speed quality

Rational®

the software development company

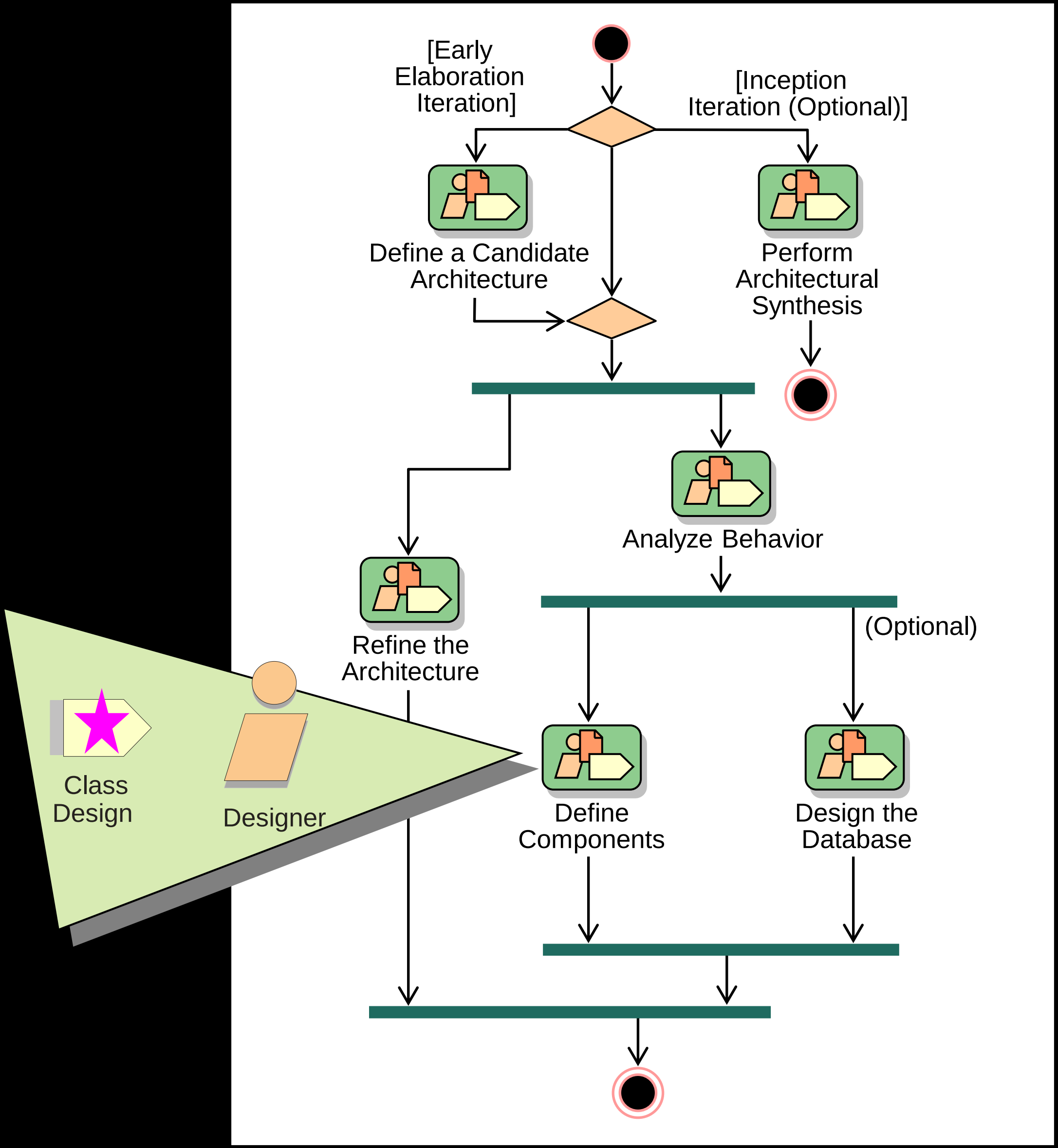
Mastering Object-Oriented Analysis and Design with UML

Module 13: Class Design

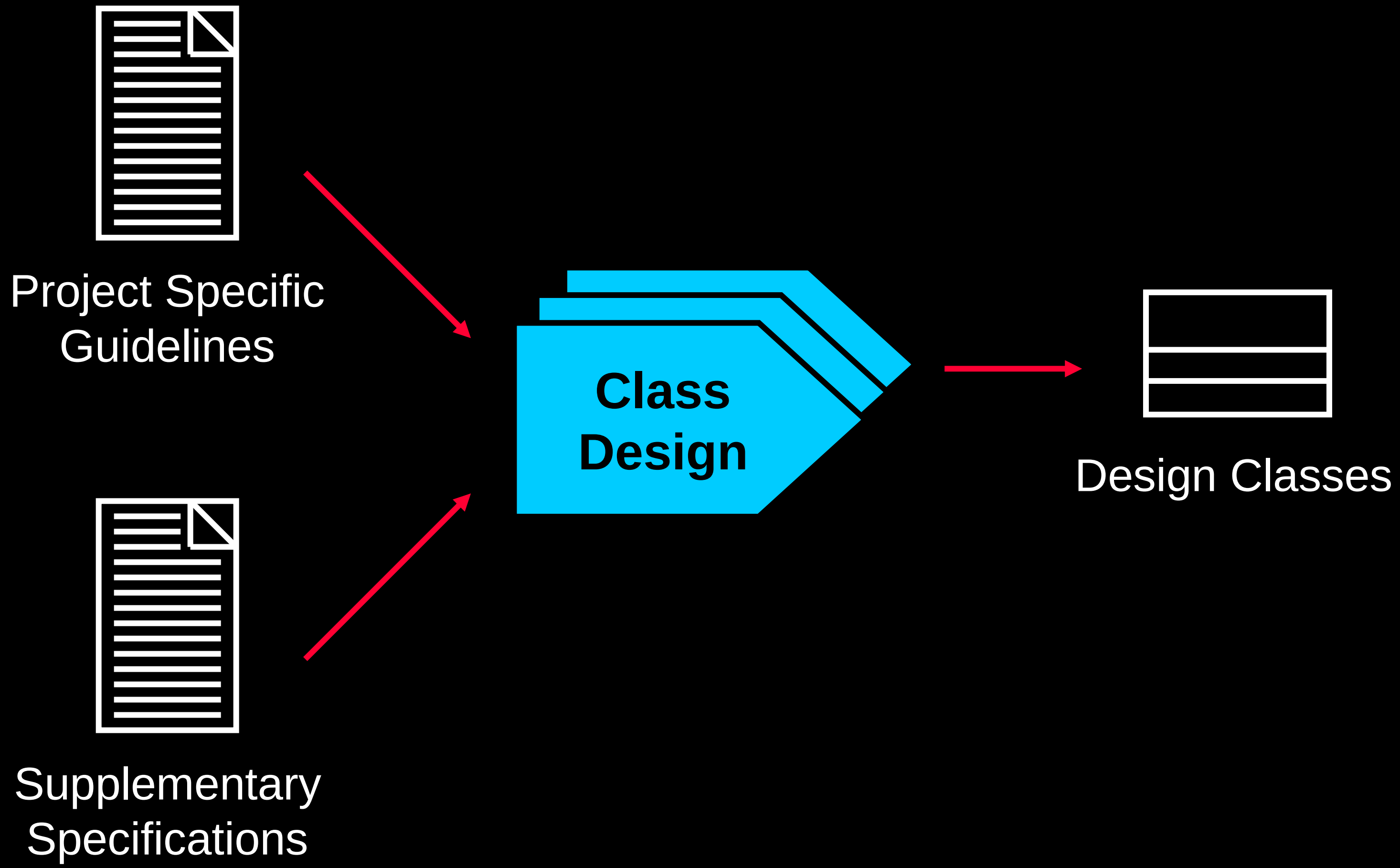
Objectives: Class Design

- 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

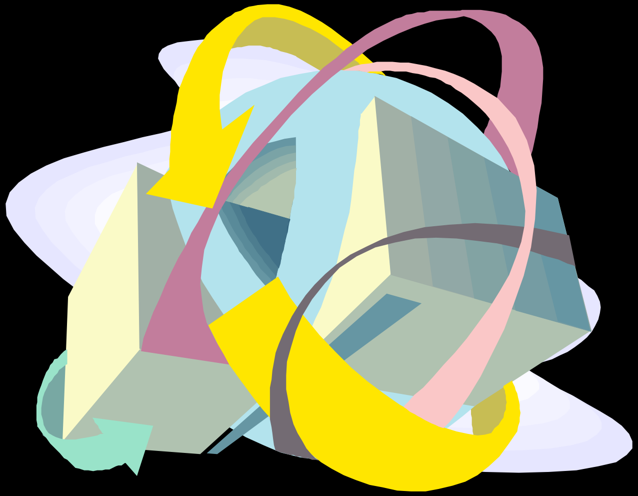

[Early Elaboration Iteration]
[Inception Iteration (Optional)]
Define a Candidate Architecture
Perform Architectural Synthesis
Analyze Behavior
Refine the Architecture
(Optional)
Class Design
Designer
Define Components
Design the Database

Class Design Overview



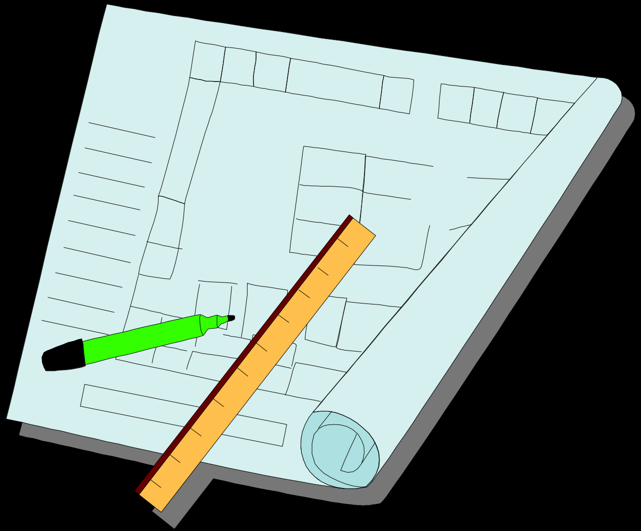
Class Design Steps

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



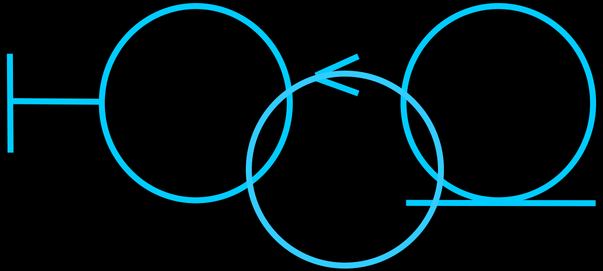
Class Design Steps

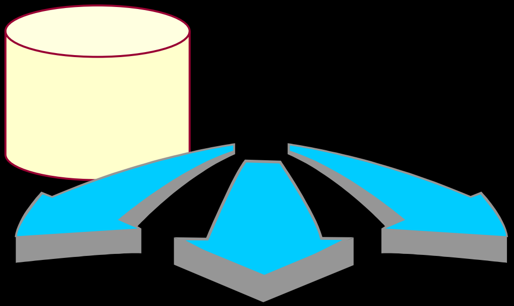
- ★ Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Class Design Considerations

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





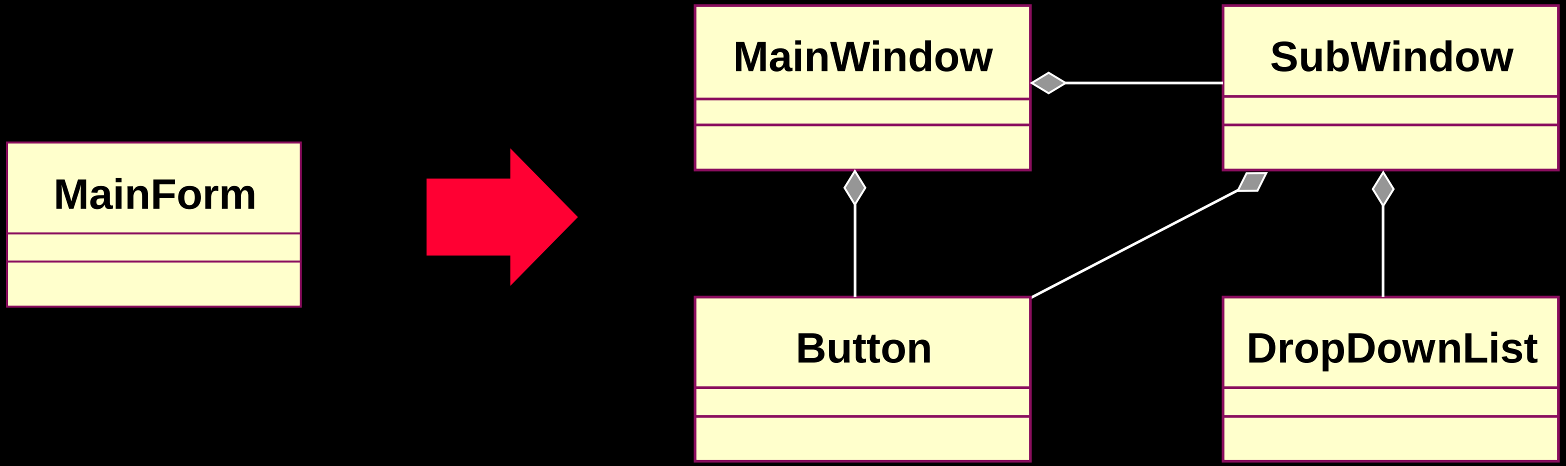
How Many Classes Are Needed?

