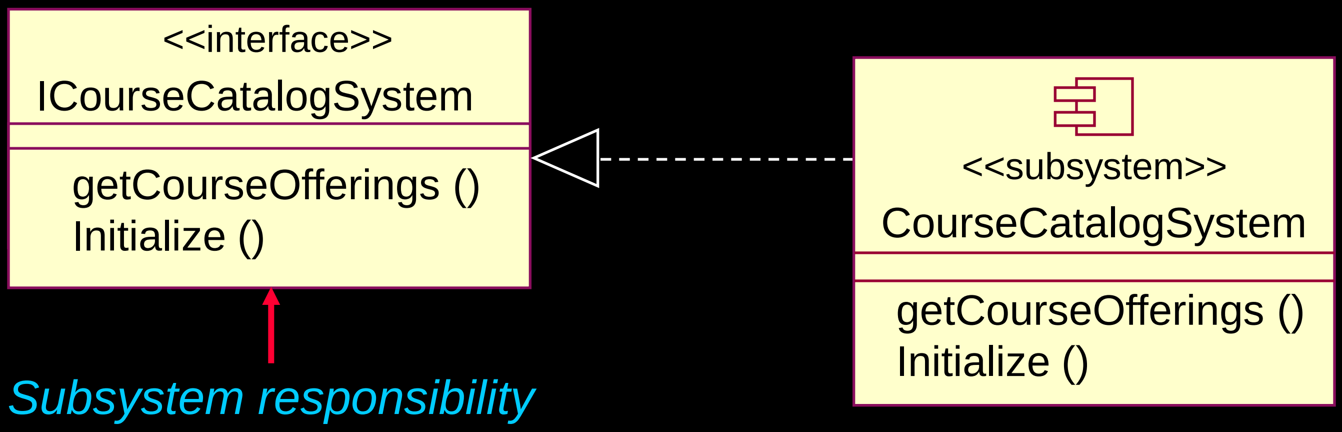
Subsystem Design Steps

- ★ Distribute subsystem behavior to subsystem elements
- Document subsystem elements
- Describe subsystem dependencies
- Checkpoints



Subsystem Responsibilities

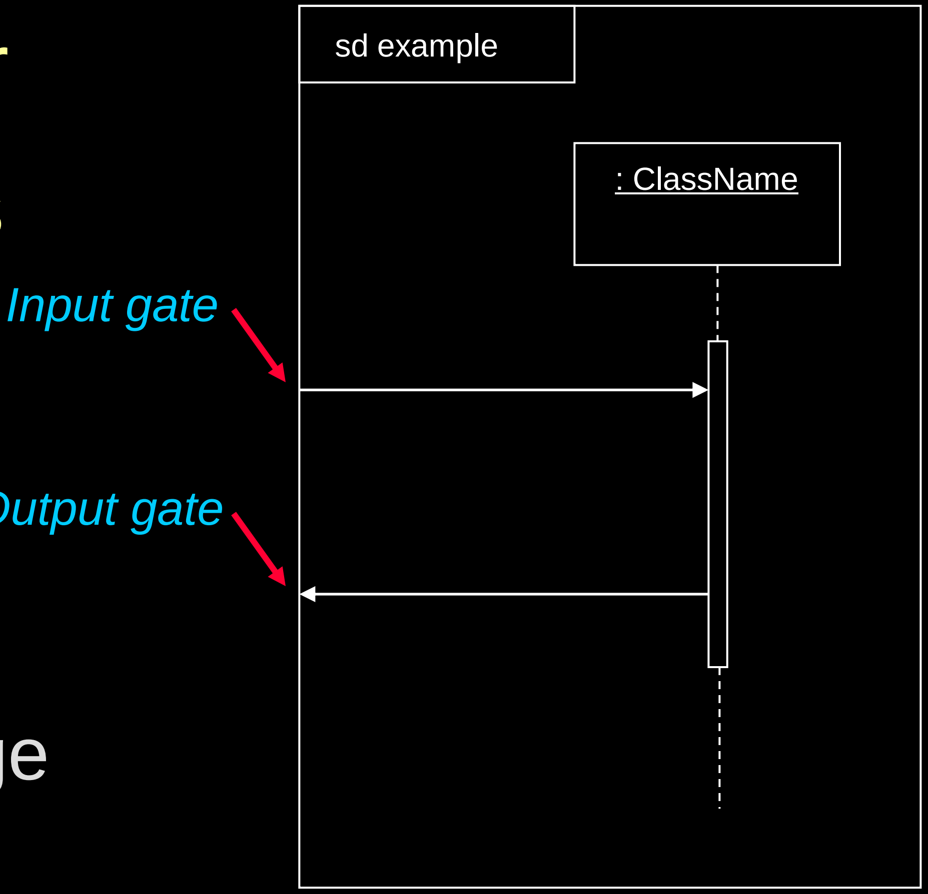
- Subsystem responsibilities defined by interface operations
  - Model interface realizations
- Interface may be realized by
  - Internal class behavior
  - Subsystem behavior



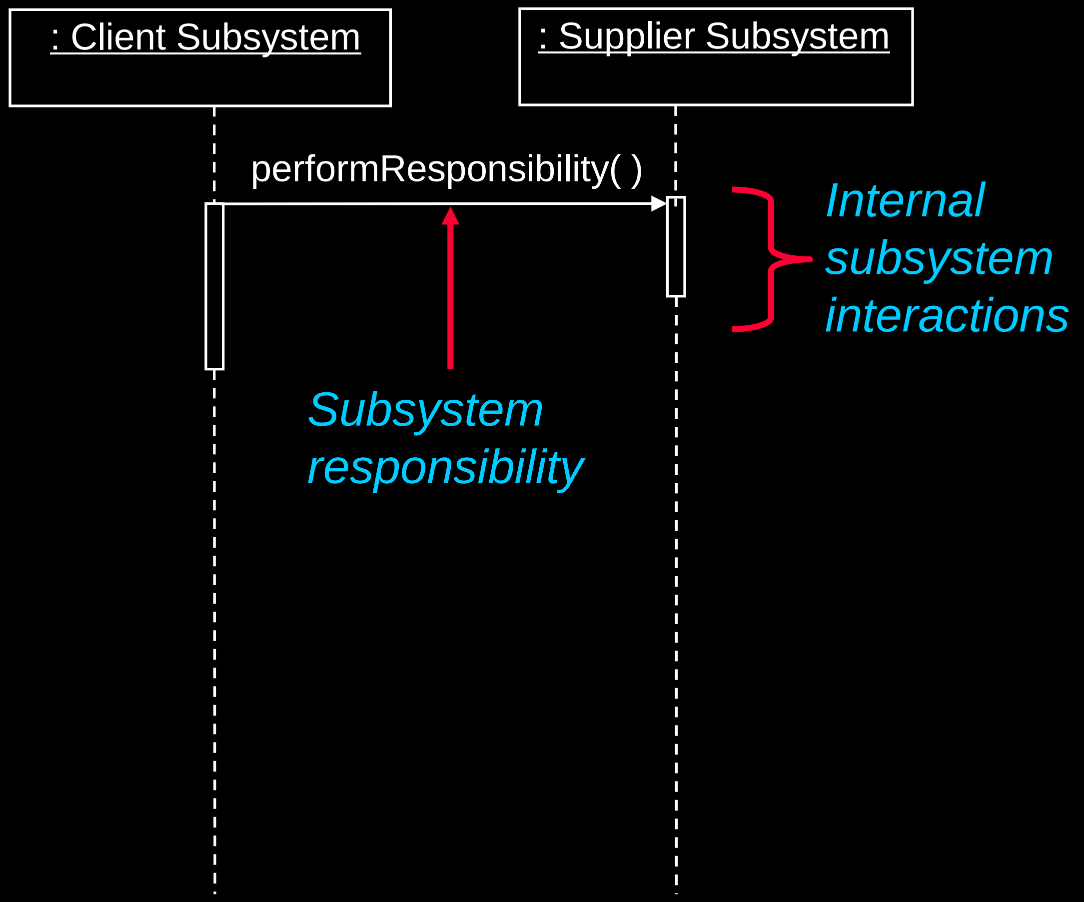
Subsystem responsibility

What Are Gates?

- A connection point in an interaction for a message that comes into or goes outside the interaction.
  - A point on the boundary of the sequence diagram
  - The name of the connected message is the name of the gate



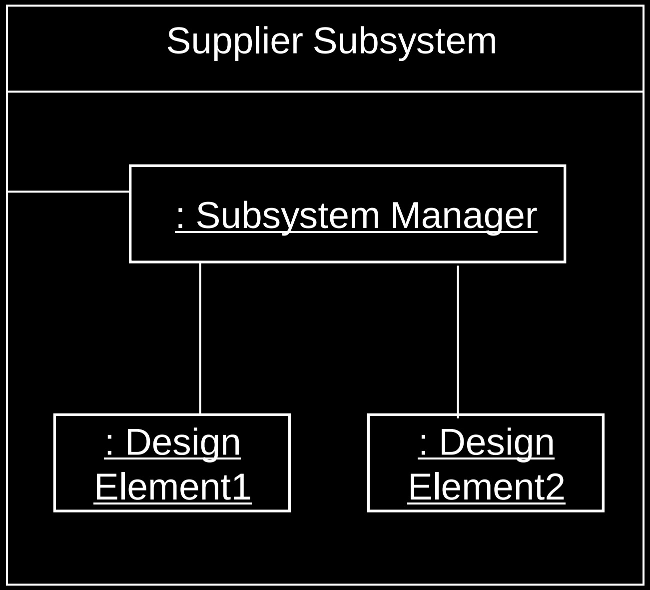
Subsystem Interaction Diagrams



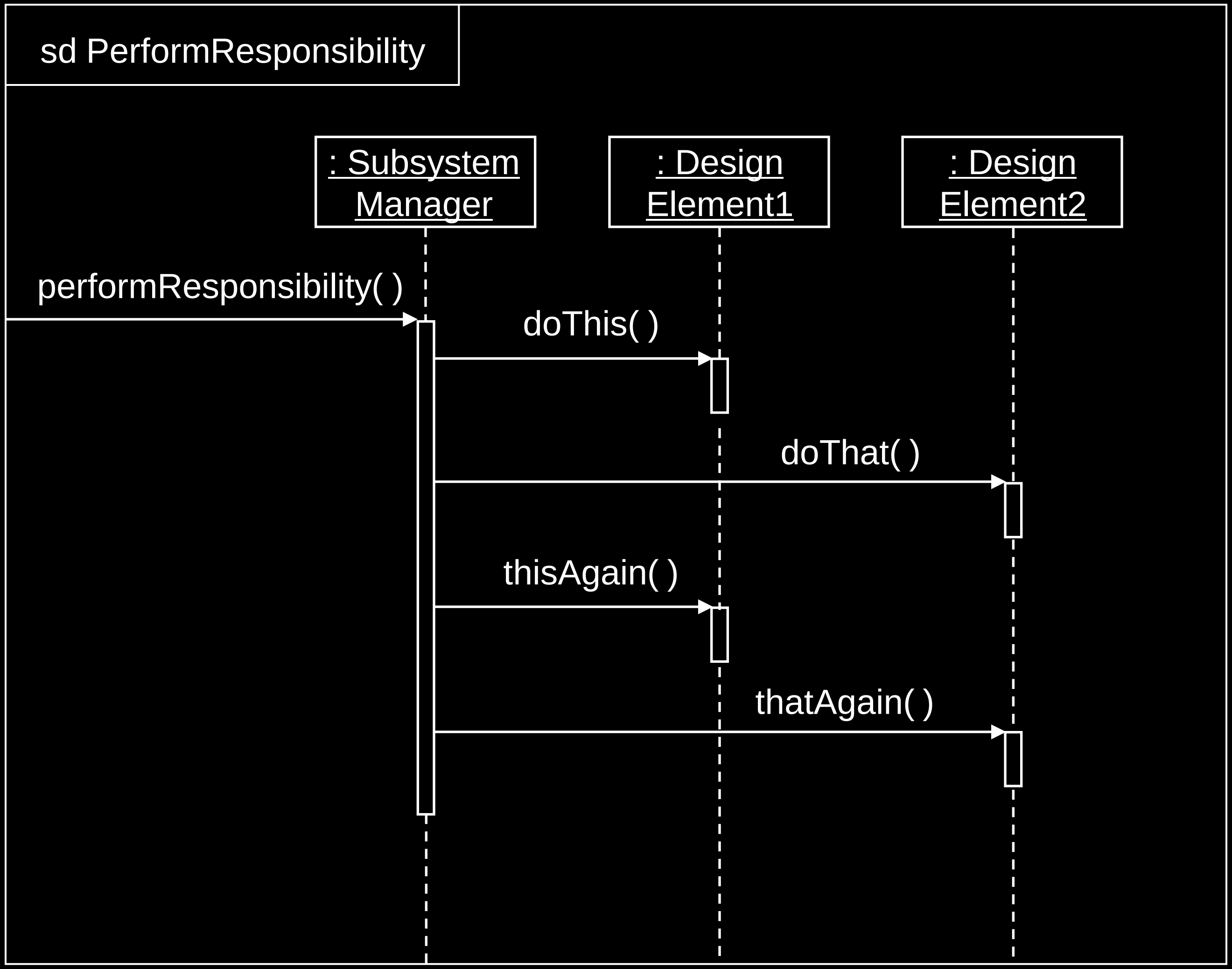
Black box view of subsystems

Internal Structure of Supplier Subsystem

- Subsystem Manager coordinates the internal behavior of the subsystem.
- The complete subsystem behavior is distributed amongst the internal Design Element classes.

