IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0

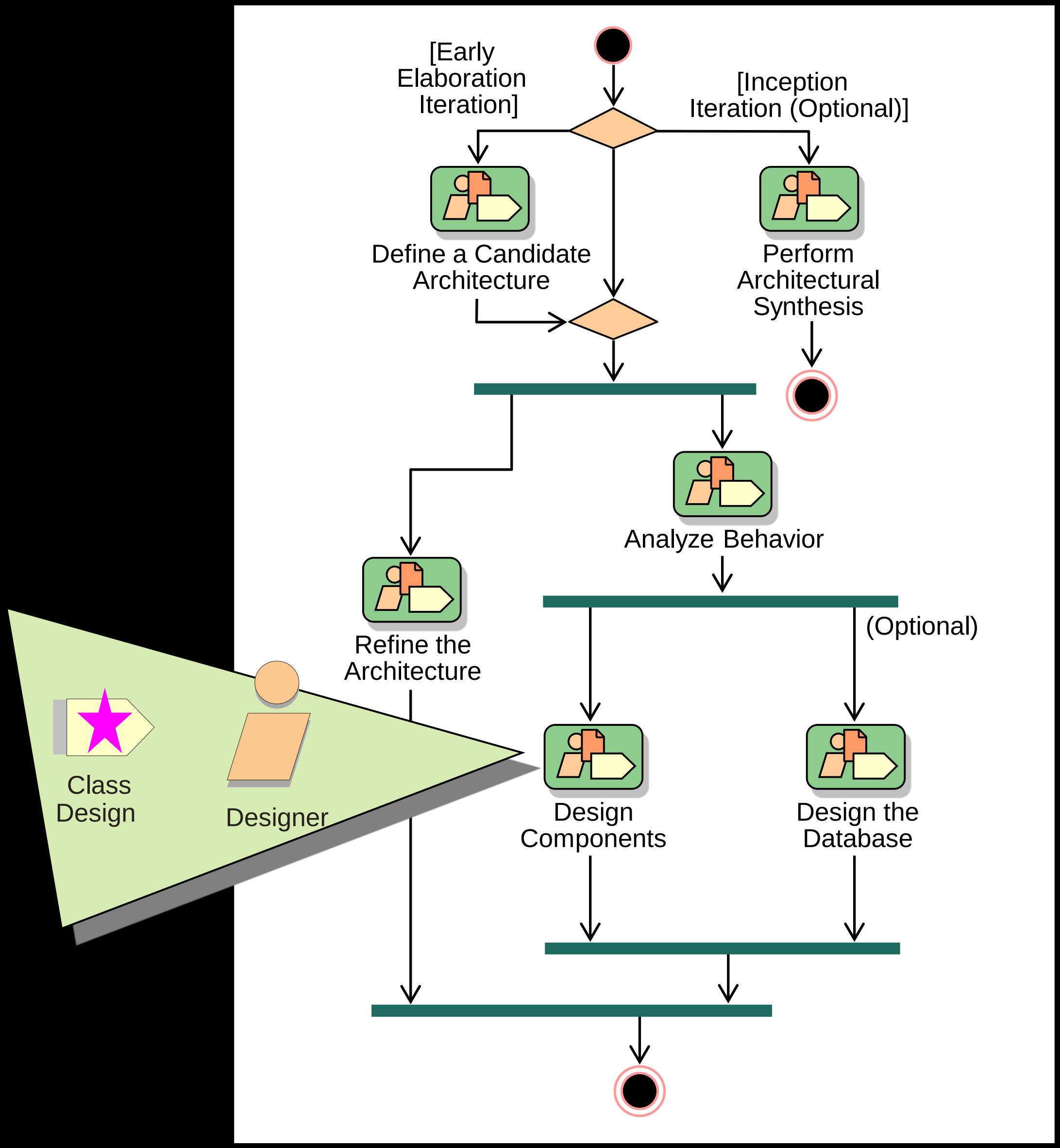
Module 13: Class Design

Rational software

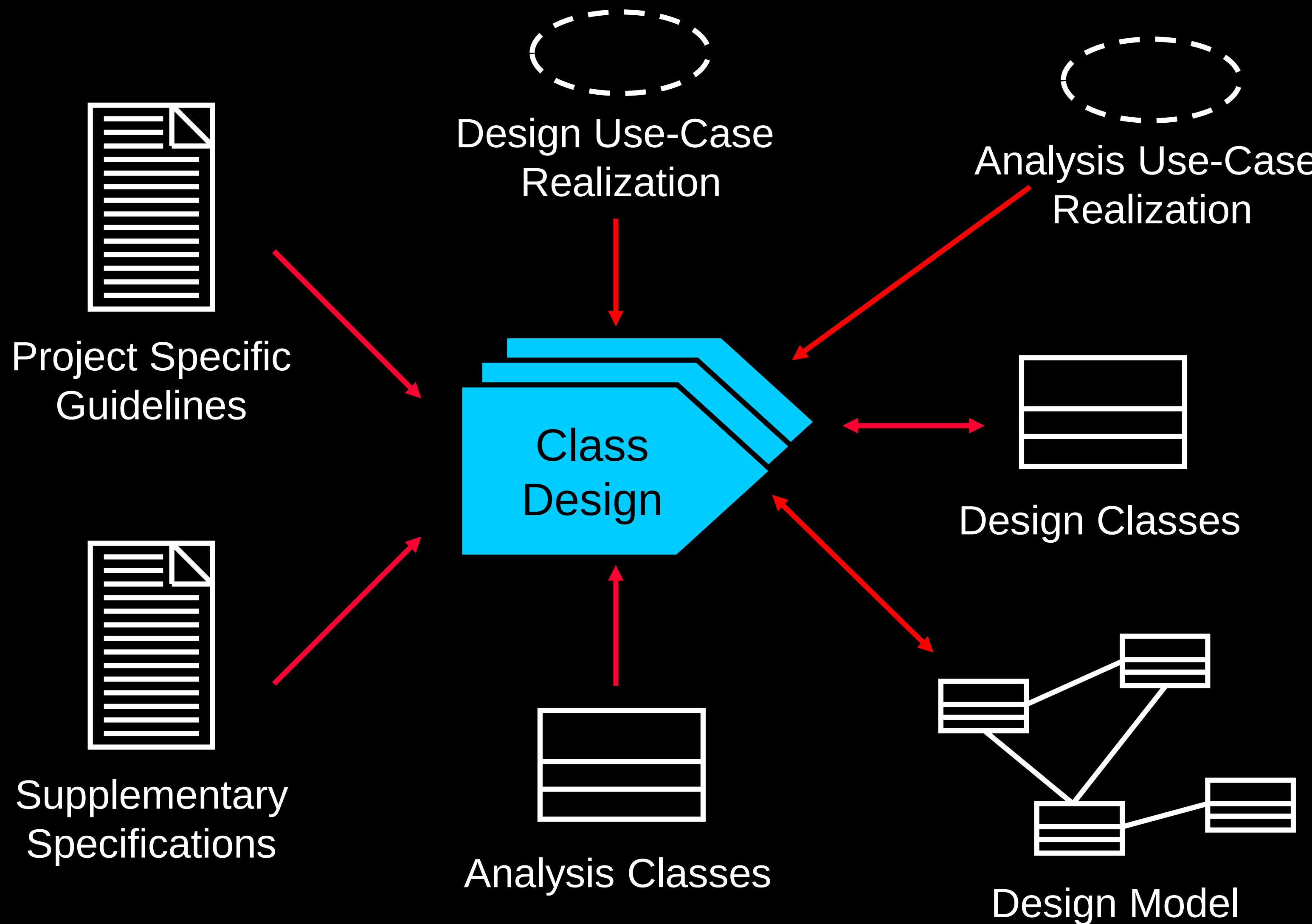
Objectives: Class Design

- Define the purpose of Class Design and where in the lifecycle it is performed
- Identify additional classes and relationships needed to support implementation of the chosen architectural mechanisms
- Identify and analyze state transitions in objects of state-controlled classes
- Refine relationships, operations, and attributes

Class Design in Context

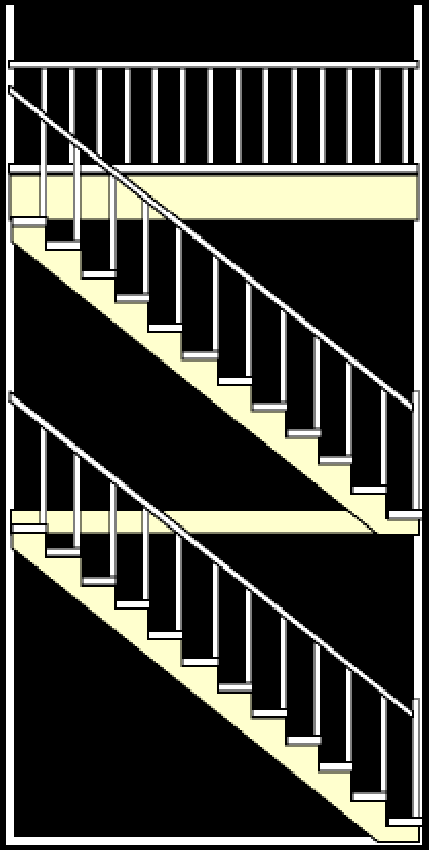


Class Design Overview



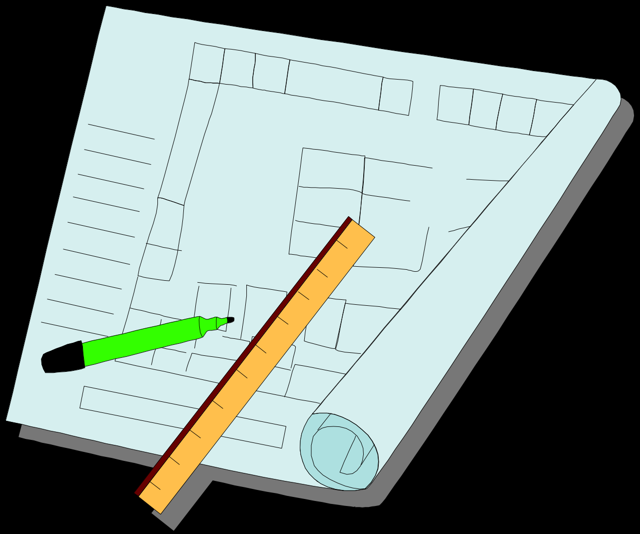
Class Design Steps

- 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



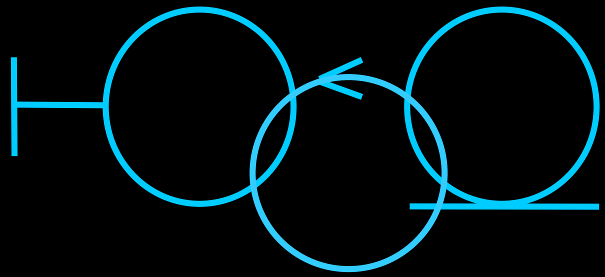
Class Design Steps

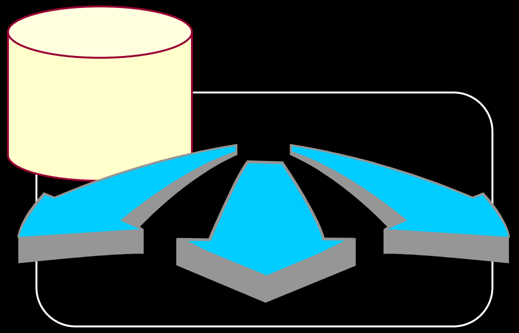
★
- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Class Design Considerations

- Class stereotype
  - Boundary
  - Entity
  - Control
- Applicable design patterns
- Architectural mechanisms
  - Persistence
  - Distribution
  - etc.





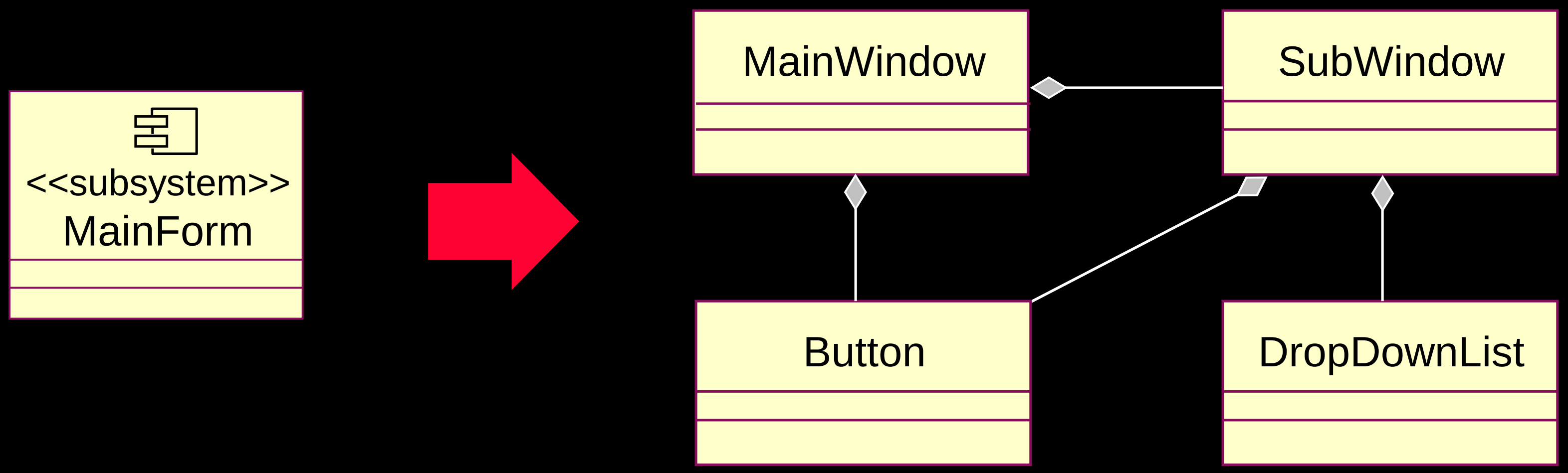
How Many Classes Are Needed?

- Many, simple classes means that each class
  - Encapsulates less of the overall system intelligence
  - Is more reusable
  - Is easier to implement
- A few, complex classes means that each class
  - Encapsulates a large portion of the overall system intelligence
  - Is less likely to be reusable
  - Is more difficult to implement

A class should have a single well-focused purpose.
A class should do one thing and do it well!

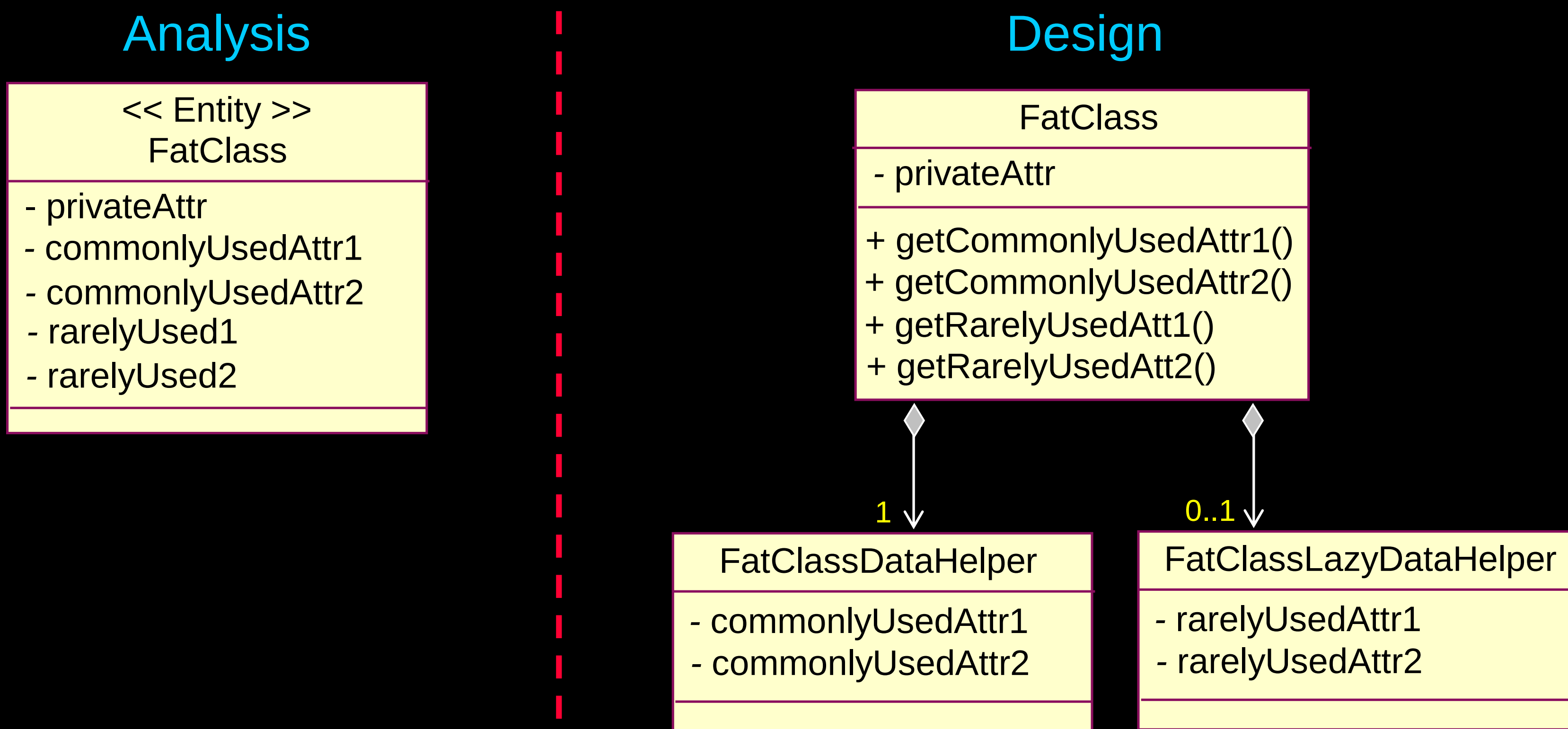
Strategies for Designing Boundary Classes

- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
- External system interface boundary classes
  - Usually model as subsystem



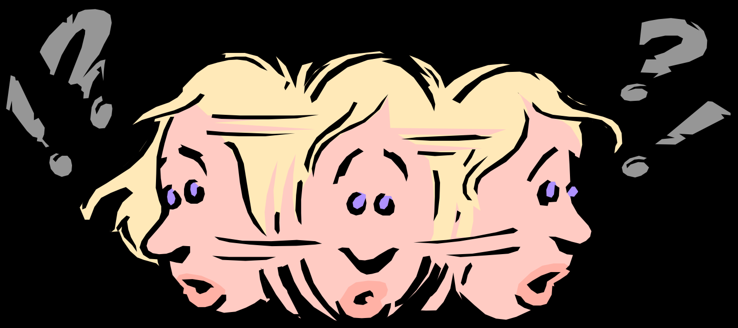
Strategies for Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring



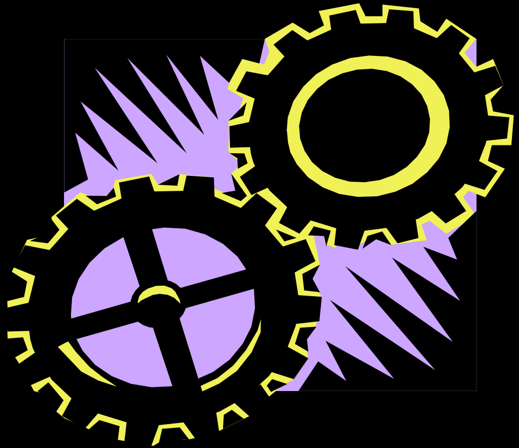
Strategies for Designing Control Classes

- What happens to Control Classes?
  - Are they really needed?
  - Should they be split?
- How do you decide?
  - Complexity
  - Change probability
  - Distribution and performance
  - Transaction management



Class Design Steps

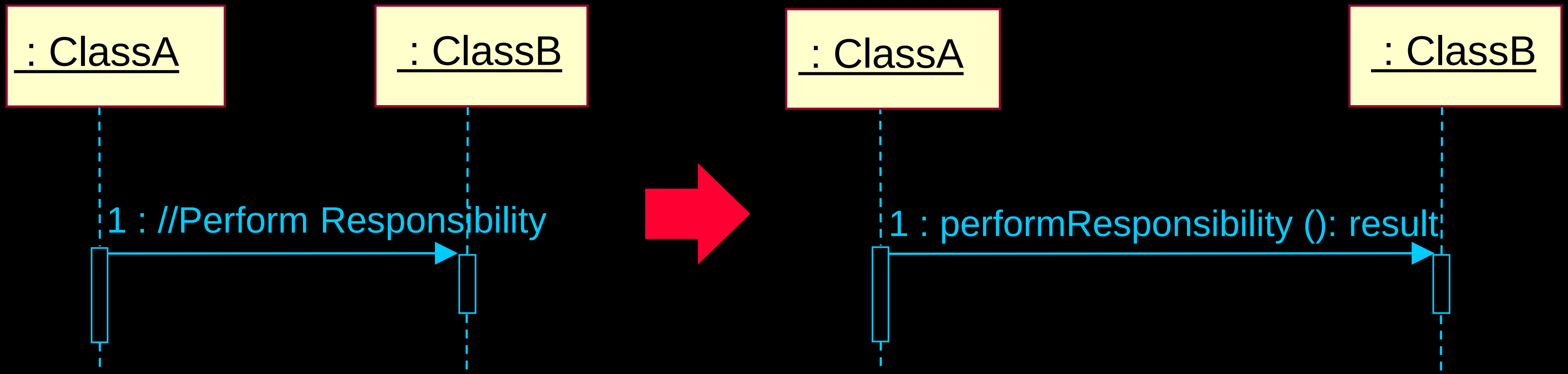
- Create Initial Design Classes
- ★ Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



IBM

Operations: Where Do You Find Them?

- Messages displayed in interaction diagrams



- Other implementation dependent functionality
  - Manager functions
  - Need for class copies
  - Need to test for equality

Name and Describe the Operations

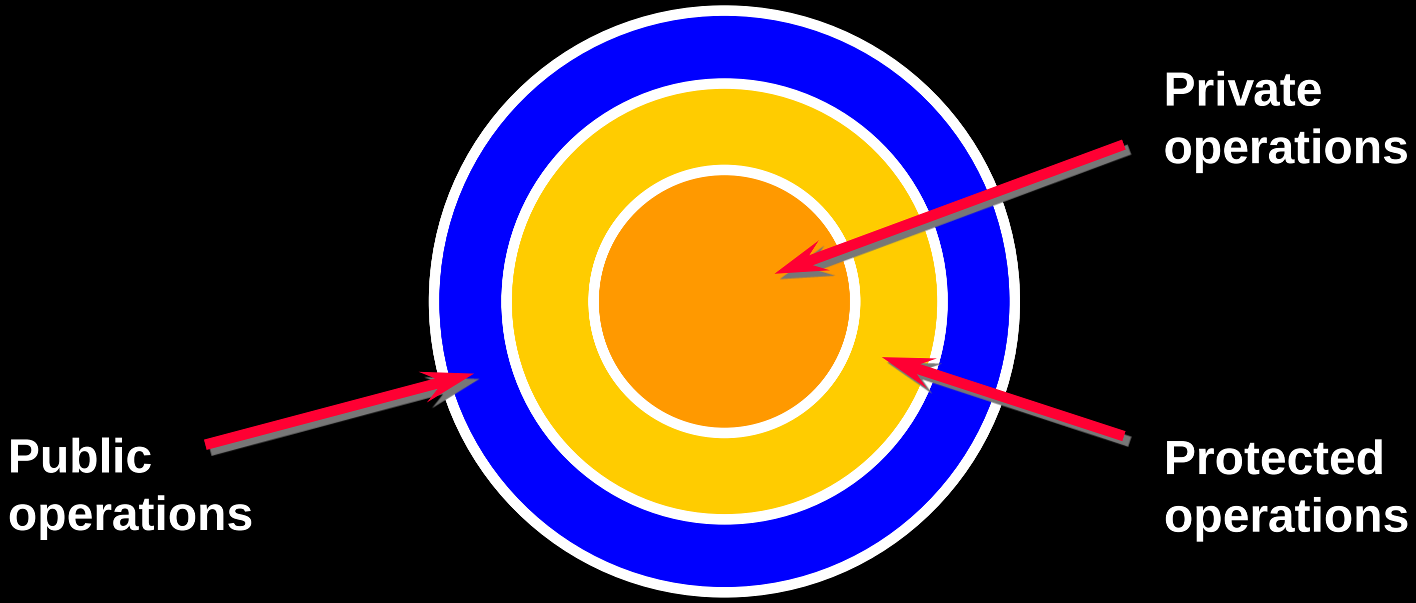
- Create appropriate operation names
  - Indicate the outcome
  - Use client perspective
  - Are consistent across classes
- Define operation signatures
  - operationName([direction]parameter : class,..) : returnType
    - Direction is **in** (default), **out** or **inout**
    - Provide short description, including meaning of all parameters

Guidelines: Designing Operation Signatures

- When designing operation signatures, consider if parameters are:
  - Passed by value or by reference
  - Changed by the operation
  - Optional
  - Set to default values
  - In valid parameter ranges
- The fewer the parameters, the better
- Pass objects instead of “data bits”

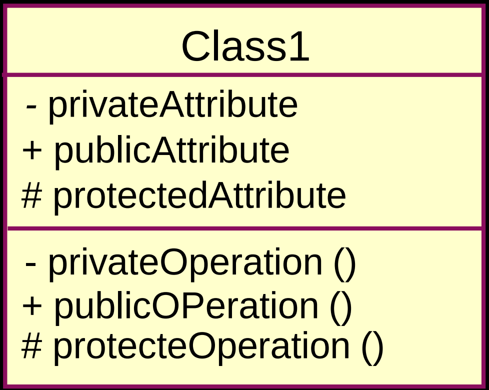
Operation Visibility

- Visibility is used to enforce encapsulation
- May be public, protected, or private



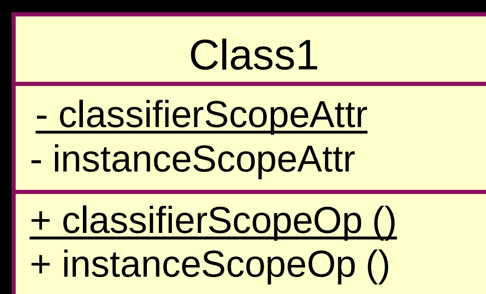
How Is Visibility Noted?