- 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!

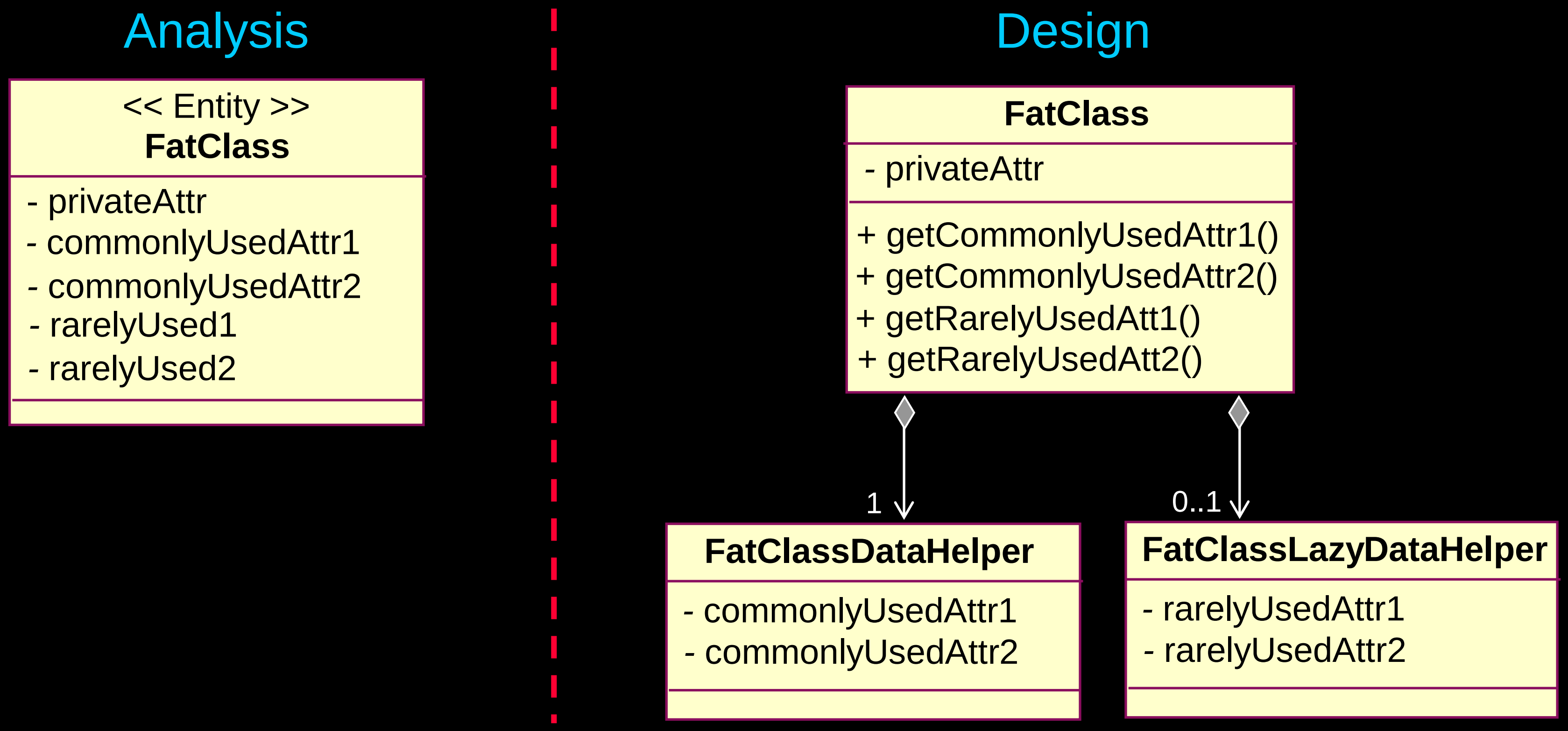
Strategies for Designing Boundary Classes

- 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



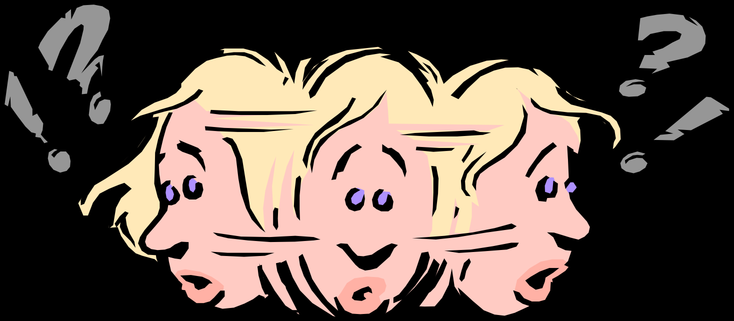
Strategies for Designing Entity Classes

- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring
- See the Identify Persistent Classes step



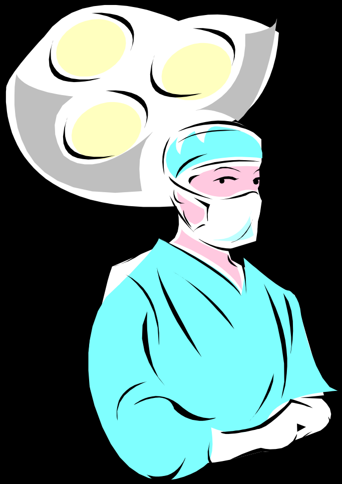
Strategies for Designing Control Classes

- 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



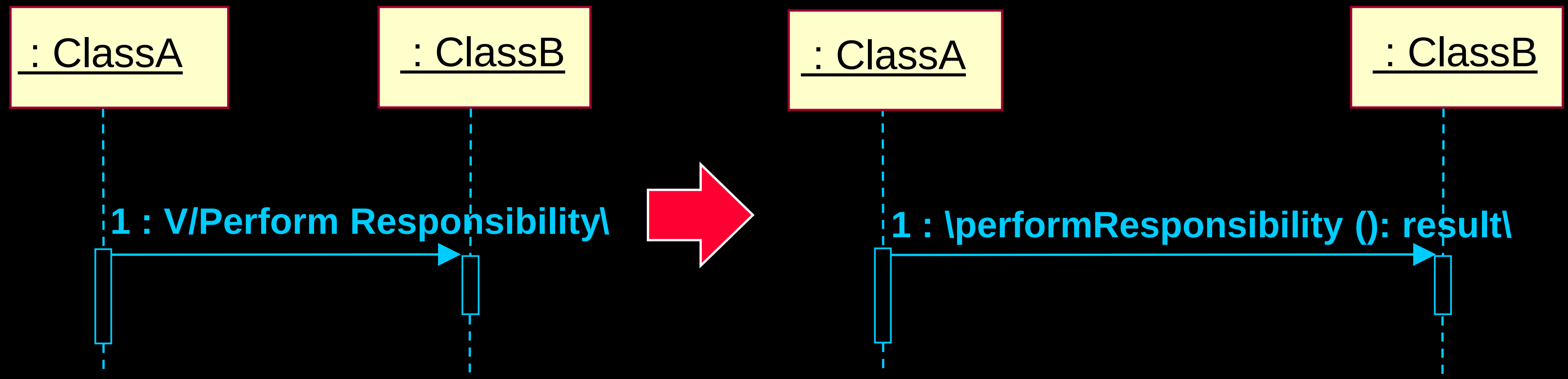
Class Design Steps

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



Operations: Where Do You Find Them?

- Messages displayed in interaction diagrams


: ClassA
: ClassB
1 : V/Perform Responsibility\
: ClassA
: ClassB
1 : \performResponsibility (): result\

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

Name and Describe the Operations

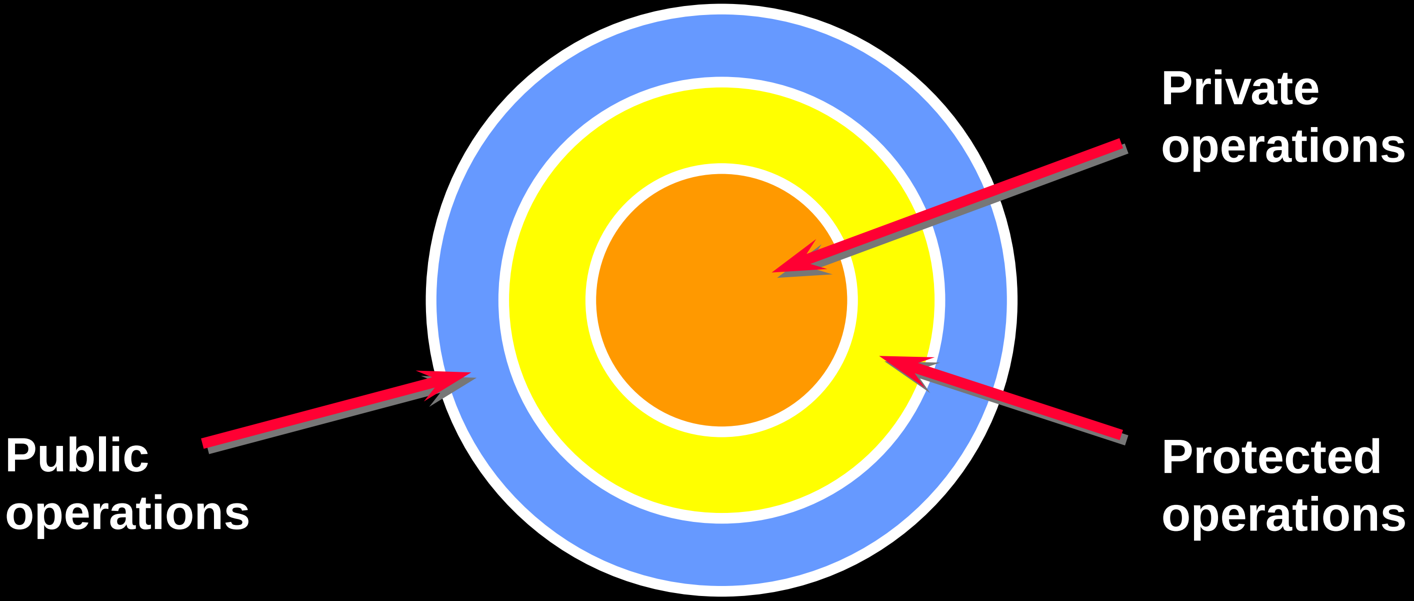
- Create appropriate operation names
  - Indicate the outcome
  - Use client perspective
  - Are consistent across classes
- Define operation signatures
  - operationName([direction]parameter : class,..) : returnType
    - Direction is **in**, **out** or **inout** with the default **in** if absent
- 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”

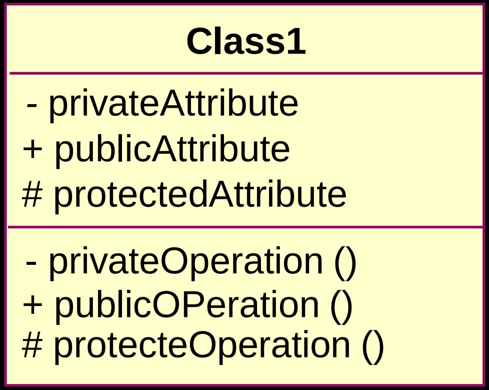
Operation Visibility

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



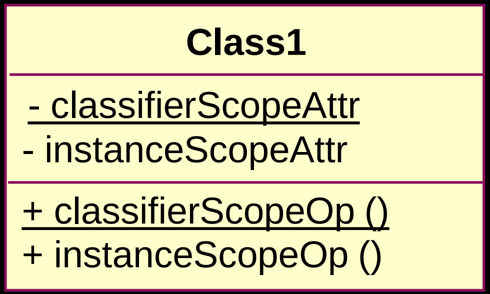
How Is Visibility Noted?