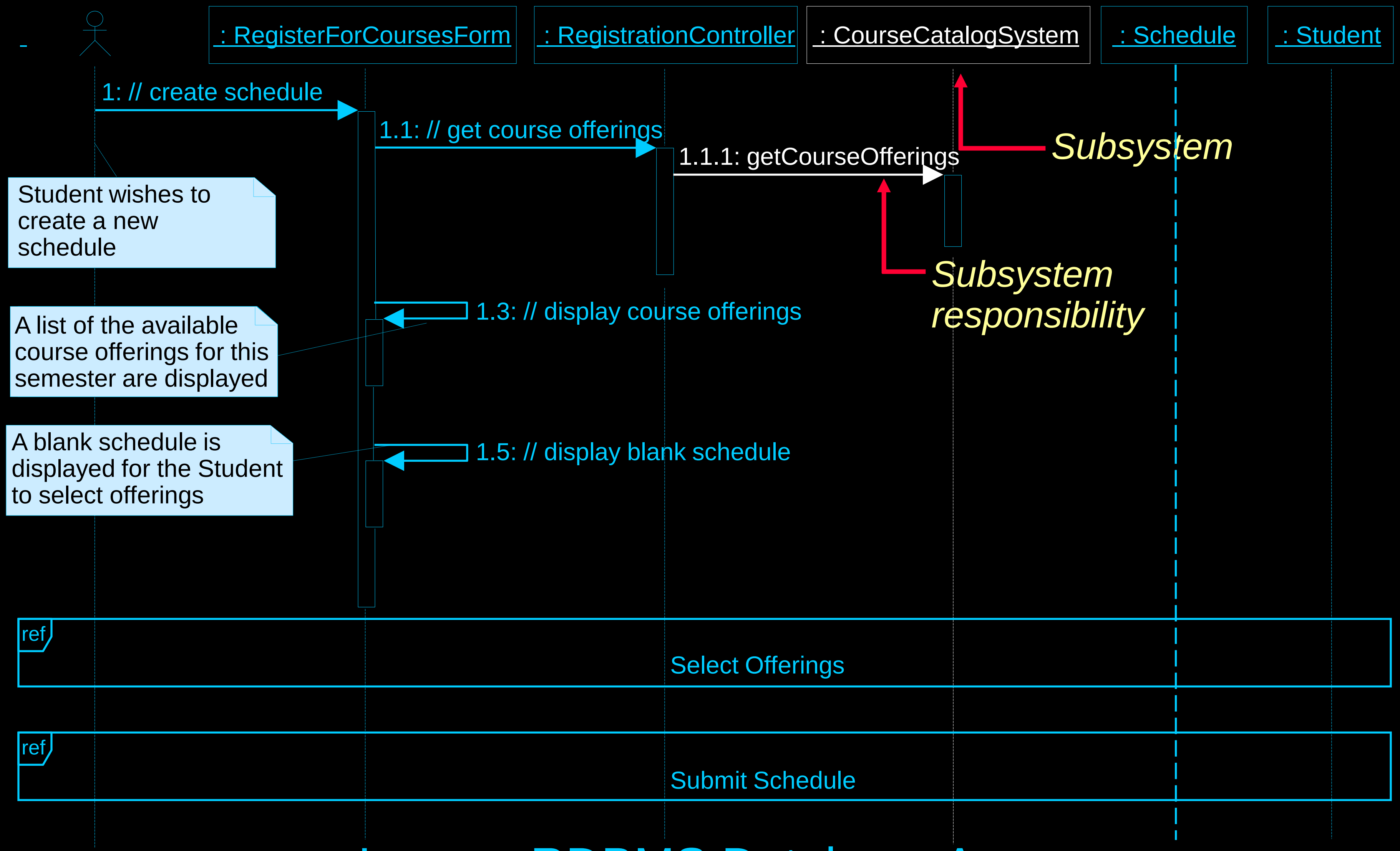


Modeling Convention: Internal Subsystem Interaction



White box view of Supplier Subsystem

Example: CourseCatalogSystem Subsystem in Context



Legacy RDBMS Database Access

Incorporating the Architectural Mechanisms: Persistency

- Analysis-Class-to-Architectural-Mechanism Map from Use-Case Analysis

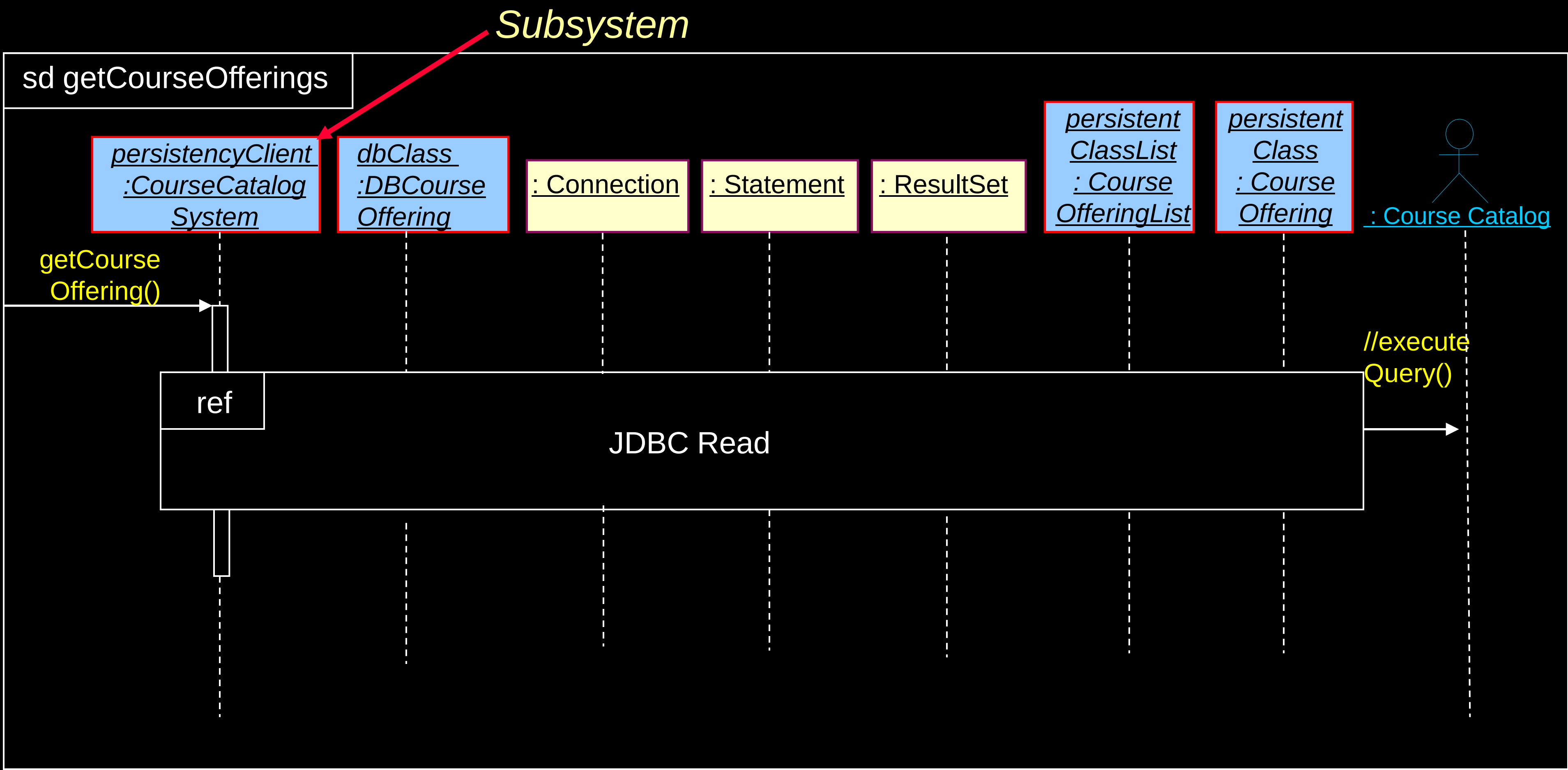
Analysis Class	Analysis Mechanism(s)
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	*Persistency, Legacy Interface*
Course	*Persistency, Legacy Interface*
RegistrationController	Distribution

OODBMS Persistency

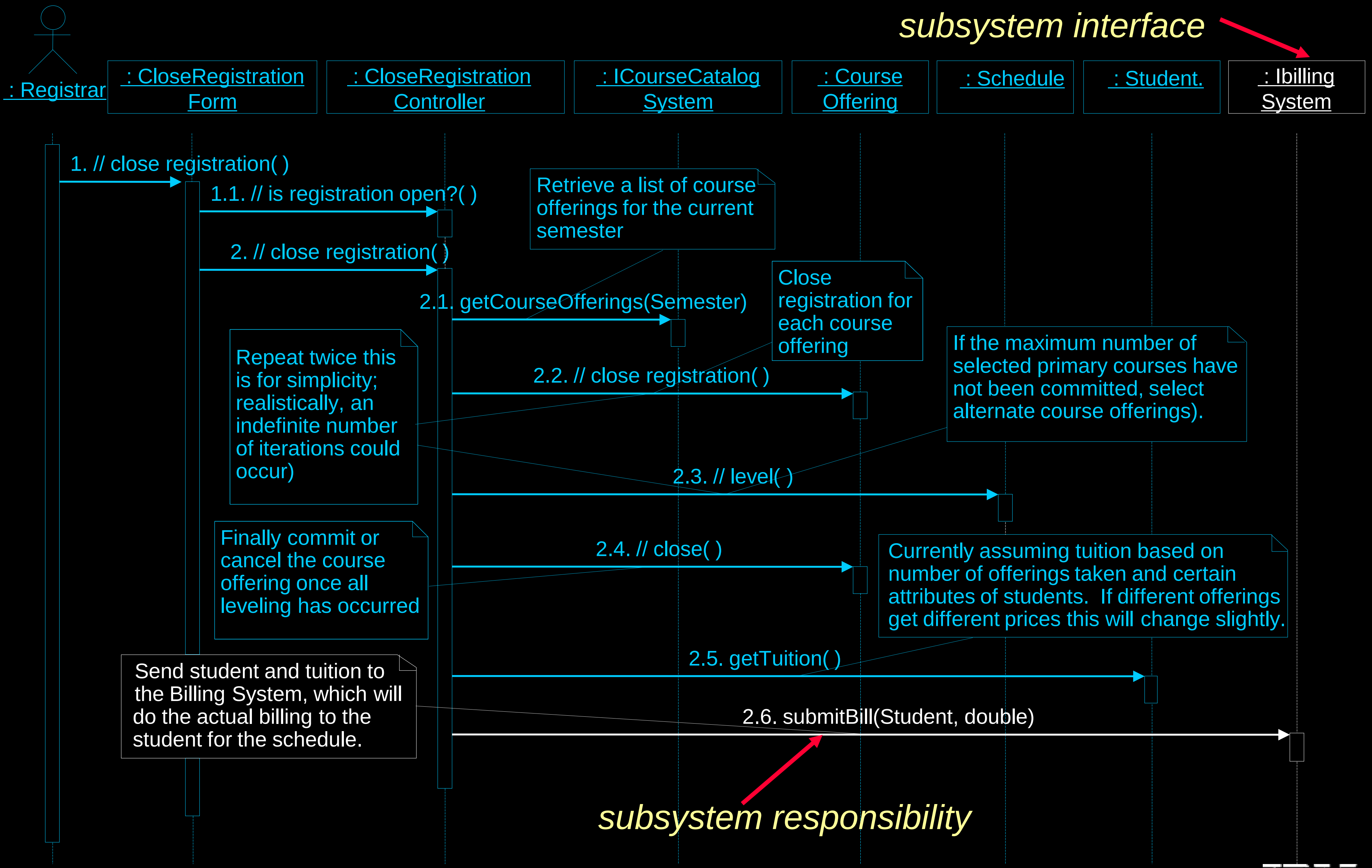
RDBMS Persistency

OODBMS Persistency was discussed in Use-Case Design

Example: Local CourseCatalogSystem Subsystem Interaction



Example: Billing System Subsystem In Context



subsystem interface

: Registrar

: CloseRegistration Form

: CloseRegistration Controller

: ICourseCatalog System

: Course Offering

: Schedule

: Student.

: Ibilling System

1. // close registration( )

1.1. // is registration open?( )

2. // close registration( )

2.1. getCourseOfferings(Semester)

2.2. // close registration( )

2.3. // level( )

2.4. // close( )

2.5. getTuition( )

2.6. submitBill(Student, double)

Retrieve a list of course offerings for the current semester

Close registration for each course offering

Repeat twice this is for simplicity; realistically, an indefinite number of iterations could occur)

If the maximum number of selected primary courses have not been committed, select alternate course offerings).

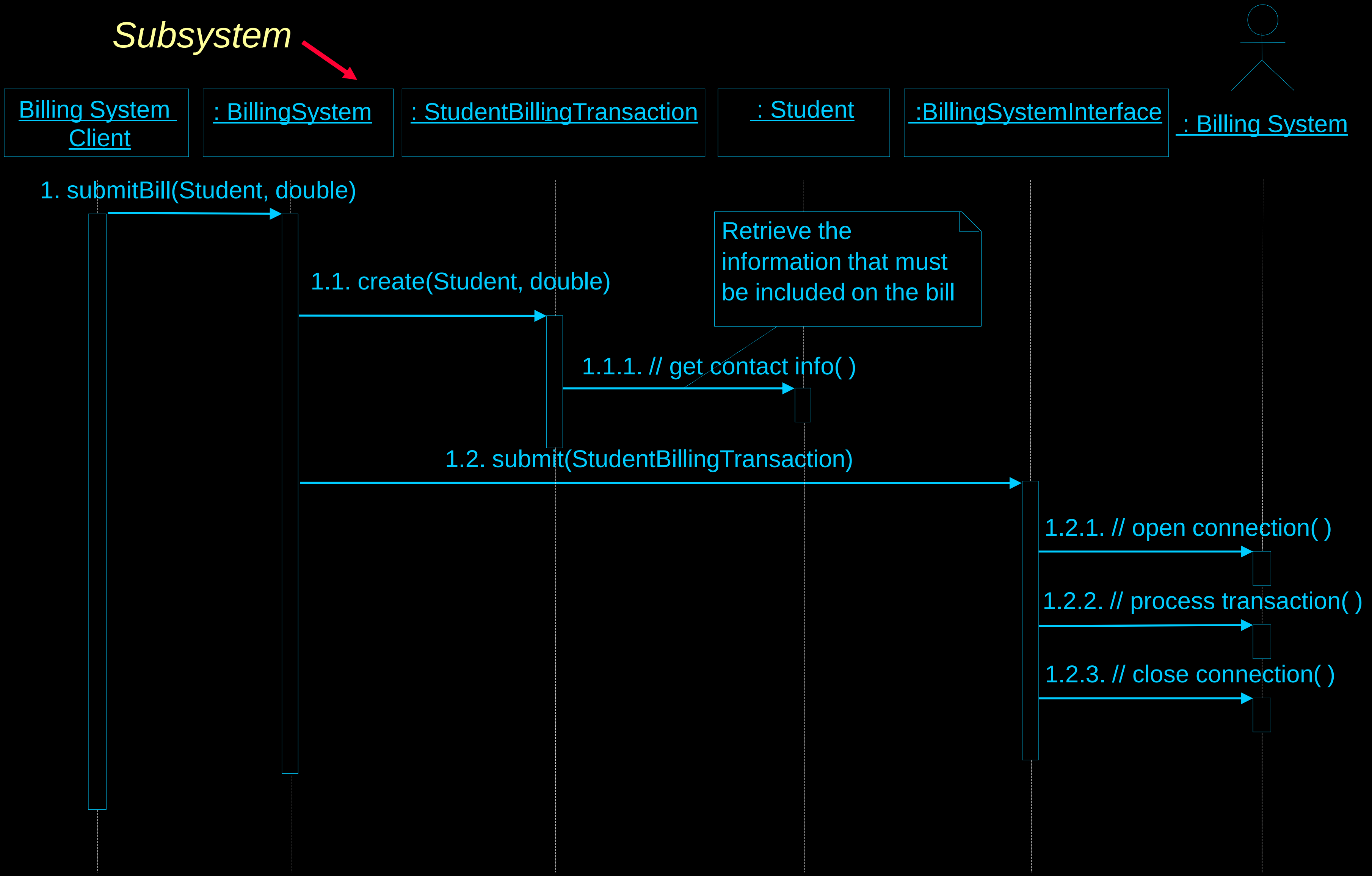
Finally commit or cancel the course offering once all leveling has occurred

Currently assuming tuition based on number of offerings taken and certain attributes of students. If different offerings get different prices this will change slightly.

Send student and tuition to the Billing System, which will do the actual billing to the student for the schedule.

subsystem responsibility

Example: Local BillingSystem Subsystem Interaction



Subsystem

Billing System Client

: BillingSystem

: StudentBillingTransaction

: Student

:BillingSystemInterface

: Billing System

1. submitBill(Student, double)

1.1. create(Student, double)

Retrieve the information that must be included on the bill

1.1.1. // get contact info( )

1.2. submit(StudentBillingTransaction)

1.2.1. // open connection( )

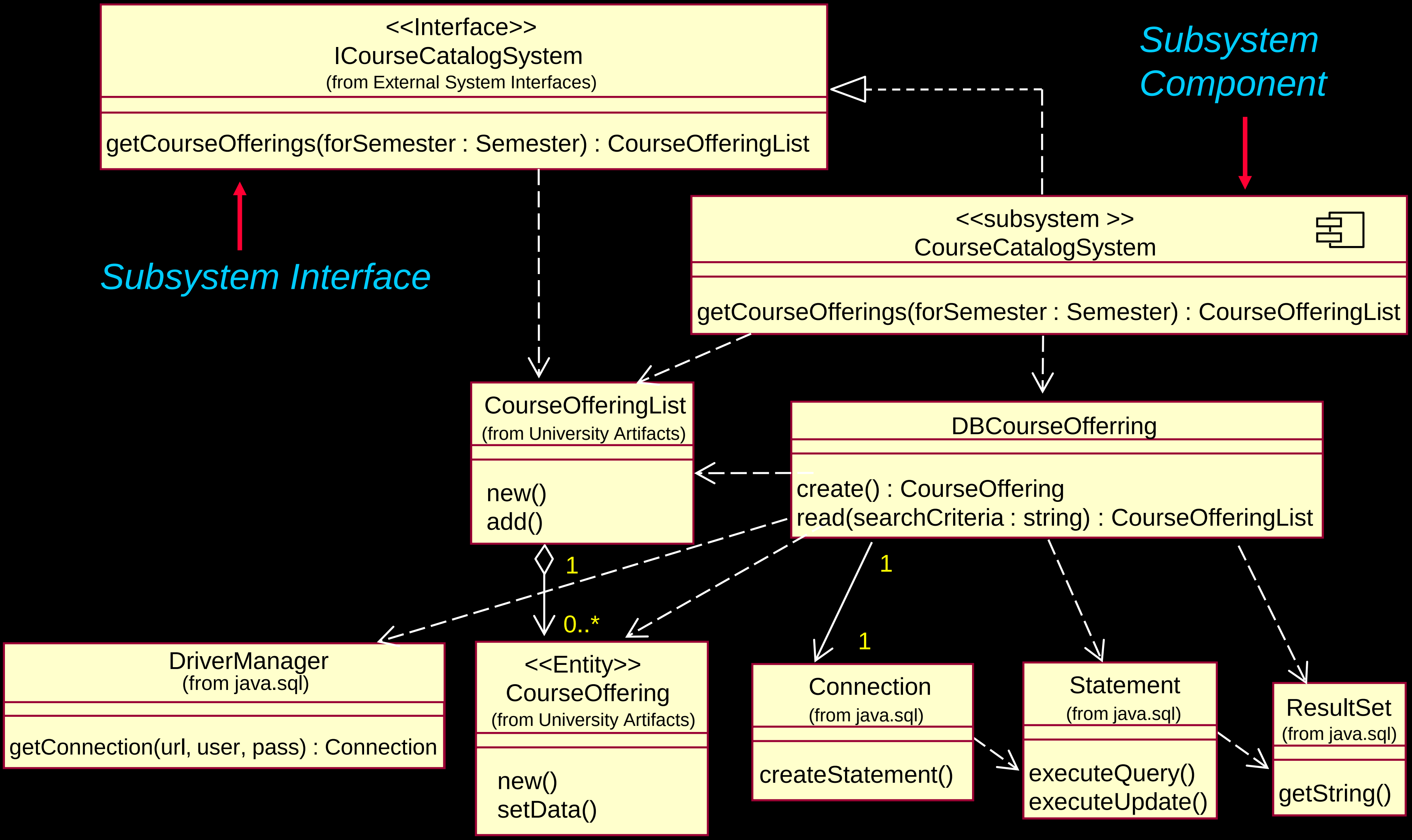
1.2.2. // process transaction( )

1.2.3. // close connection( )

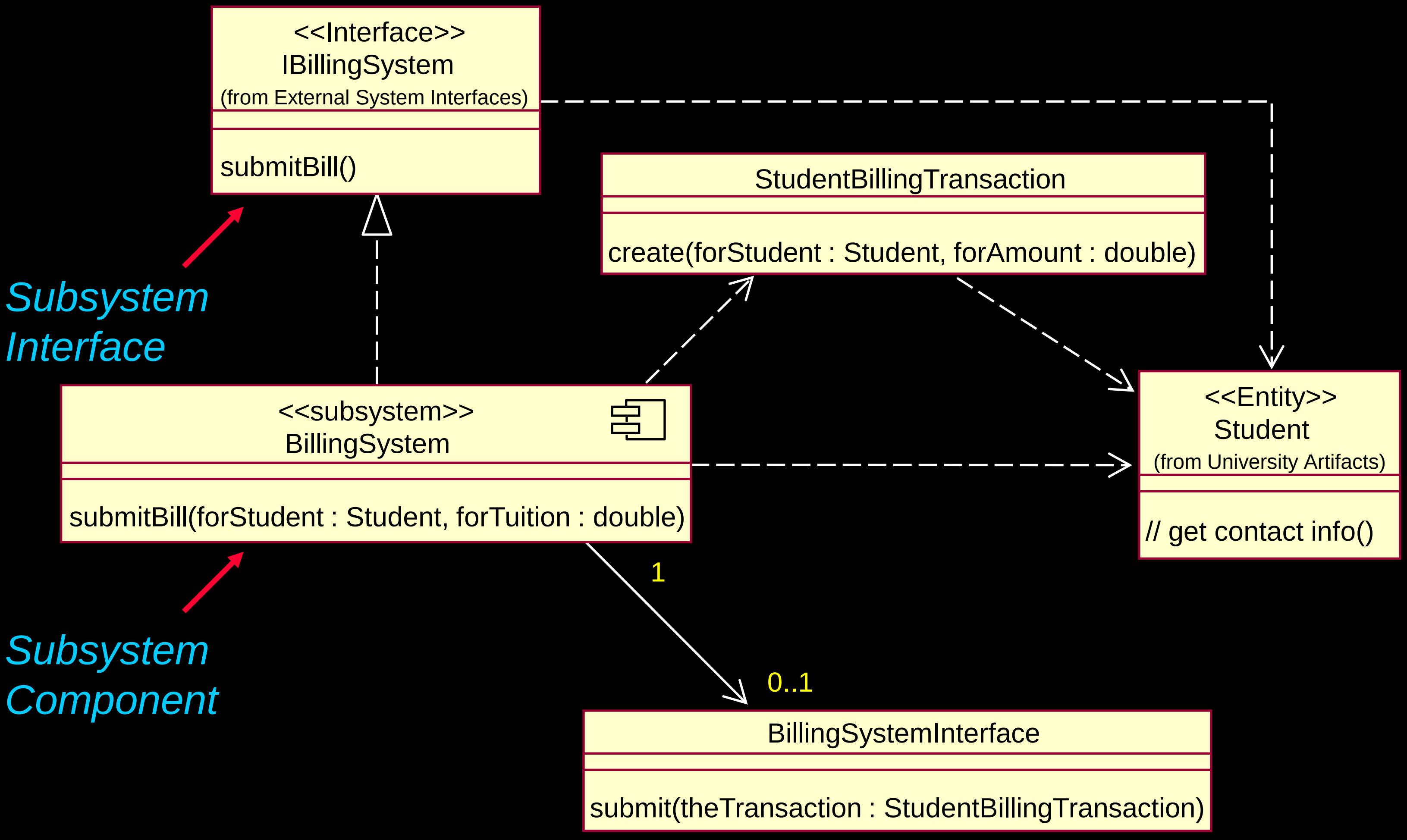
Subsystem Design Steps

- Distribute subsystem behavior to subsystem elements
- ★ Document subsystem elements
- Describe subsystem dependencies
- Checkpoints