- The following symbols are used to specify export control:
  - + Public access
  - # Protected access
  - - Private access



Scope

- Determines number of instances of the attribute/operation
  - Instance: one instance for each class instance
  - Classifier: one instance for all class instances
- Classifier scope is denoted by underlining the attribute/operation name



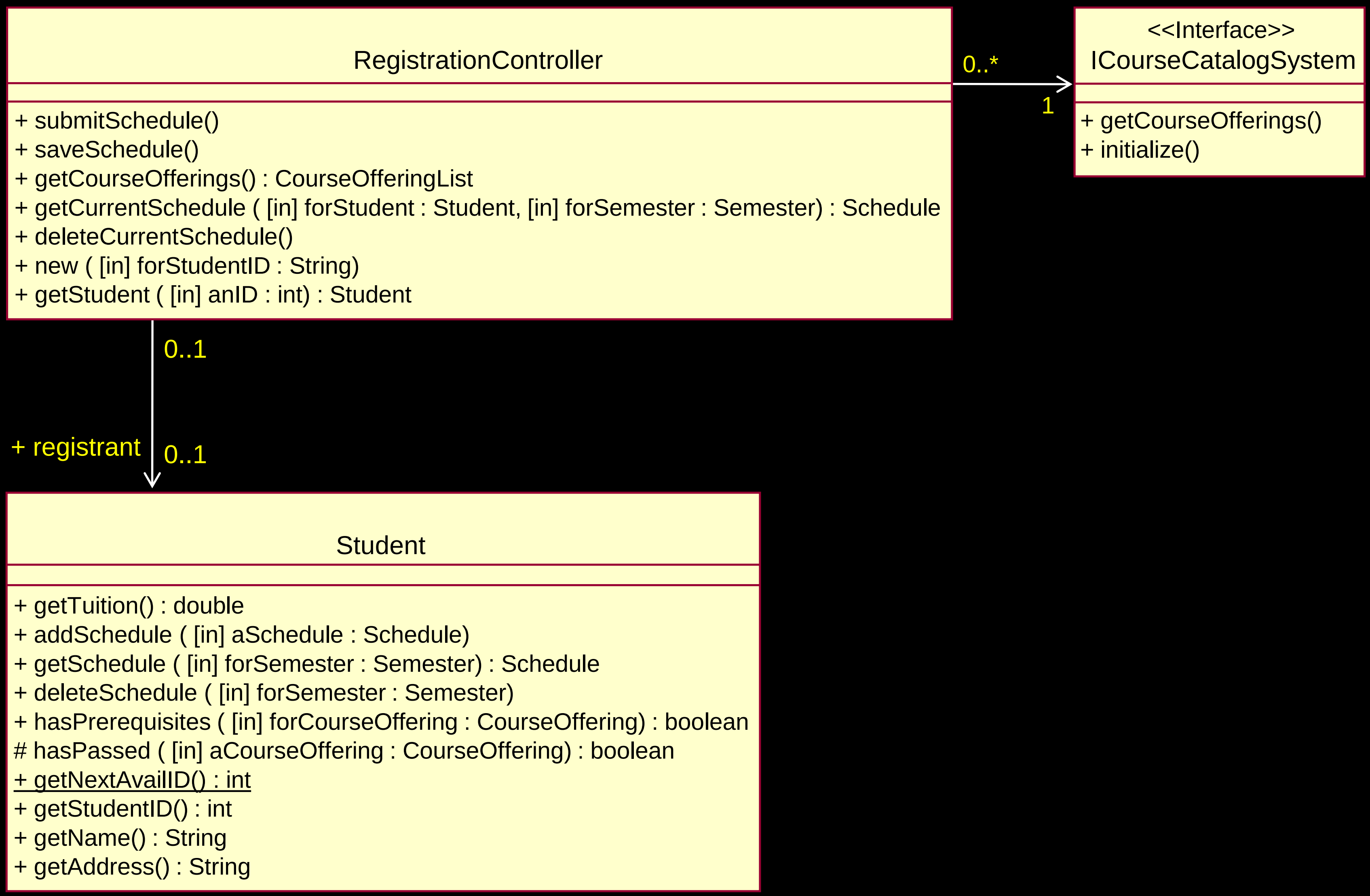
Example: Scope

Student

- name
- address
- studentID
- nextAvailID : int

+ addSchedule ([in] theSchedule : Schedule, [in] forSemester : Semester)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] theCourseOffering : CourseOffering) : boolean
+ getNextAvailID () : int

Example: Define Operations



Class Design Steps

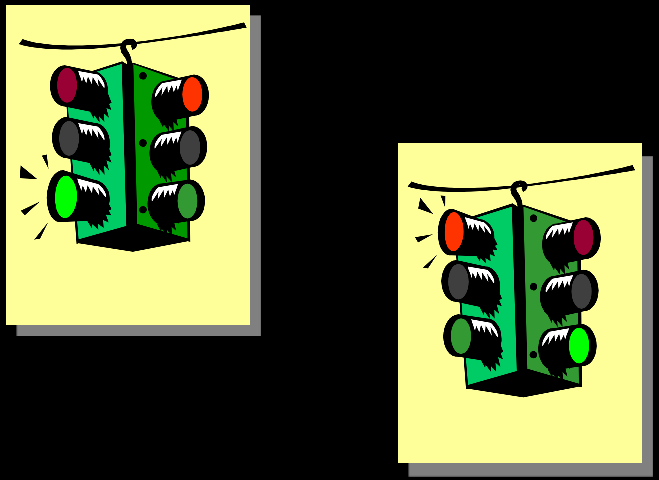
- Create Initial Design Classes
- Define Operations
- ★ Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Internal Structure
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

Define Methods

- What is a method?
  - Describes operation implementation
- Purpose
  - Define special aspects of operation implementation
- Things to consider:
  - Special algorithms
  - Other objects and operations to be used
  - How attributes and parameters are to be implemented and used
  - How relationships are to be implemented and used

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

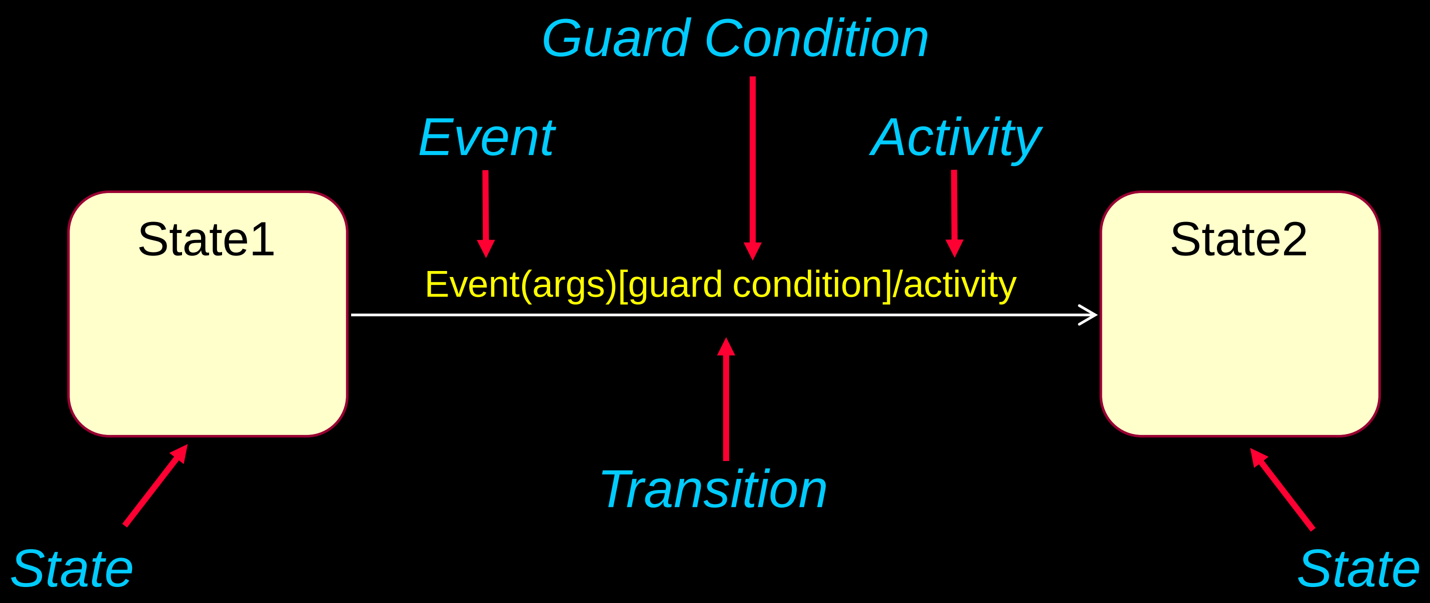


Define States

- 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?

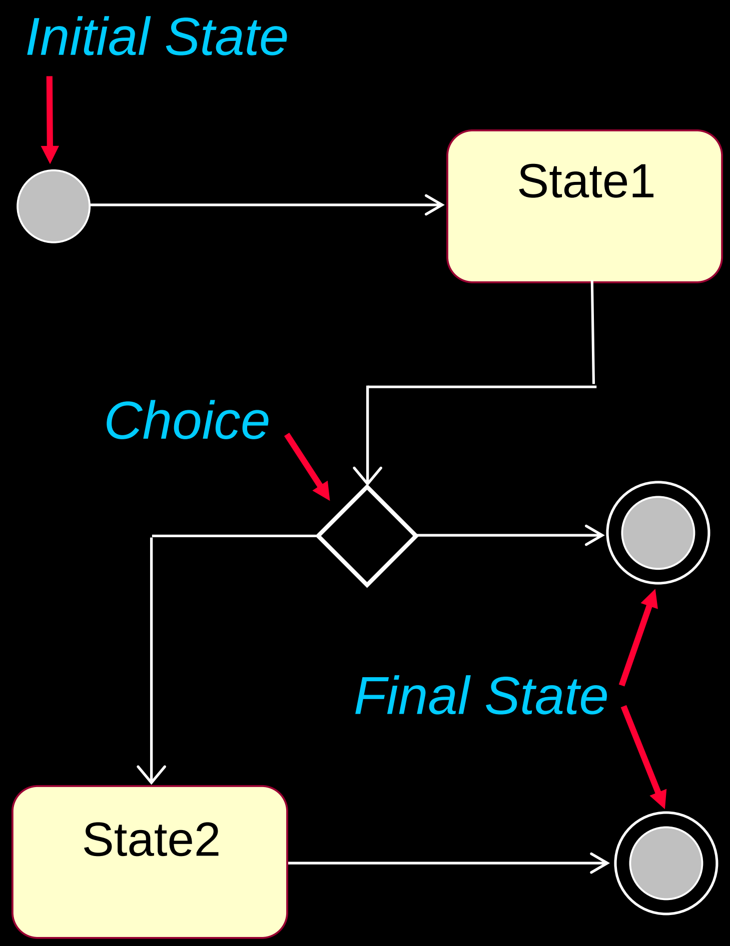
What is a State Machine?

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



Pseudo States

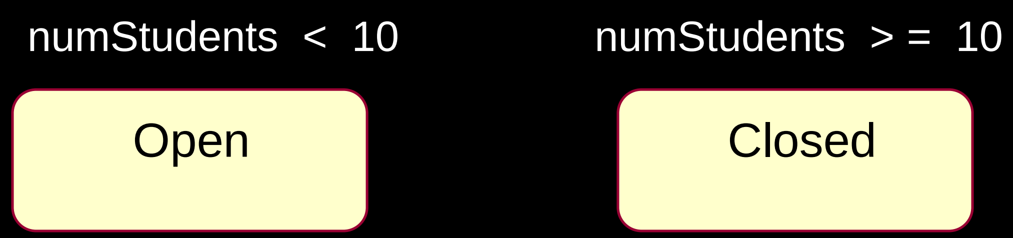
- Initial state
  - The state entered when an object is created
  - Mandatory, can only have one initial state
- Choice
  - Dynamic evaluation of subsequent guard conditions
  - Only first segment has a trigger
- Final state
  - Indicates the object's end of life
  - Optional, may have more than one



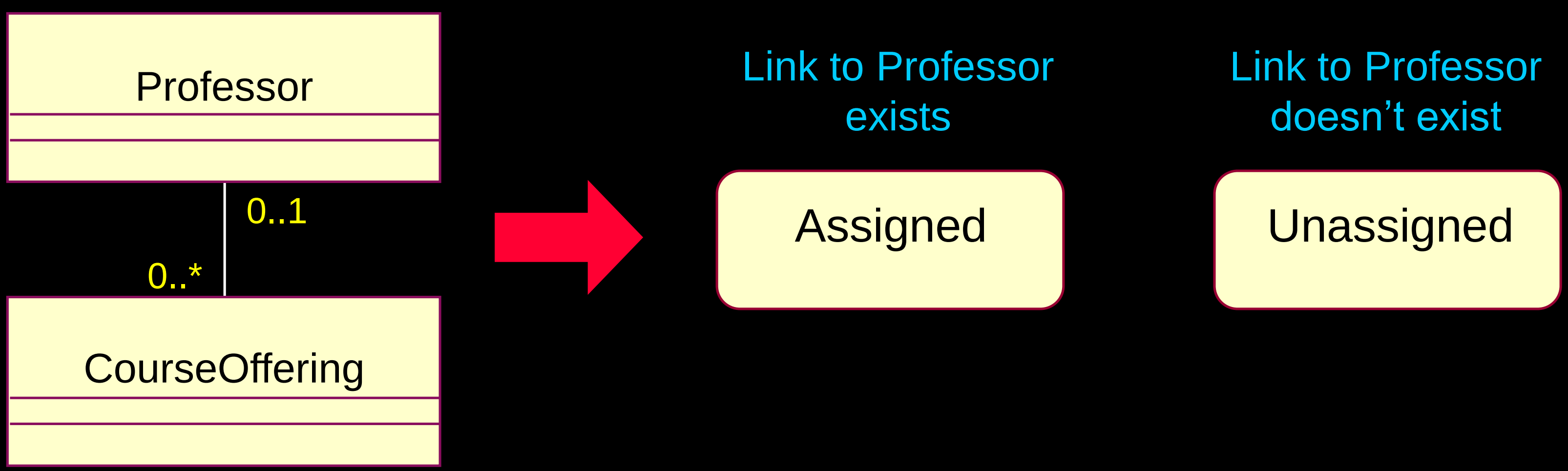
Identify and Define the States

- Significant, dynamic attributes

The maximum number of students per course offering is 10

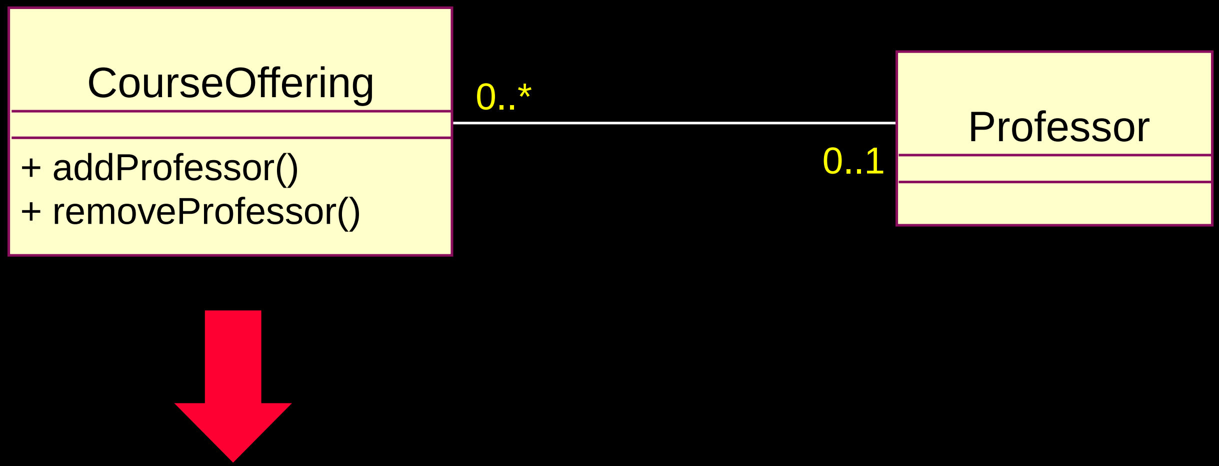


- Existence and non-existence of certain links



Identify the Events

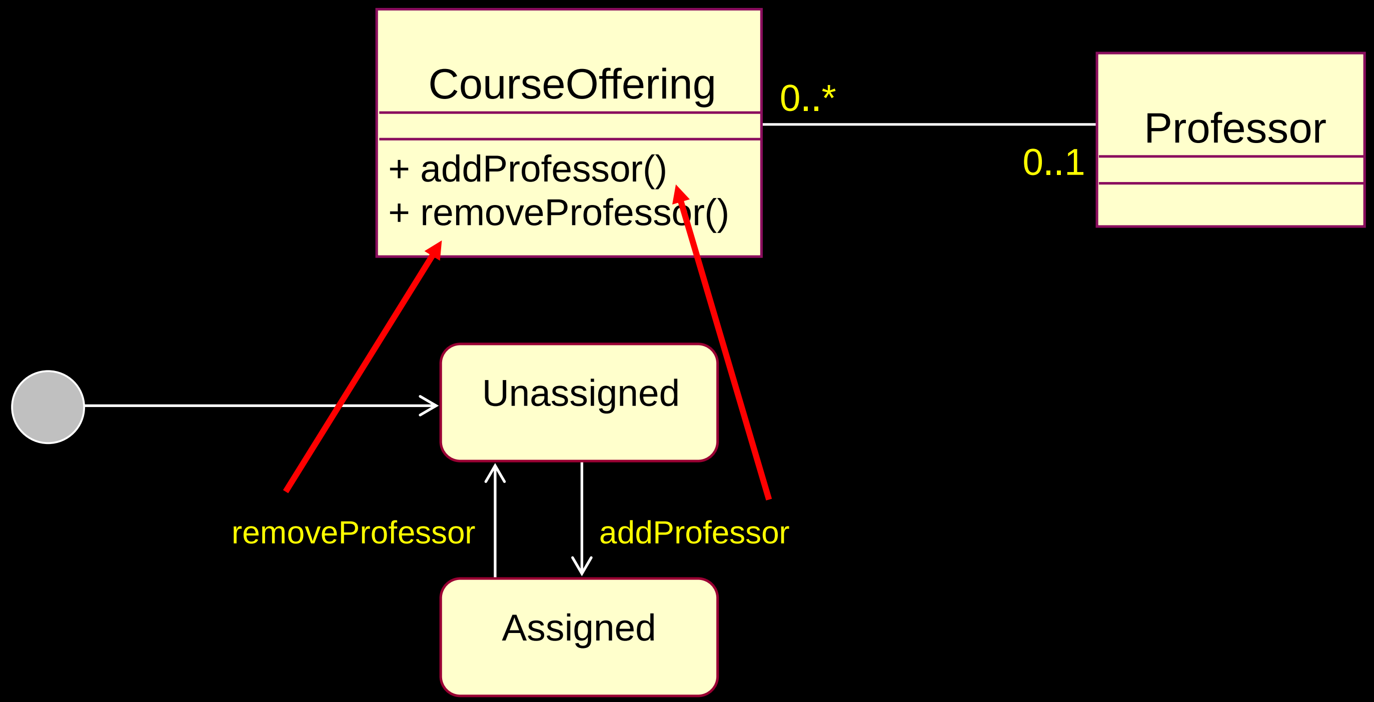
- Look at the class interface operations