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

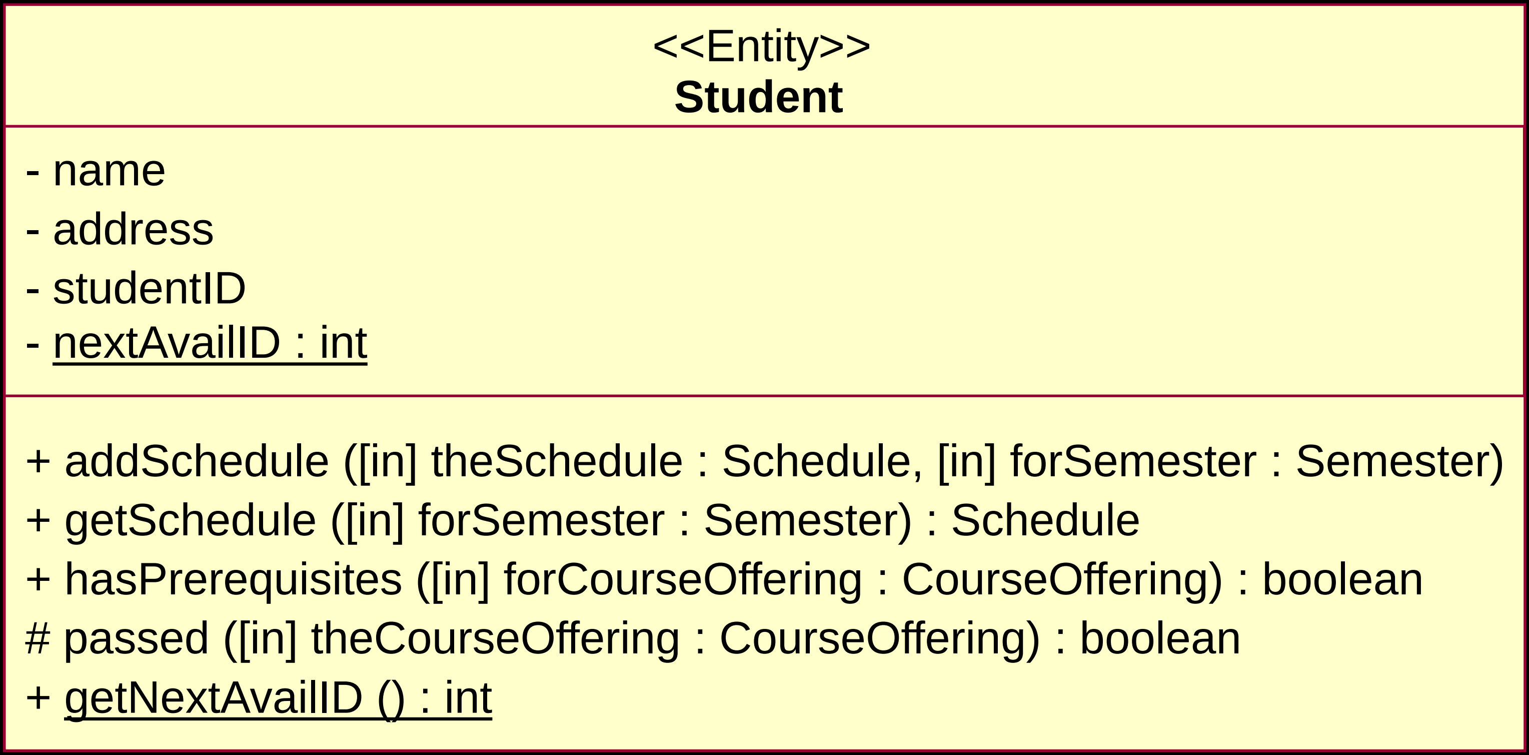


Scope

- 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

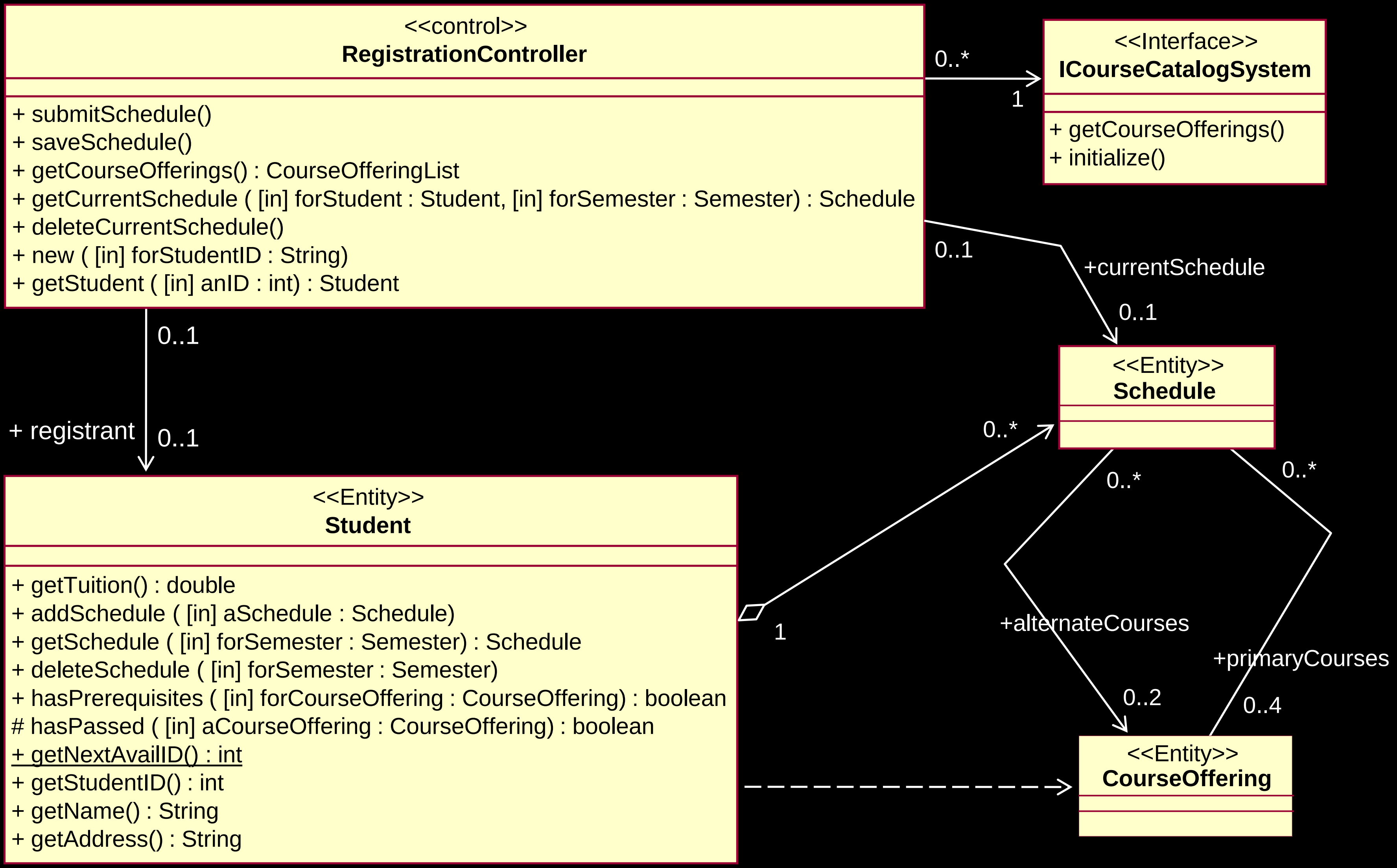


Example: Scope



<<Entity>>
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



<<control>>
RegistrationController
+ submitSchedule()
+ saveSchedule()
+ getCourseOfferings() : CourseOfferingList
+ getCurrentSchedule ([in] forStudent : Student, [in] forSemester : Semester) : Schedule
+ deleteCurrentSchedule()
+ new ([in] forStudentID : String)
+ getStudent ([in] anID : int) : Student
0..*
1
<<Interface>>
ICourseCatalogSystem
+ getCourseOfferings()
+ initialize()
0..1
+currentSchedule
0..1
<<Entity>>
Schedule
0..1
+ registrant
0..1
<<Entity>>
Student
+ getTuition() : double
+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ deleteSchedule ([in] forSemester : Semester)
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# hasPassed ([in] aCourseOffering : CourseOffering) : boolean
+ getNextAvailID() : int
+ getStudentID() : int
+ getName() : String
+ getAddress() : String
0..*
1
0..*
0..*
+alternateCourses
+primaryCourses
0..2
0..4
<<Entity>>
CourseOffering

Rational
the software development company

Class Design Steps

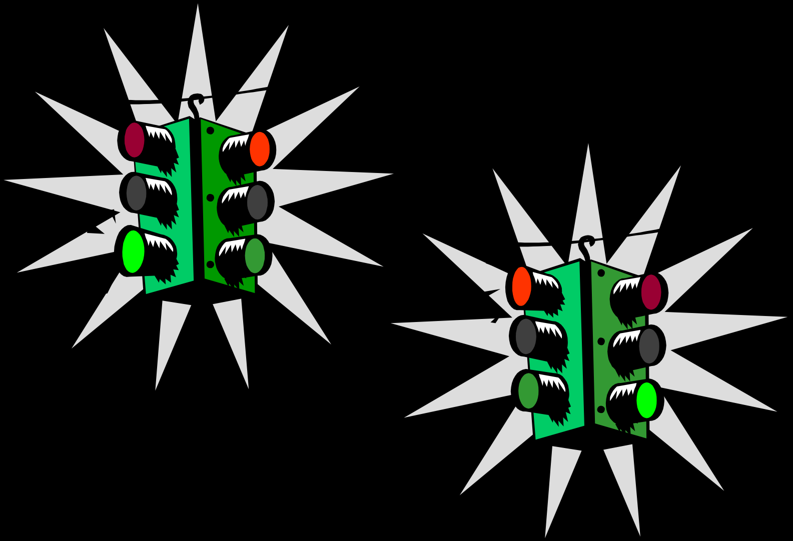
- Create Initial Design Classes
- Define Operations
- ★ Define Methods
- Define States
- Define Attributes
- Define Dependencies
- Define Associations
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

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 Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

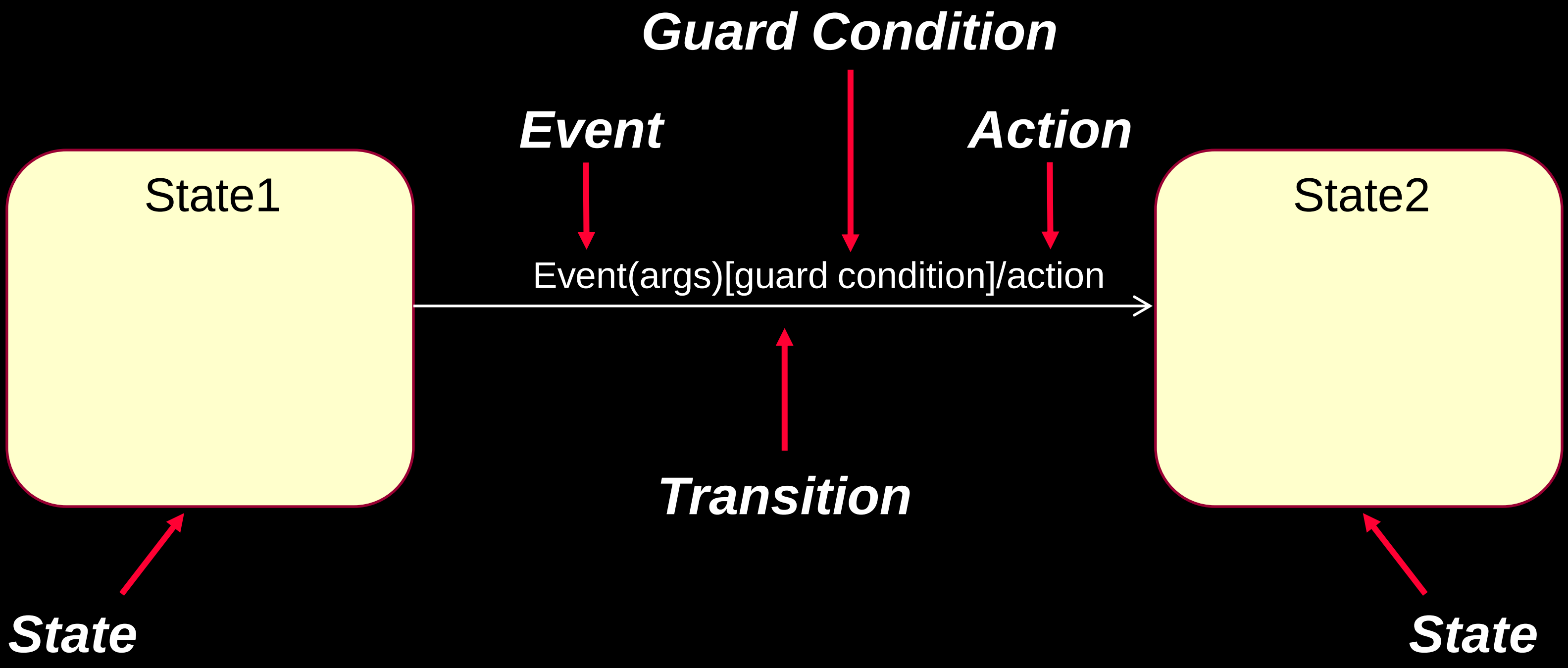


Define States

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

What is a Statechart?