Example: CourseCatalogSystem Subsystem Elements



Example: Billing System Subsystem Elements



Subsystem Interface

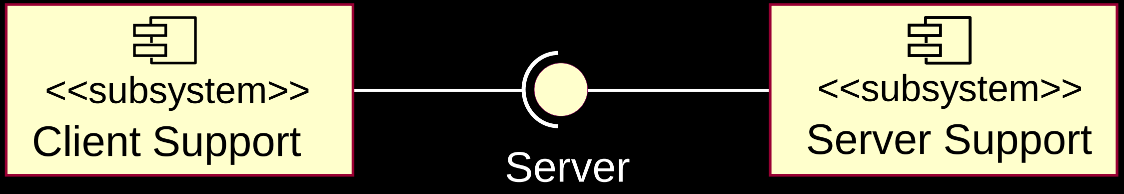
Subsystem Component

Subsystem Design Steps

- Distribute subsystem behavior to subsystem elements
- Document subsystem elements
- ★ Describe subsystem dependencies
- Checkpoints

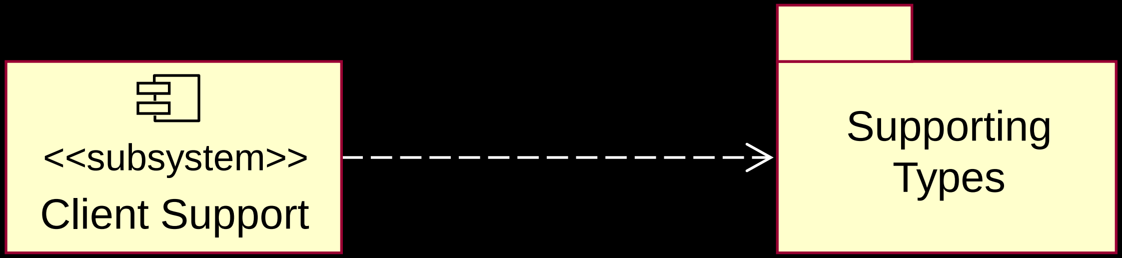
Subsystem Dependencies: Guidelines

- Subsystem dependency on a subsystem



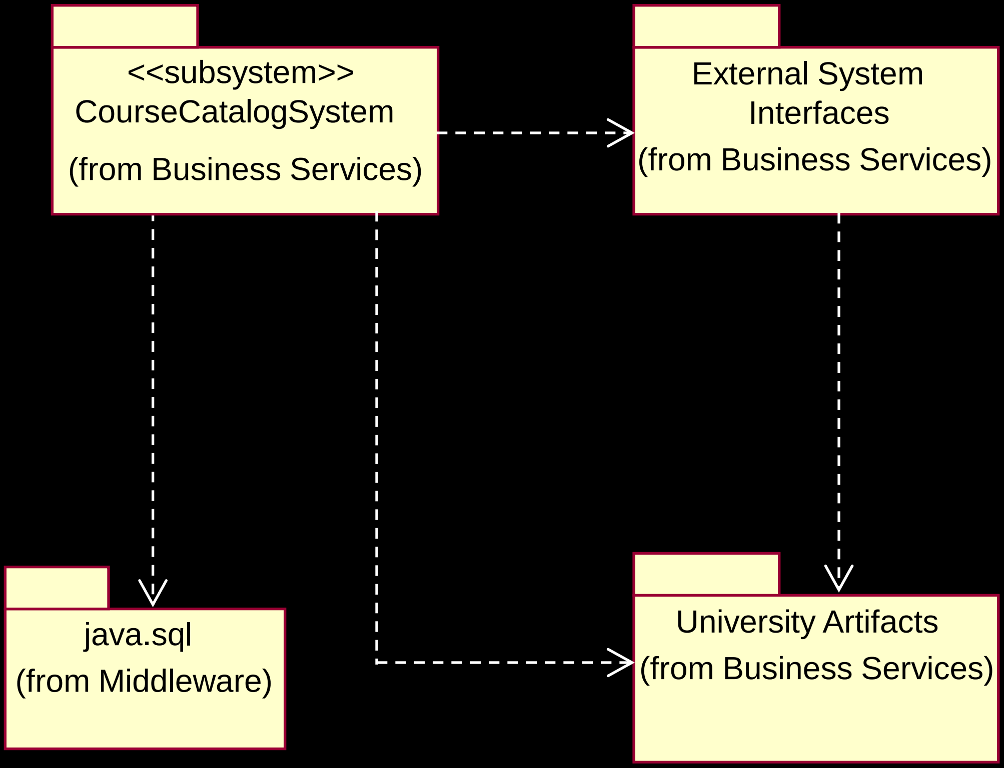
Flexible, Preferred

- Subsystem dependency on a package

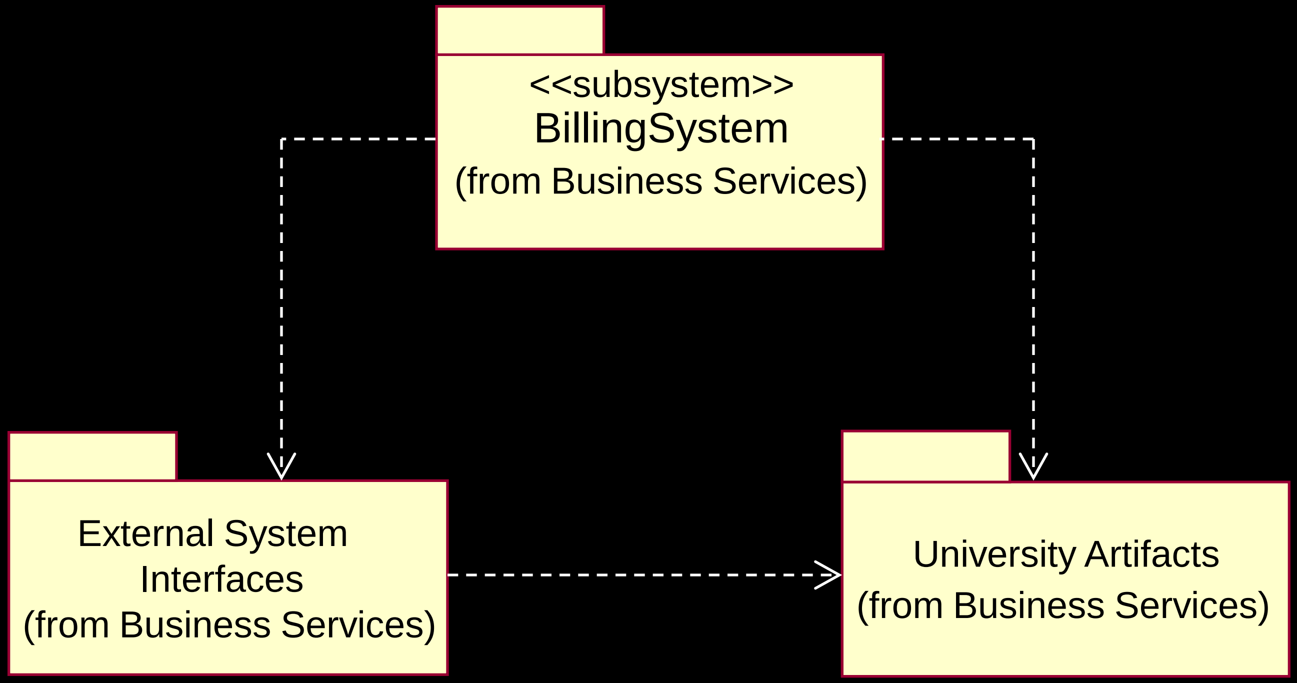


Use with care

Example: CourseCatalogSystem Subsystem Dependencies



Example: BillingSystem Subsystem Dependencies



11 Class Design

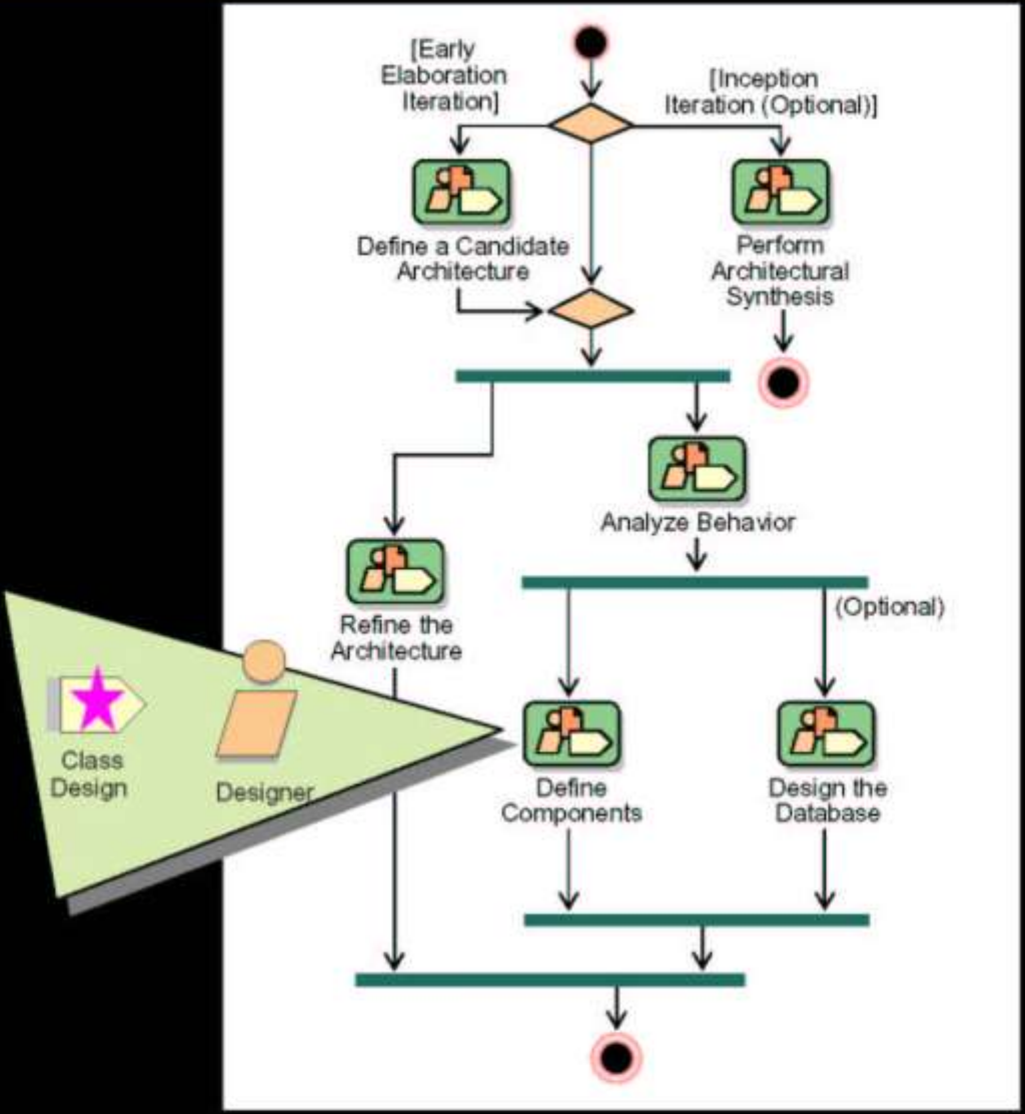
IBM Software Group

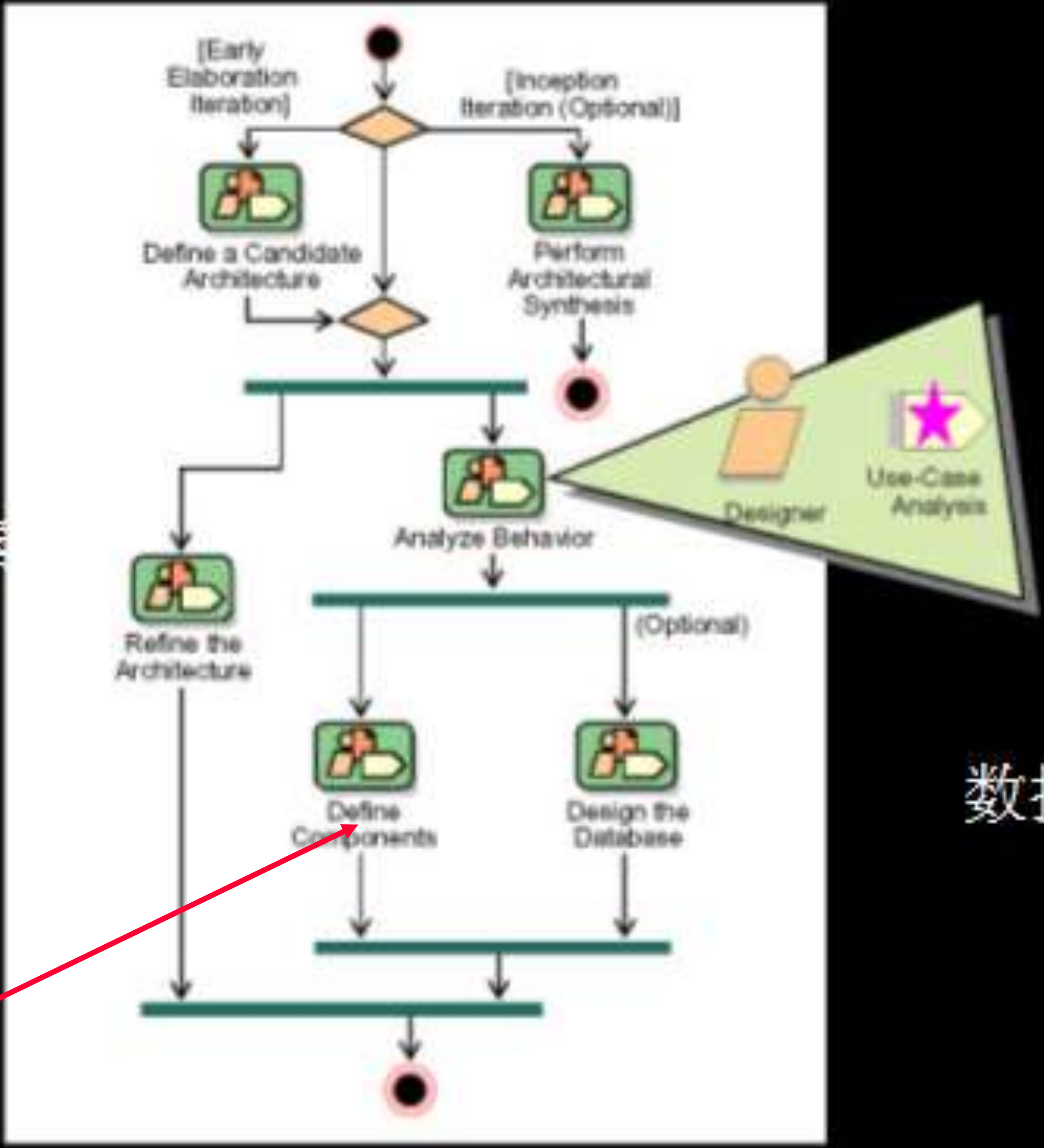
Mastering Object-Oriented Analysis and Design with UML