Events: addProfessor,
removeProfessor

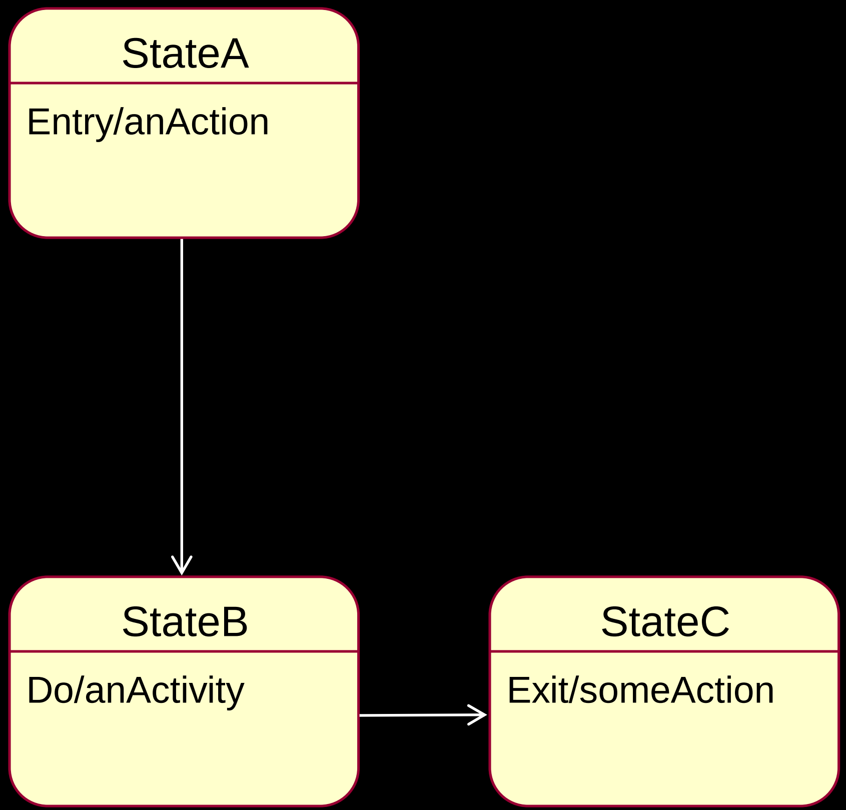
Identify the Transitions

- 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

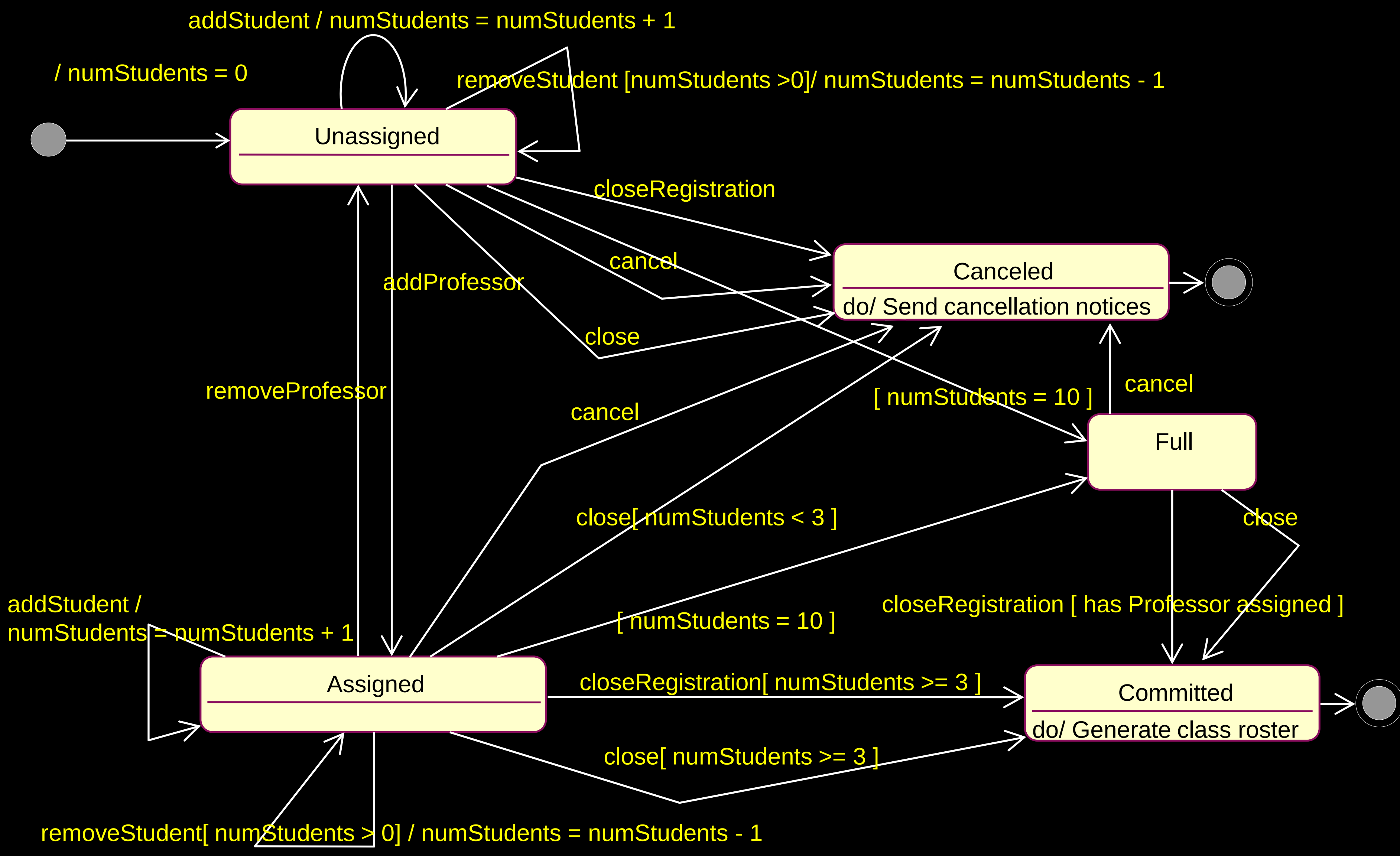


Add Activities

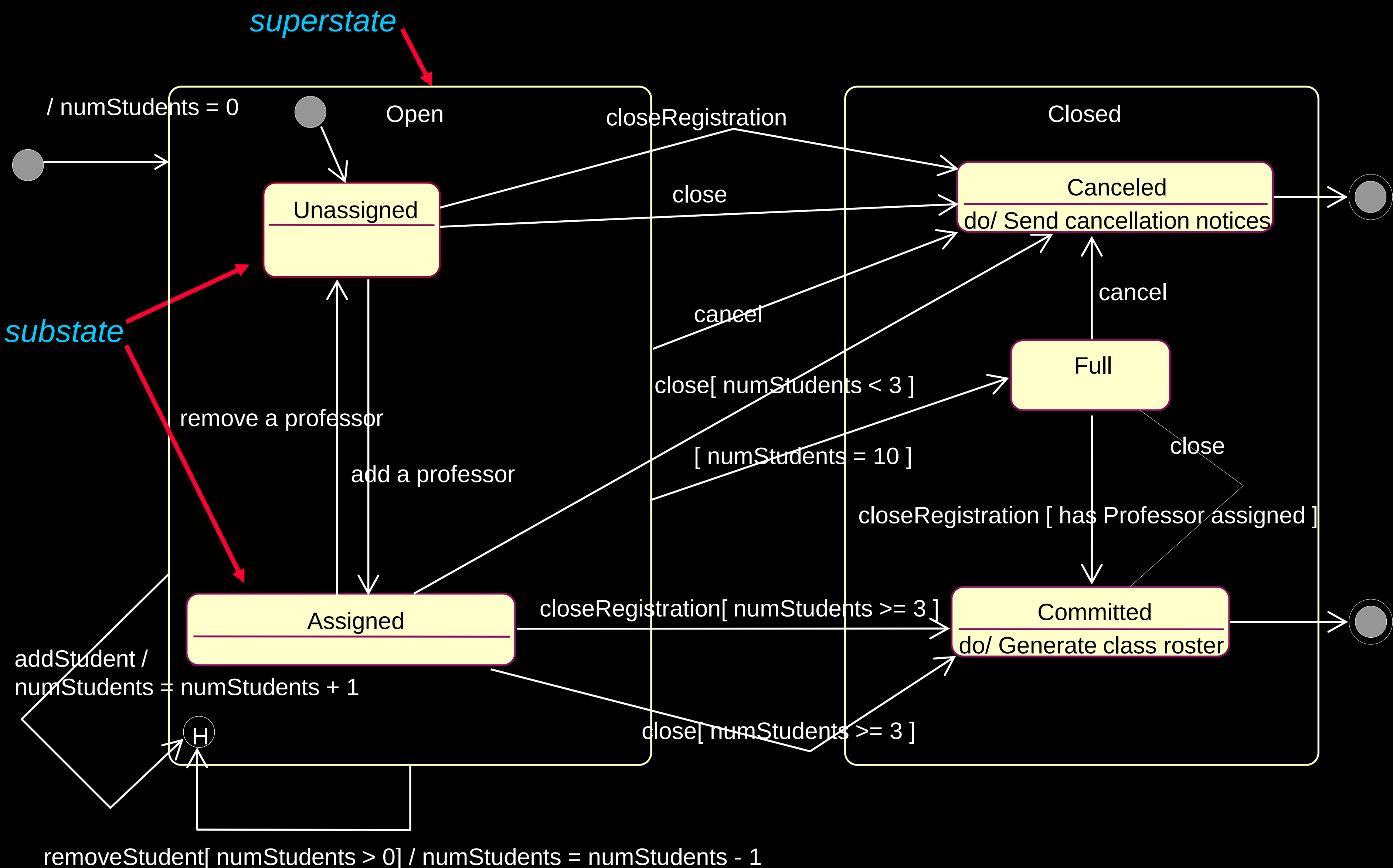
- Entry
  - Executed when the state is entered
- Do
  - Ongoing execution
- Exit
  - Executed when the state is exited



Example: State Machine



Example: State Machine with Nested States and History

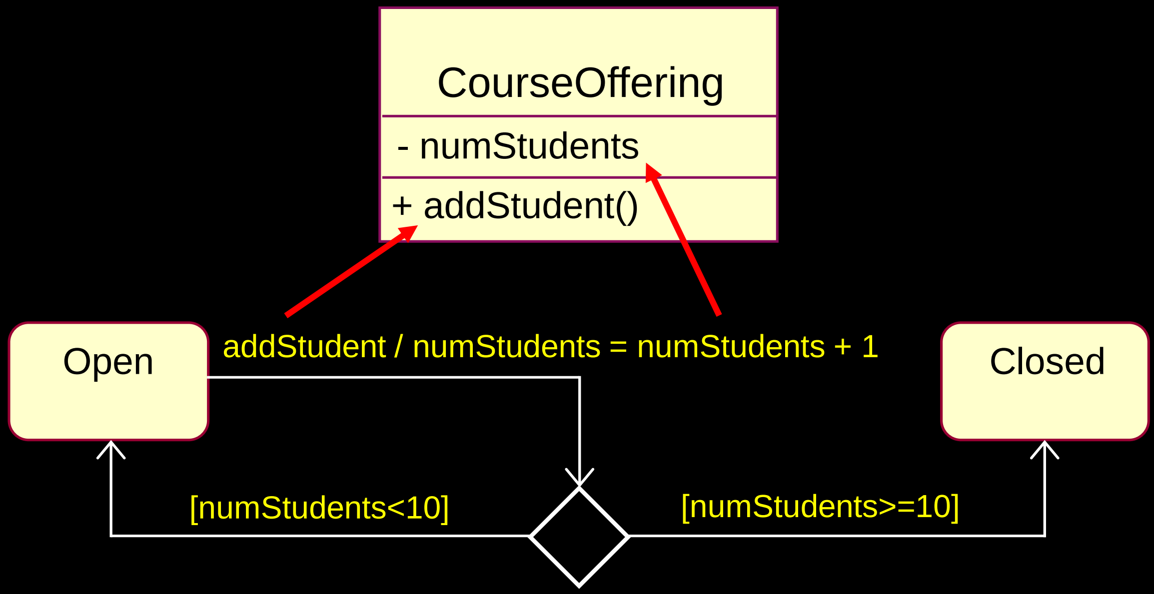


Which Objects Have Significant State?

- Objects whose role is clarified by state transitions
- Complex use cases that are state-controlled
- It is not necessary to model objects such as:
  - Objects with straightforward mapping to implementation
  - Objects that are not state-controlled
  - Objects with only one computational state

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

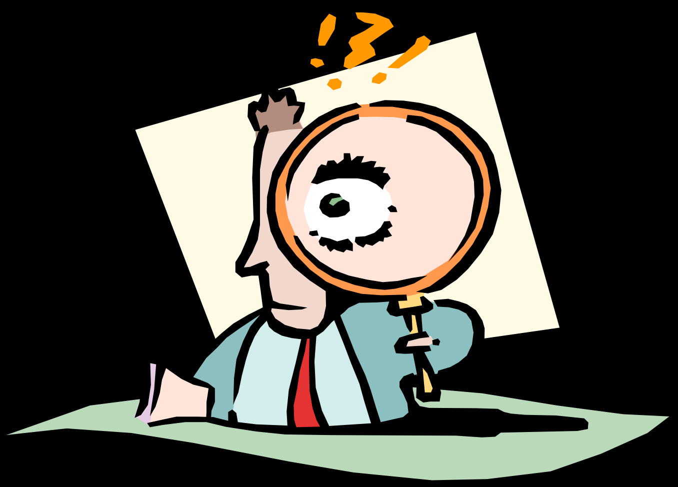


Class Design Steps

- 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

Attributes: How Do You Find Them?

- Examine method descriptions
- Examine states
- Examine any information the class itself needs to maintain



Attribute Representations

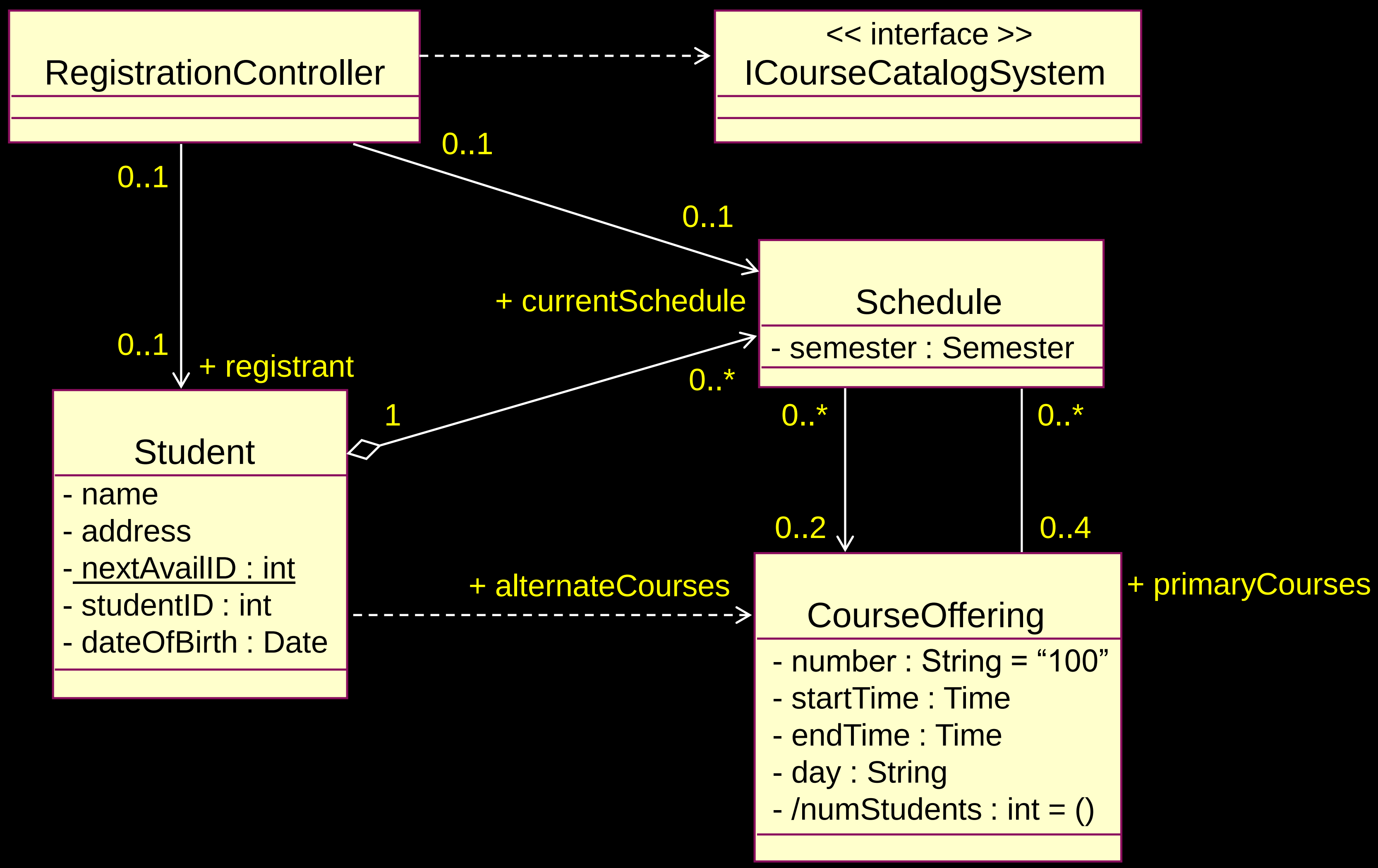
- Specify name, type, and optional default value
  - attributeName : Type = Default
- Follow naming conventions of implementation language and project
- Type should be an elementary data type in implementation language
  - Built-in data type, user-defined data type, or user-defined class
- Specify visibility
  - Public: +
  - Private: -
  - Protected: #

Derived Attributes

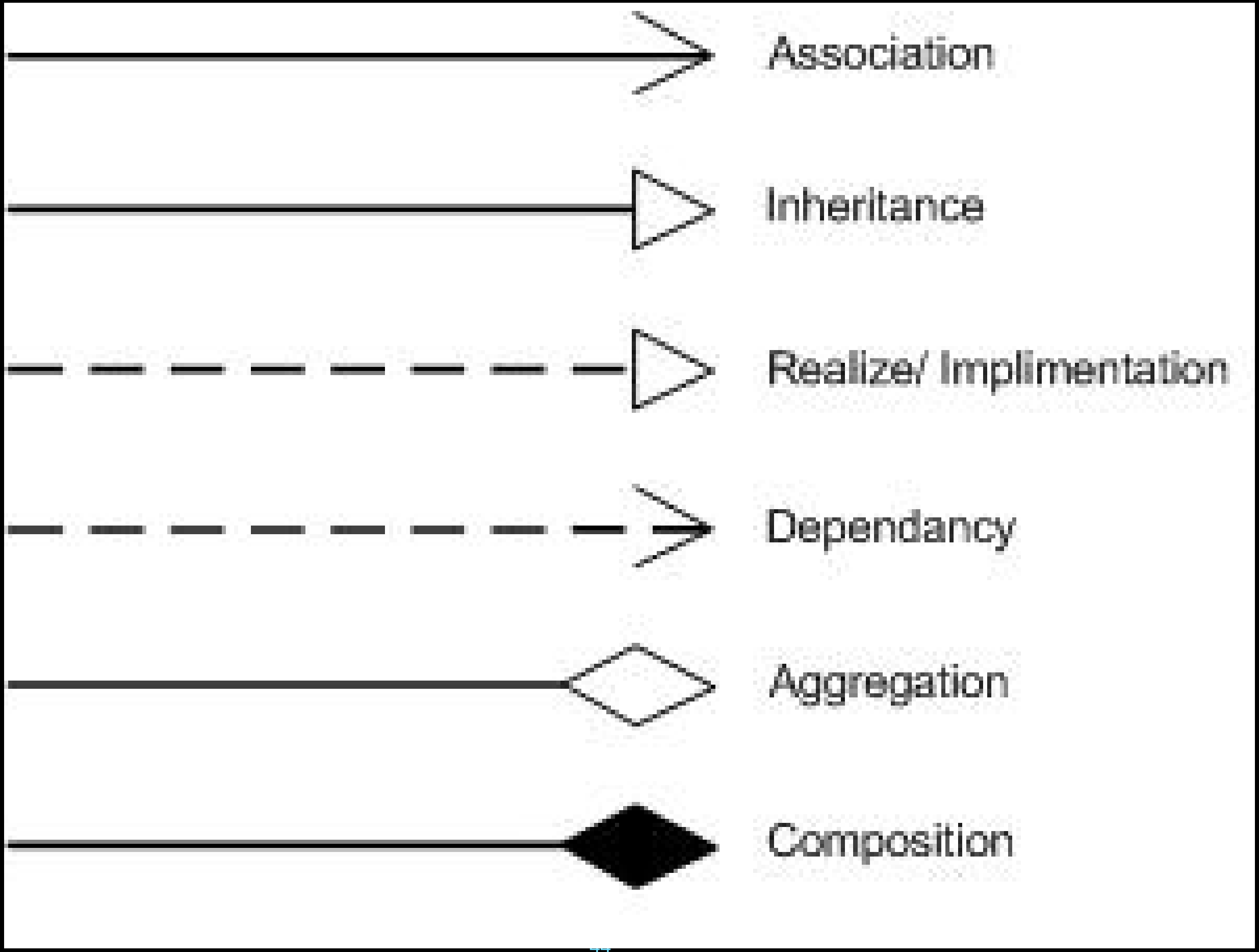
- What is a derived attribute?
  - An attribute whose value may be calculated based on the value of other attribute(s)
- When do you use it?
  - When there is not enough time to re-calculate the value every time it is needed
  - When you must trade-off runtime performance versus memory required

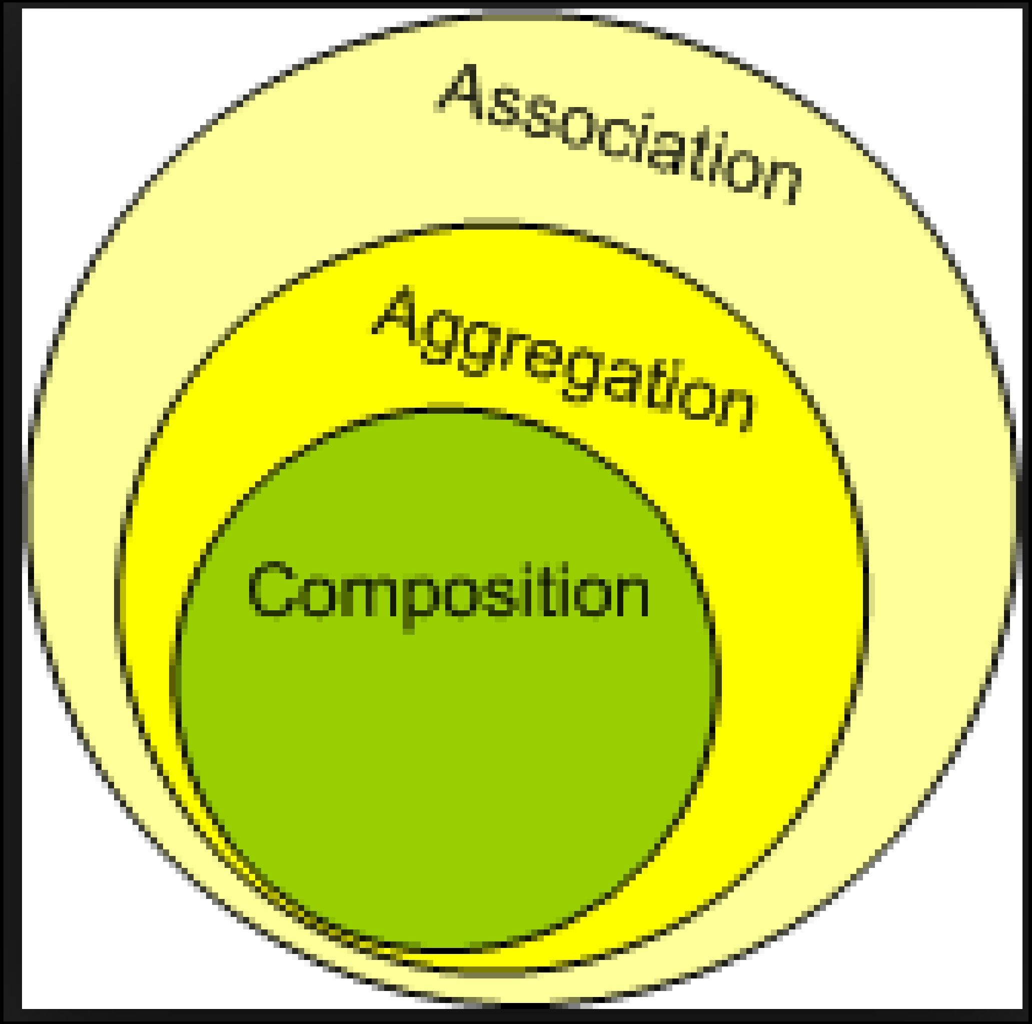
IBM

Example: Define Attributes

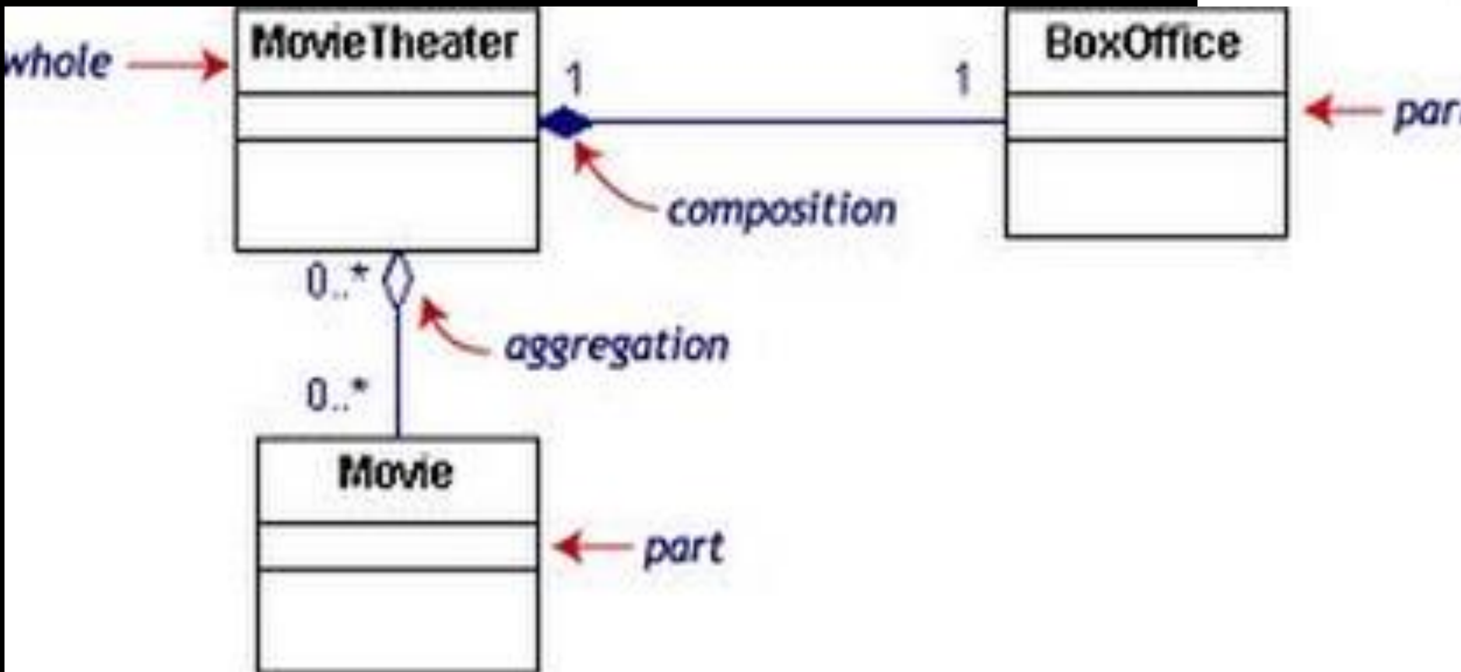


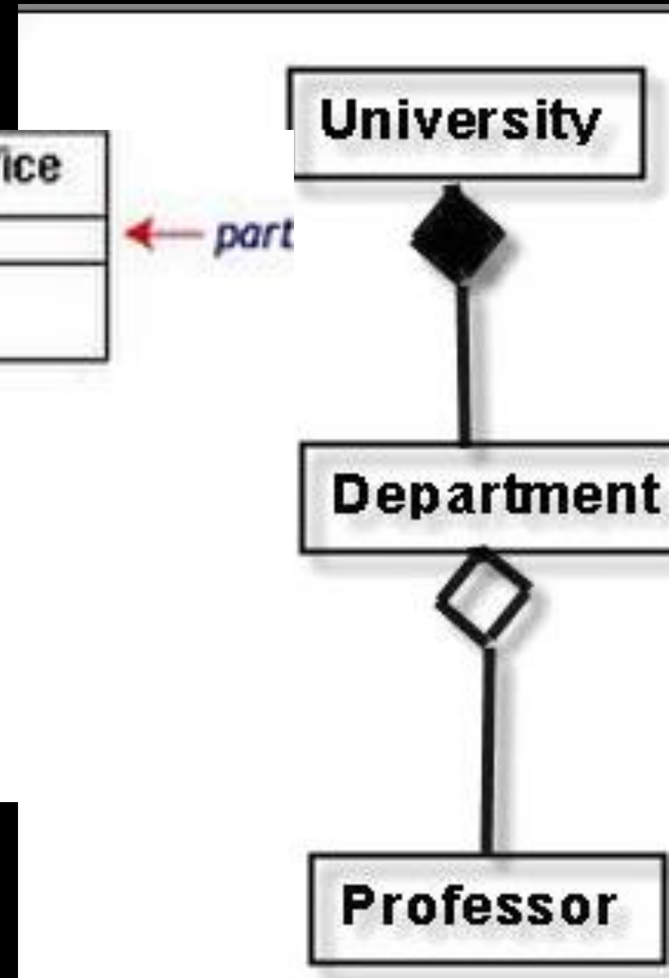
Relationship between classes




Association
Aggregation
Composition

Example


whole
MovieTheater
BoxOffice
1
1
part
composition
0..*
aggregation
0..*
Movie
part

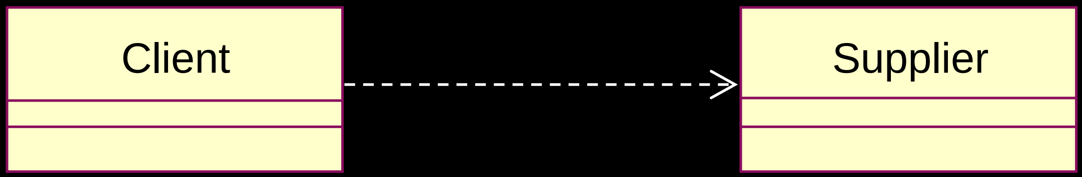

University
Department
Professor

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

Define Dependency

- What Is a Dependency?
  - A relationship between two objects



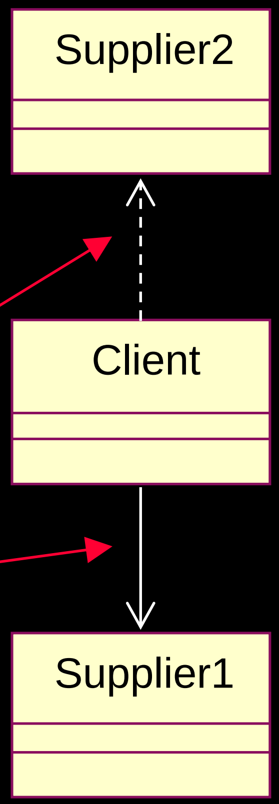
- Purpose
  - Determine where structural relationships are NOT required
- Things to look for :
  - What causes the supplier to be visible to the client

Dependencies vs. Associations

- Associations are structural relationships
- Dependencies are non-structural relationships
- In order for objects to “know each other” they must be visible
  - Local variable reference
  - Parameter reference
  - Global reference
  - Field reference

Dependency

Association



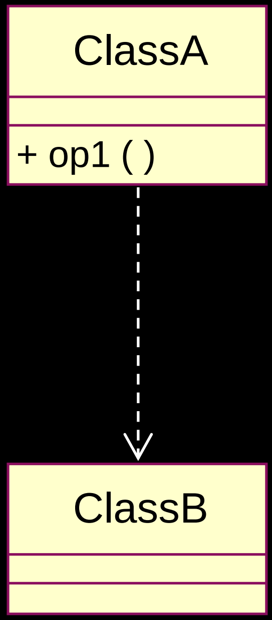
Associations vs. Dependencies in Collaborations

- An instance of an association is a link
  - All links become associations unless they have global, local, or parameter visibility
  - Relationships are context-dependent
- Dependencies are transient links with:
  - A limited duration
  - A context-independent relationship
  - A summary relationship

A dependency is a secondary type of relationship in that it doesn't tell you much about the relationship. For details you need to consult the collaborations.