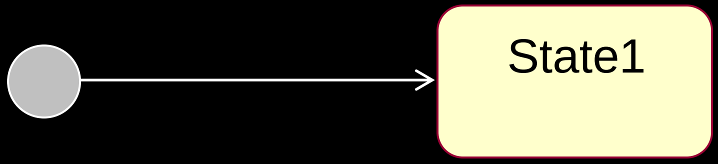
- A directed graph of states (nodes) connected by transitions (directed arcs)
- Describes the life history of a reactive object

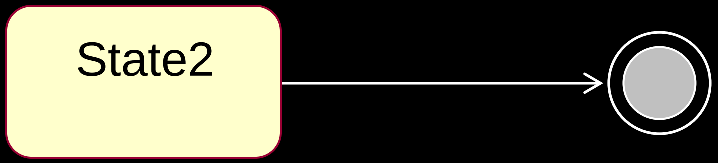


Special States

- Initial state
  - The state entered when an object is created
  - Mandatory
  - Can only have one initial state

- Final state
  - Indicates the object's end of life
  - Optional
  - May have more than one

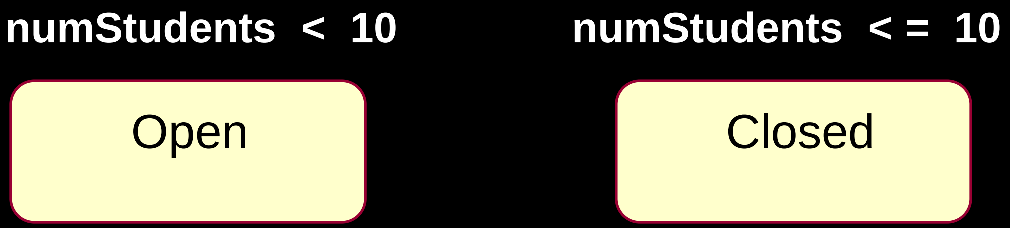




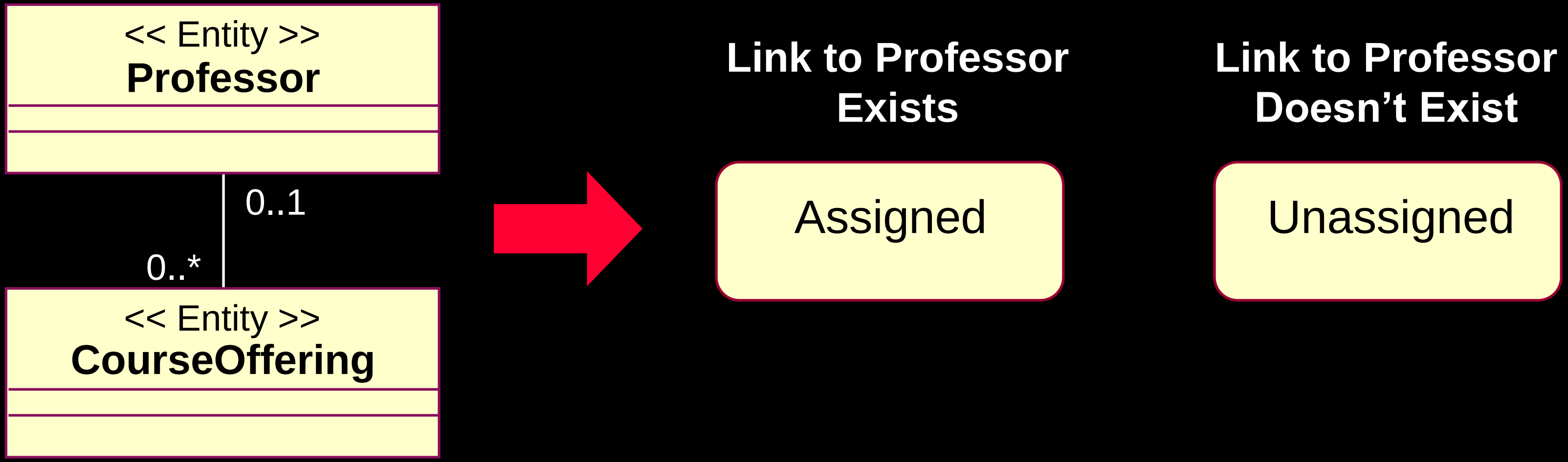
Identify and Define the States

- Significant, dynamic attributes

The maximum number of students per course offering is 10

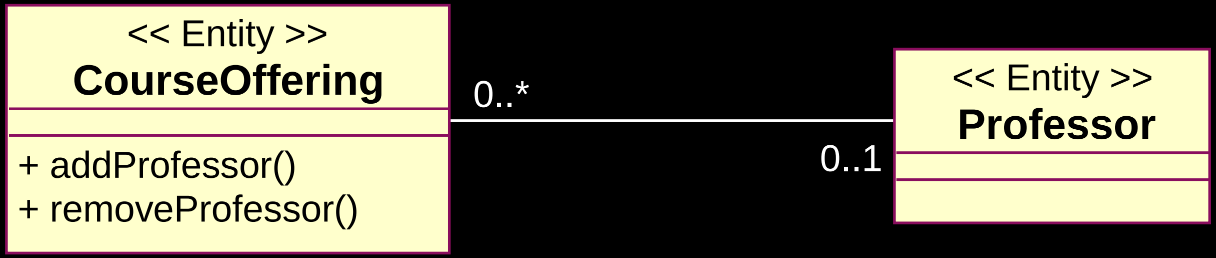


- Existence and non-existence of certain links



Identify the Events

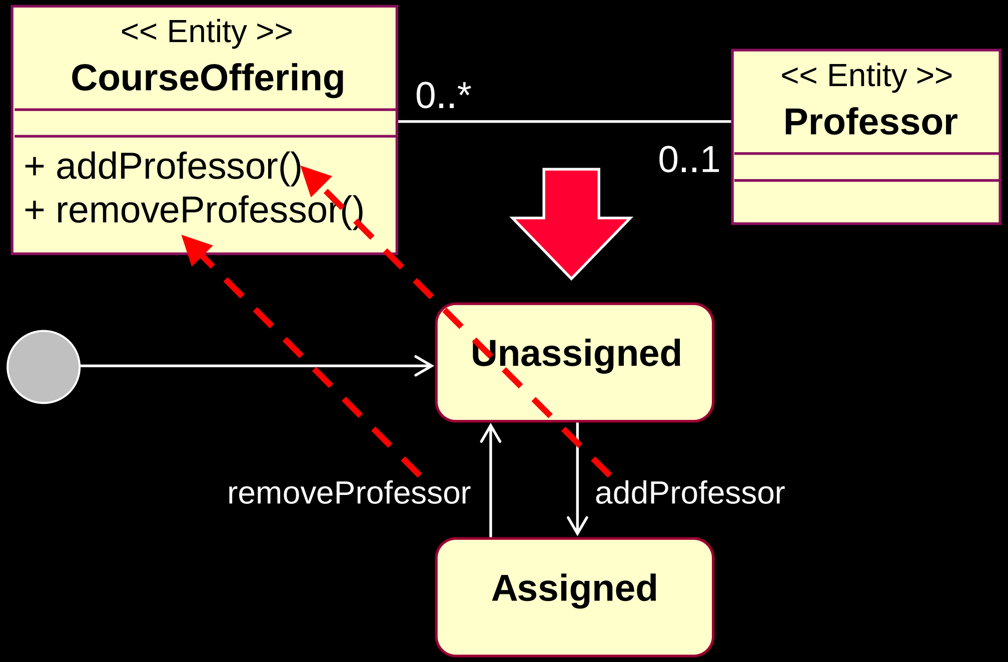
- Look at the class interface operations