Module 11: Class Design

Rational. software

Class Design in Context





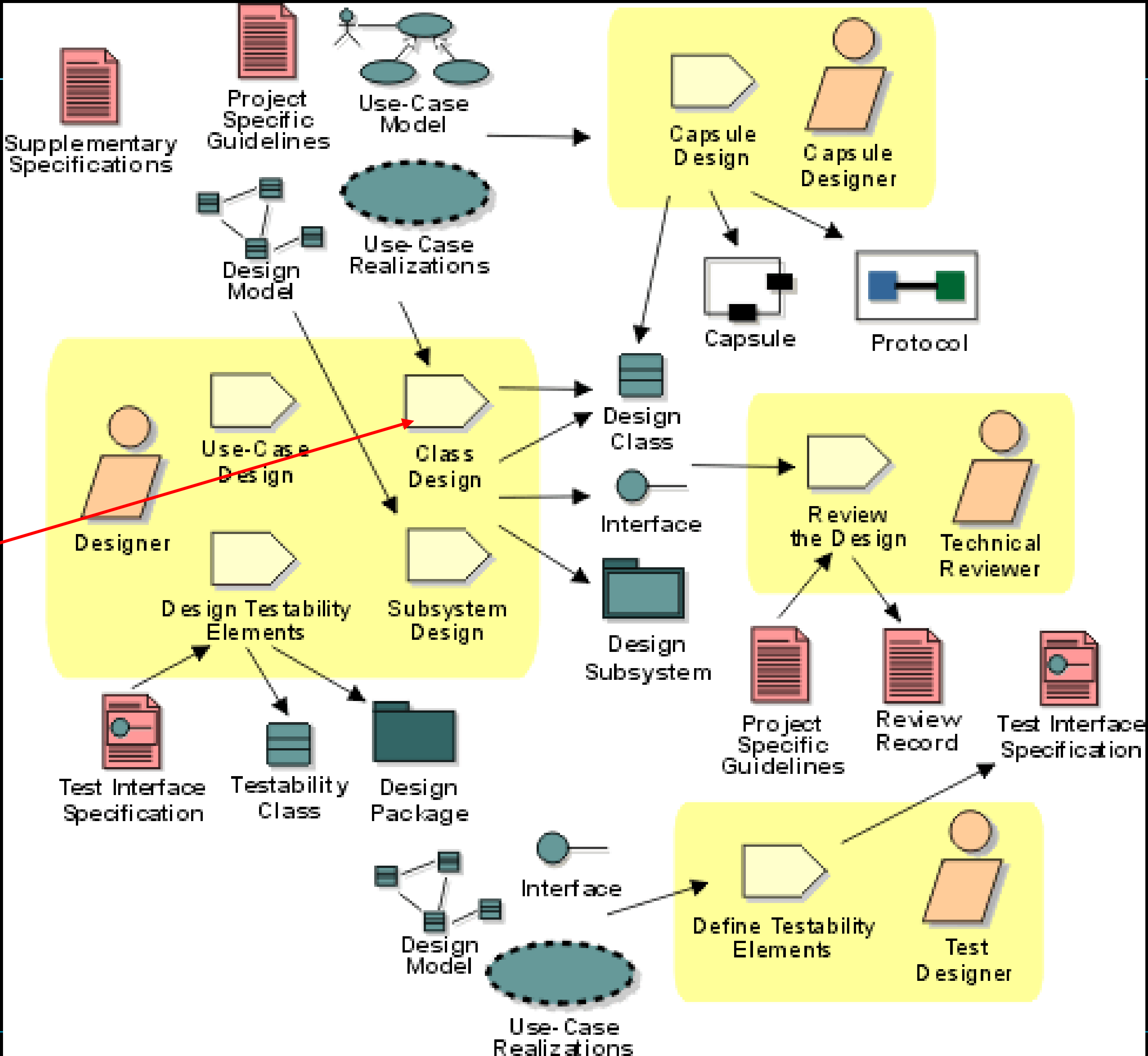
架构分析

用例分析

识别设计元素
运行时架构
描述分布

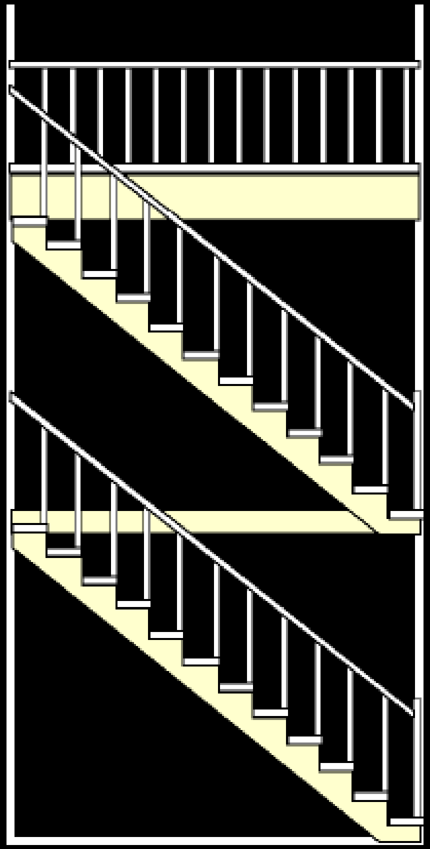
数据库设计

用例设计
子系统设计
类设计


Supplementary Specifications
Project Specific Guidelines
Use-Case Model
Capsule Design
Capsule Designer
Design Model
Use-Case Realizations
Capsule
Protocol
Designer
Use-Case Design
Class Design
Design Class
Interface
Review the Design
Technical Reviewer
Design Testability Elements
Subsystem Design
Design Subsystem
Project Specific Guidelines
Review Record
Test Interface Specification
Test Interface Specification
Testability Class
Design Package
Design Model
Interface
Define Testability Elements
Test Designer
Use-Case Realizations

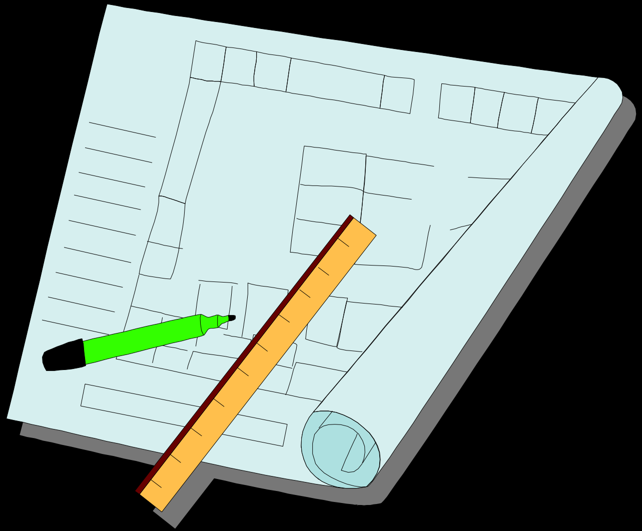
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Nonfunctional Requirements in General
- Checkpoints



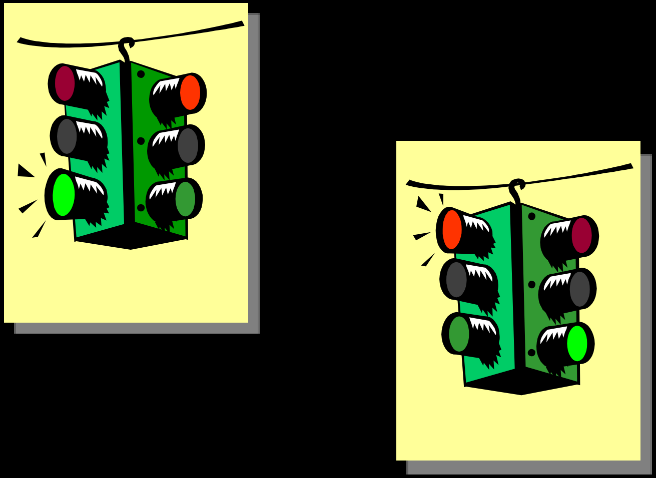
Class Design Steps

- ★ Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- ★ Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Identify and Define the States

- Significant, dynamic attributes

The maximum number of students per course offering is 10

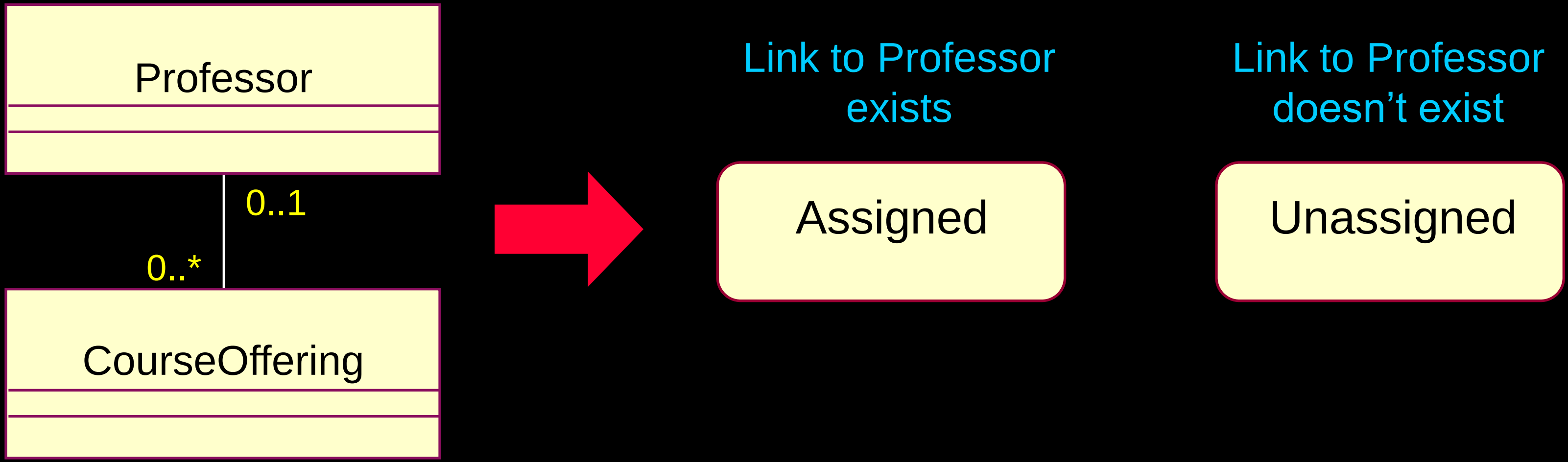
numStudents < 10

Open

numStudents >= 10

Closed

- Existence and non-existence of certain links



Link to Professor exists

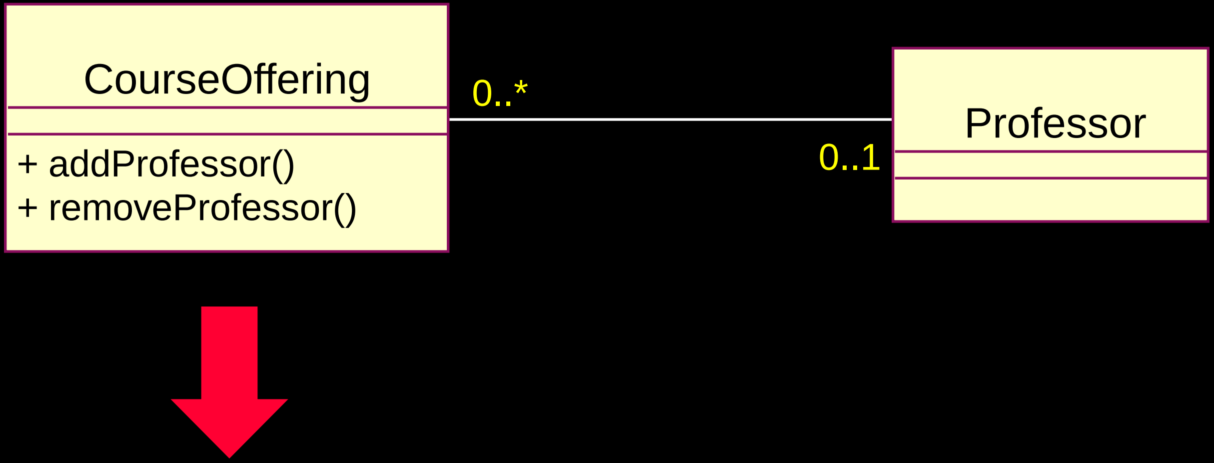
Assigned

Link to Professor doesn’t exist

Unassigned

Identify the Events

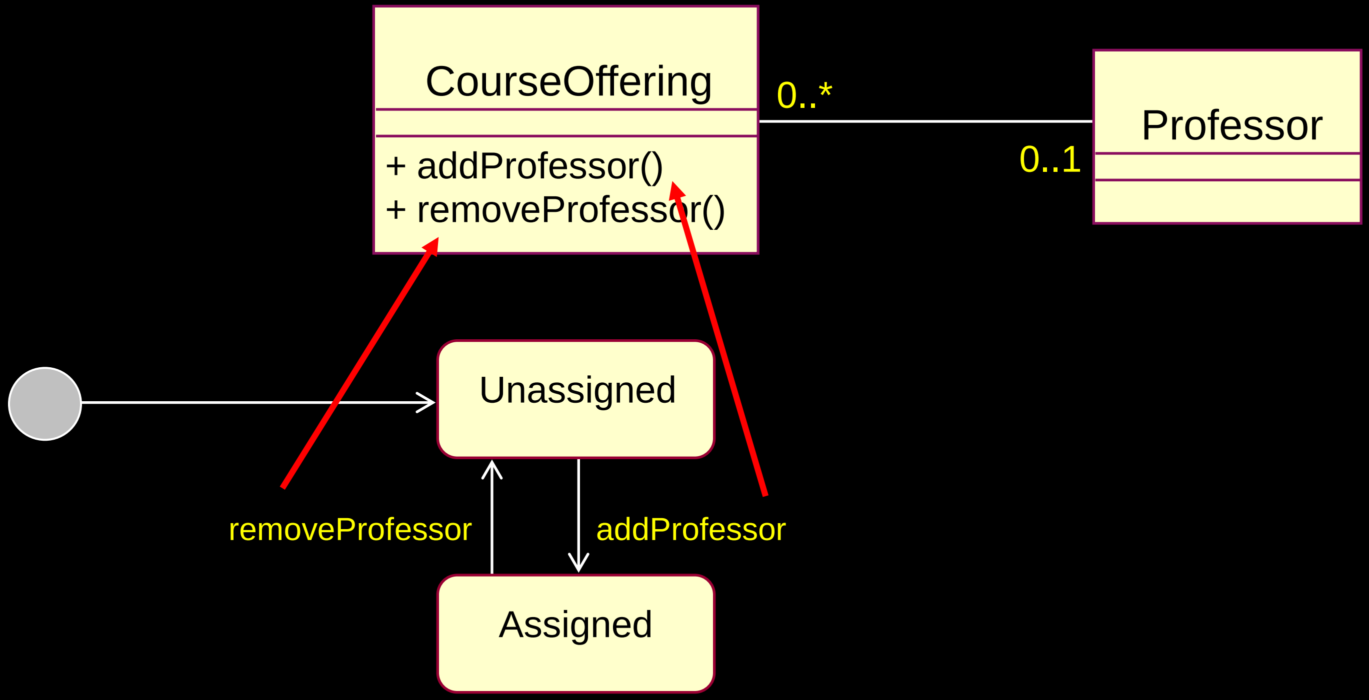
- Look at the class interface operations



Events: addProfessor,
removeProfessor

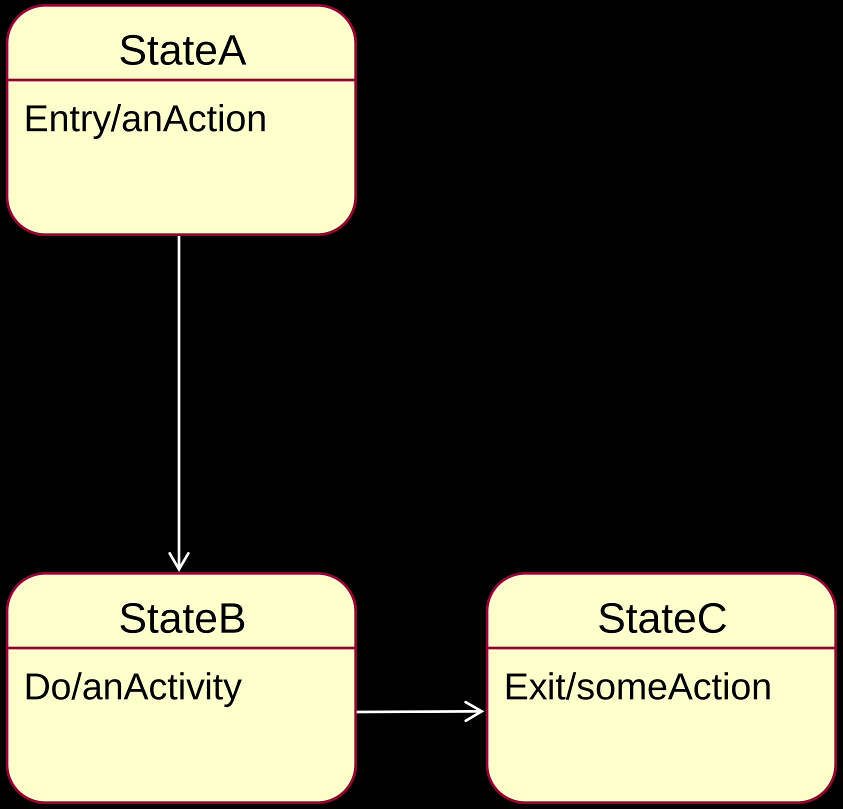
Identify the Transitions

- For each state, determine what events cause transitions to what states, including guard conditions, when needed
- Transitions describe what happens in response to the receipt of an event

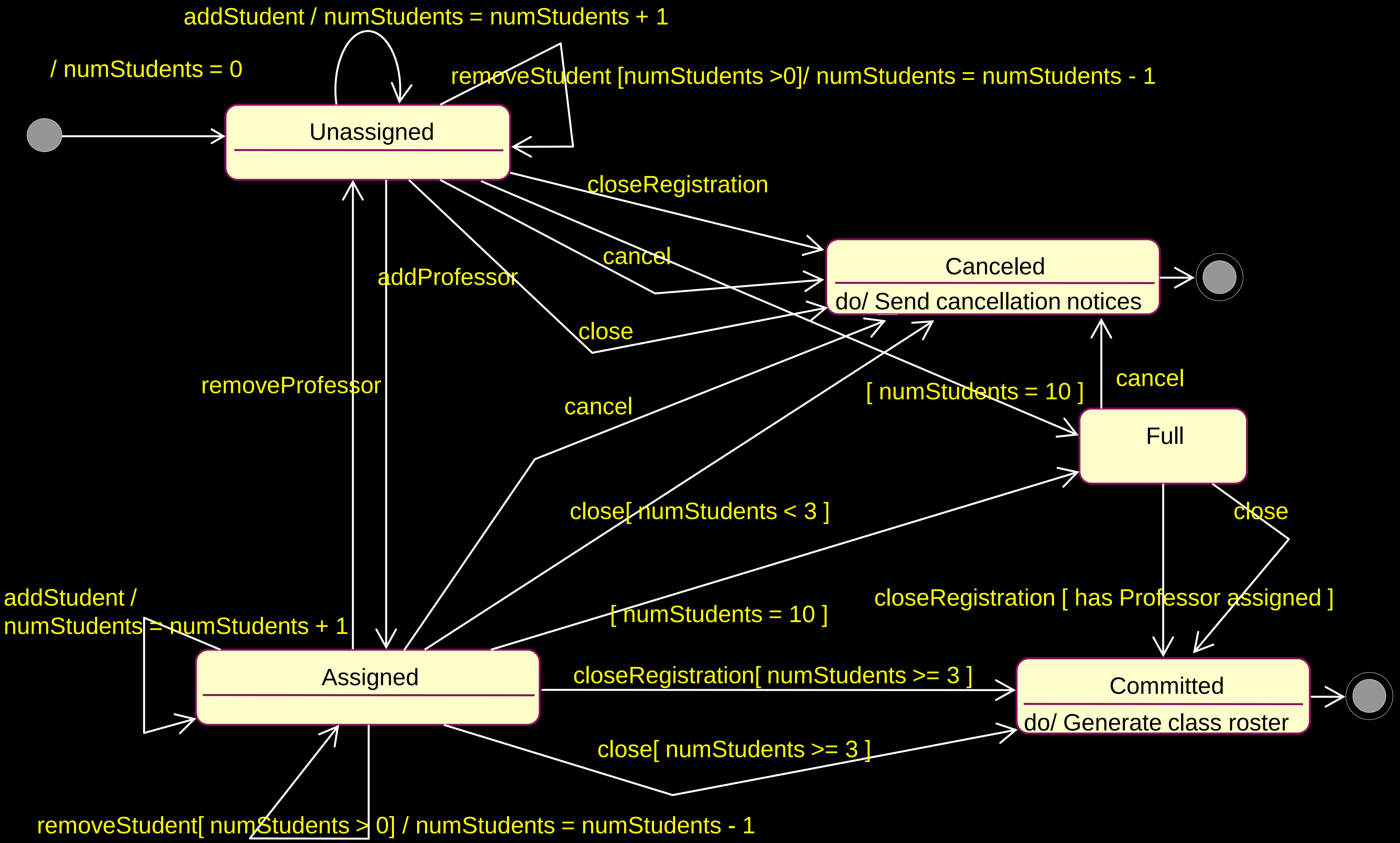


Add Activities

- Entry
  - Executed when the state is entered
- Do
  - Ongoing execution
- Exit
  - Executed when the state is exited