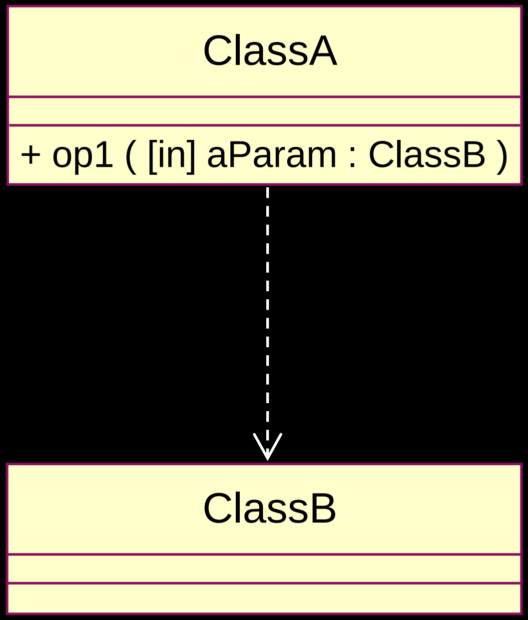
Local Variable Visibility

- The op1() operation contains a local variable of type ClassB



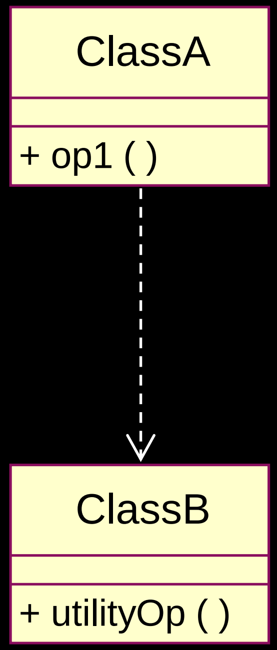
Parameter Visibility

- The ClassB instance is passed to the ClassA instance



Global Visibility

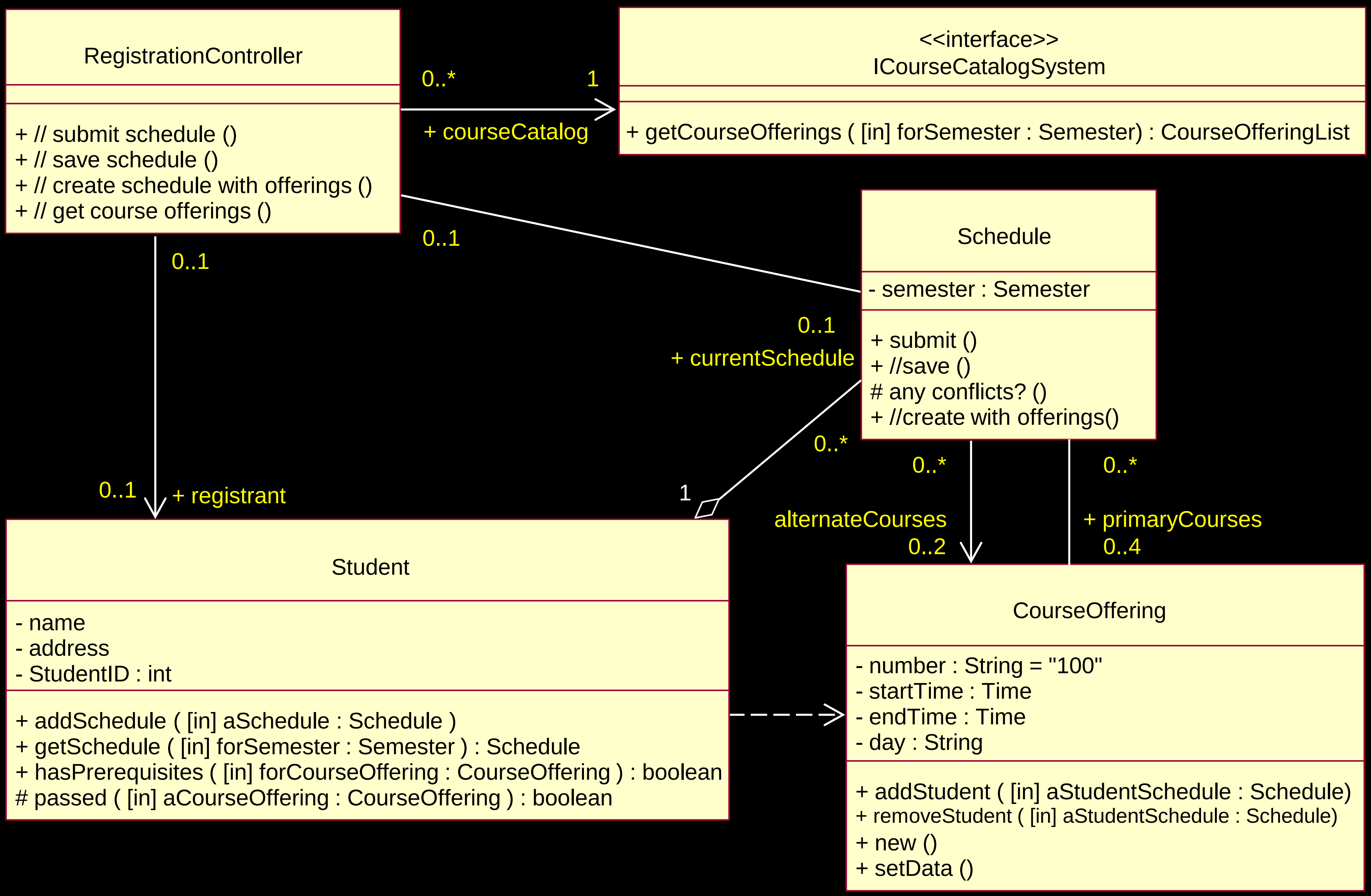
- The ClassUtility instance is visible because it is global



Identifying Dependencies: Considerations

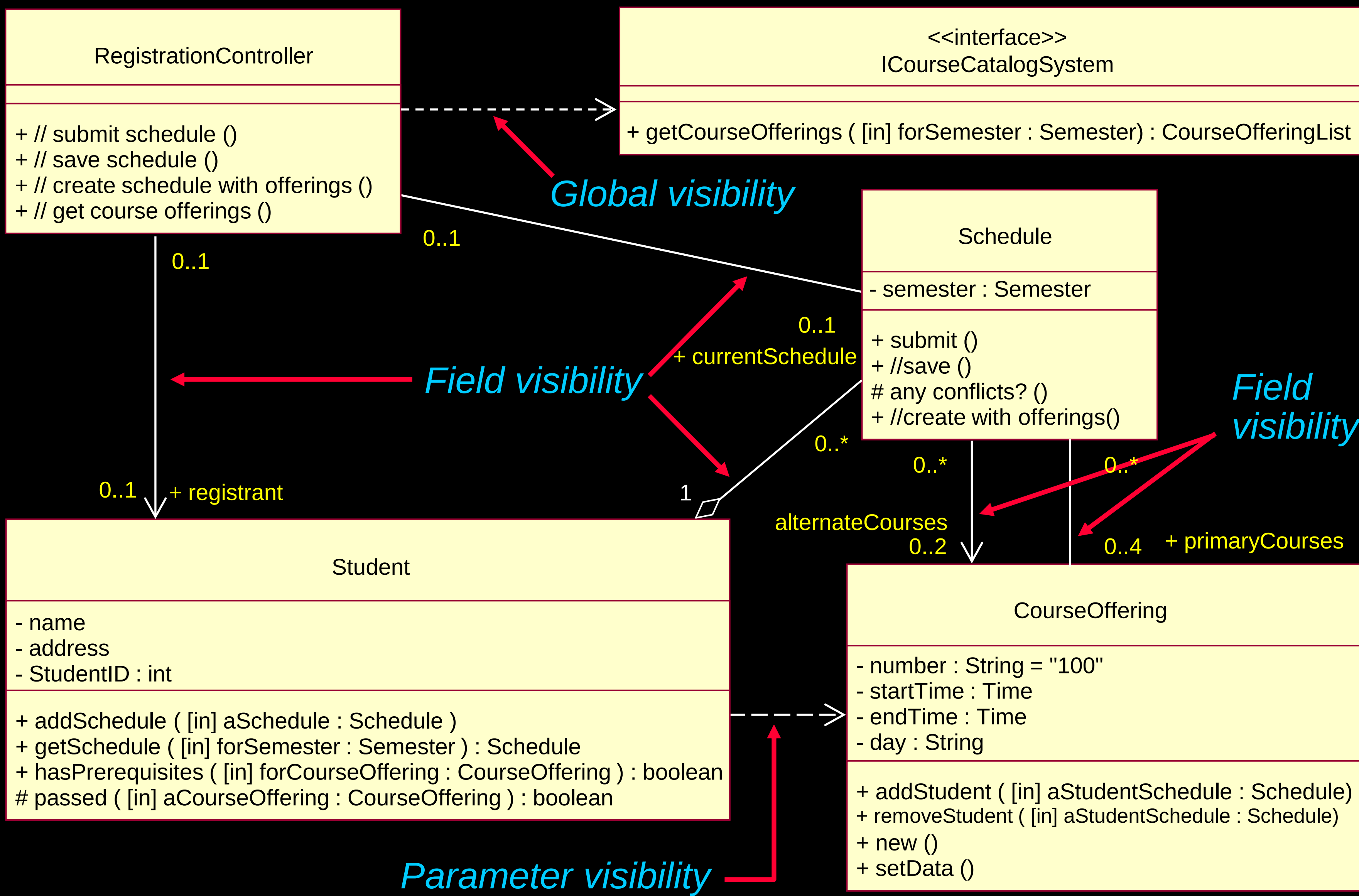
- Permanent relationships — Association (field visibility)
- Transient relationships — Dependency
  - Multiple objects share the same instance
    - Pass instance as a parameter (parameter visibility)
    - Make instance a managed global (global visibility)
  - Multiple objects don't share the same instance (local visibility)
- How long does it take to create/destroy?
  - Expensive? Use field, parameter, or global visibility
  - Strive for the lightest relationships possible

Example: Define Dependencies (before)



IBM

Example: Define Dependencies (after)



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

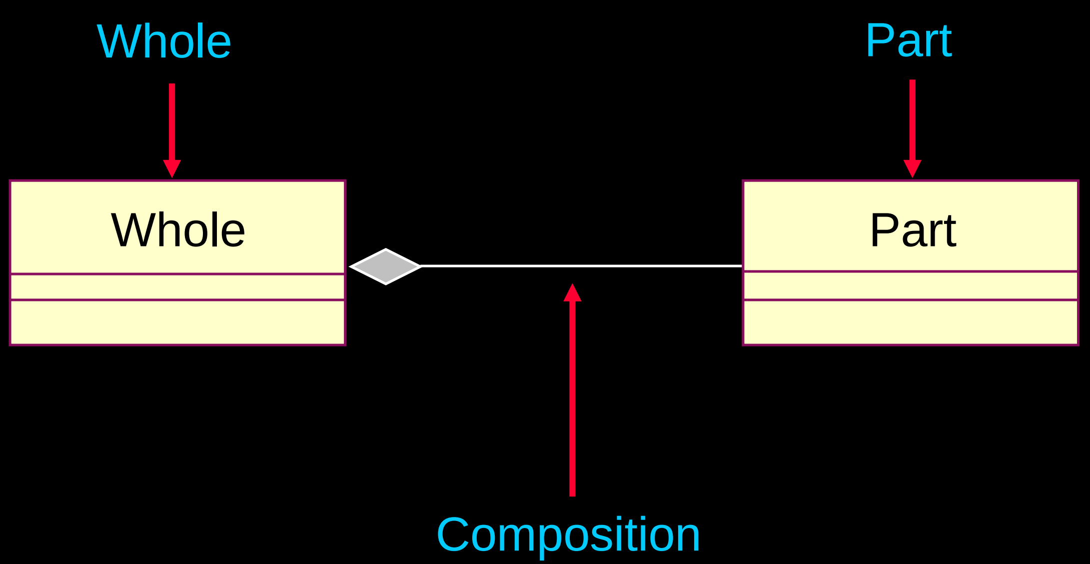
Define Associations

- Purpose
  - Refine remaining associations
- Things to look for :
  - Association vs. Aggregation
  - Aggregation vs. Composition
  - Attribute vs. Association
  - Navigability
  - Association class design
  - Multiplicity design

IBM

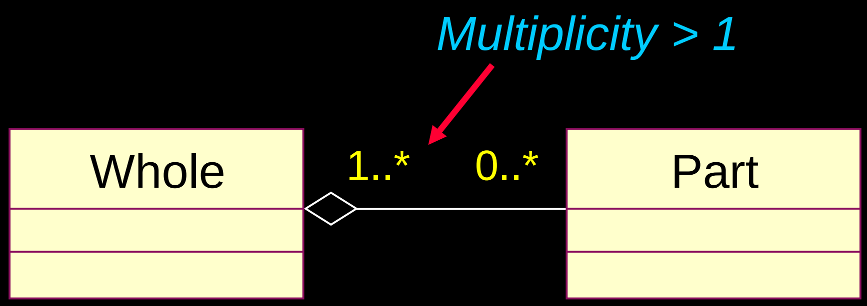
What Is Composition?

- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate

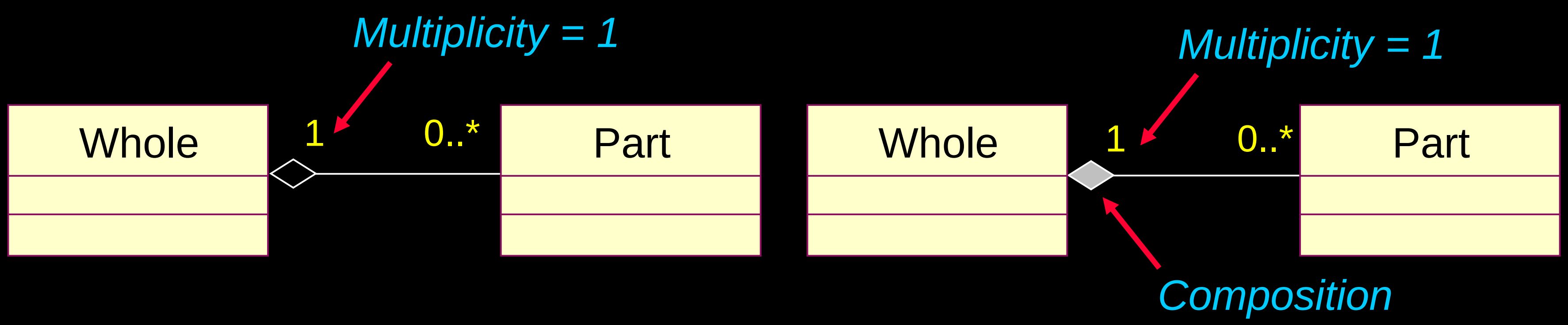


Aggregation: Shared vs. Non-shared

- Shared Aggregation



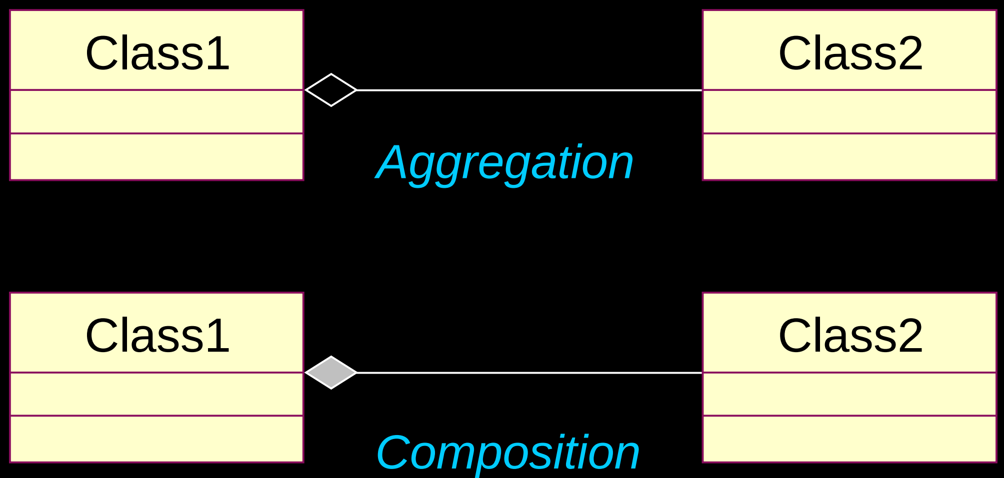
- Non-shared Aggregation



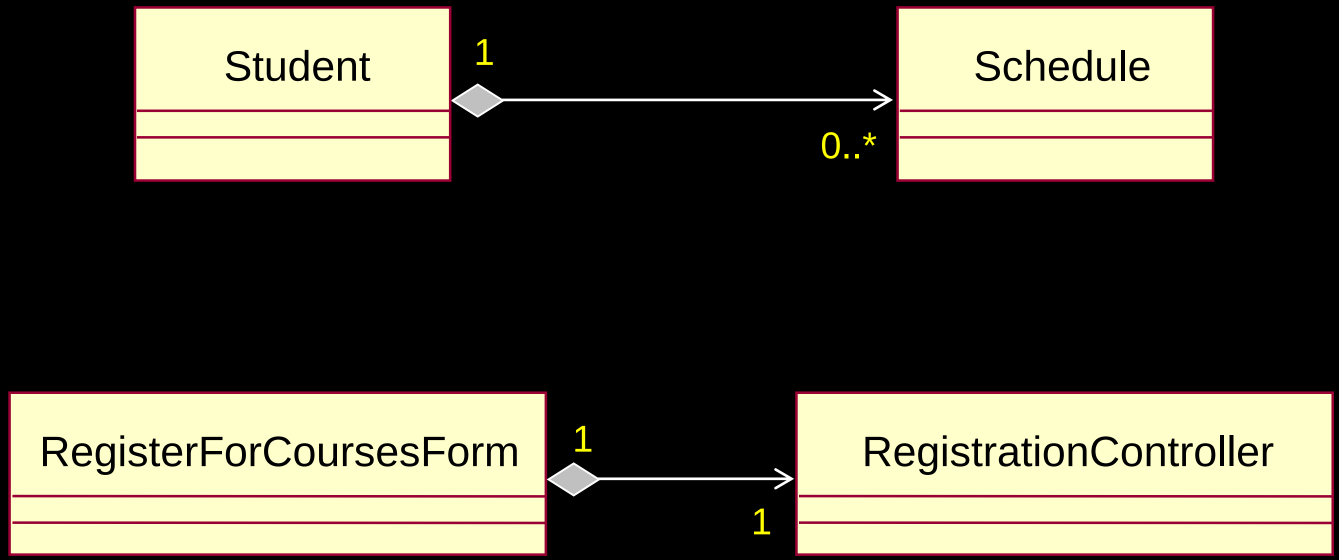
By definition, composition is non-shared aggregation.

Aggregation or Composition?

- Consideration
  - Lifetimes of Class1 and Class2



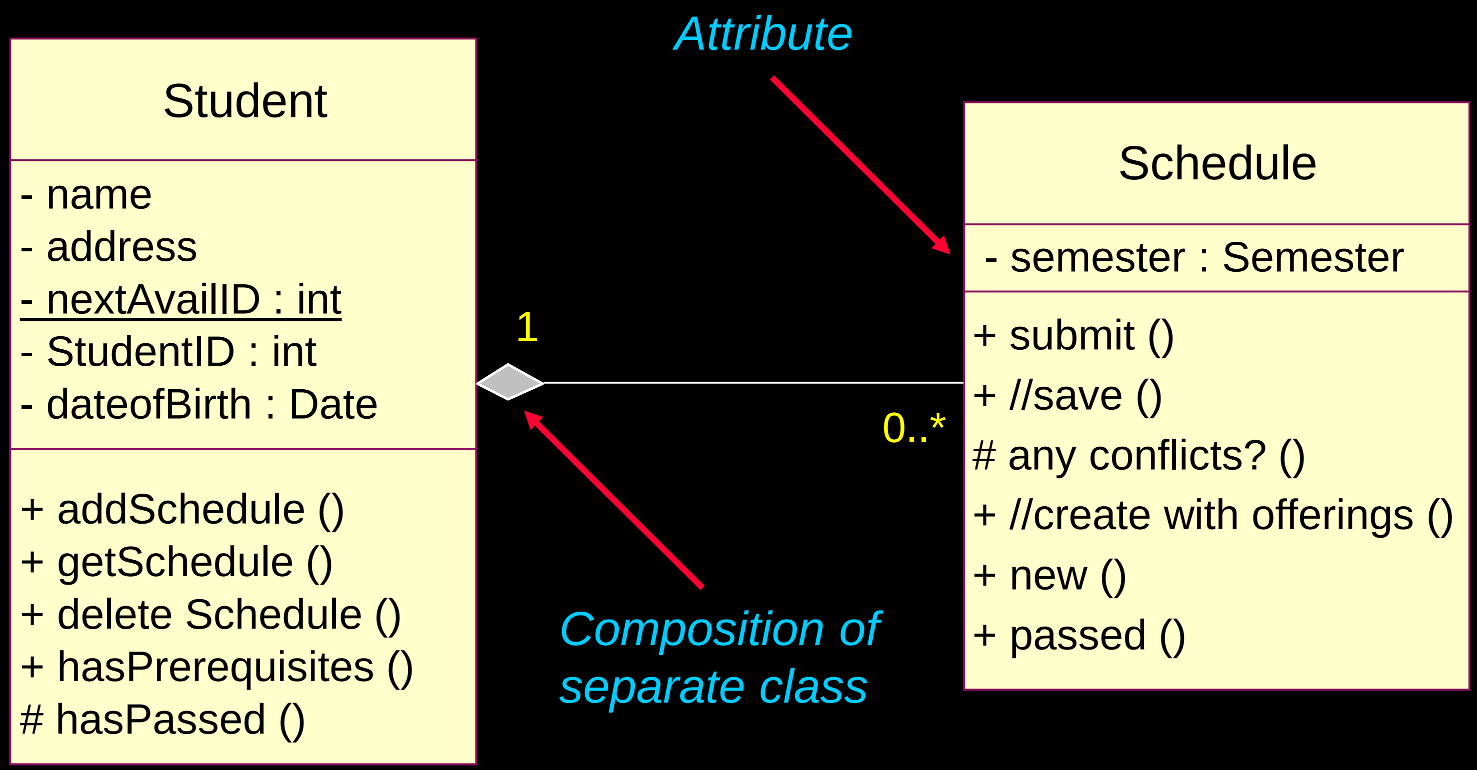
Example: Composition



Attributes vs. Composition

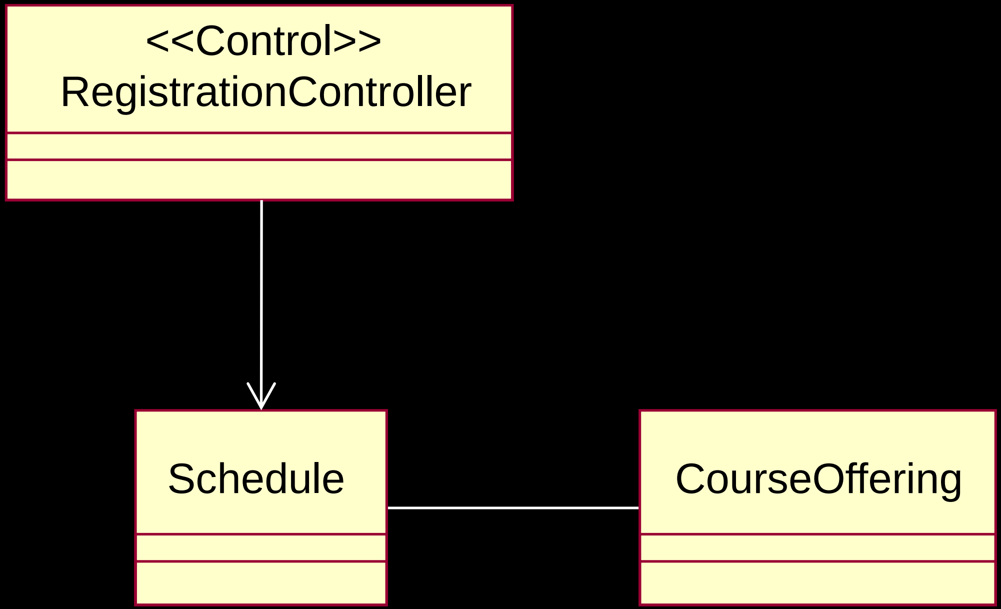
- Use composition when
  - Properties need independent identities
  - Multiple classes have the same properties
  - Properties have a complex structure and properties of their own
  - Properties have complex behavior of their own
  - Properties have relationships of their own
- Otherwise use attributes