Events: addProfessor, removeProfessor

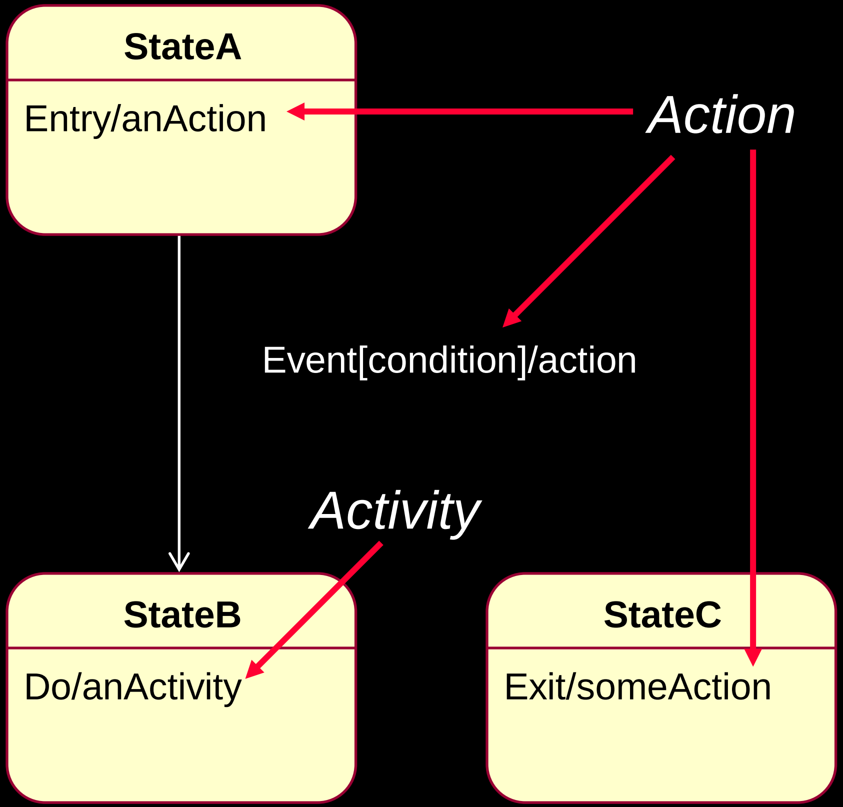
Identify the Transitions

- 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

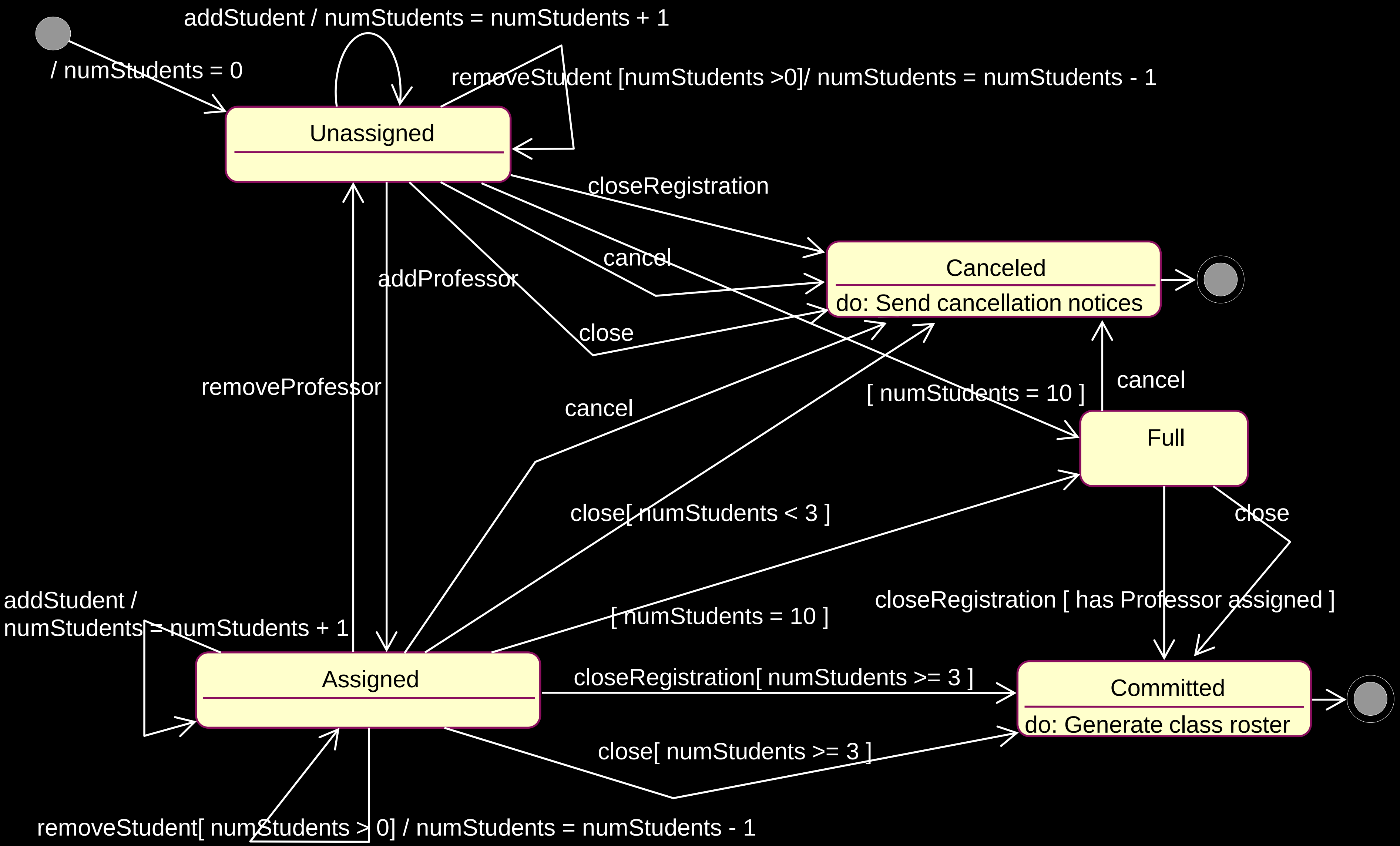


Add Activities and Actions

- Activities
  - Are associated with a state
  - Start when the state is entered
  - Take time to complete
  - Interruptible
- Actions
  - Associated with a transition
  - Take an insignificant amount of time to complete
  - Are non-interruptible

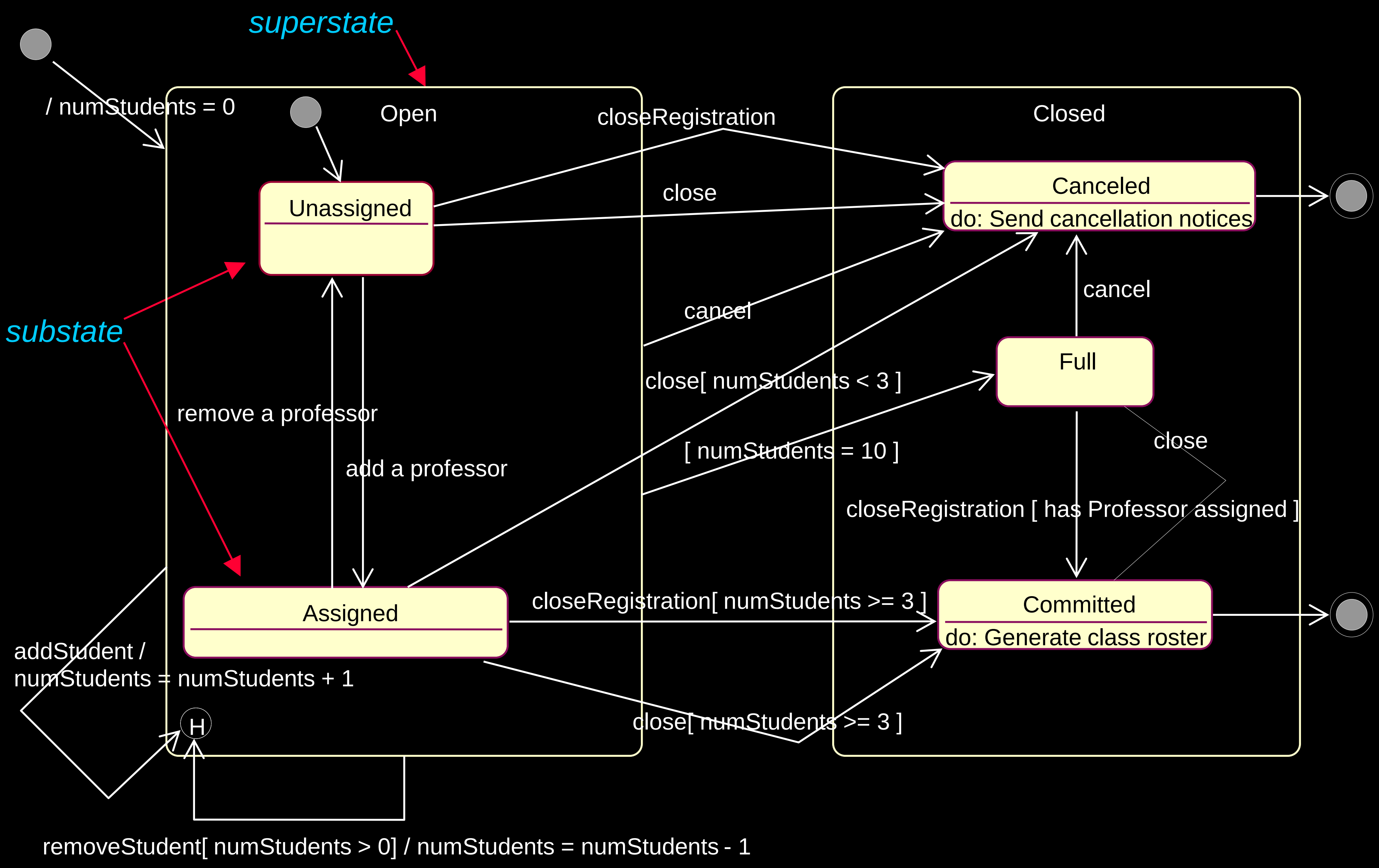


Example: Statechart



addStudent / numStudents = numStudents + 1
/ numStudents = 0
removeStudent [numStudents >0]/ numStudents = numStudents - 1
Unassigned
closeRegistration
cancel
addProfessor
close
Canceled
do: Send cancellation notices
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: Statechart with Nested States and History

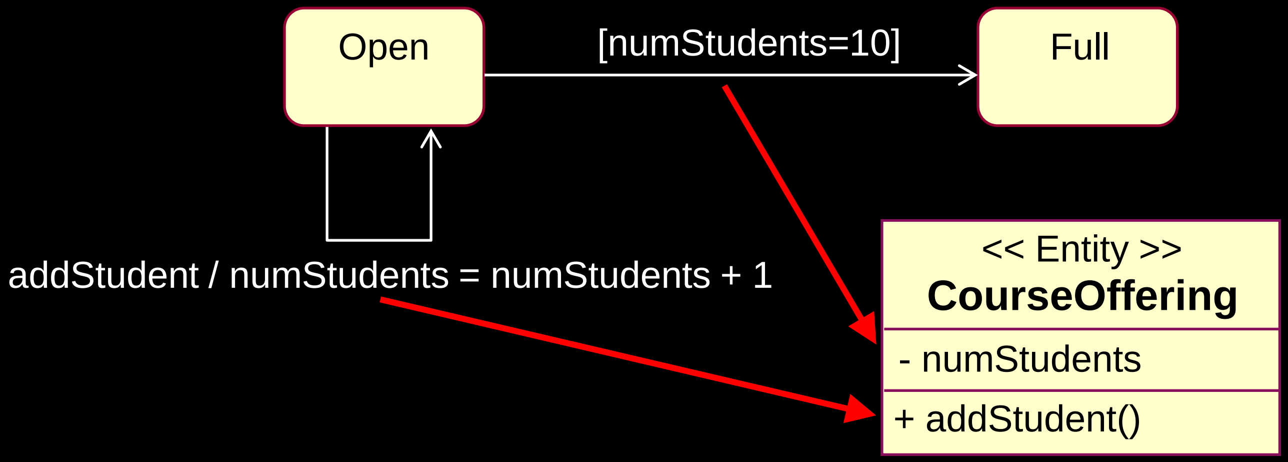

superstate
/ numStudents = 0
Open
closeRegistration
Closed
Unassigned
close
Canceled
do: Send cancellation notices
cancel
substate
cancel
Full
close[numStudents < 3]
remove a professor
[numStudents = 10]
close
add a professor
closeRegistration [has Professor assigned]
closeRegistration[numStudents >= 3]
Assigned
Committed
do: Generate class roster
addStudent /
numStudents = numStudents + 1
H
close[numStudents >= 3]
removeStudent[numStudents > 0] / numStudents = numStudents - 1

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 Statecharts 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



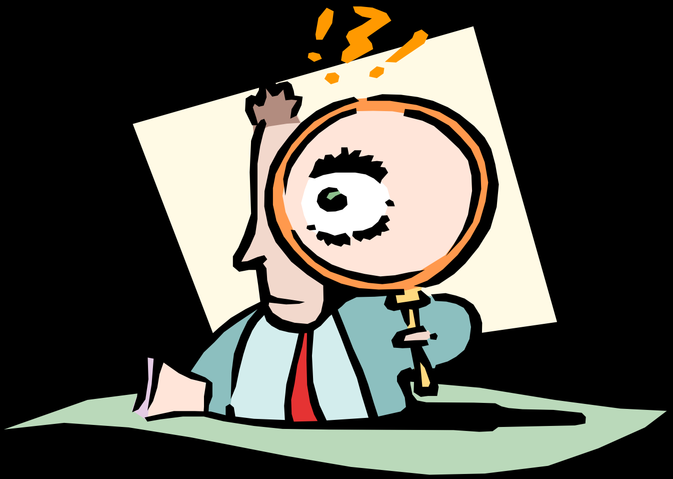
(Stay tuned for derived attributes)

Class Design Steps

- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- ★ Define Attributes
- Define Dependencies
- Define Associations
- 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



Rational the software development company

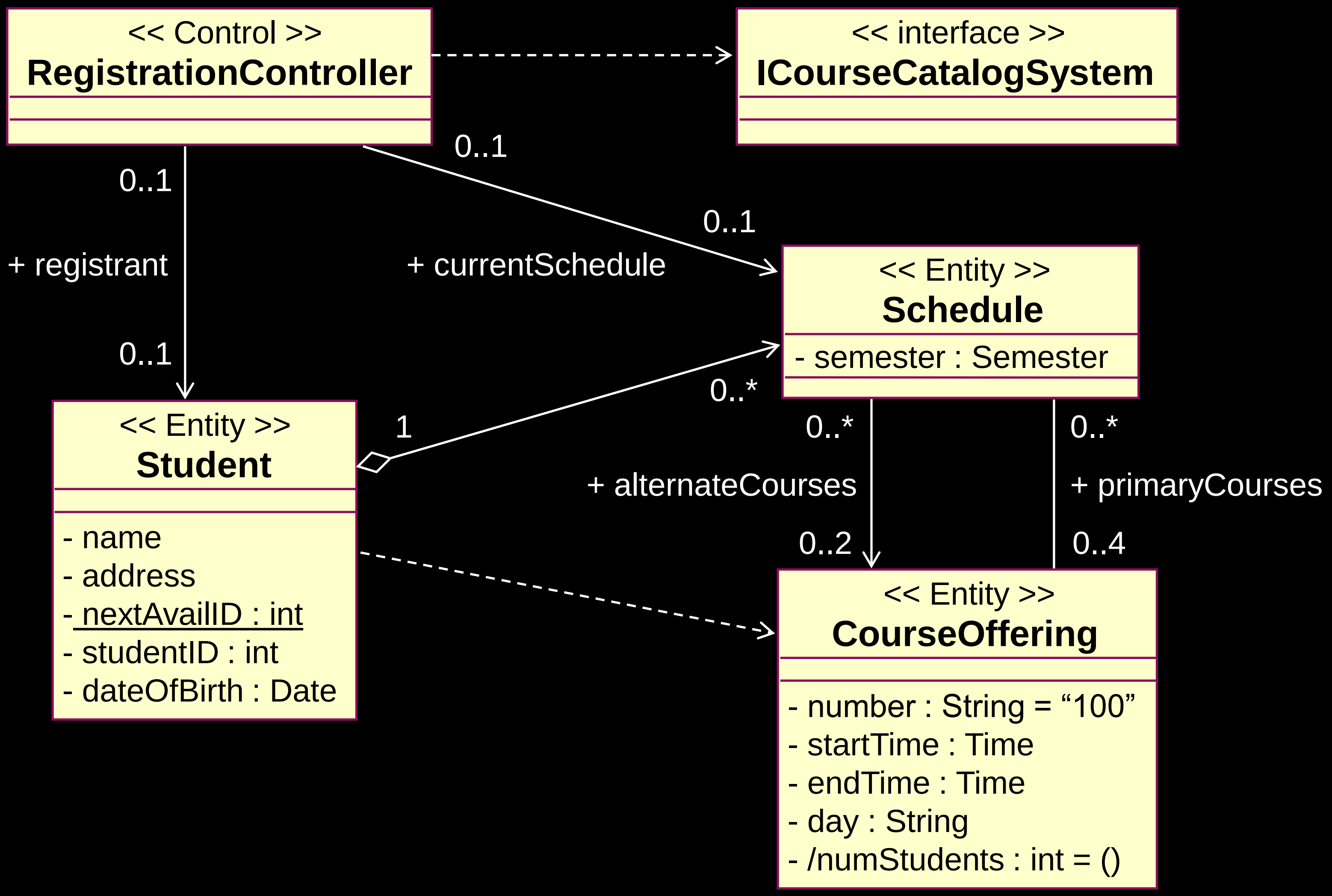
Attribute Representations

- 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

Example: Define Attributes



Exercise 1: Class Design

- Given the following:
  - The architectural layers, their packages, and their dependencies
  - Design classes for a particular use case

(continued)

Exercise 1: Class Design (cont.)