Example: State Machine



addStudent / numStudents = numStudents + 1

/ numStudents = 0

removeStudent [numStudents >0]/ numStudents = numStudents - 1

Unassigned

closeRegistration

Canceled

do/ Send cancellation notices

cancel

addProfessor

close

removeProfessor

cancel

[numStudents = 10]

cancel

Full

close[numStudents < 3]

close

addStudent /
numStudents = numStudents + 1

closeRegistration [has Professor assigned]

[numStudents = 10]

Assigned

closeRegistration[numStudents >= 3]

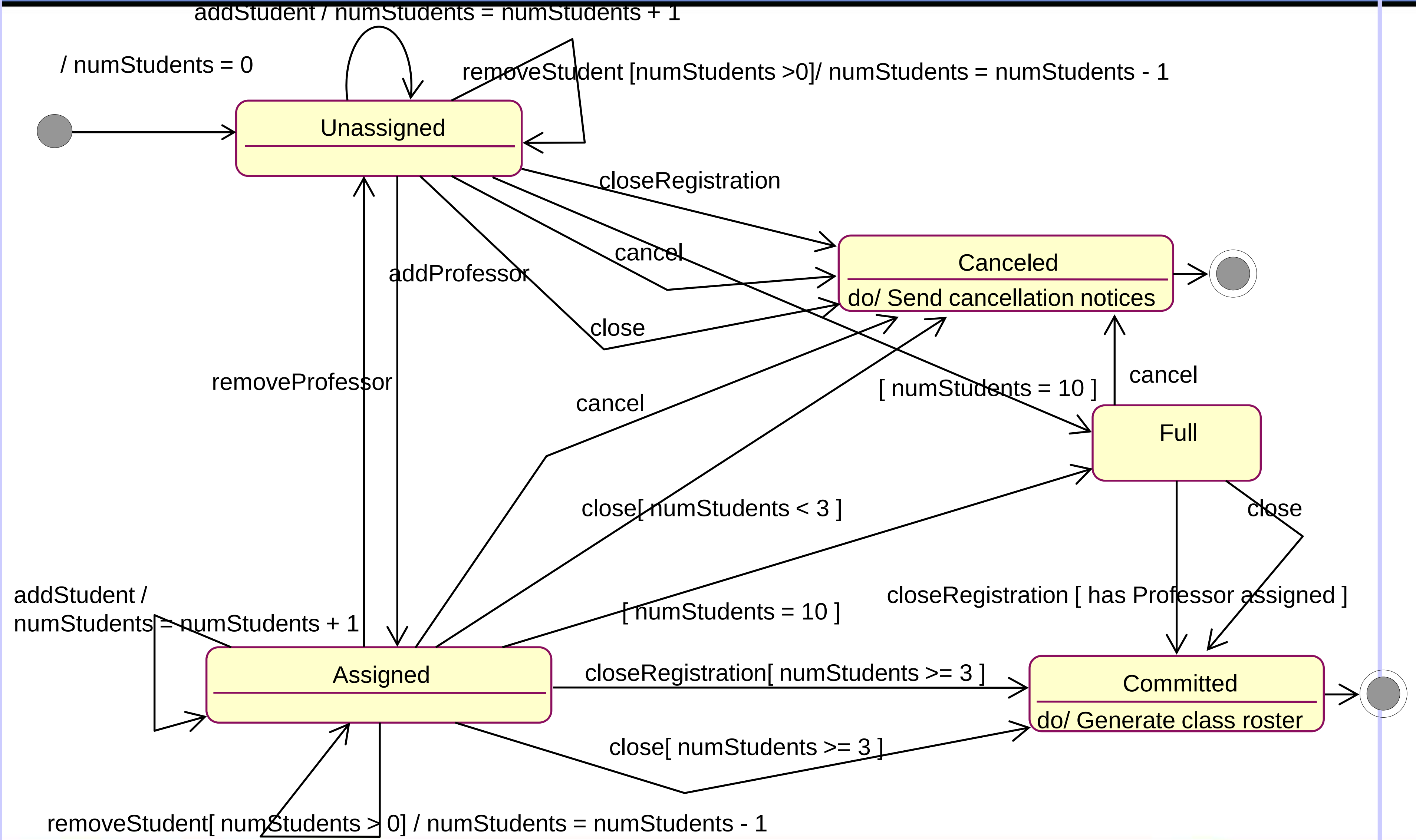
Committed

do/ Generate class roster

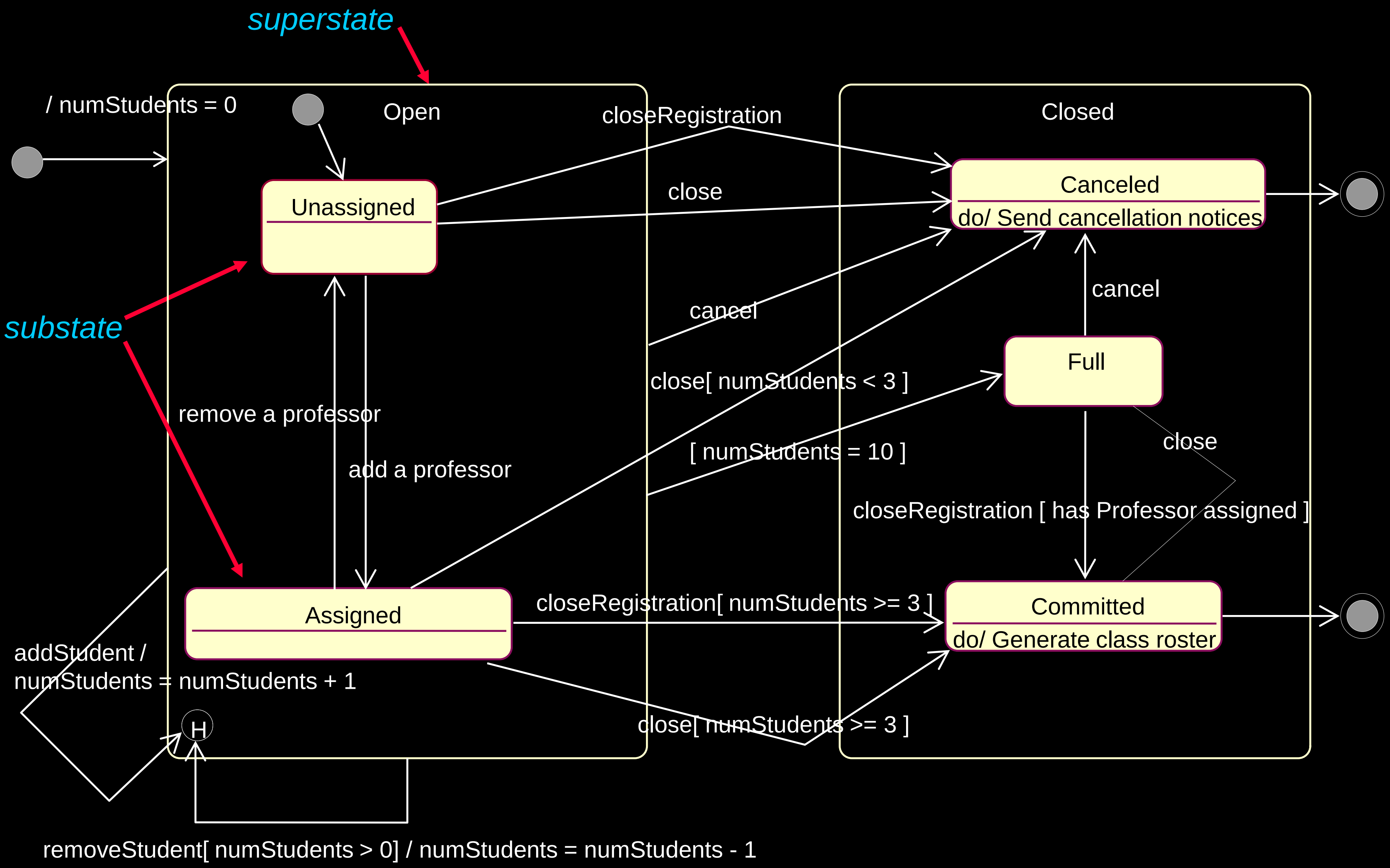
close[numStudents >= 3]

removeStudent[numStudents > 0] / numStudents = numStudents - 1

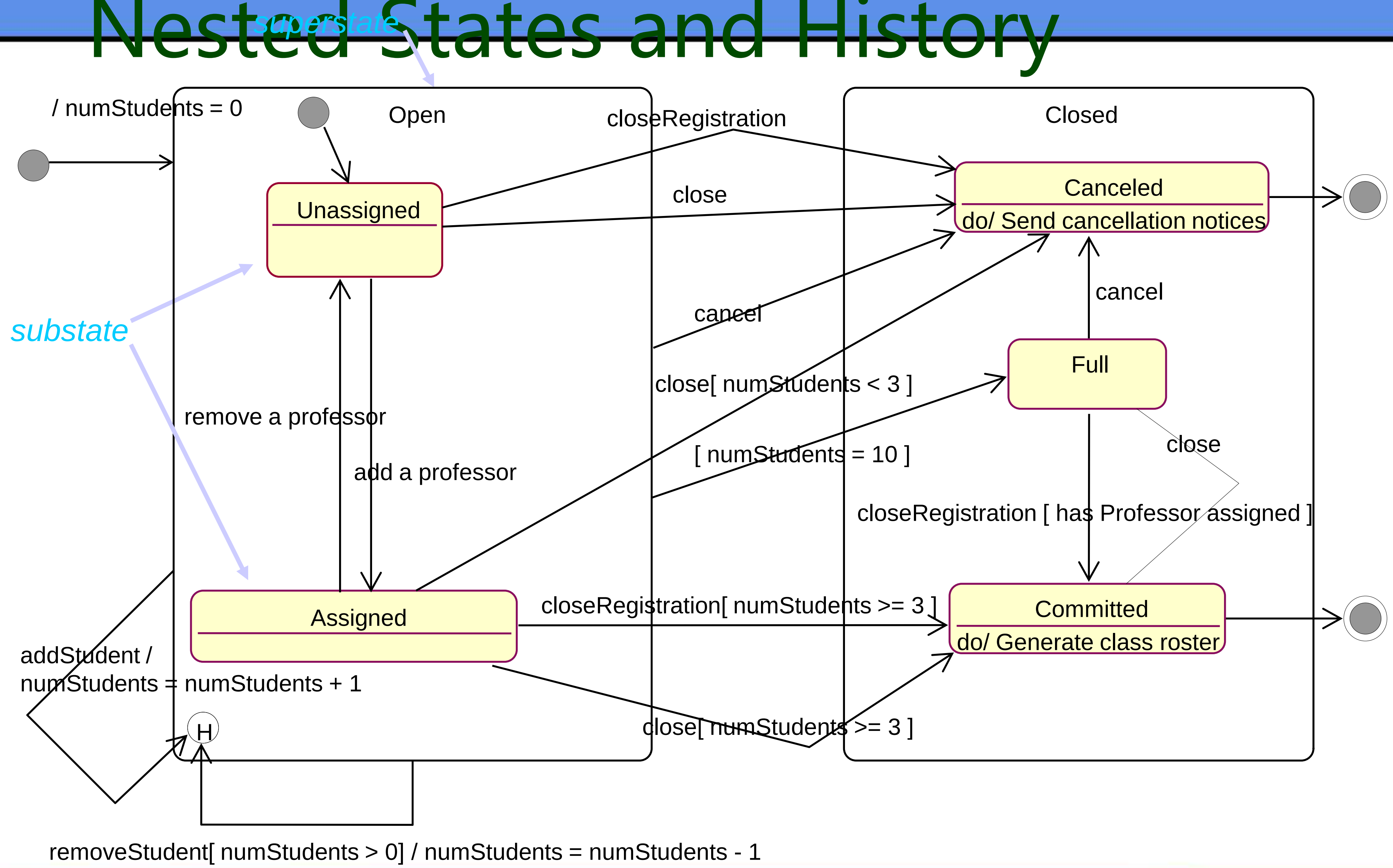
Example: State Machine



Example: State Machine with Nested States and History

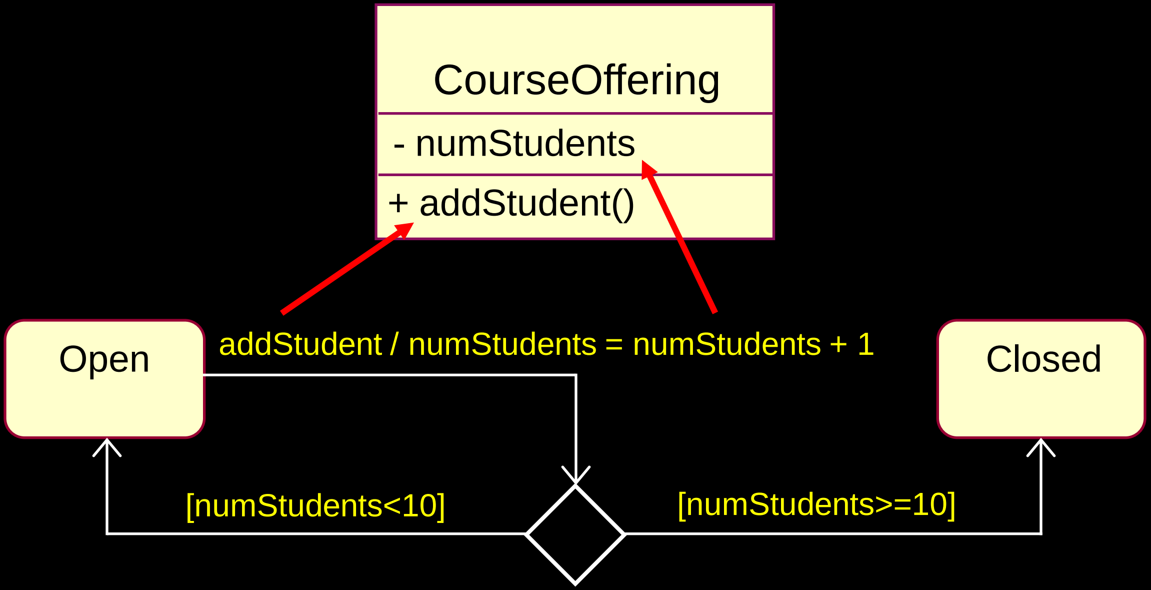


Example: State Machine with Nested States and History



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

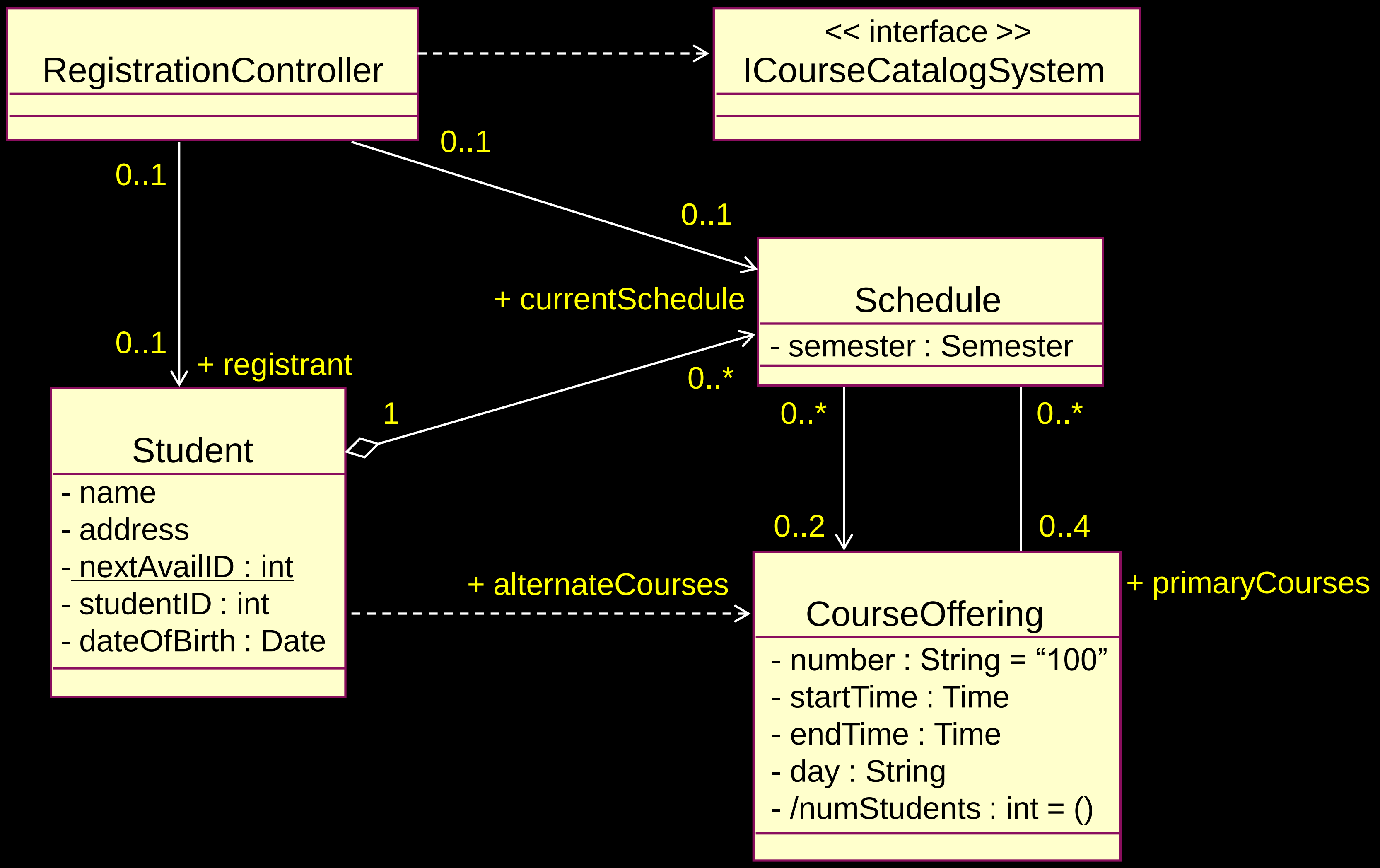
- Events may map to operations
- Methods should be updated with state-specific information
- States are often represented using attributes
  - This serves as input into the "*Define Attributes*" step



Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- ★ Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

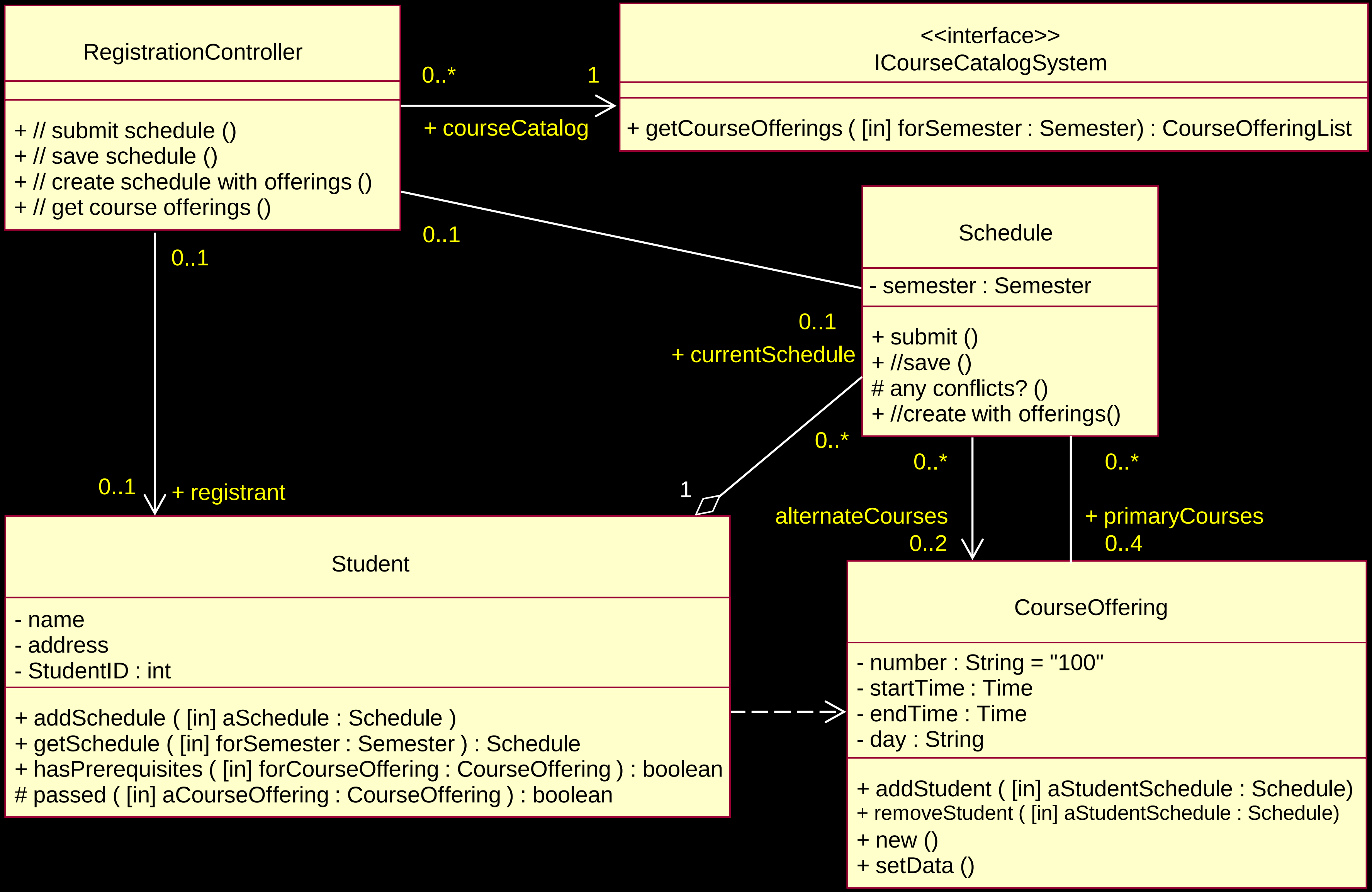
Example: Define Attributes



Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- ★ Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Example: Define Dependencies (before)



RegistrationController

+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()

0..* 1
+ courseCatalog

<<interface>>
ICourseCatalogSystem

+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList

0..1

Schedule

- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()

0..1
+ currentSchedule

0..1

0..1
+ registrant

0..*
1

0..*
alternateCourses
0..2

0..*
+ primaryCourses
0..4

Student

- name
- address
- StudentID : int

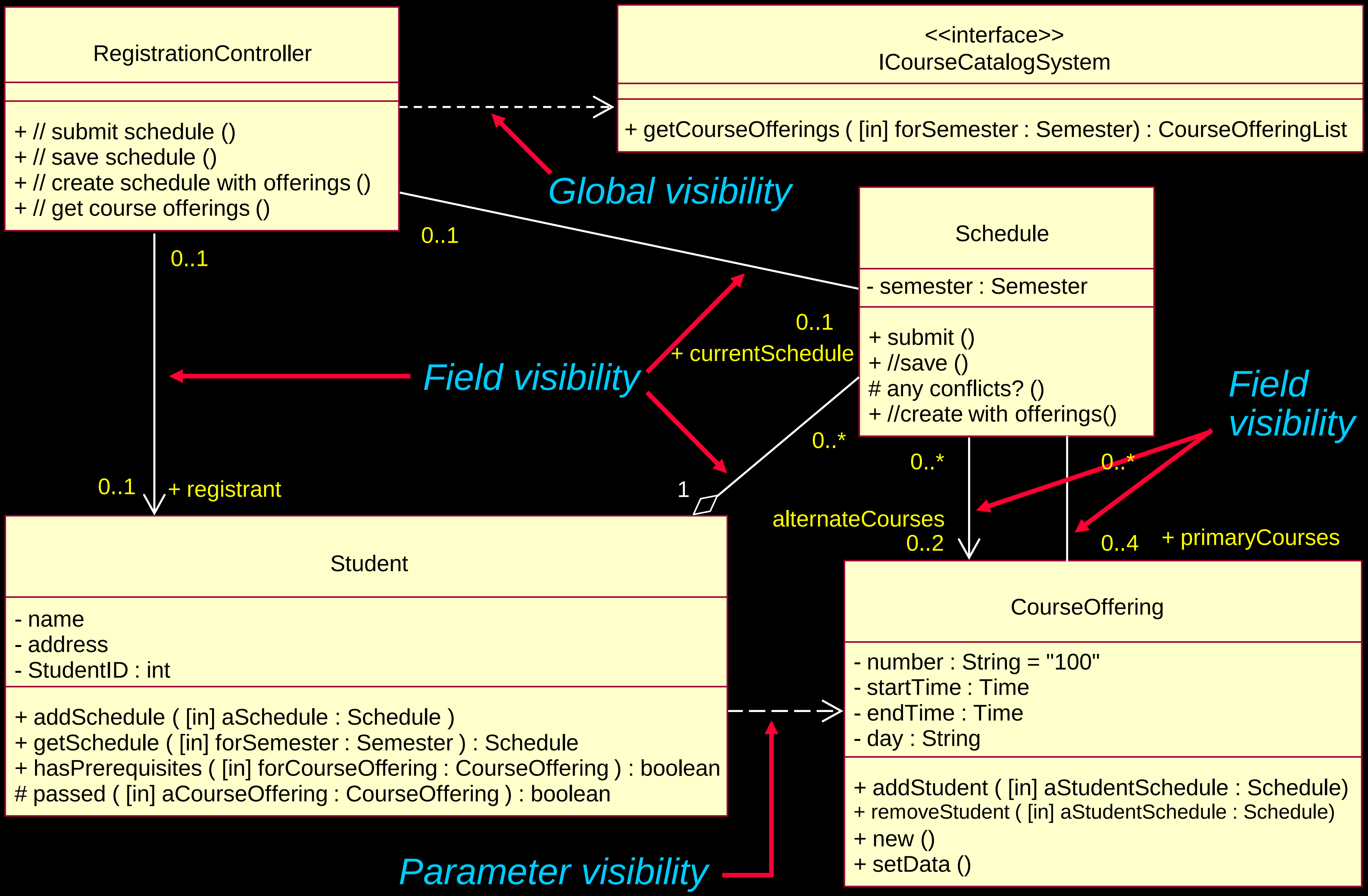
+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean

CourseOffering

- number : String = "100"
- startTime : Time
- endTime : Time
- day : String

+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()

Example: Define Dependencies (after)



RegistrationController

+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()

<<interface>>
ICourseCatalogSystem

+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList

Schedule

- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()

Student

- name
- address
- StudentID : int

+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean

CourseOffering

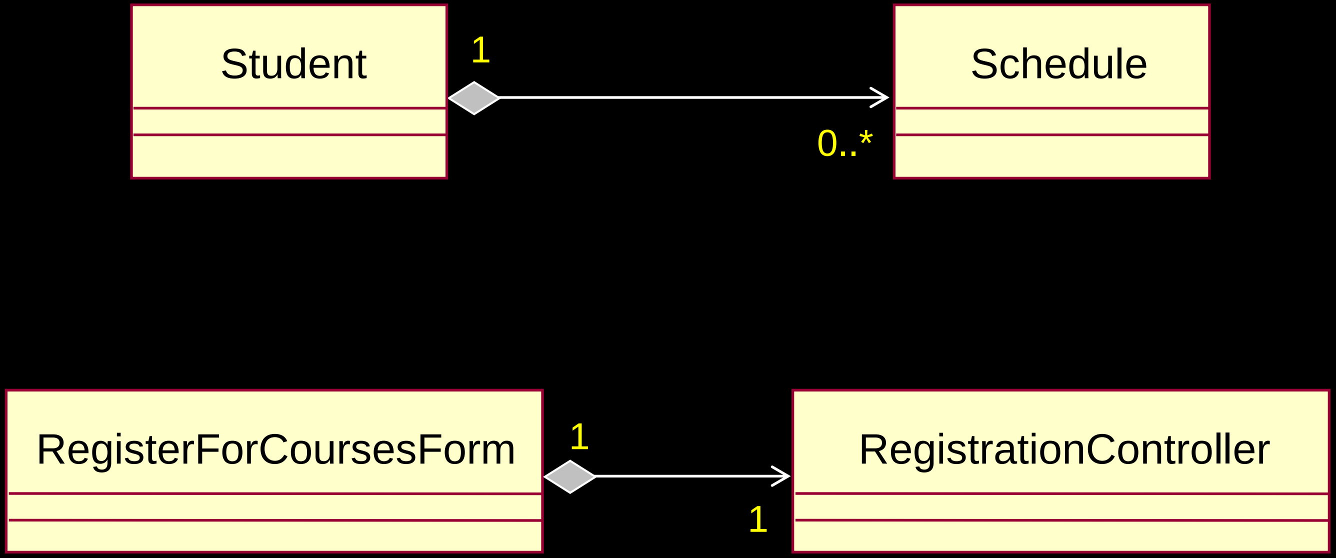
- number : String = "100"
- startTime : Time
- endTime : Time
- day : String

+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()

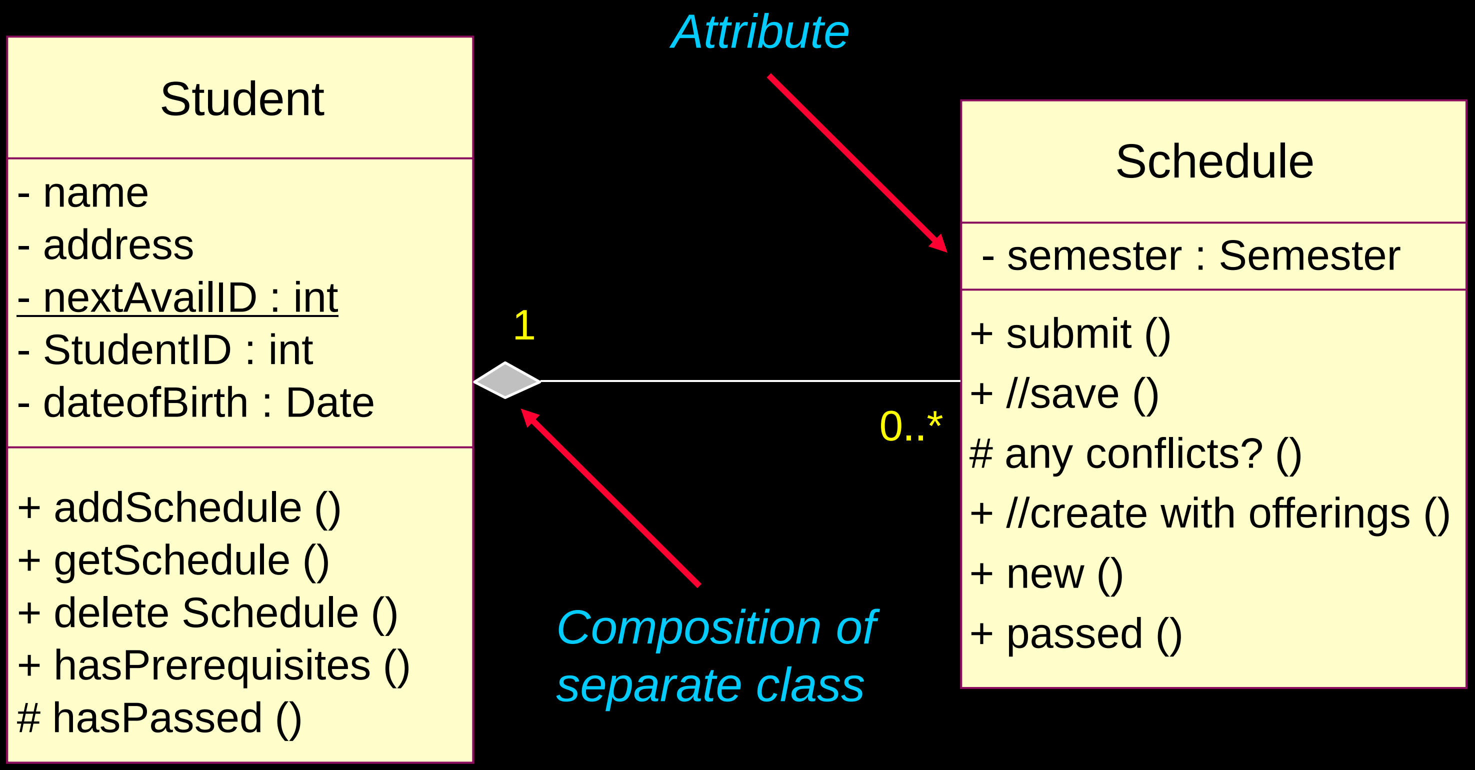
Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- ★ Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Example: Composition



Example: Attributes vs. Composition



Student

- name
- address
- nextAvailID : int
- StudentID : int
- dateofBirth : Date

+ addSchedule ()
+ getSchedule ()
+ delete Schedule ()
+ hasPrerequisites ()
# hasPassed ()

1

0..*

Attribute

Schedule

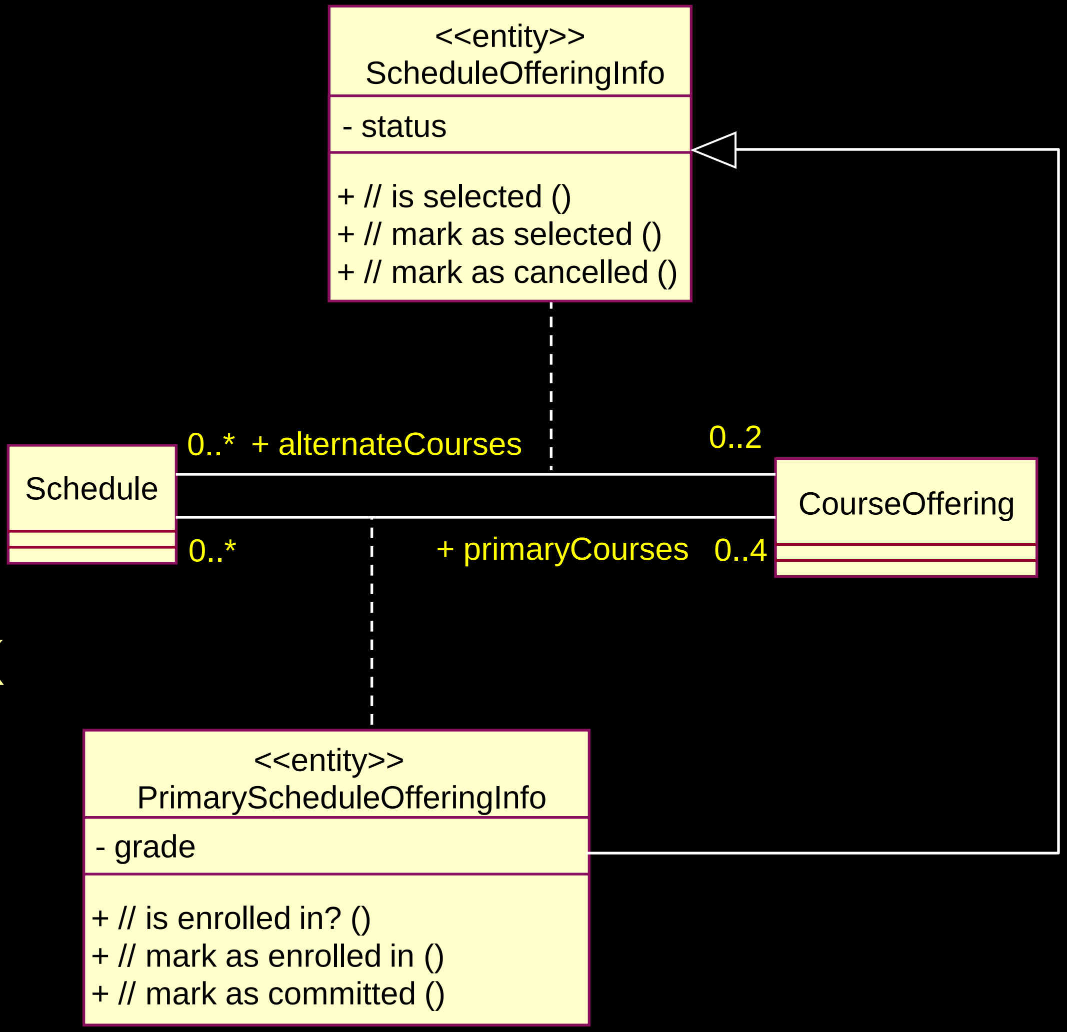
- semester : Semester

+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings ()
+ new ()
+ passed ()