Example: Attributes vs. Composition



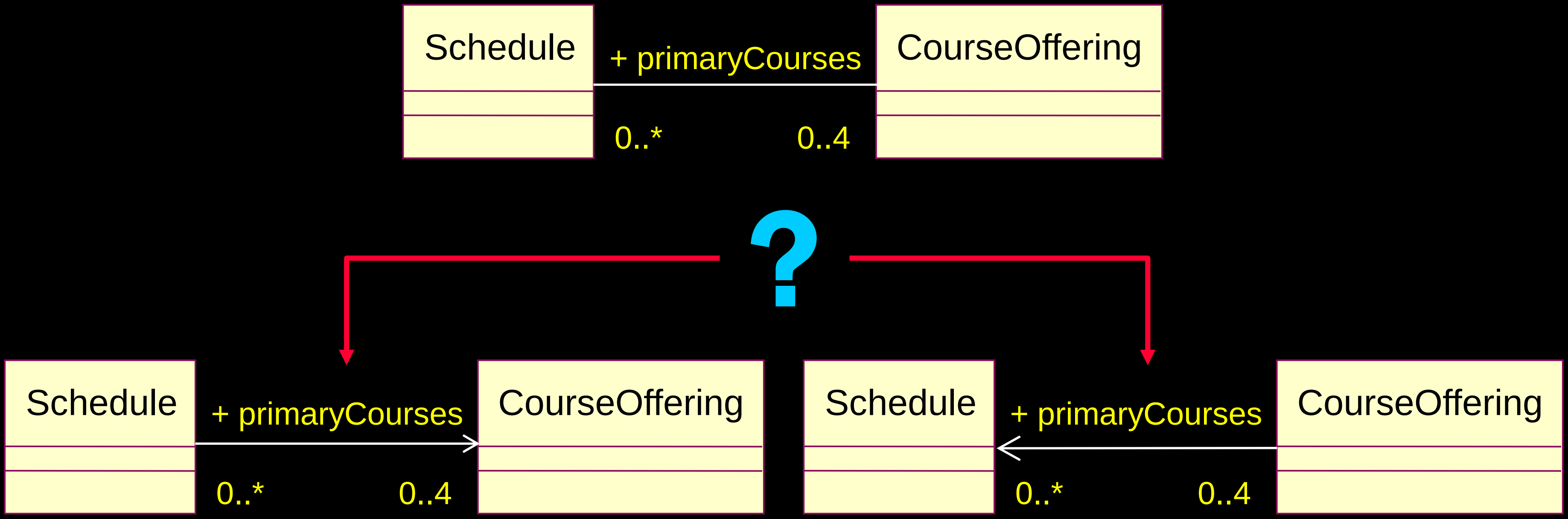
Review: What Is Navigability?

- Indicates that it is possible to navigate from an associating class to the target class using the association



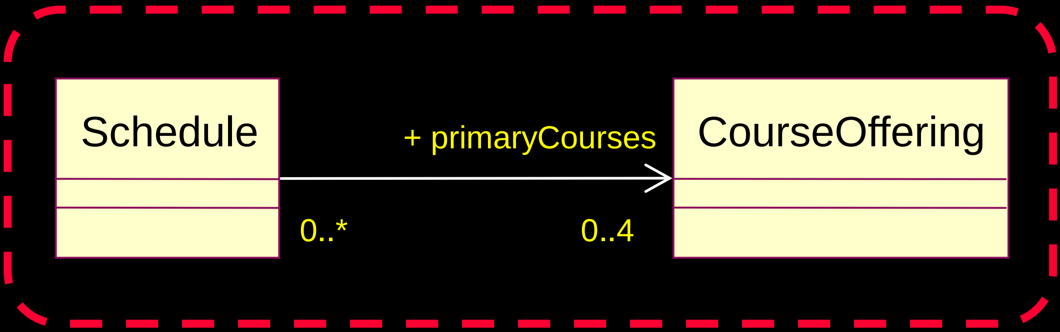
Navigability: Which Directions Are Really Needed?

- Explore interaction diagrams
- Even when both directions seem required, one may work
  - Navigability in one direction is infrequent
  - Number of instances of one class is small

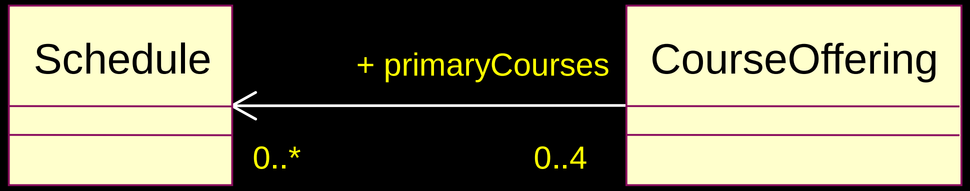


Example: Navigability Refinement

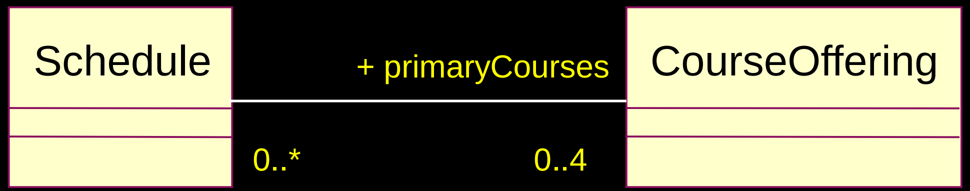
- Total number of Schedules is small, or
- Never need a list of the Schedules on which the CourseOffering appears



- Total number of CourseOfferings is small, or
- Never need a list of CourseOfferings on a Schedule



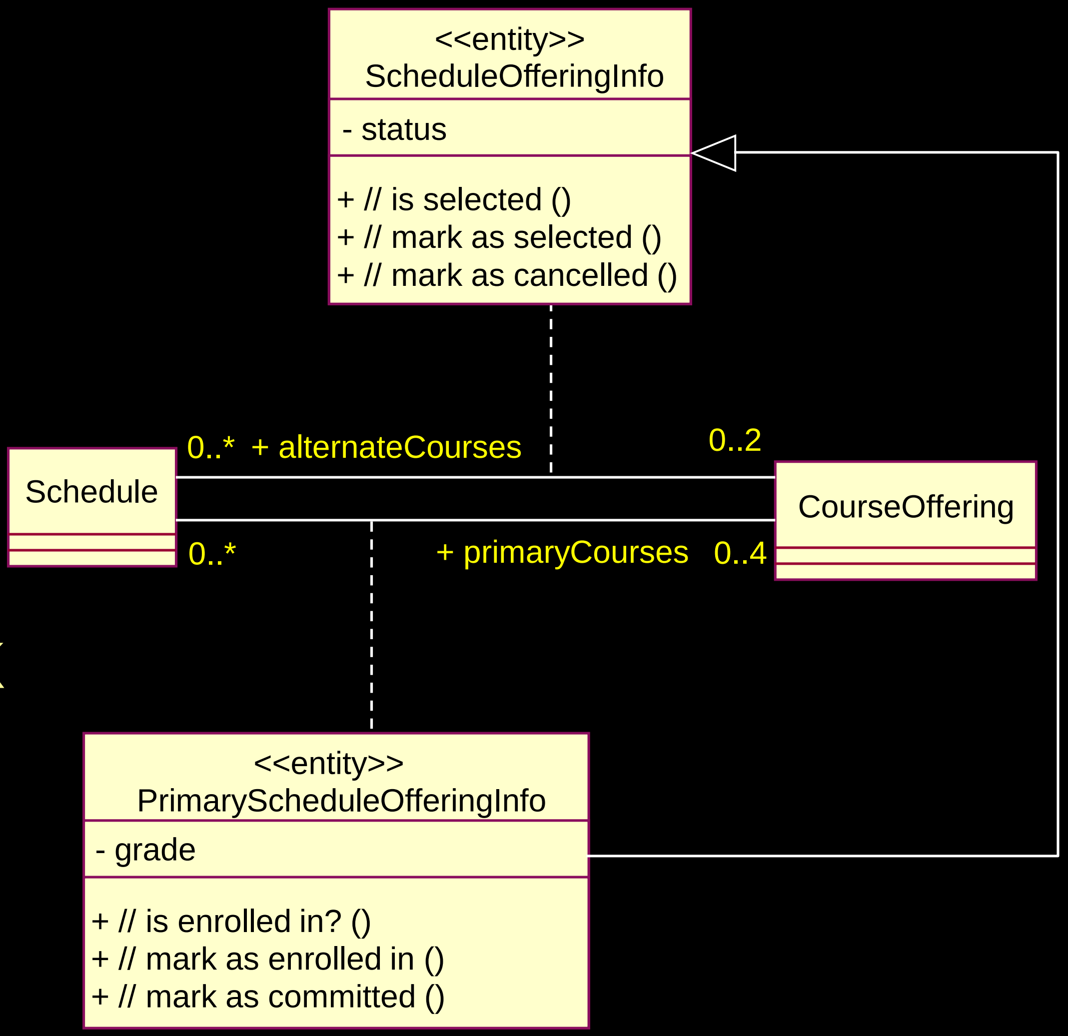
- Total number of CourseOfferings and Schedules are not small
- Must be able to navigate in both directions



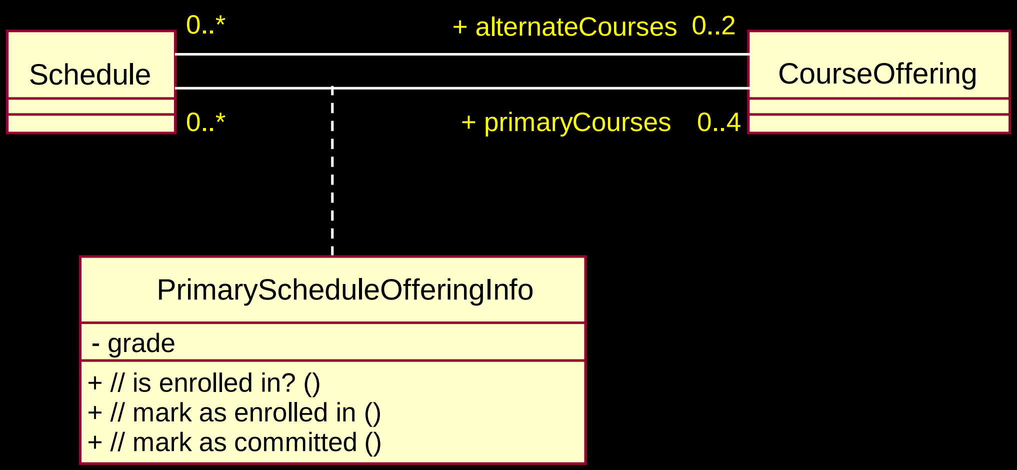
IBM

Association Class

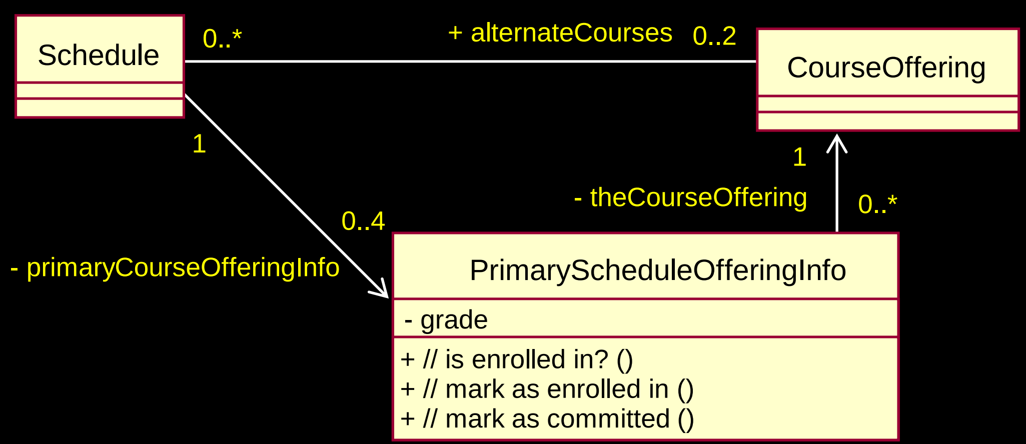
- A class is “attached” to an association
- Contains properties of a relationship
- Has one instance per link



Example: Association Class Design

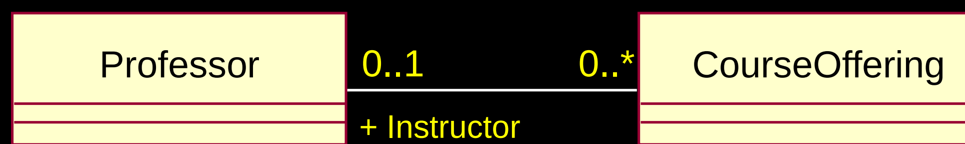


Design Decisions



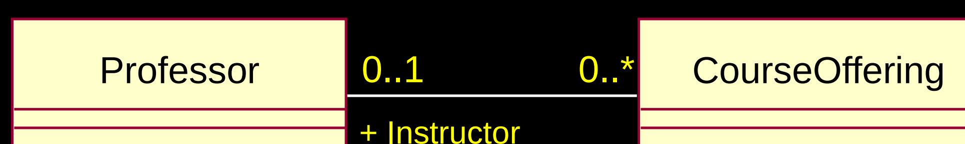
Multiplicity Design

- Multiplicity = 1, or Multiplicity = 0..1
  - May be implemented directly as a simple value or pointer
  - No further “design” is required

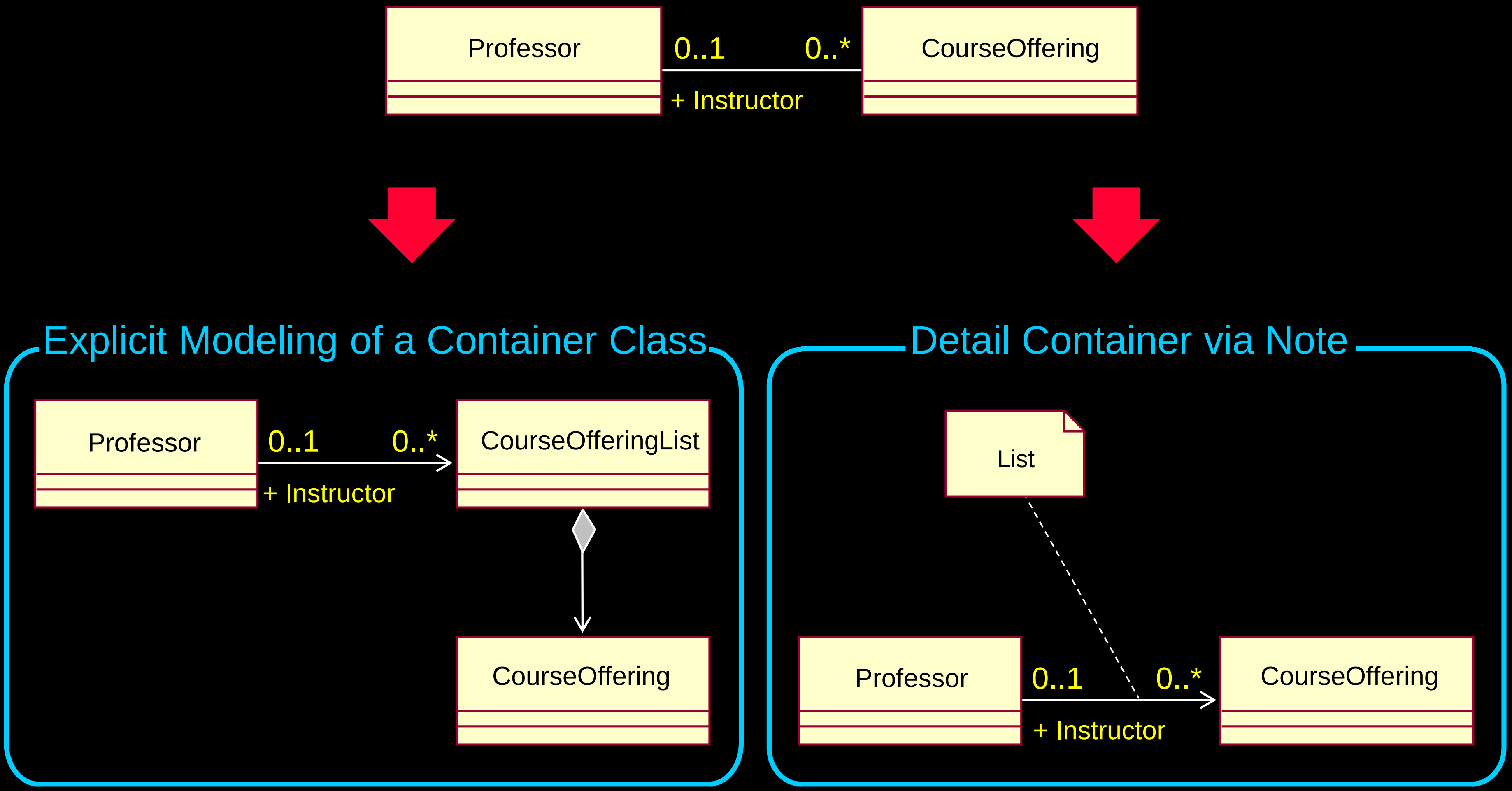


- Multiplicity > 1
  - Cannot use a simple value or pointer
  - Further “design” may be required

Needs a container for CourseOfferings

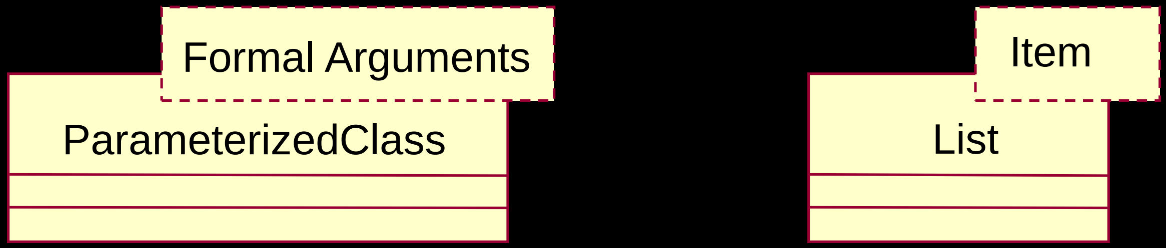


Multiplicity Design Options

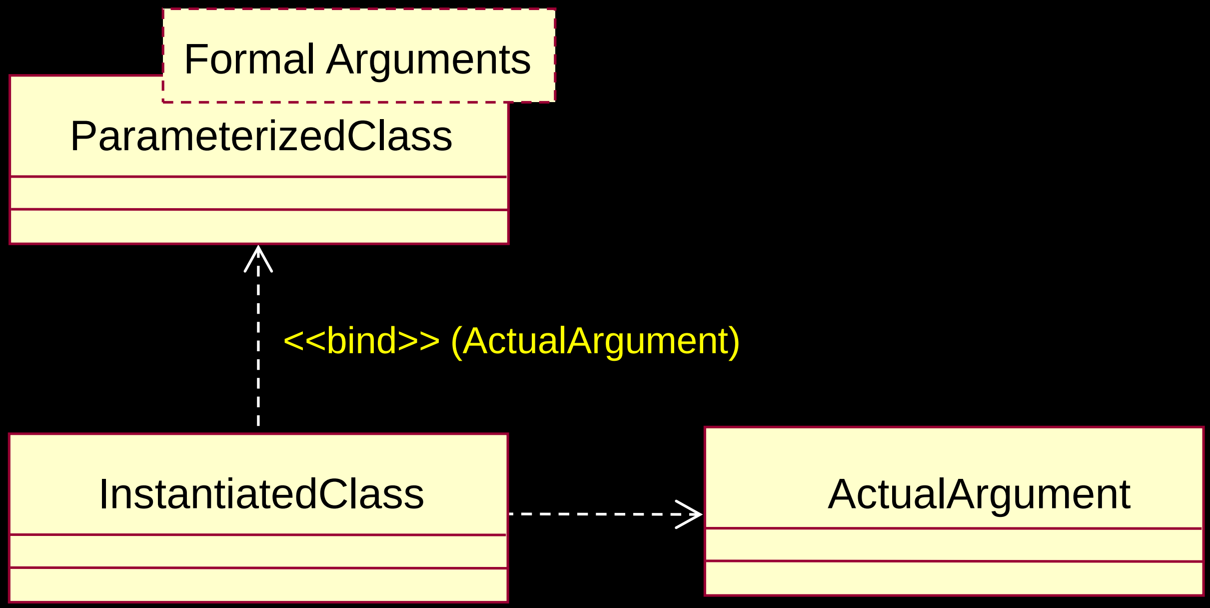


What Is a Parameterized Class (Template)?

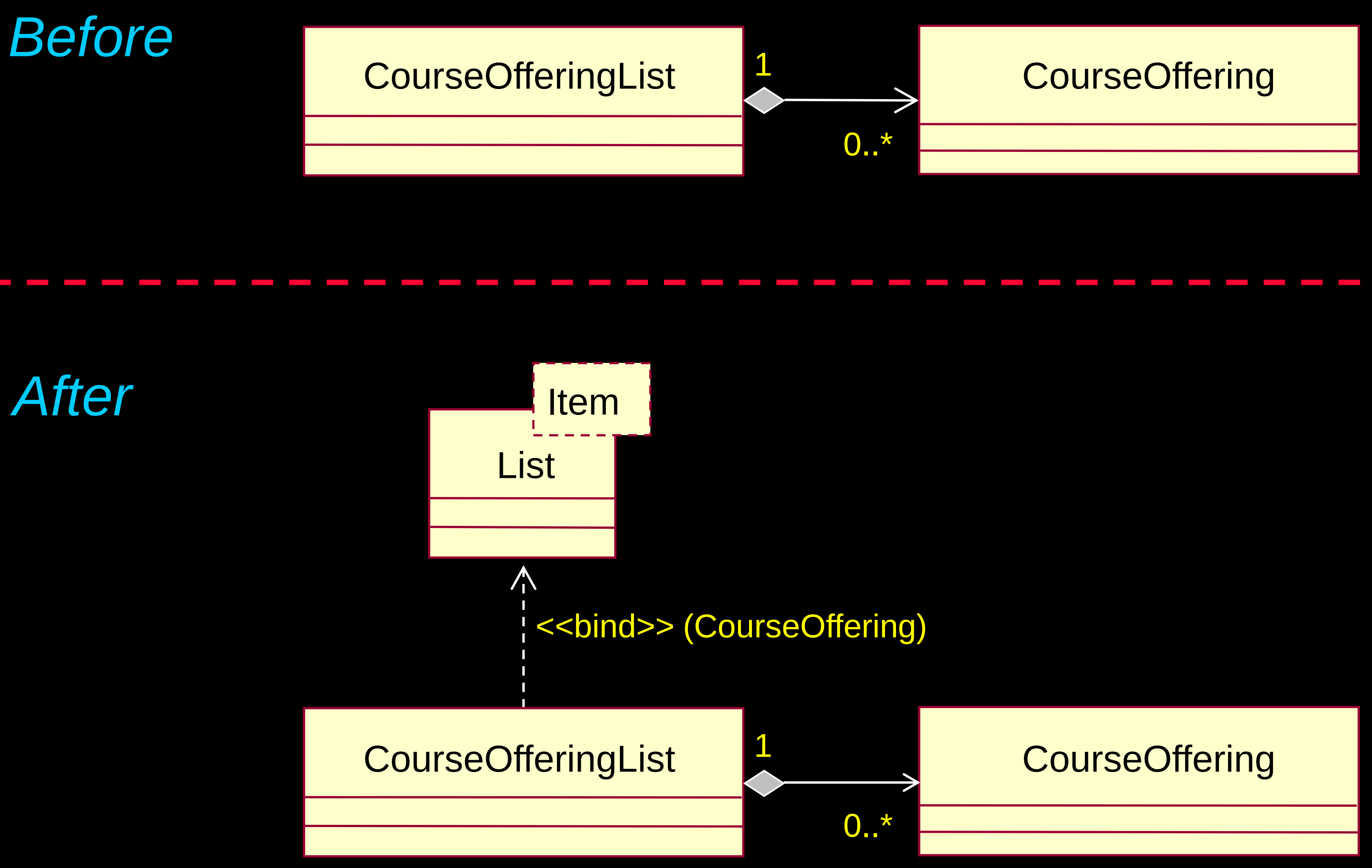
- A class definition that defines other classes
- Often used for container classes
  - Some common container classes:
    - Sets, lists, dictionaries, stacks, queues



Instantiating a Parameterized Class



Example: Instantiating a Parameterized Class



IBM

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

IBM

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.