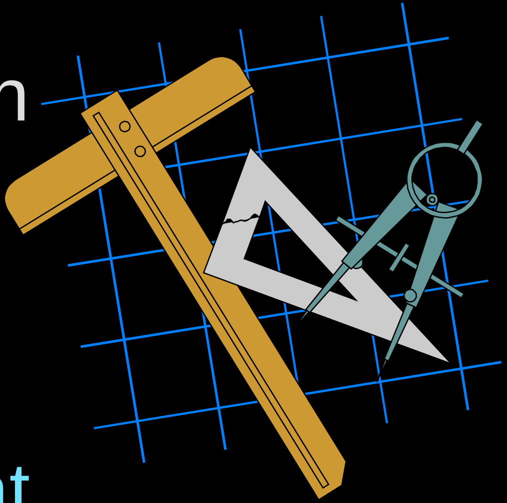
- Identify the following:
  - Attributes, operations, and their complete attribute signatures
  - Attribute and operation scope and visibility
  - Any additional relationships and/or classes to support the defined attributes and attribute signatures
  - Class(es) with significant state-controlled behavior
  - The important states and transitions for the identified class

(continued)

Rational
the software development company

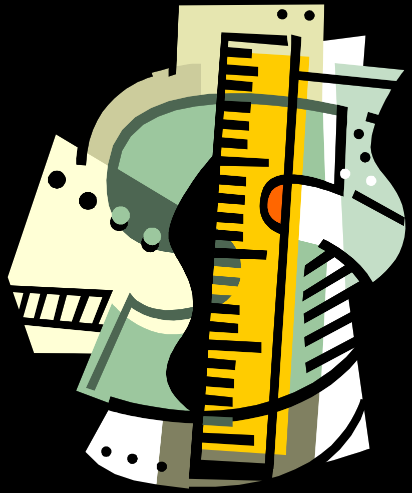
Exercise 1: Class Design

- Produce the following:
  - Design Use-Case Realization
    - Statechart for one of the classes that exhibits state-controlled behavior
    - Class diagram (VOPC) that includes all operations, operation signatures, attributes, and attribute signatures



Exercise 1: Review

- Compare your results
  - Is the name of each operation descriptive and understandable? Does the name of the operation indicate its outcome?
  - Does each attribute represent a single conceptual thing? Is the name of each attribute descriptive and does it correctly convey the information it stores?
  - Is the state machine understandable? Do state names and transitions reflect the context of the domain of the system? Does the state machine contain any superfluous states or transitions?

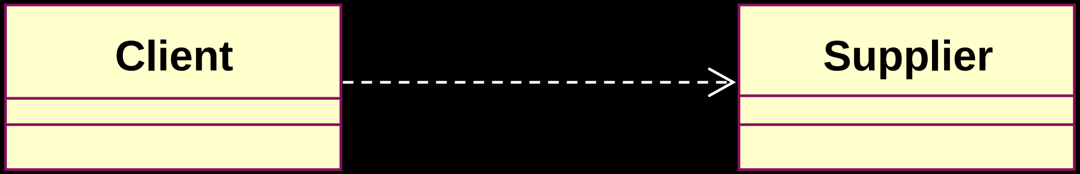

Payroll System

Class Design Steps

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

Define Dependency

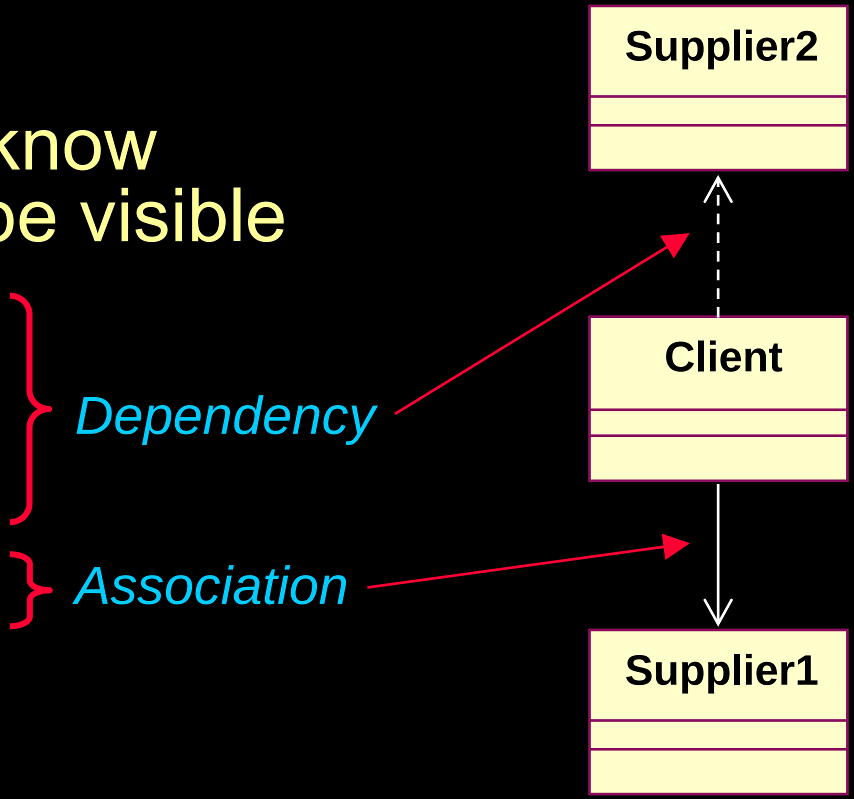
- What Is a Dependency?
  - A relationship between two objects



- 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



Associations vs. Dependencies in Collaborations

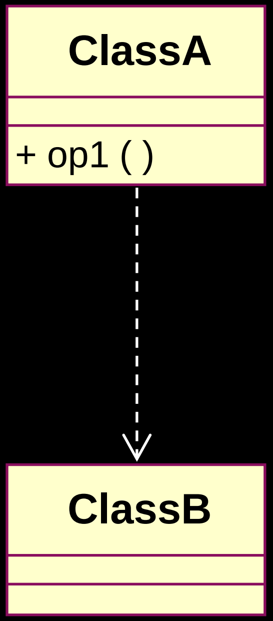
- An instance of an association is a link
  - All links become associations unless they have global, local, or parameter visibility
  - Relationships are context-dependent
- Dependencies are transient links with:
  - A limited duration
  - A context-independent relationship
  - A summary relationship

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.