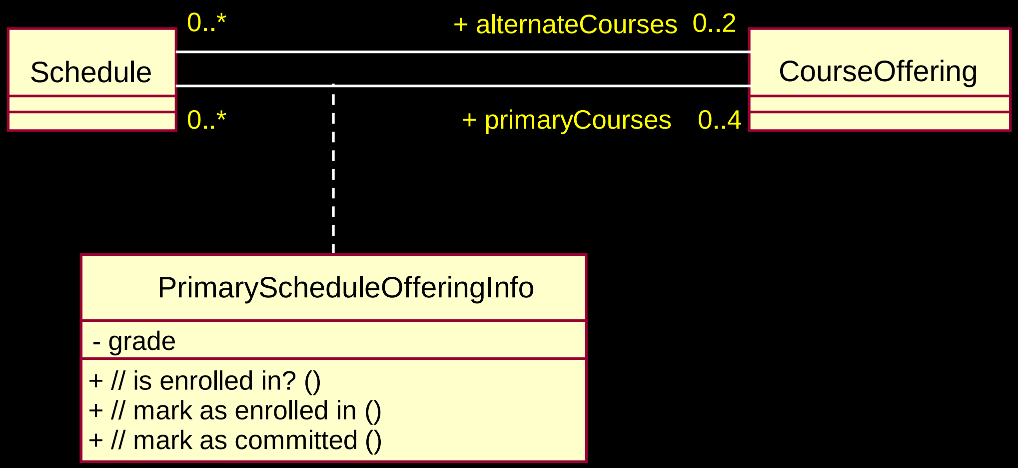
Composition of separate class

Association Class

- A class is "attached" to an association
- Contains properties of a relationship
- Has one instance per link



Example: Association Class Design



Schedule

0..*

+ alternateCourses 0..2

CourseOffering

0..*

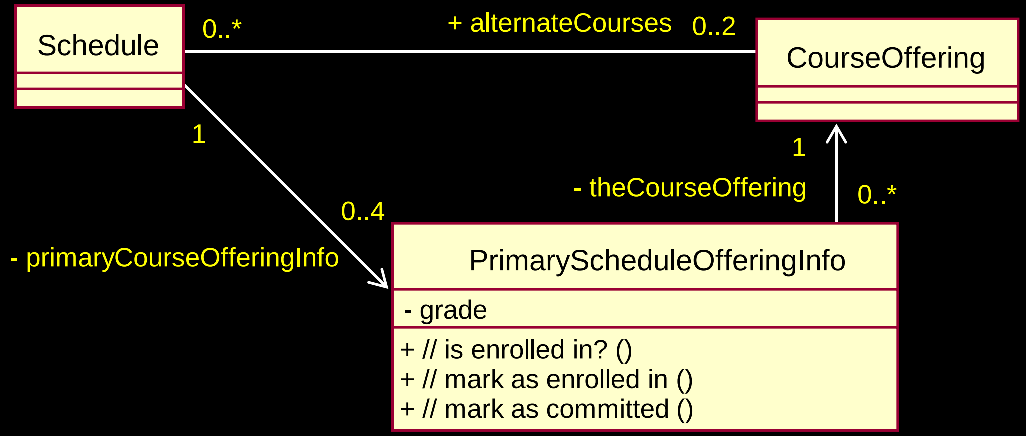
+ primaryCourses 0..4

PrimaryScheduleOfferingInfo

- grade

+ // is enrolled in? ()
+ // mark as enrolled in ()
+ // mark as committed ()

Design Decisions



Schedule

0..*

+ alternateCourses 0..2

CourseOffering

1

0..4

- primaryCourseOfferingInfo

1

- theCourseOffering

0..*

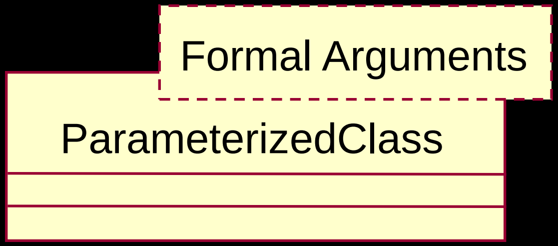
PrimaryScheduleOfferingInfo

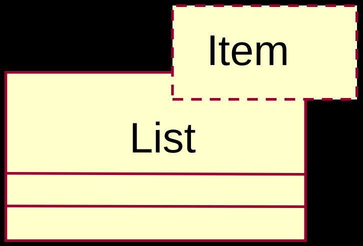
- grade

+ // is enrolled in? ()
+ // mark as enrolled in ()
+ // mark as committed ()

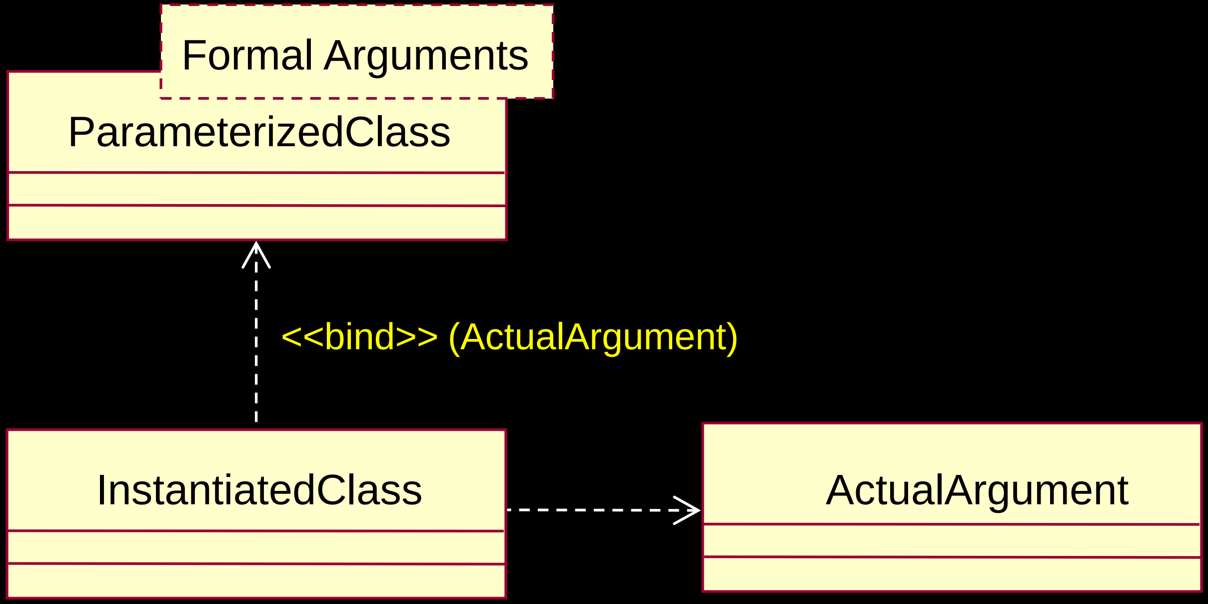
What Is a Parameterized Class (Template)?

- A class definition that defines other classes
- Often used for container classes
  - Some common container classes:
    - Sets, lists, dictionaries, stacks, queues



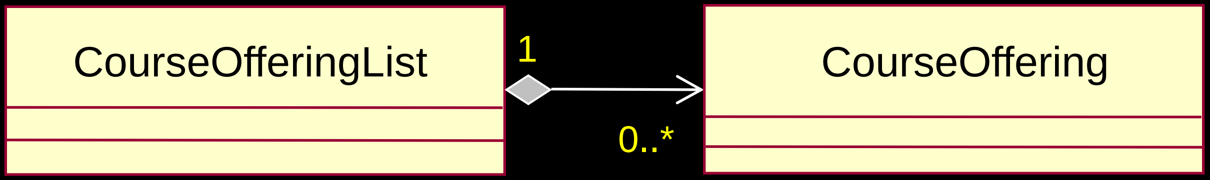


Instantiating a Parameterized Class

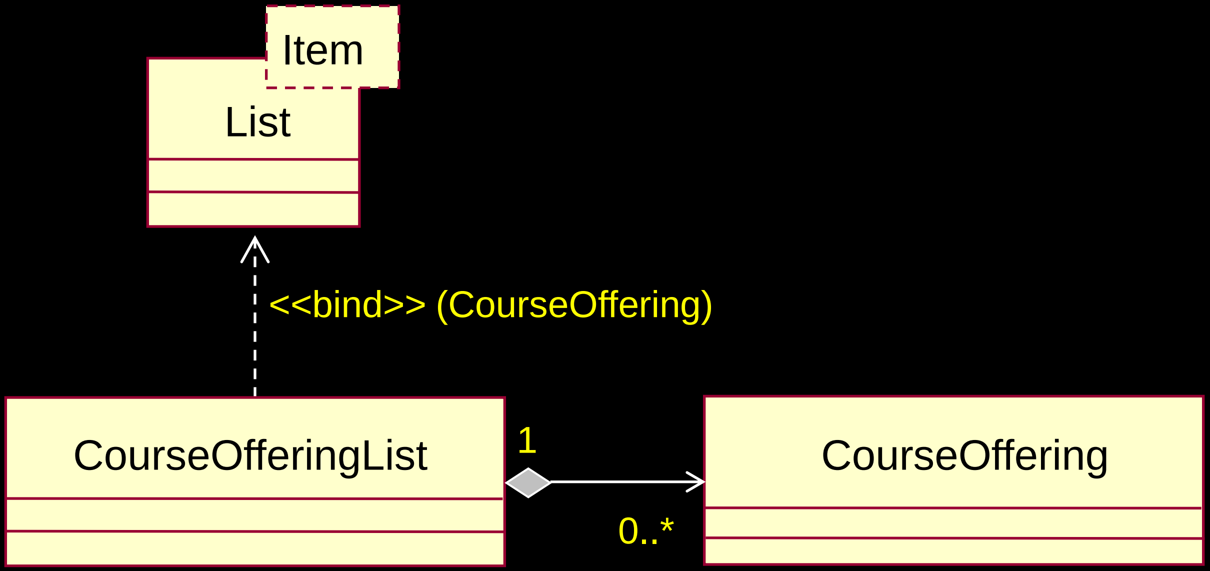


Example: Instantiating a Parameterized Class

Before

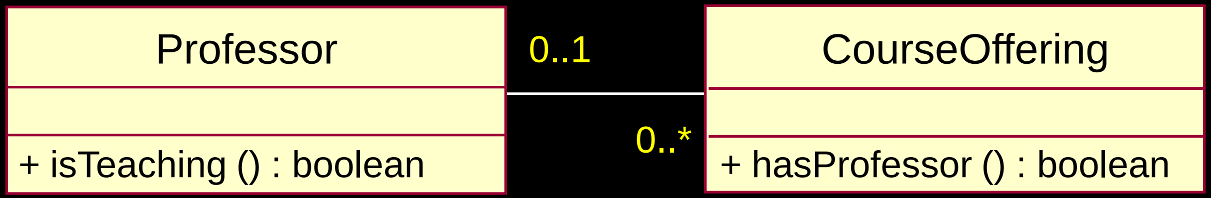


After



Multiplicity Design: Optionality

- If a link is optional, make sure to include an operation to test for the existence of the link



Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- ★ Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

What is Internal Structure?

- The interconnected parts and connectors that compose the contents of a structured class.
  - It contains parts or roles that form its structure and realize its behavior.
  - Connectors model the communication link between interconnected parts.

The interfaces describe what a class must do; its internal structure describes how the work is accomplished.

Review: What Is a Structured Class?

- A structured class contains parts or roles that form its structure and realize its behavior
  - Describes the internal implementation structure
- The parts themselves may also be structured classes
  - Allows hierarchical structure to permit a clear expression of multilevel models.
- A connector is used to represent an association in a particular context
  - Represents communications paths among parts

What Is a Connector?

- A connector models the communication link between interconnected parts. For example:
  - Assembly connectors
    - Reside between two elements (parts or ports) in the internal implementation specification of a structured class.
  - Delegation connectors
    - Reside between an external (relay) port and an internal part in the internal implementation specification of a structured class.

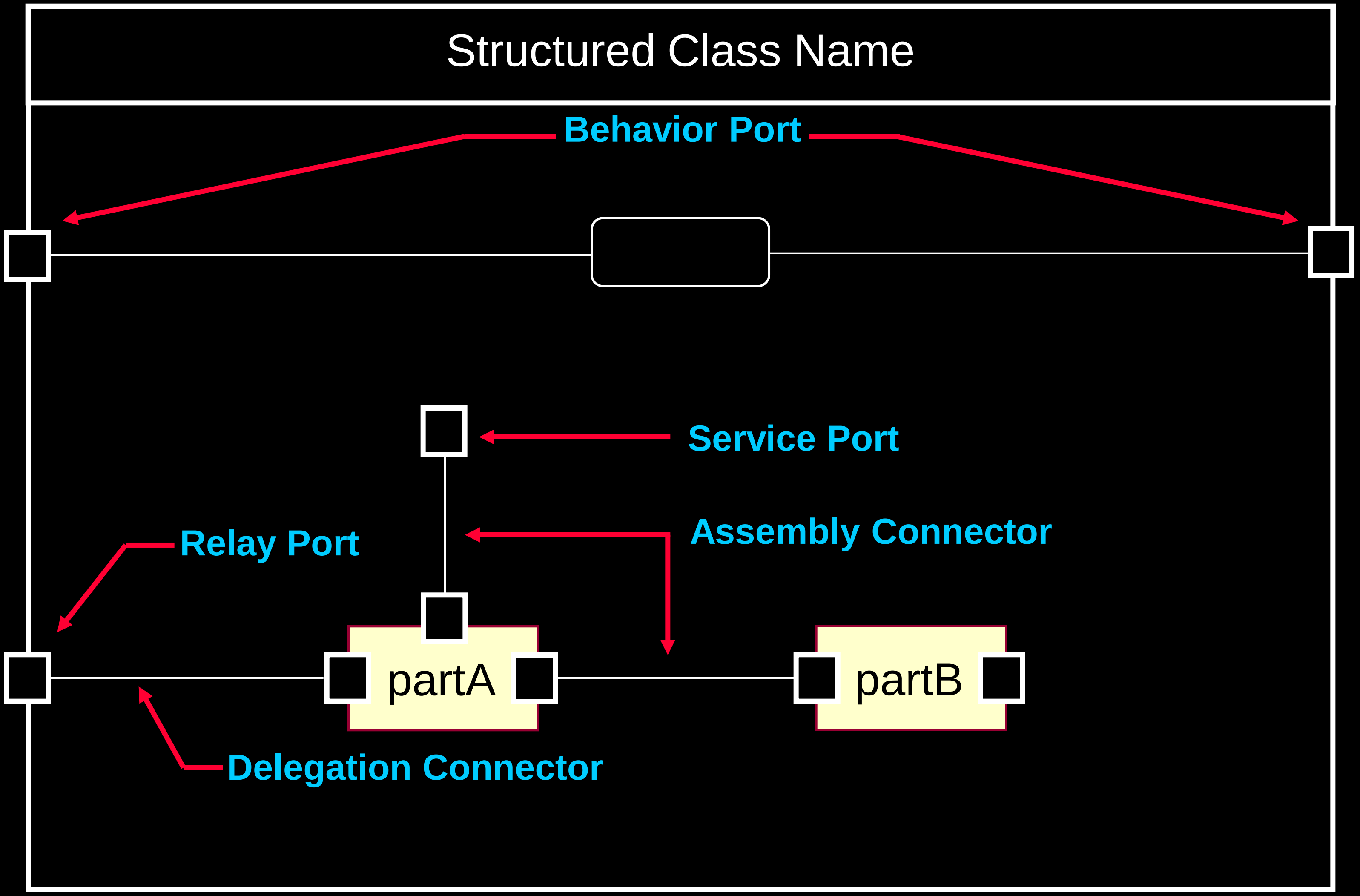
Review: What Is a Port?

- A port is a structural feature that encapsulates the interaction between the contents of a class and its environment.
  - Port behavior is specified by its provided and required interfaces
    - They permit the internal structure to be modified without affecting external clients
      - External clients have no visibility to internals
- A class may have a number of ports
  - Each port has a set of provided and required interfaces

Review: Port Types

- Ports can have different implementation types
  - Service ports are only used for the internal implementation of the class.
  - Behavior ports are used where requests on the port are implemented directly by the class.
  - Relay ports are used where requests on the port are transmitted to internal parts for implementation.