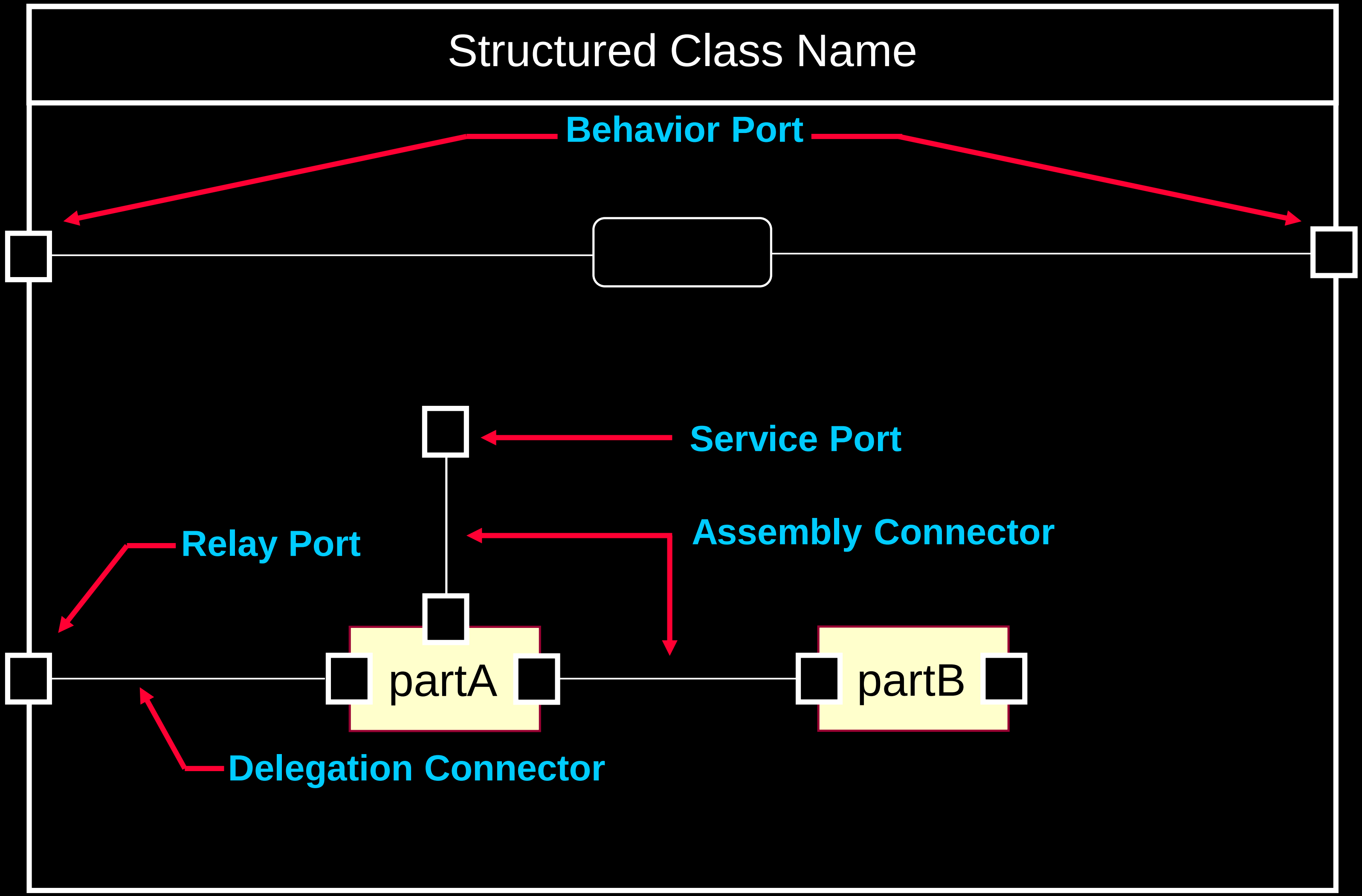
Review: What Is a Port?

- 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

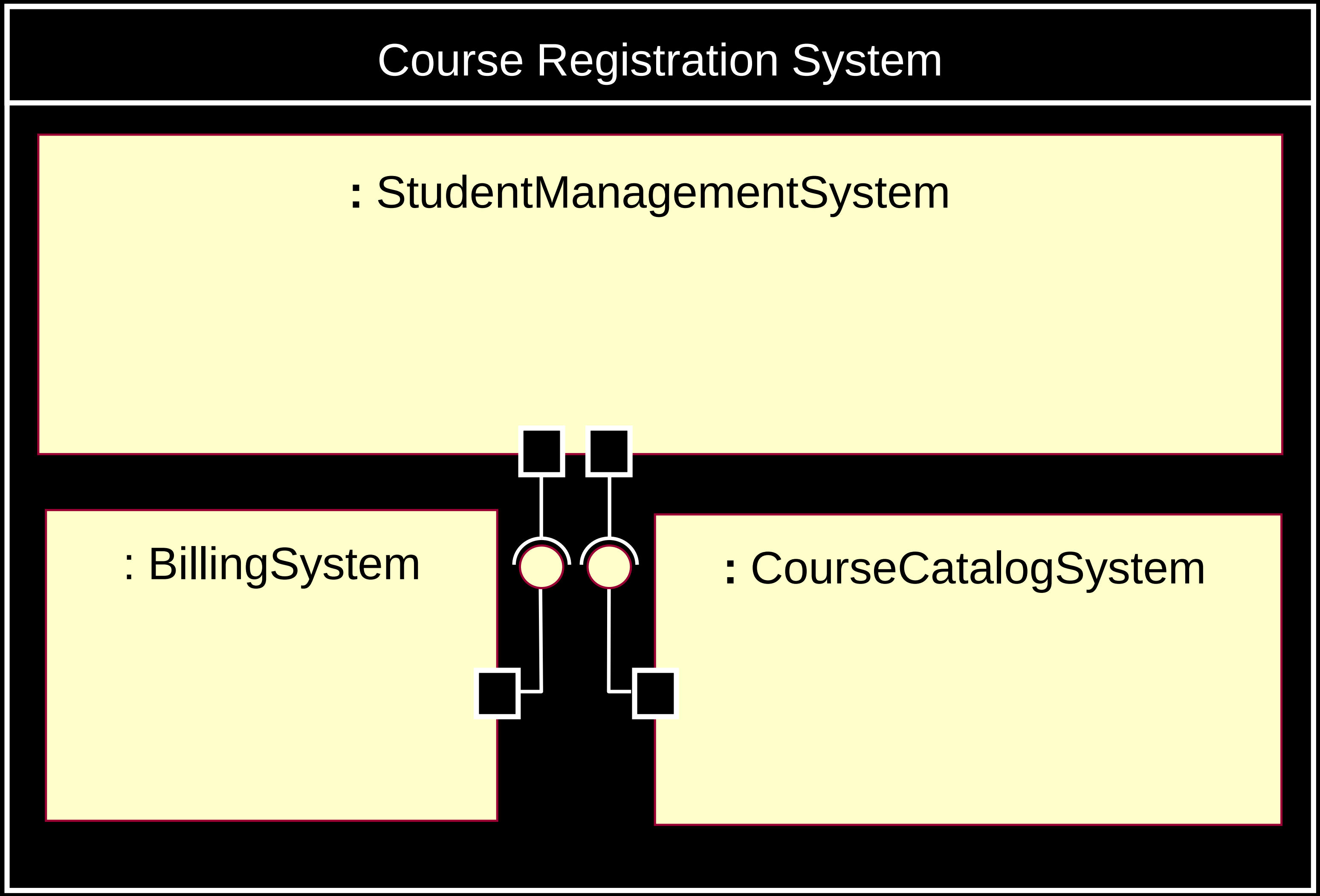
Review: Port Types

- 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.

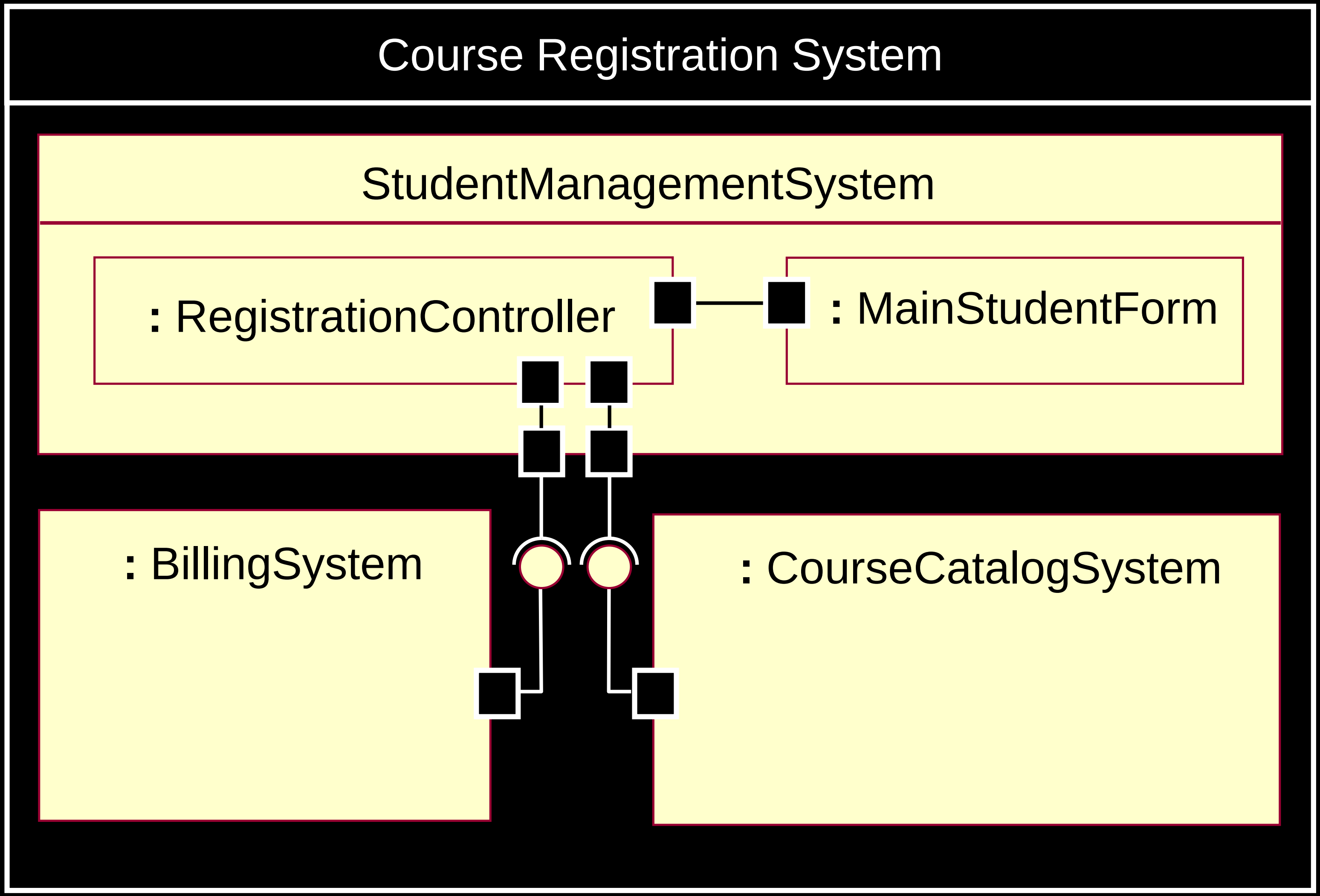
Review: Structure Diagram With Ports



Review: Structure Diagram



Example: Structure Diagram Detailed



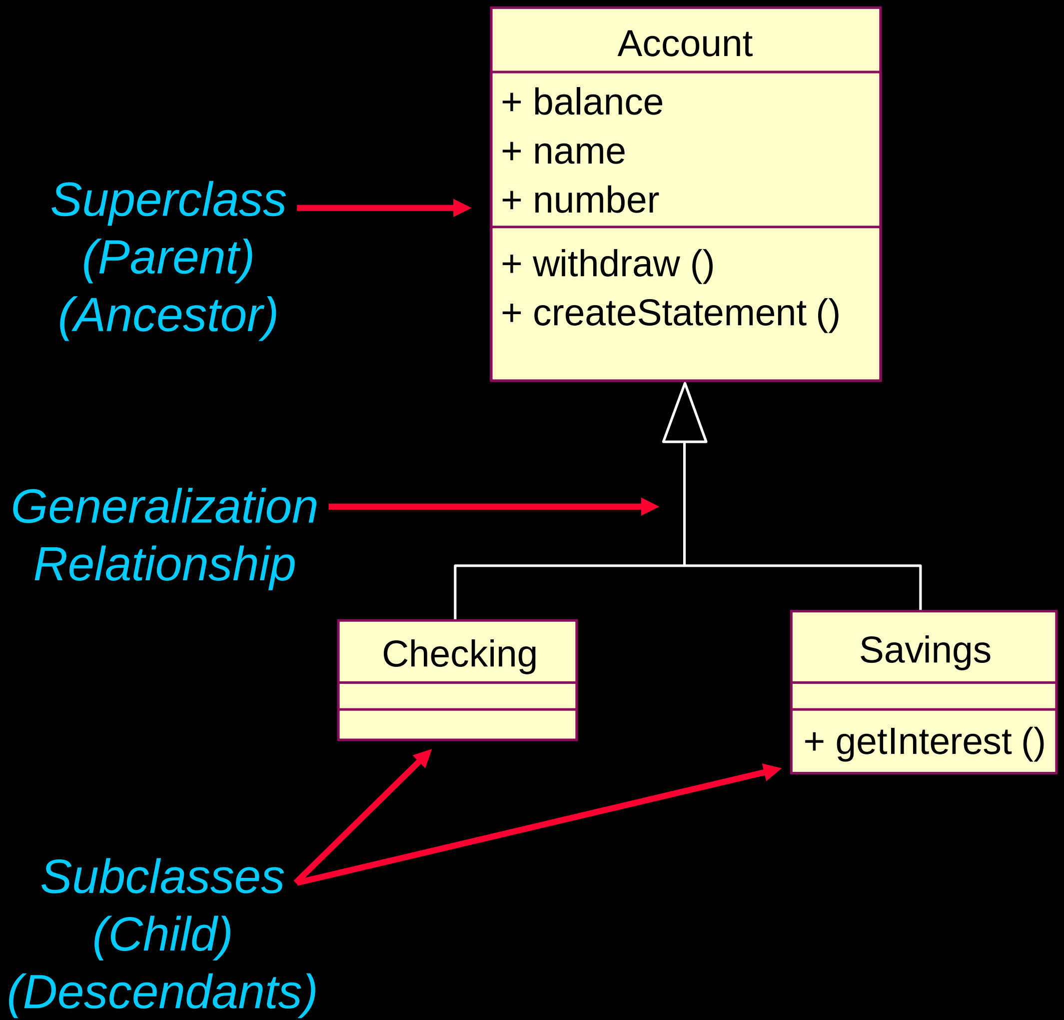
Class Design Steps

- 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

IBM

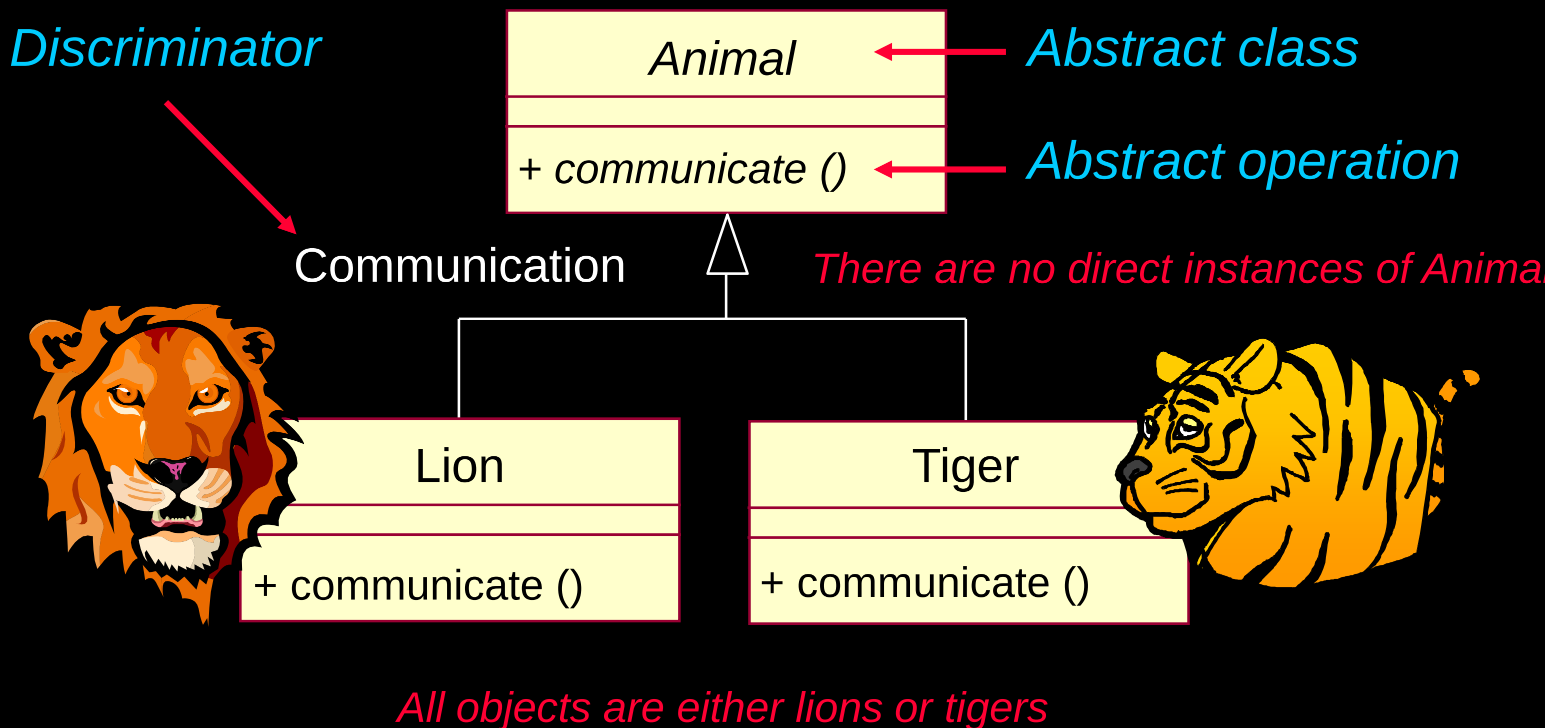
Review: Generalization

- One class shares the structure and/or behavior of one or more classes
- “Is a kind of” relationship
- In Analysis, use sparingly



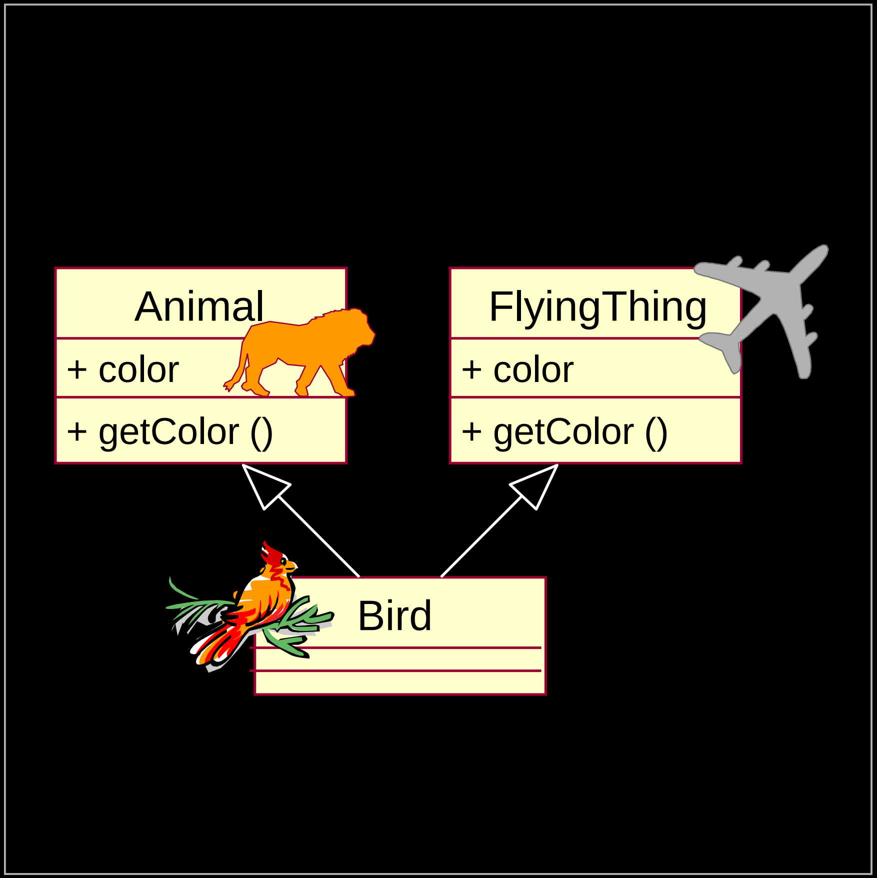
Abstract and Concrete Classes

- Abstract classes cannot have any objects
- Concrete classes can have objects

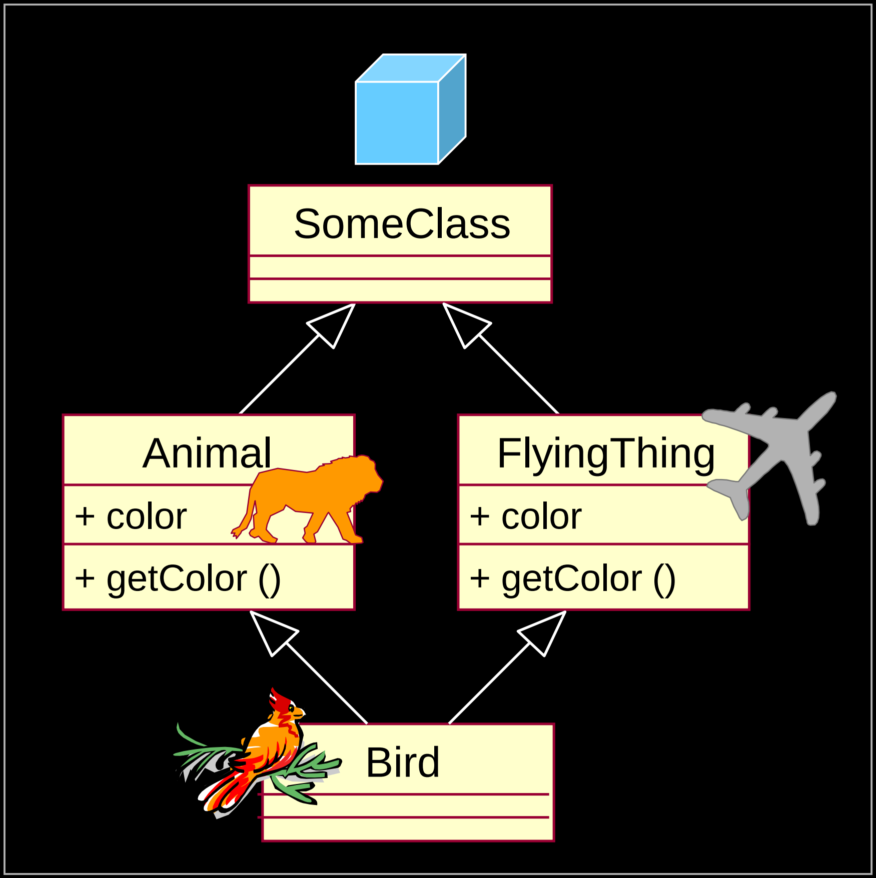


Multiple Inheritance: Problems

Name clashes on attributes or operations



Repeated inheritance



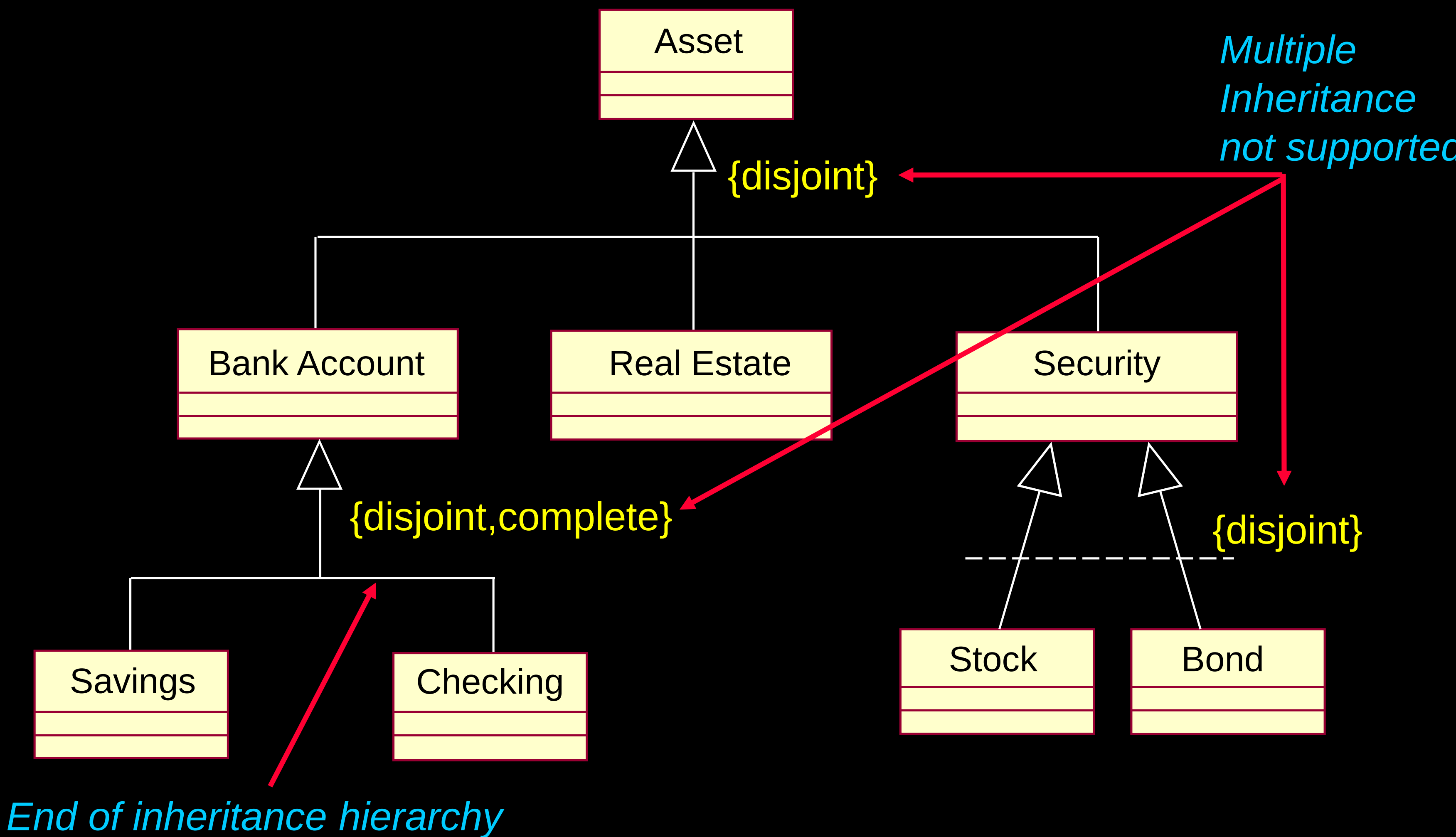
Resolution of these problems is implementation-dependent.