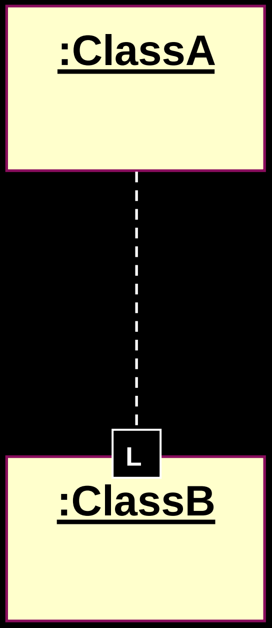
Local Variable Visibility

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

Class Diagram

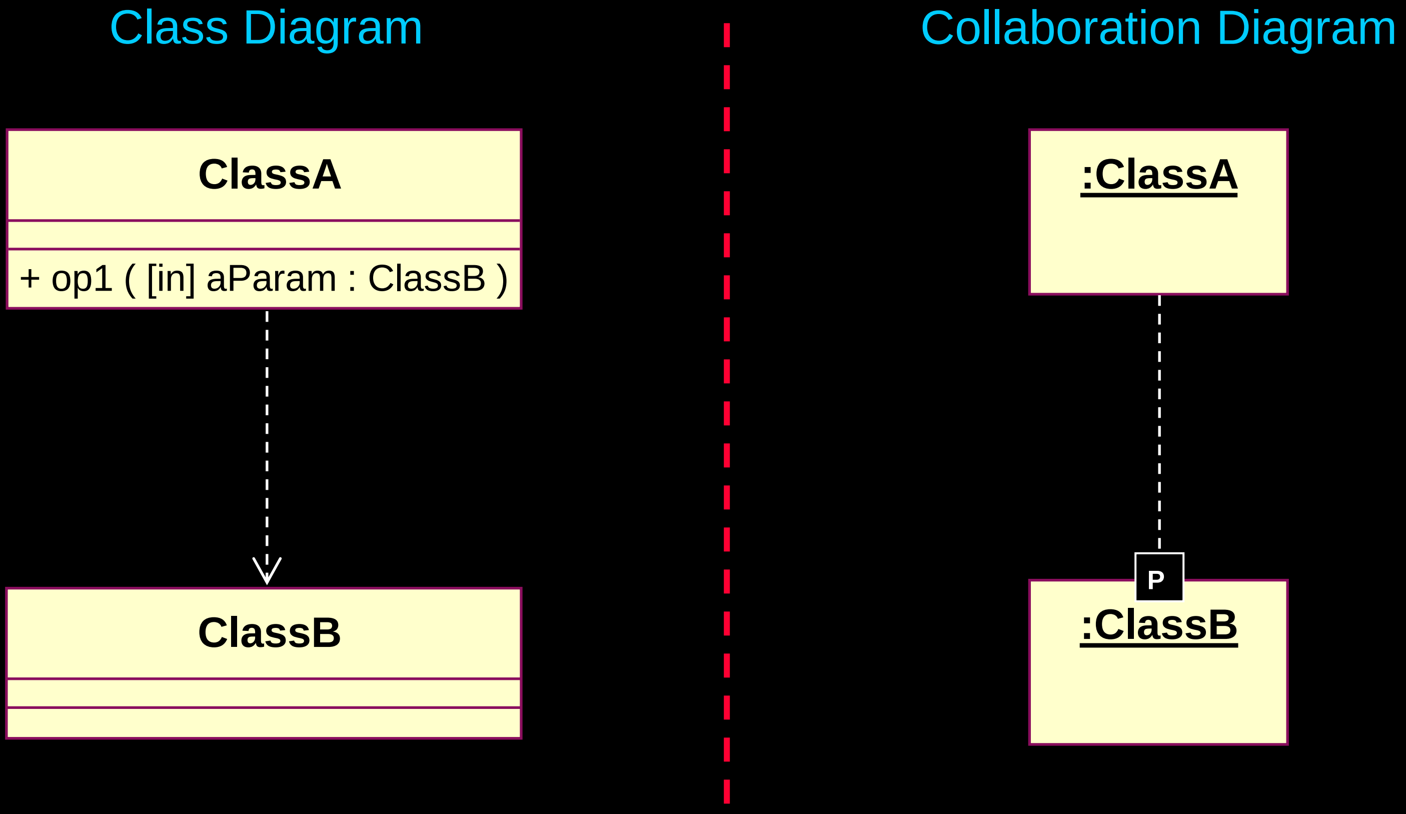


Collaboration Diagram



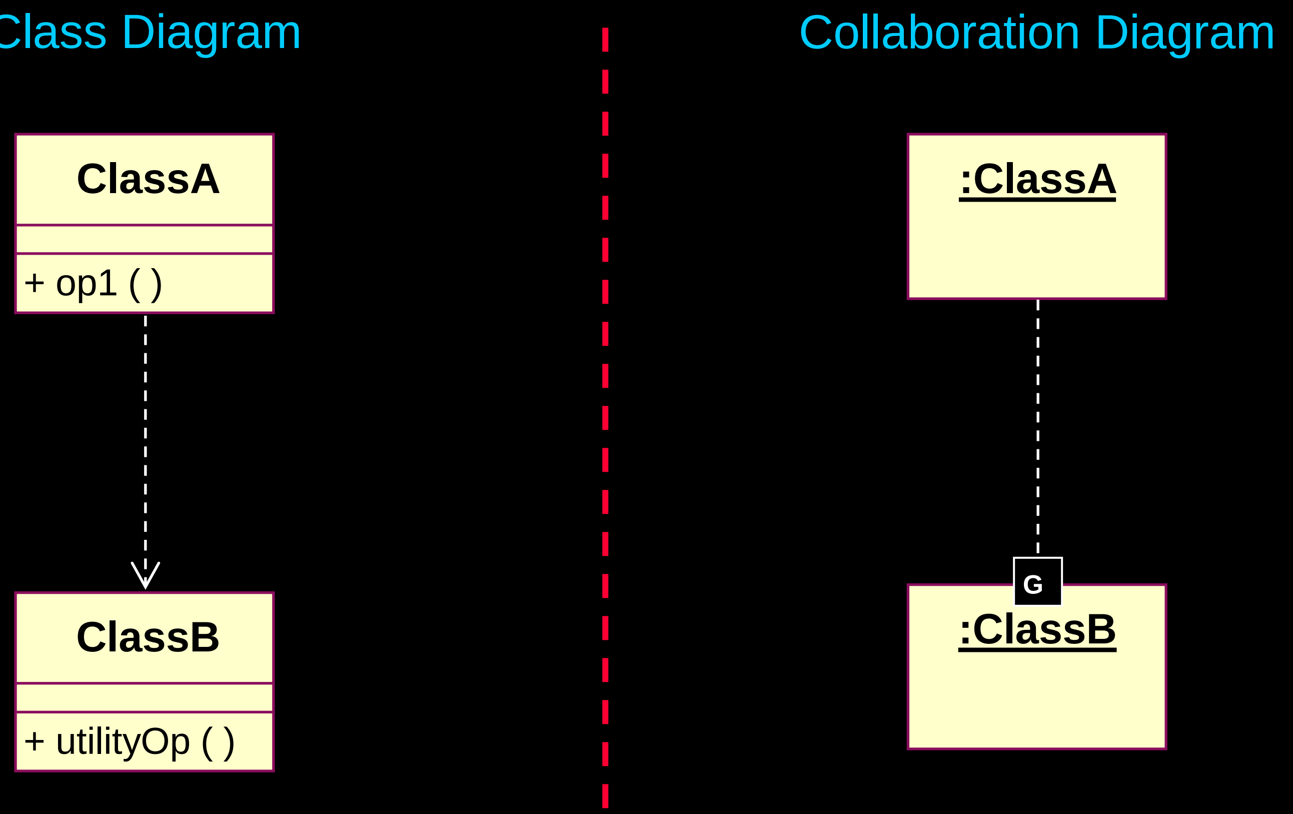
Parameter Visibility

- The ClassB instance is passed to the ClassA instance



Global Visibility

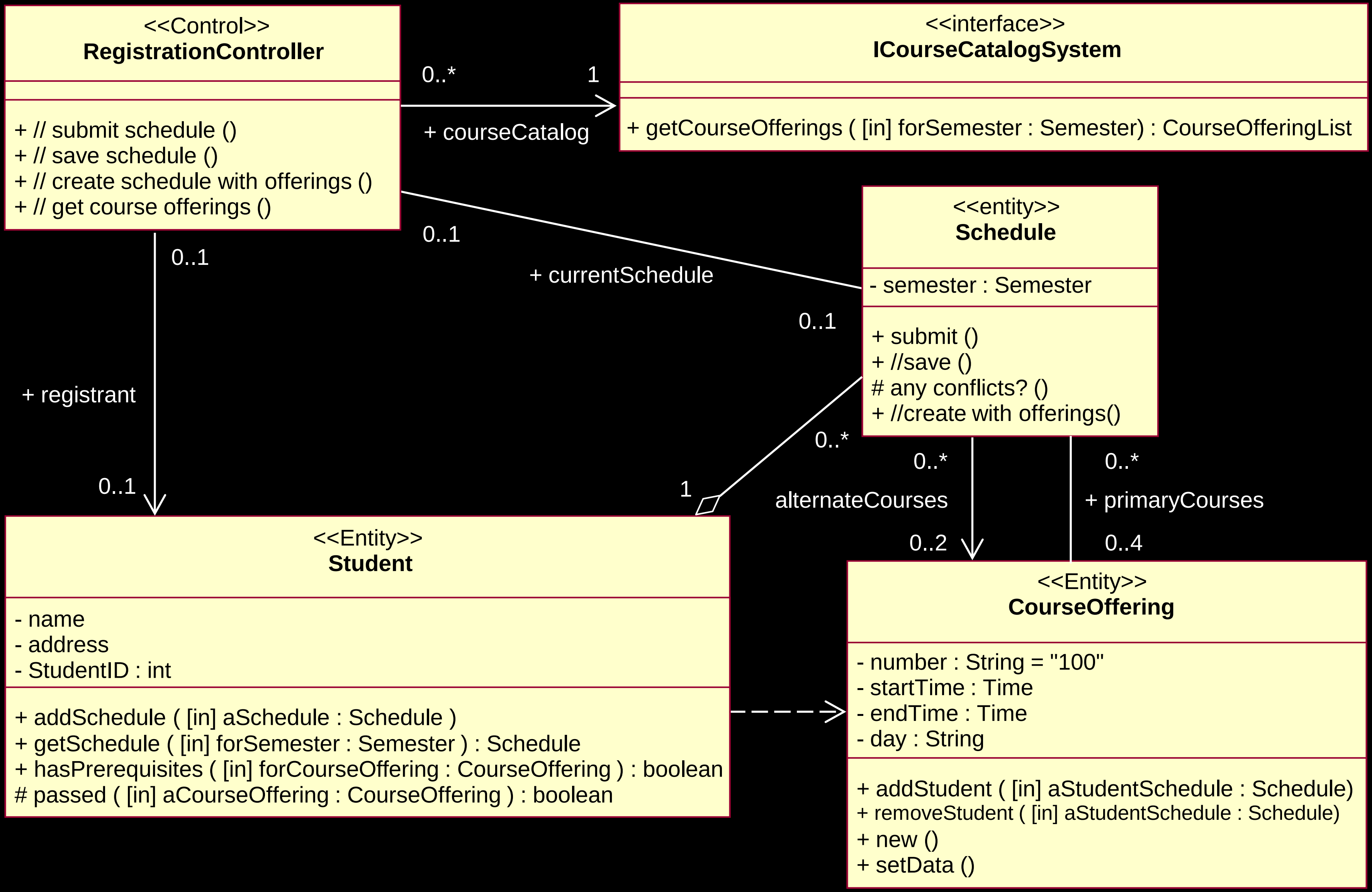
- The ClassUtility instance is visible because it is global



Identifying Dependencies: Considerations

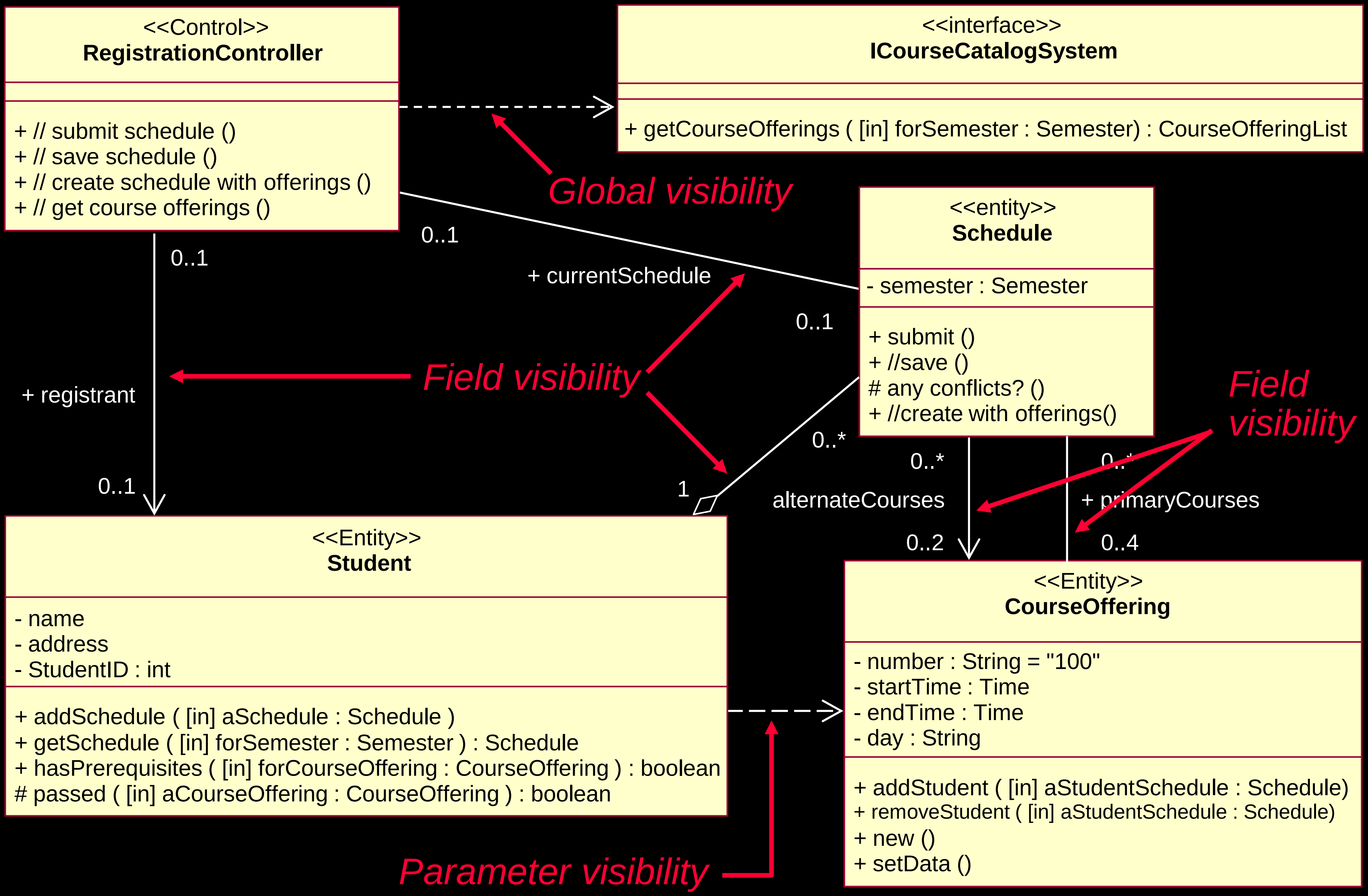
- Permanent relationships — Association (field visibility)
- Transient relationships — Dependency
  - Multiple objects share the same instance
    - Pass instance as a parameter (parameter visibility)
    - Make instance a managed global (global visibility)
  - Multiple objects don't share the same instance (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)



<<Control>>
RegistrationController
+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()
0..*
1
+ courseCatalog
<<interface>>
ICourseCatalogSystem
+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList
0..1
+ currentSchedule
0..1
<<entity>>
Schedule
- semester : Semester
+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()
0..1
+ registrant
0..1
0..*
1
0..*
alternateCourses
0..2
0..*
+ primaryCourses
0..4
<<Entity>>
Student
- name
- address
- StudentID : int
+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean
<<Entity>>
CourseOffering
- number : String = "100"
- startTime : Time
- endTime : Time
- day : String
+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()

Example: Define Dependencies (after)



<<Control>>
RegistrationController
+ // submit schedule ()
+ // save schedule ()
+ // create schedule with offerings ()
+ // get course offerings ()
<<interface>>
ICourseCatalogSystem
+ getCourseOfferings ([in] forSemester : Semester) : CourseOfferingList
Global visibility
<<entity>>
Schedule
- semester : Semester
+ submit ()
+ //save ()
# any conflicts? ()
+ //create with offerings()
0..1
0..1
+ currentSchedule
0..1
+ registrant
Field visibility
Field visibility
0..*
0..*
0..*
alternateCourses
+ primaryCourses
0..1
1
0..2
0..4
<<Entity>>
Student
- name
- address
- StudentID : int
+ addSchedule ([in] aSchedule : Schedule)
+ getSchedule ([in] forSemester : Semester) : Schedule
+ hasPrerequisites ([in] forCourseOffering : CourseOffering) : boolean
# passed ([in] aCourseOffering : CourseOffering) : boolean
<<Entity>>
CourseOffering
- number : String = "100"
- startTime : Time
- endTime : Time
- day : String
+ addStudent ([in] aStudentSchedule : Schedule)
+ removeStudent ([in] aStudentSchedule : Schedule)
+ new ()
+ setData ()
Parameter visibility