Review: Structure Diagram With Ports


Structured Class Name

Behavior Port

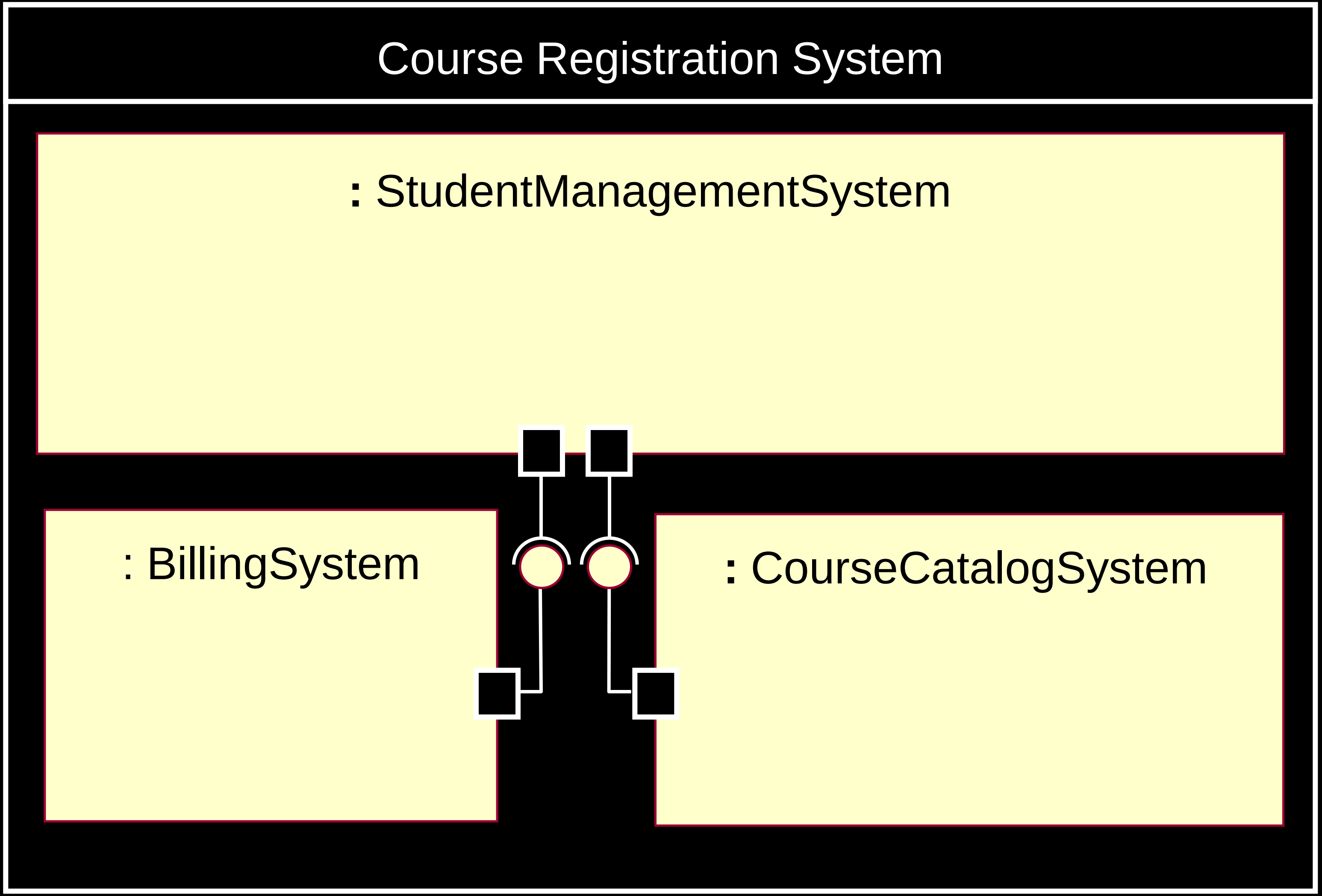
Service Port

Assembly Connector

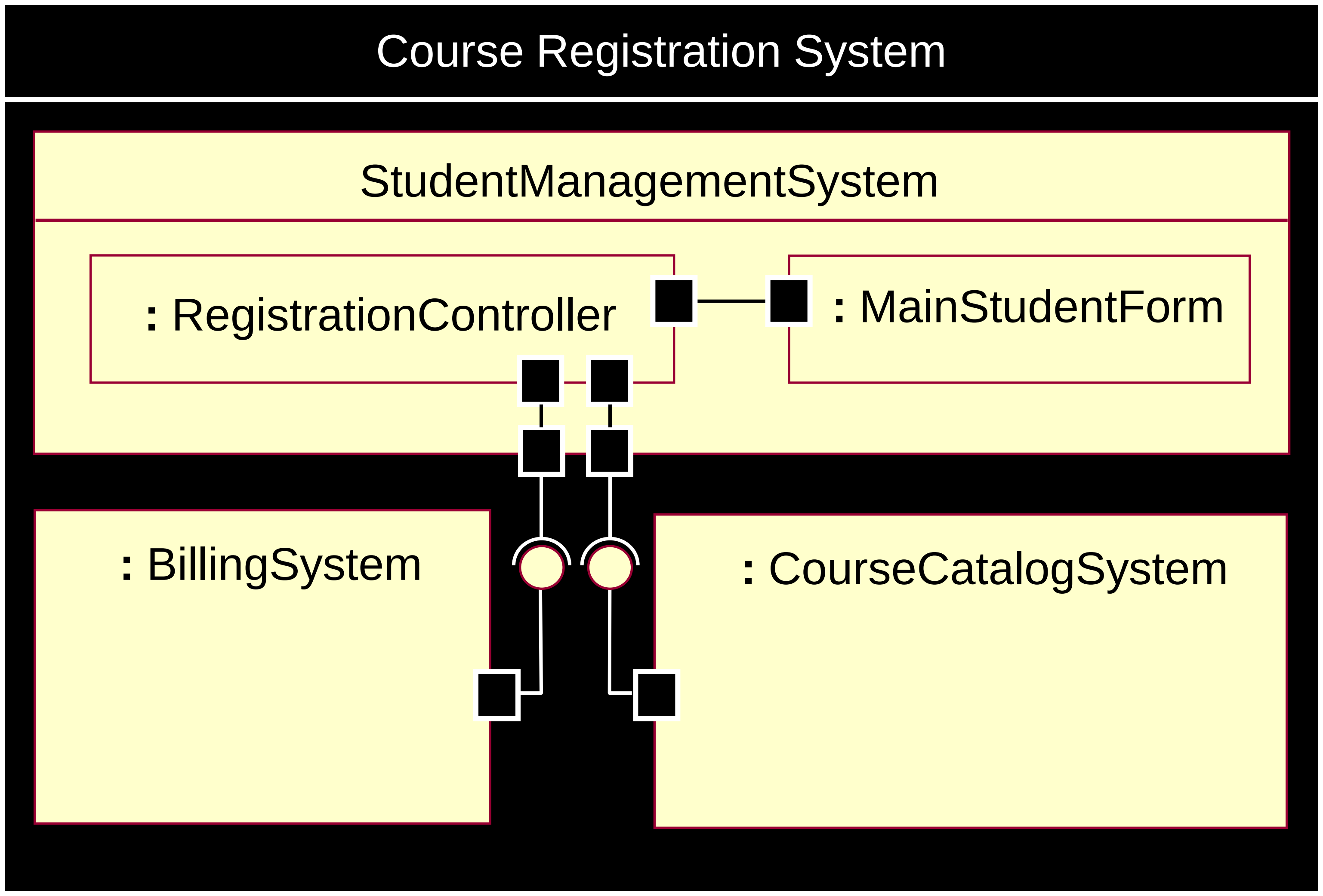
Relay Port

Delegation Connector

Review: Structure Diagram



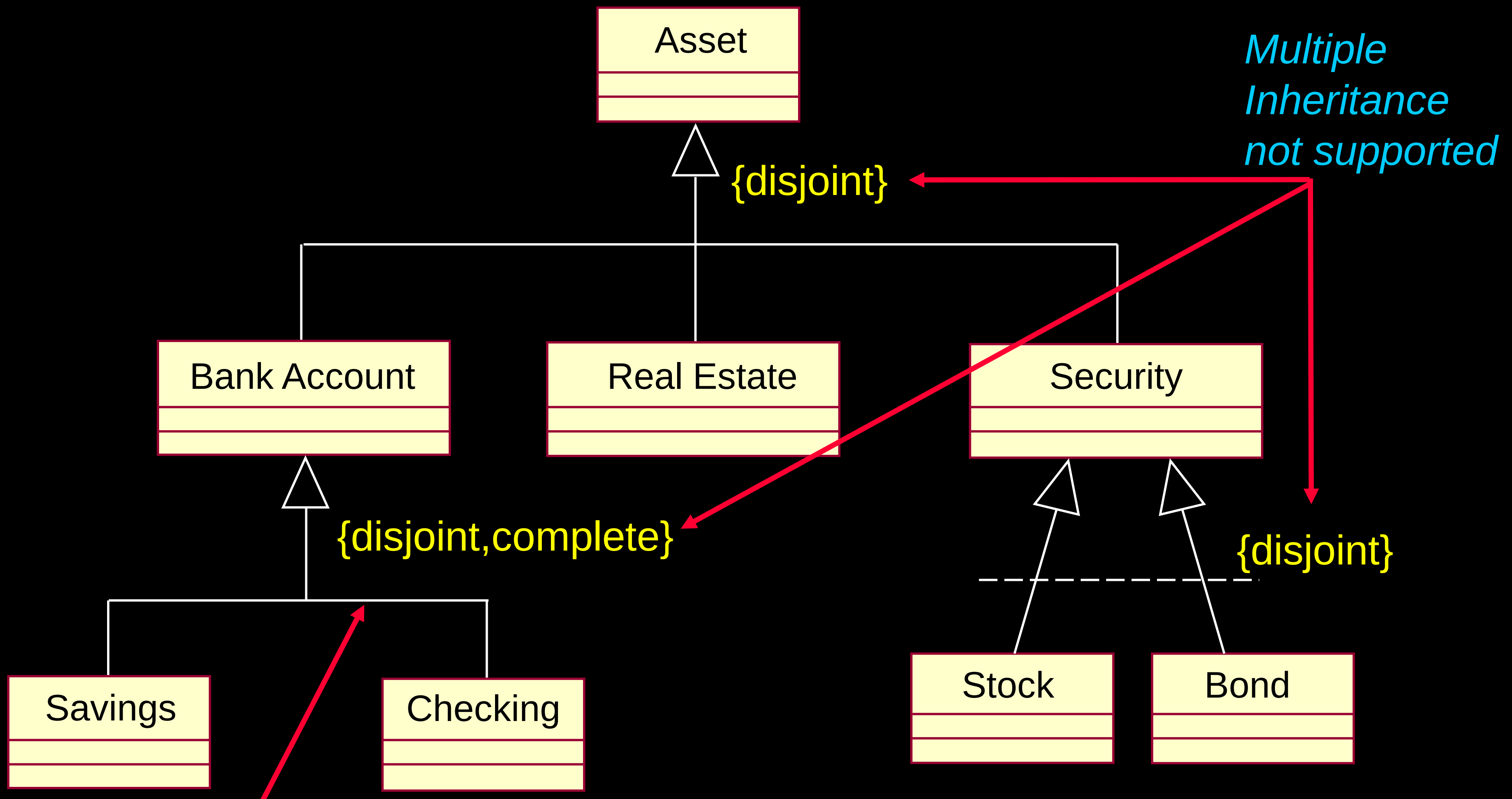
Example: Structure Diagram Detailed



Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- ★ Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

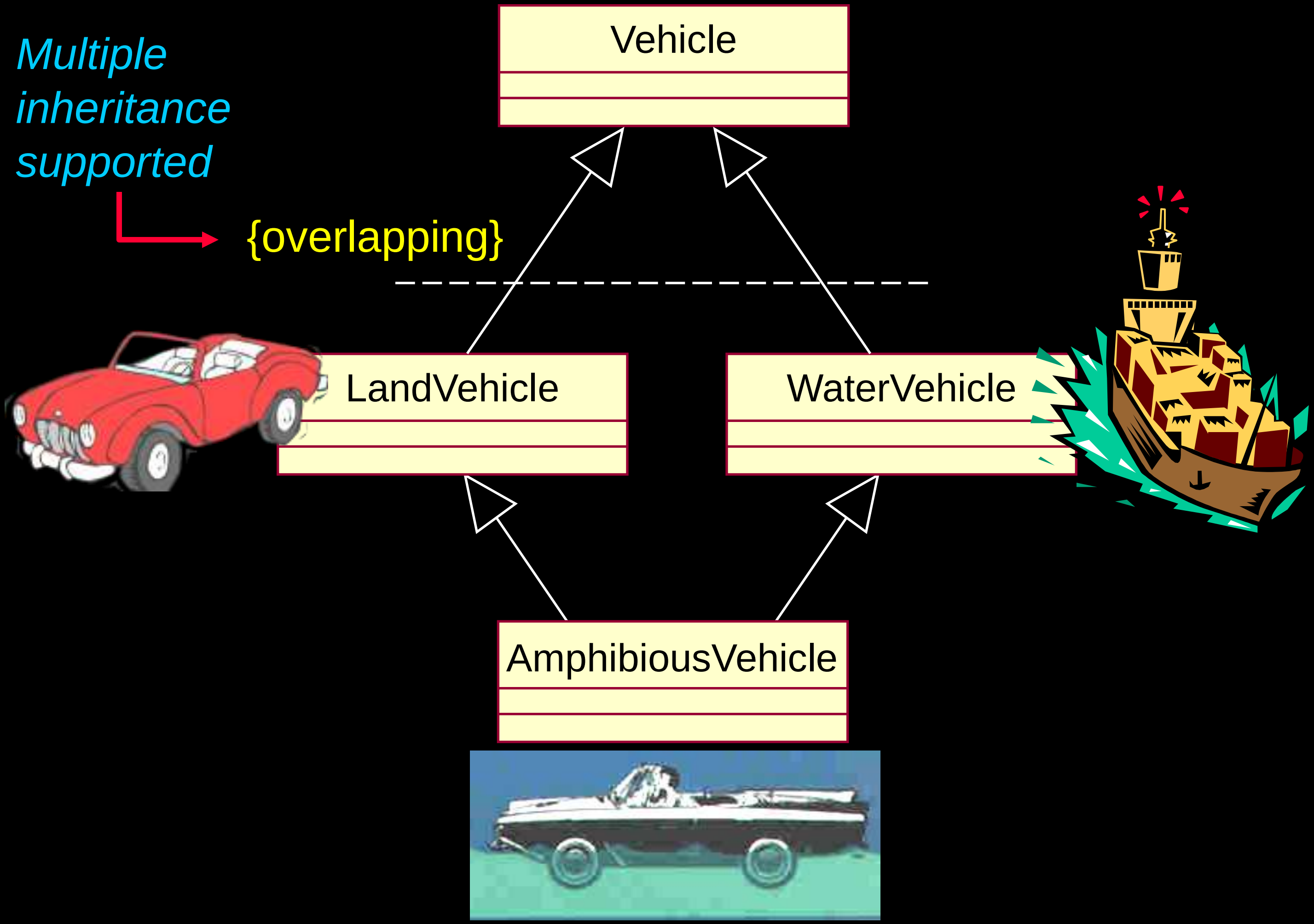
Example: Generalization Constraints



Multiple Inheritance not supported

End of inheritance hierarchy

Example: Generalization Constraints (continued)

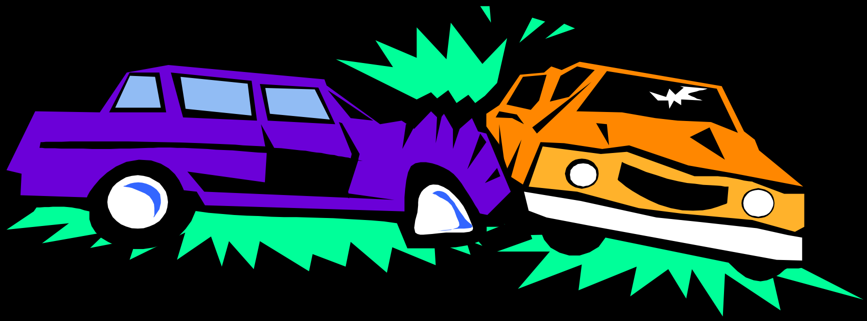


Multiple inheritance supported

{overlapping}

Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- ★ Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Resolve Use-Case Collisions

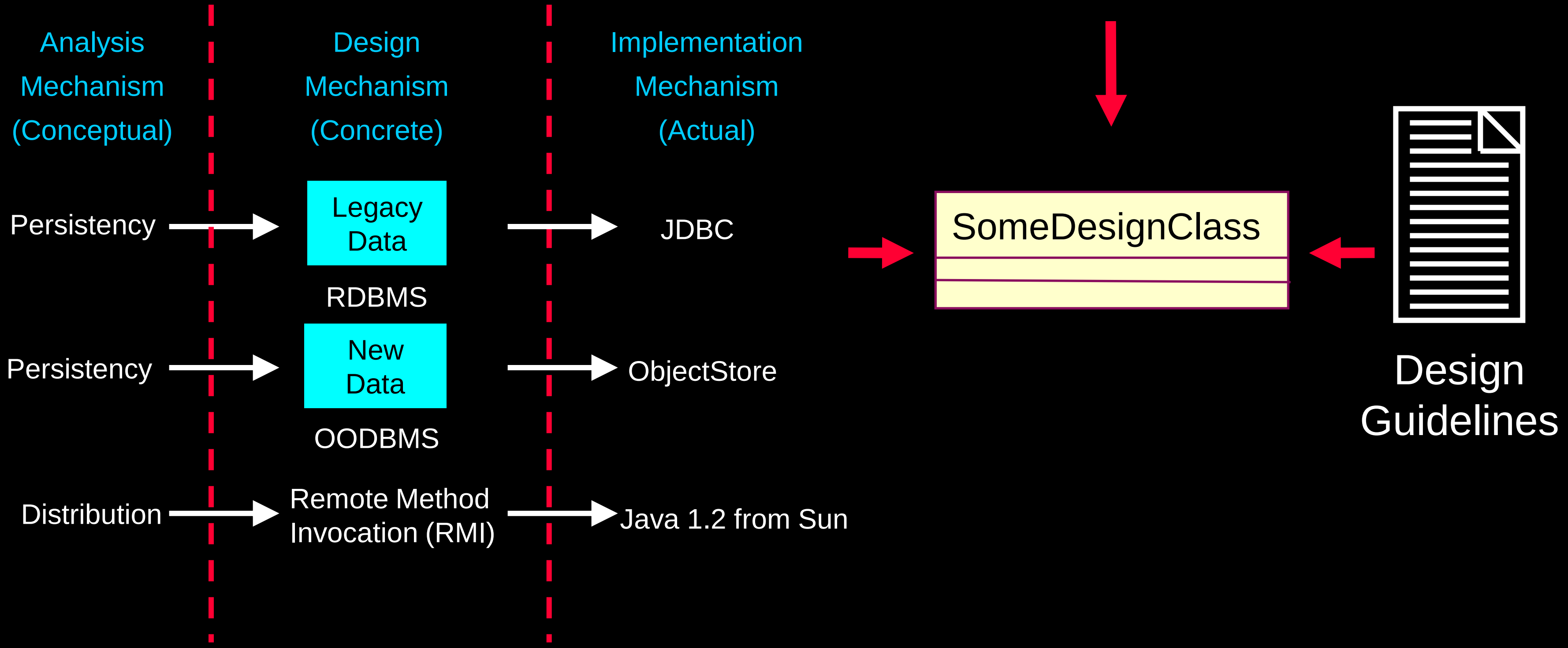
- Multiple use cases may simultaneously access design objects
- Options
  - Use synchronous messaging => first-come first-serve order processing
  - Identify operations (or code) to protect
  - Apply access control mechanisms
    - Message queuing
    - Semaphores (or “tokens”)
    - Other locking mechanism
- Resolution is highly dependent on implementation environment

Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- ★ Handle Non-Functional Requirements in General
- Checkpoints

Handle Non-Functional Requirements in General

Analysis Class	**Analysis Mechanism(s)**
Student	Persistency, Security
Schedule	Persistency, Security
CourseOffering	Persistency, Legacy Interface
Course	Persistency, Legacy Interface
RegistrationController	Distribution



Analysis Mechanism (Conceptual)	Design Mechanism (Concrete)	Implementation Mechanism (Actual)
Persistency	Legacy Data RDBMS	JDBC
Persistency	New Data OODBMS	ObjectStore
Distribution	Remote Method Invocation (RMI)	Java 1.2 from Sun

SomeDesignClass

Design Guidelines