IBM

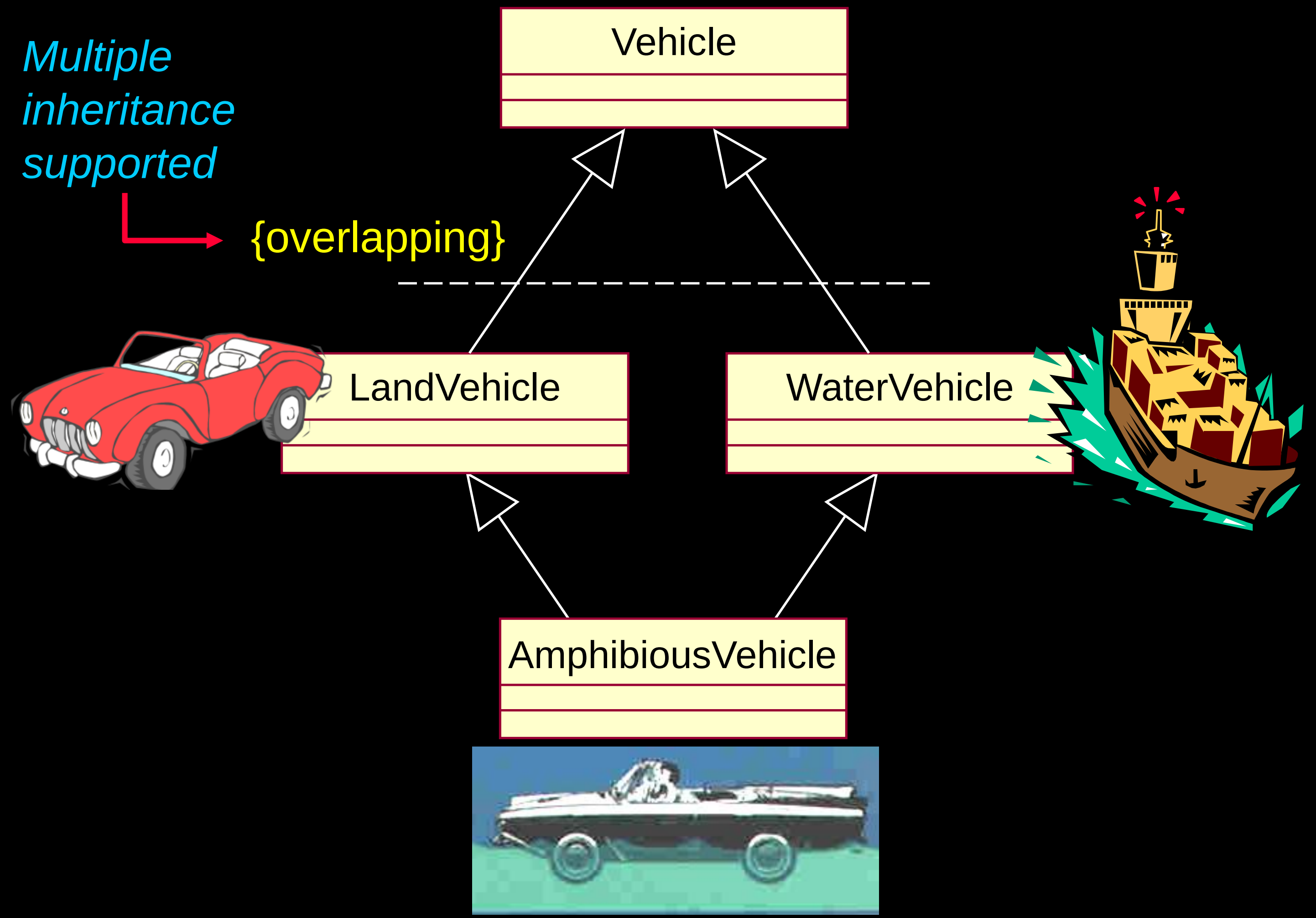
Generalization Constraints

- Complete
  - End of the inheritance tree
- Incomplete
  - Inheritance tree may be extended
- Disjoint
  - Subclasses mutually exclusive
  - Doesn’t support multiple inheritance
- Overlapping
  - Subclasses are not mutually exclusive
  - Supports multiple inheritance

Example: Generalization Constraints

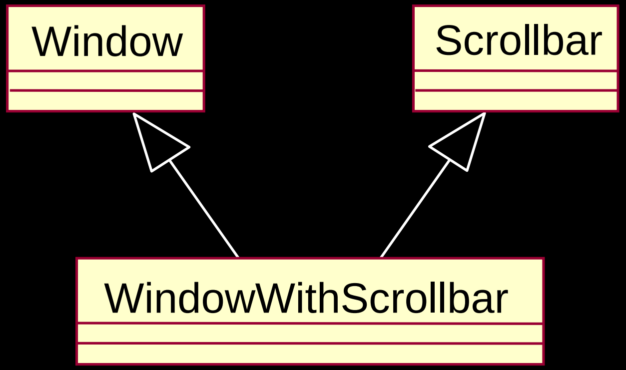


Example: Generalization Constraints (continued)



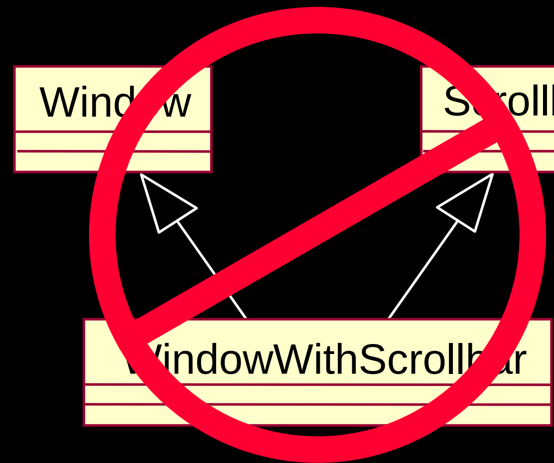
Generalization vs. Aggregation

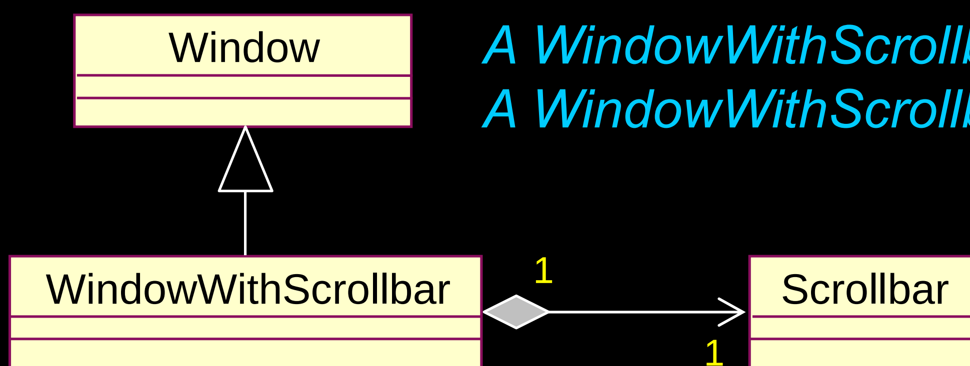
- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship



Is this correct?

Generalization vs. Aggregation

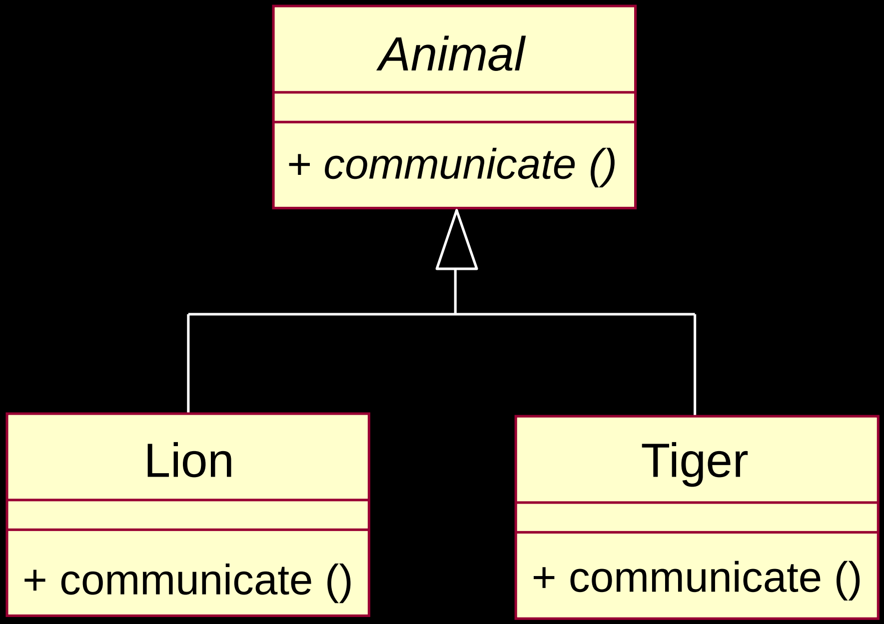


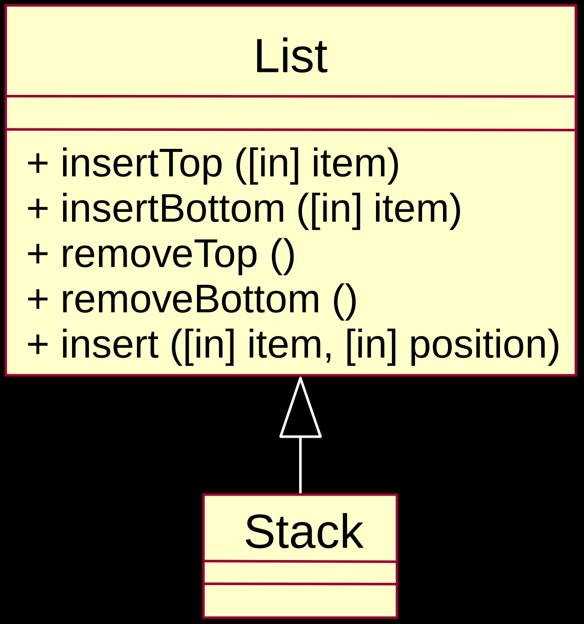


A WindowWithScrollbar “is a” Window
A WindowWithScrollbar “contains a” Scrollbar

Generalization: Share Common Properties and Behavior

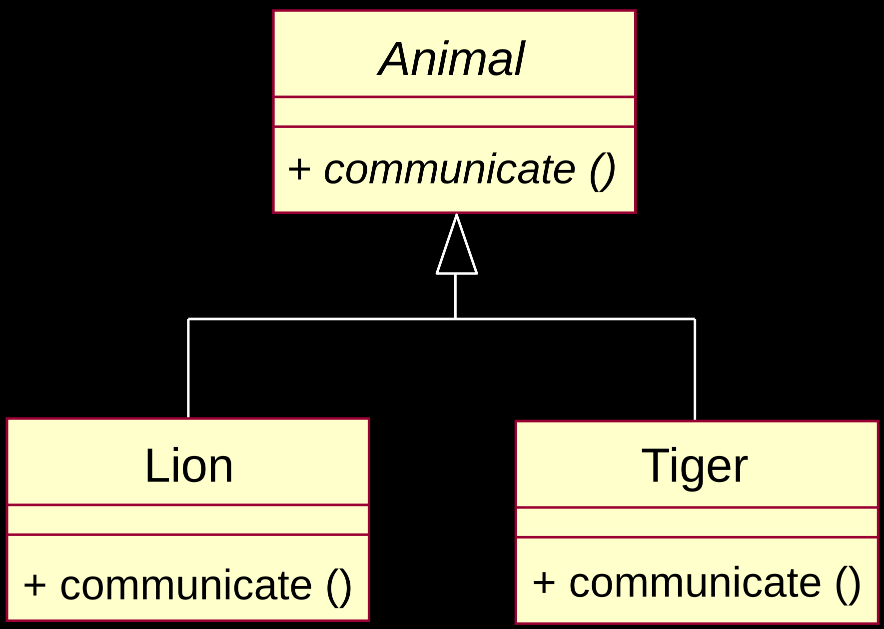
- Follows the "is a" style of programming
- Class substitutability

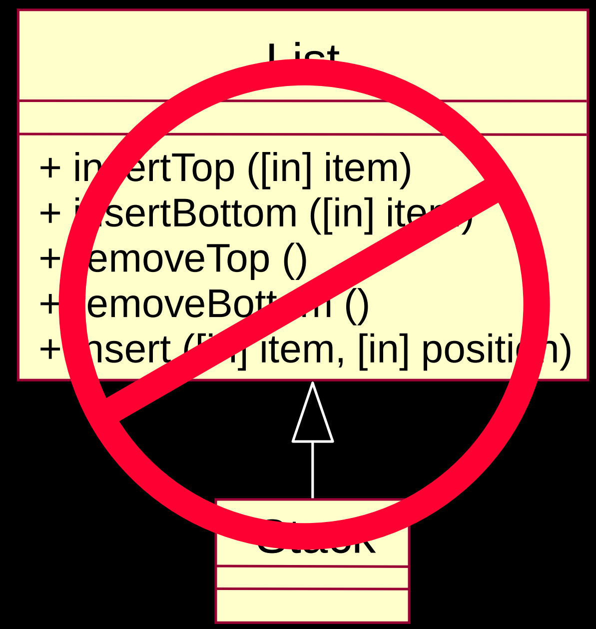




Do these classes follow the "is a" style of programming?

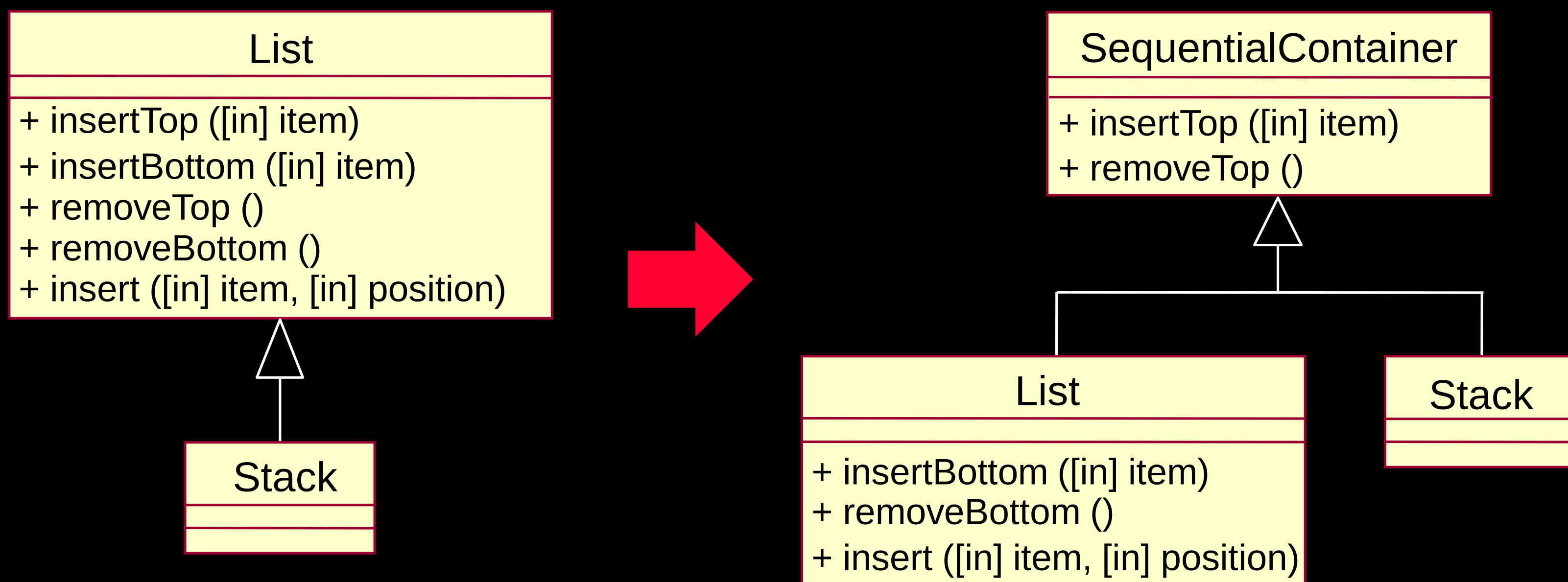
Generalization: Share Common Properties and Behavior (cont.)





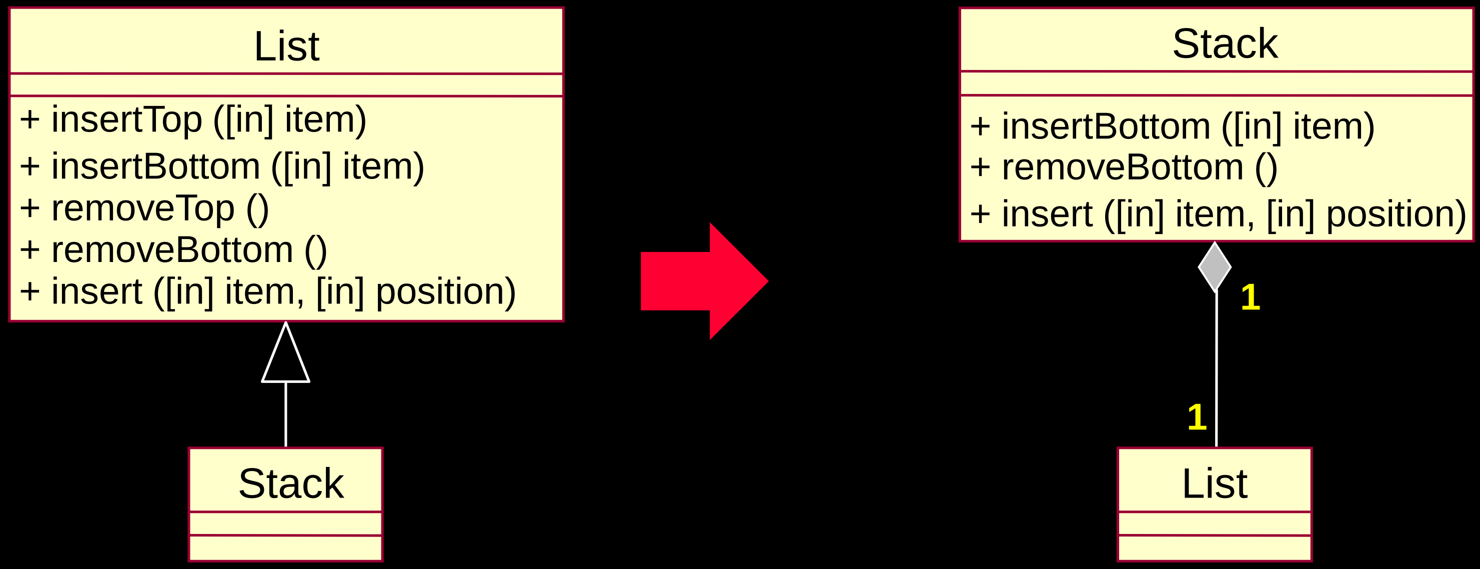
Generalization: Share Implementation: Factoring

- Supports the reuse of the implementation of another class
- Cannot be used if the class you want to "reuse" cannot be changed



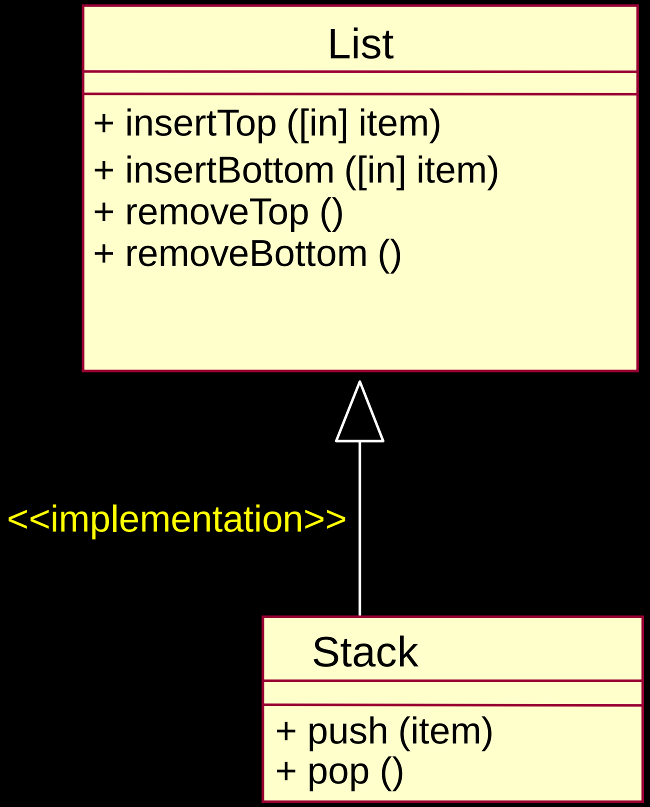
Generalization Alternative: Share Implementation: Delegation

- Supports the reuse of the implementation of another class
- Can be used if the class you want to "reuse" cannot be changed



Implementation Inheritance

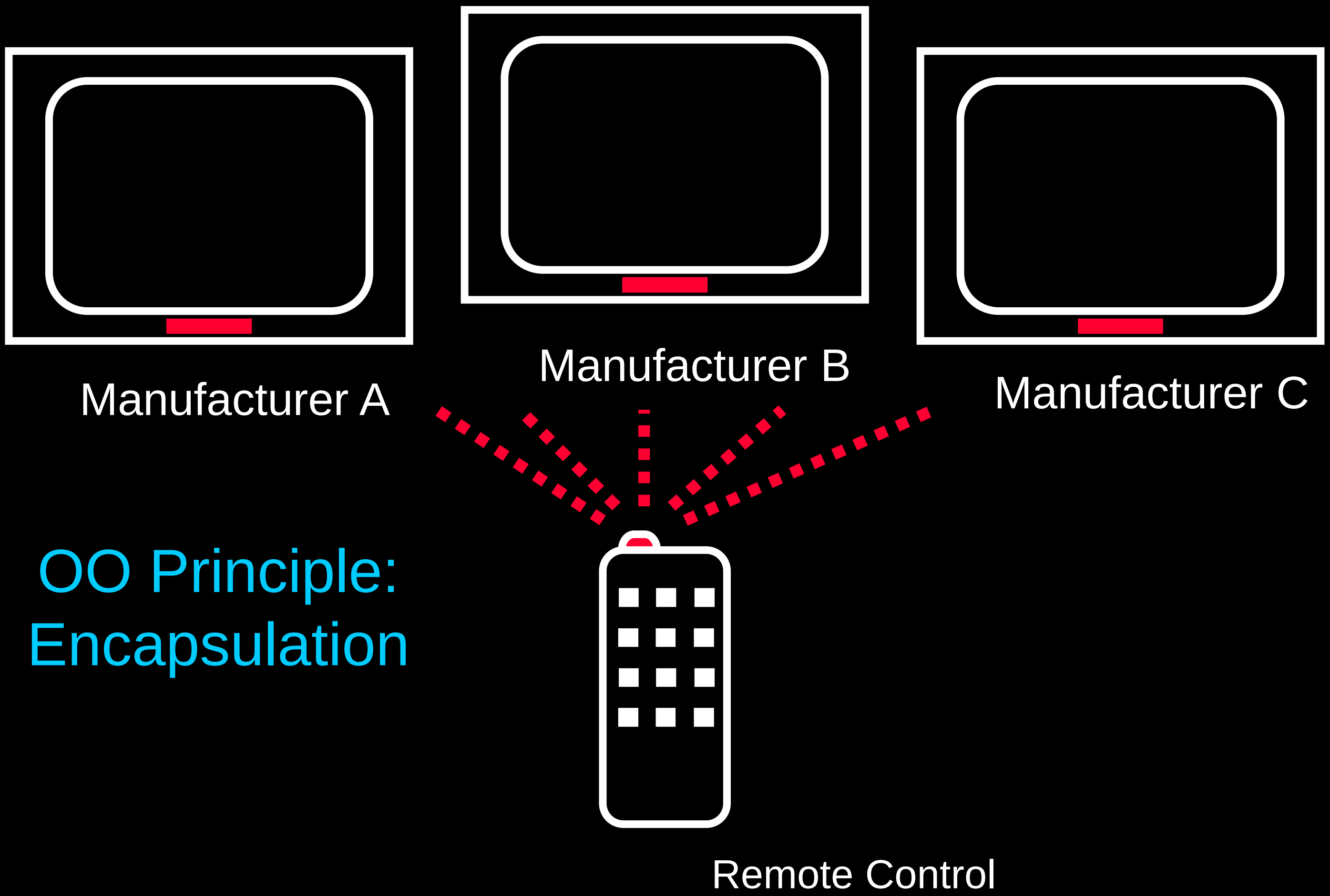
- Ancestor public operations, attributes, and relationships are NOT visible to clients of descendent class instances
- Descendent class must define all access to ancestor operations, attributes, and relationships



push() and pop() can access methods of List but instances of Stack cannot

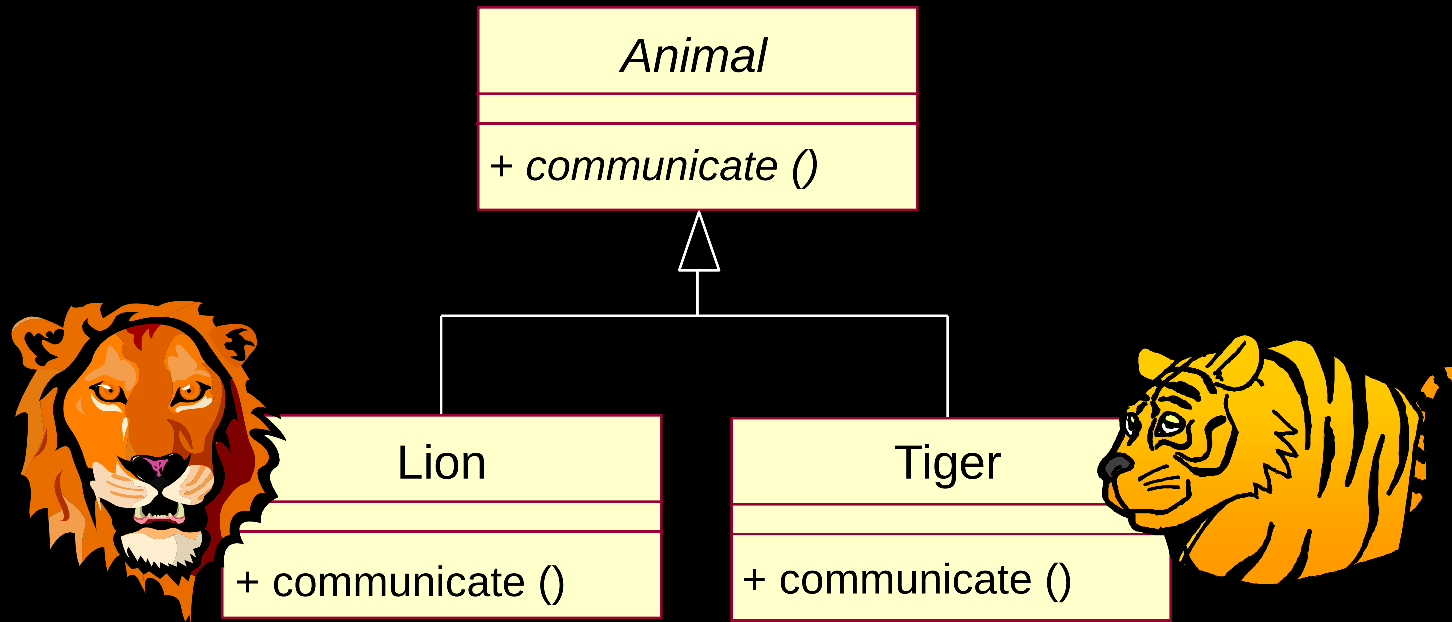
Review: What Is Polymorphism?

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



OO Principle:
Encapsulation

Generalization: Implement Polymorphism



Without Polymorphism

if animal = "Lion" then
    Lion communicate
else if animal = "Tiger" then
    Tiger communicate
end

With Polymorphism

Animal communicate

Polymorphism: Use of Interfaces vs. Generalization

- Interfaces support implementation-independent representation of polymorphism
  - Realization relationships can cross generalization hierarchies
- Interfaces are pure specifications, no behavior
  - Abstract base class may define attributes and associations
- Interfaces are totally independent of inheritance
  - Generalization is used to re-use implementations
  - Interfaces are used to re-use behavioral specifications
- Generalization provides a way to implement polymorphism

Polymorphism via Generalization Design Decisions

- Provide interface only to descendant classes?
  - Design ancestor as an abstract class
  - All methods are provided by descendent classes
- Provide interface and default behavior to descendent classes?
  - Design ancestor as a concrete class with a default method
  - Allow polymorphic operations
- Provide interface and mandatory behavior to descendent classes?
  - Design ancestor as a concrete class
  - Do not allow polymorphic operations

What Is Metamorphosis?

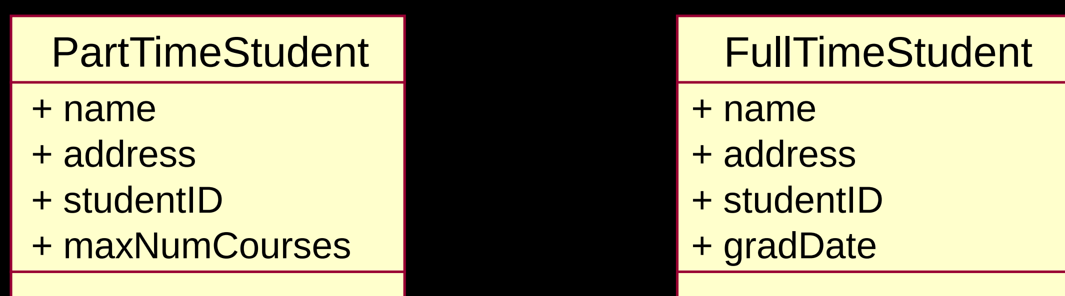
- Metamorphosis
  - 1. A change in form, structure, or function; specifically the physical change undergone by some animals, as of the tadpole to the frog.
  - 2. Any marked change, as in character, appearance, or condition.

~ Webster's New World Dictionary, Simon & Schuster, Inc.

Metamorphosis exists in the real world.
How should it be modeled?

Example: Metamorphosis

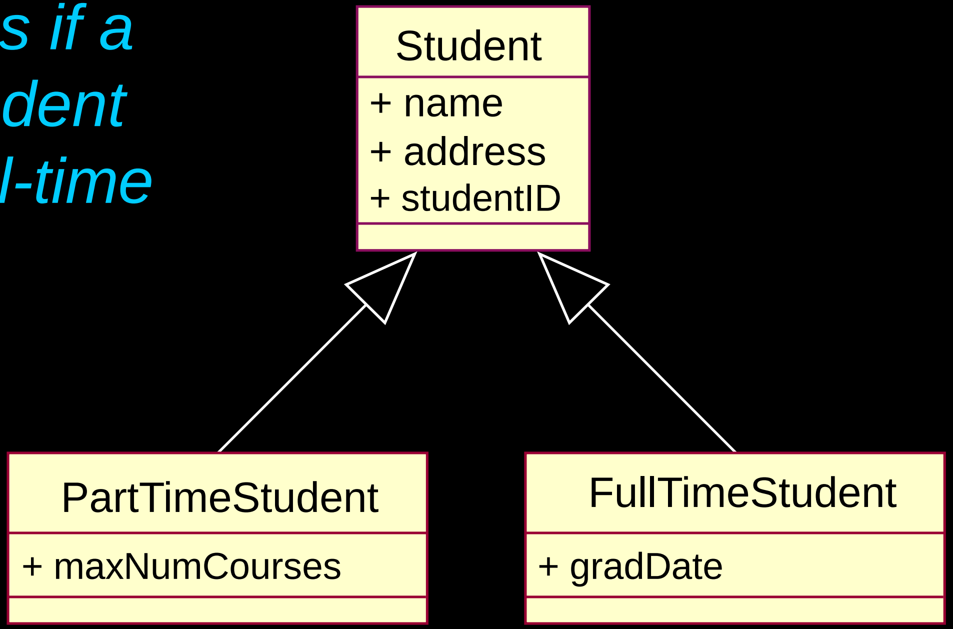
- In the university, there are full-time students and part-time students
  - Part-time students may take a maximum of three courses but there is no maximum for full-time students
  - Full-time students have an expected graduation date but part-time students do not



Modeling Metamorphosis: One Approach

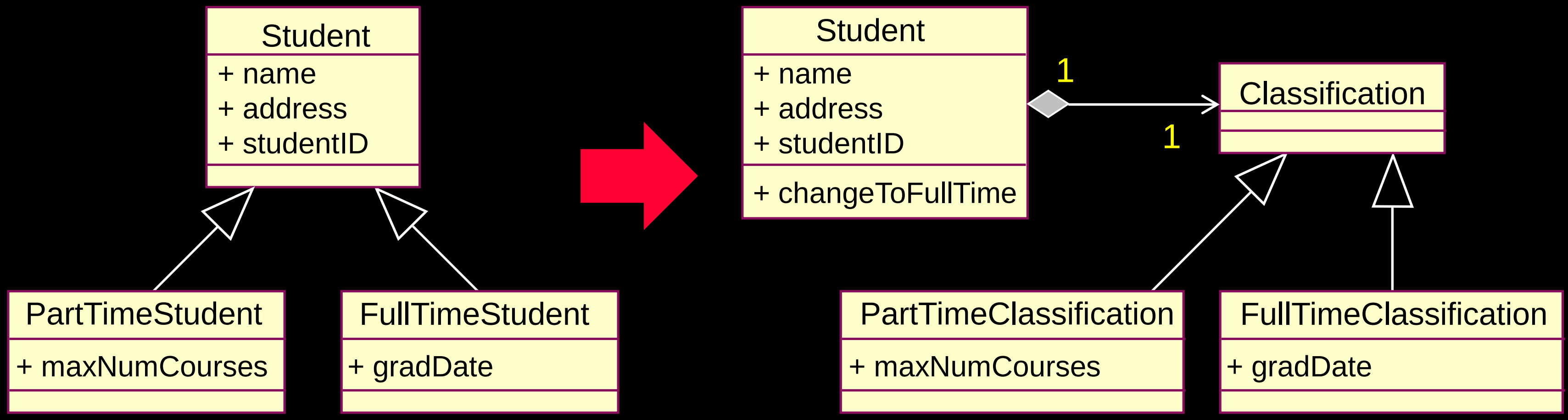
- A generalization relationship may be created

What happens if a part-time student becomes a full-time student?



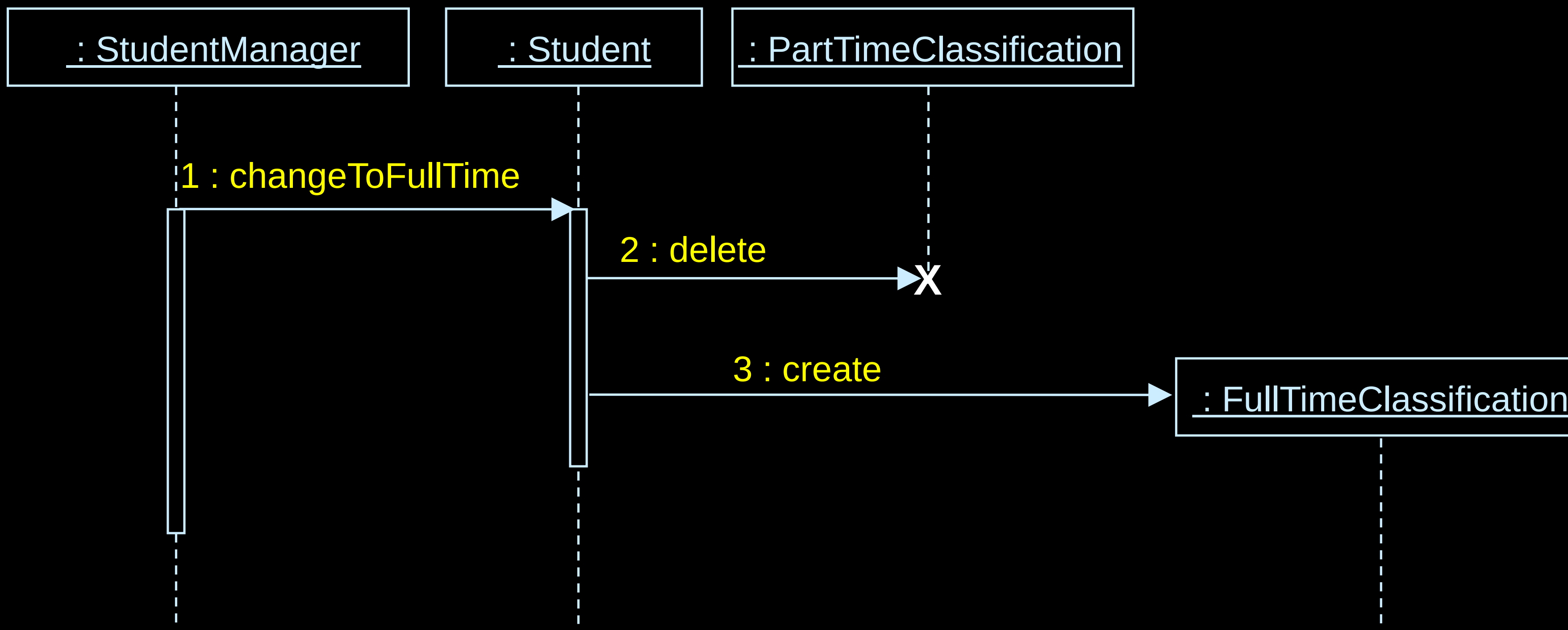
Modeling Metamorphosis: Another Approach

- Inheritance may be used to model common structure, behavior, and/or relationships to the “changing” parts



Modeling Metamorphosis: Another Approach (continued)

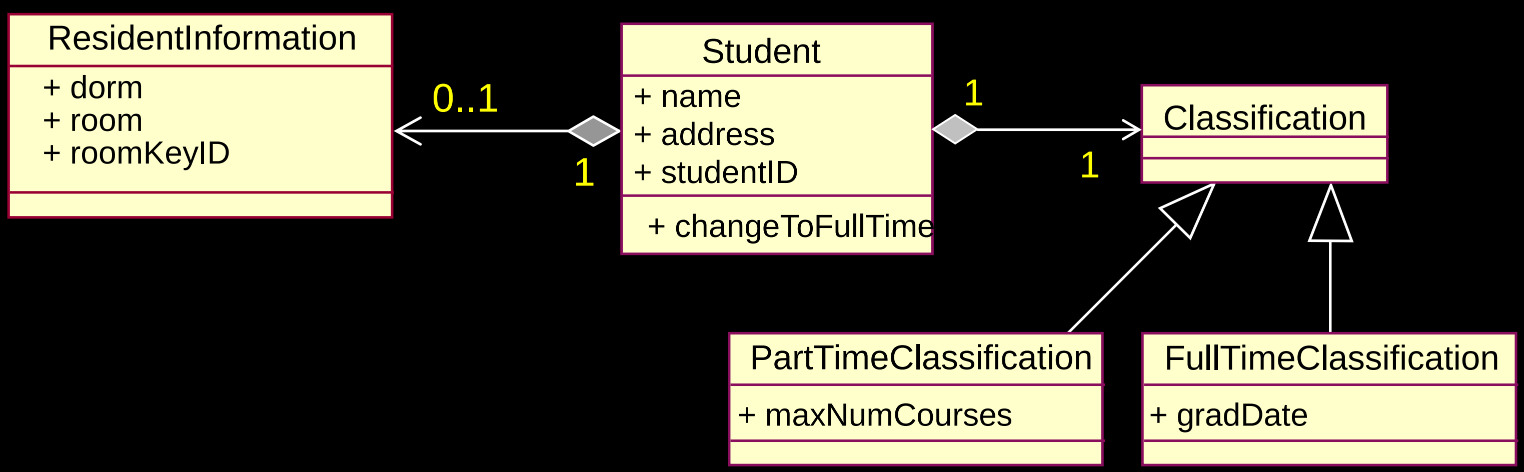
- Metamorphosis is accomplished by the object "talking" to the changing parts



IBM

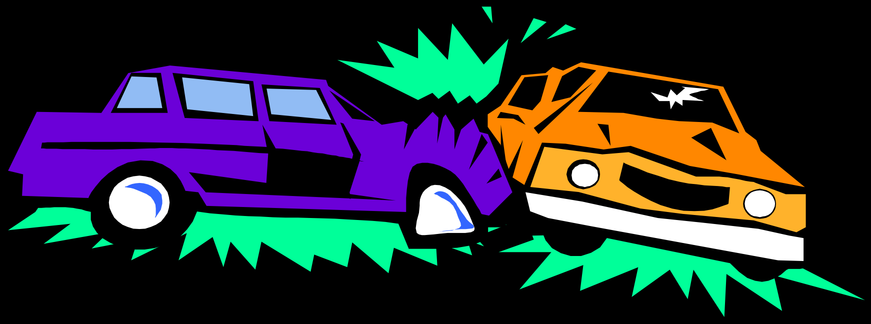
Metamorphosis and Flexibility

- This technique also adds to the flexibility of the model



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

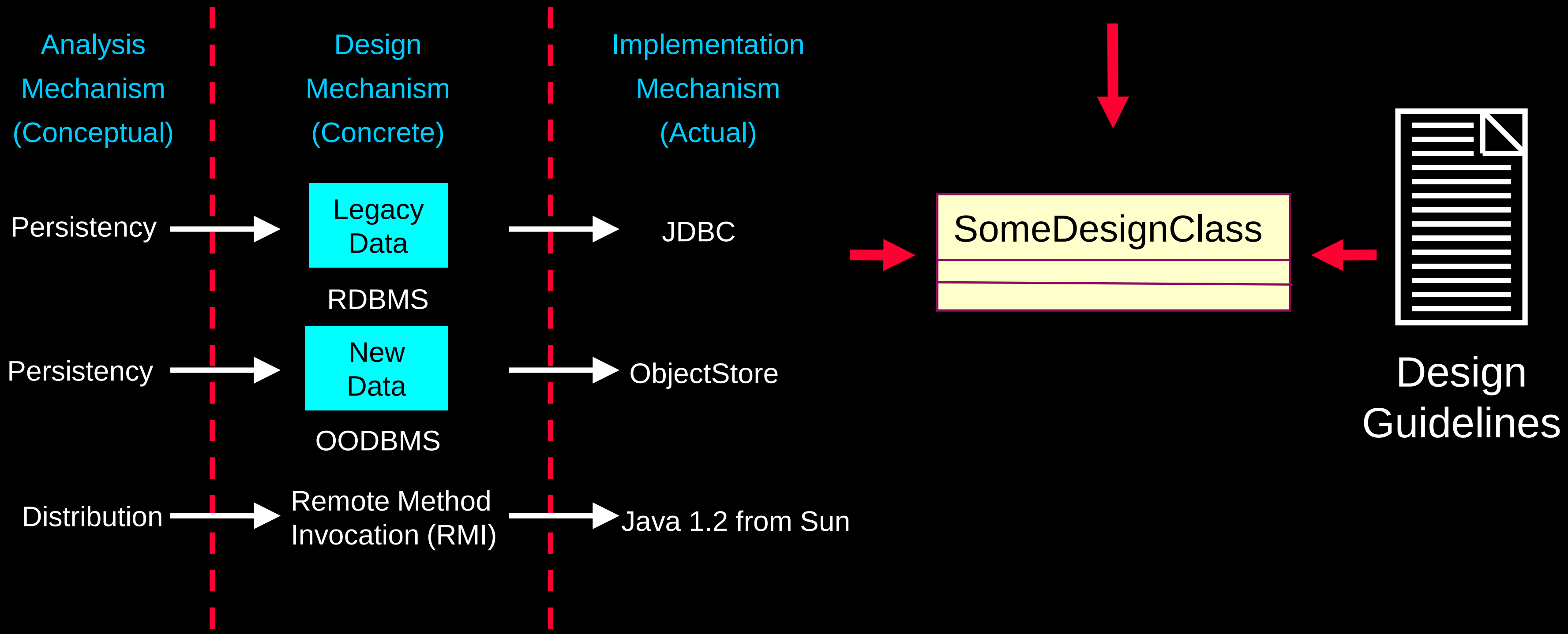
IBM

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



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

Checkpoints: Classes

- Clear class names
- One well-defined abstraction
- Functionally coupled attributes/behavior
- Generalizations were made
- All class requirements were addressed
- Demands are consistent with state machines
- Complete class instance life cycle is described
- The class has the required behavior

Checkpoints: Operations

- Operations are easily understood
- State description is correct
- Required behavior is offered
- Parameters are defined correctly
- Messages are completely assigned operations
- Implementation specifications are correct
- Signatures conform to standards
- All operations are needed by Use-Case Realizations

Checkpoints: Attributes

- A single concept
- Descriptive names
- All attributes are needed by Use-Case Realizations

Checkpoints: Relationships

- Descriptive role names
- Correct multiplicities

Review: Class Design

- What is the purpose of Class Design?
- In what ways are classes refined?
- Are state machines created for every class?
- What are the major components of a state machine? Provide a brief description of each.
- What is the difference between a dependency and an association?
- What is a structured class? What is a connector?

Exercise 2: Class Design

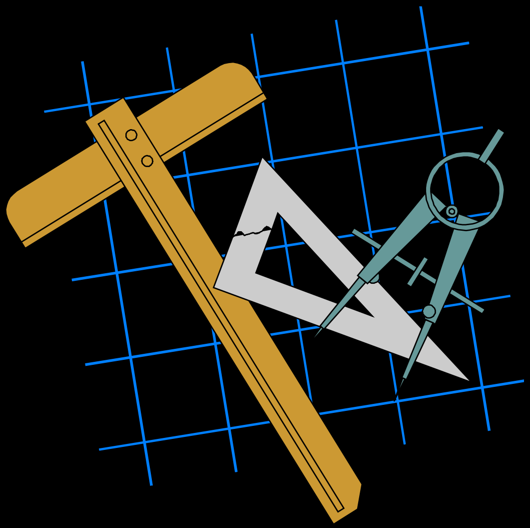
- Given the following:
  - The Use-Case Realization for a use case and/or the detailed design of a subsystem
    - Payroll Exercise Solution, Exercise: Use-Case Design, Part 1
  - The design of all participating design elements
    - Payroll Exercise Solution, Exercise: Subsystem Design

Exercise 2: Class Design (continued)

- Identify the following:
  - The required navigability for each relationship
  - Any additional classes to support the relationship design
  - Any associations refined into dependencies
  - Any associations refined into aggregations or compositions
  - Any refinements to multiplicity
  - Any refinements to existing generalizations
  - Any new applications of generalization
    - Make sure any metamorphosis is considered

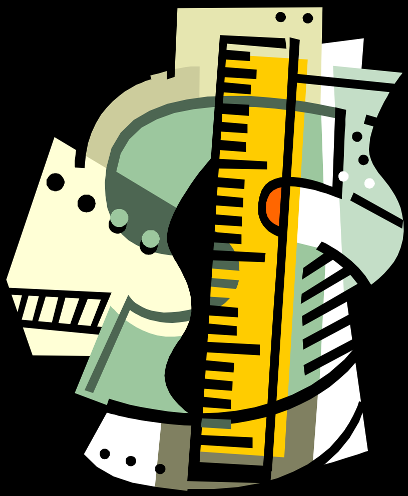
Exercise 2: Class Design (continued)

- Produce the following:
  - An updated VOPC, including the relationship refinements (generalization, dependency, association)



Exercise 2: Review

- Compare your results
  - Do your dependencies represent context independent relationships?
  - Are the multiplicities on the relationships correct?
  - Does the inheritance structure capture common design abstractions, and not implementation considerations?
  - Is the obvious commonality reflected in the inheritance hierarchy?



Payroll System