Class Design Steps

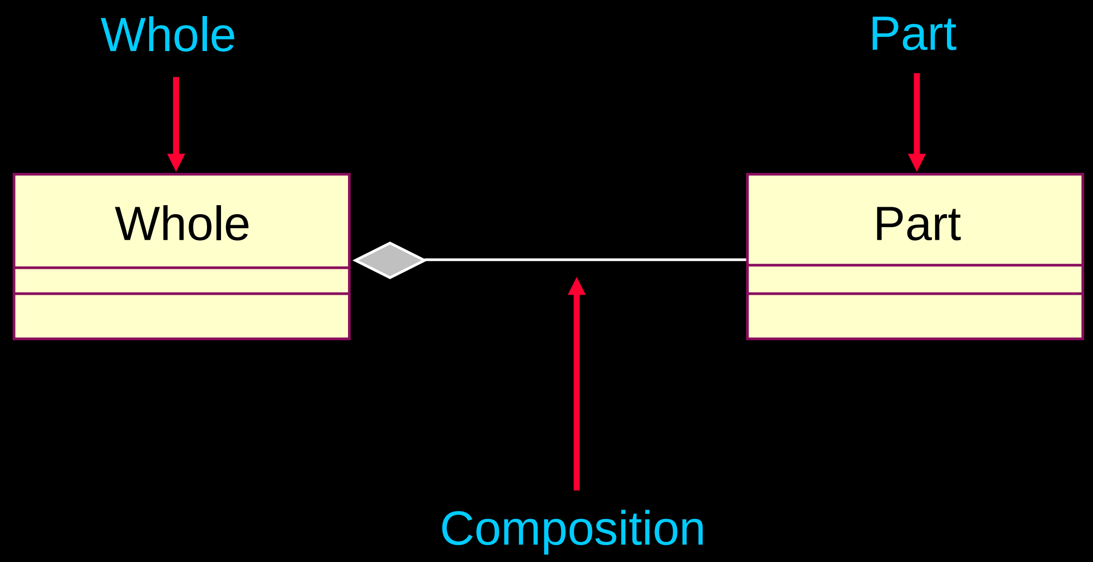
- Create Initial Design Classes
- Define Operations
- Define Methods
- Define States
- Define Attributes
- Define Dependencies
- ★ Define Associations
- Define Generalizations
- Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints

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

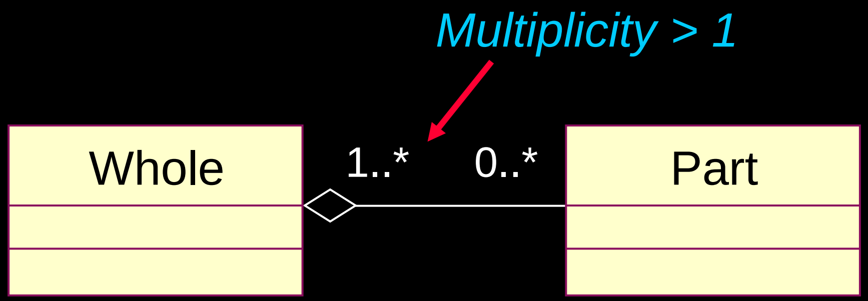
What Is Composition?

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

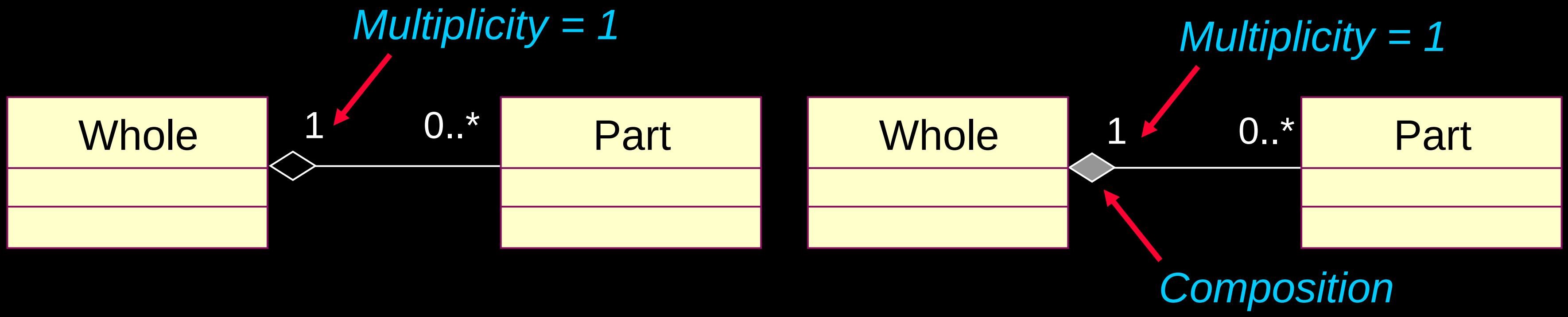


Aggregation: Shared vs. Non-shared

- Shared Aggregation



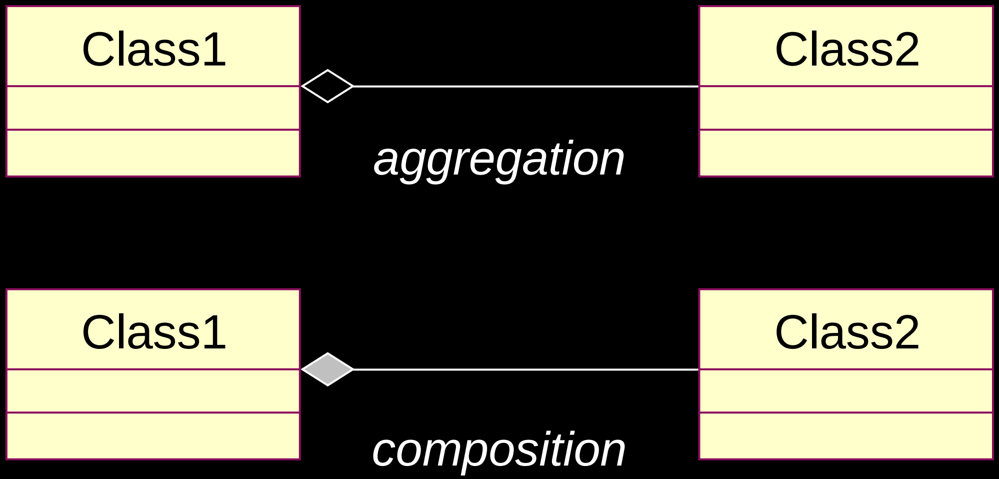
- Non-shared Aggregation



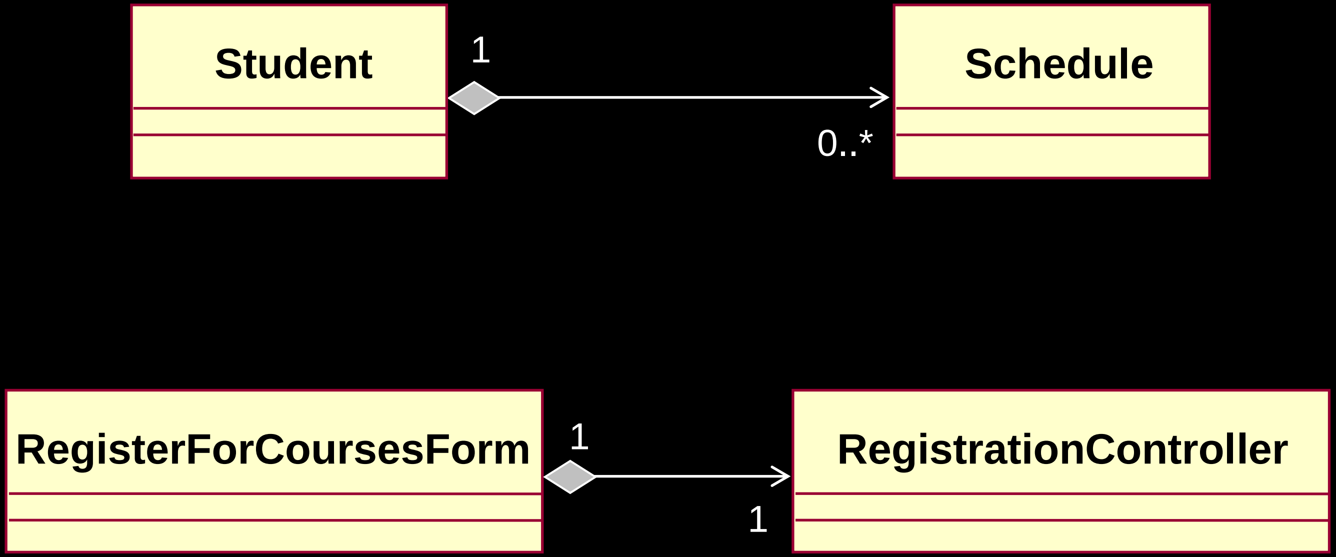
By definition, composition is non-shared aggregation

Aggregation or Composition?

- Consideration
  - Lifetimes of Class1 and Class2



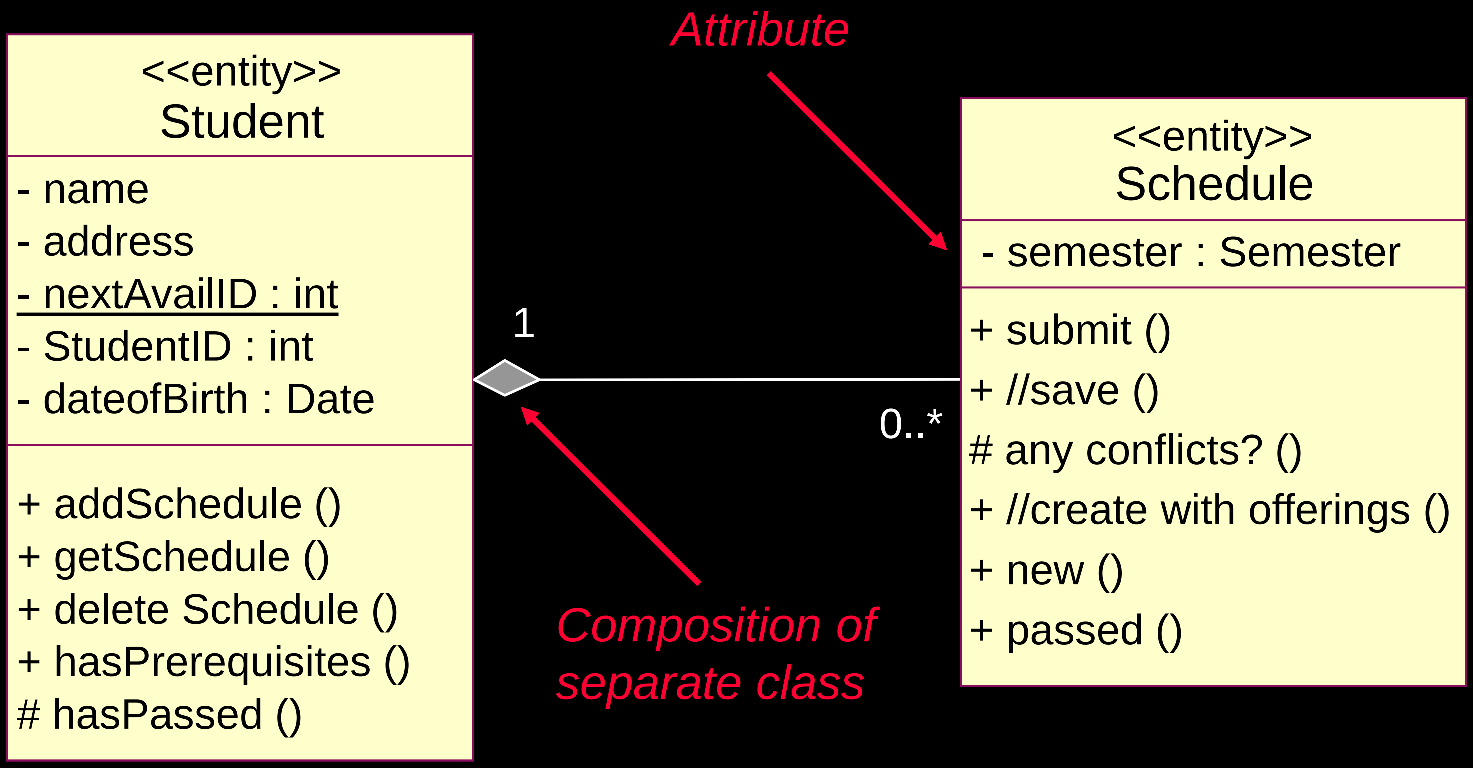
Example: Composition



Attributes Vs Composition

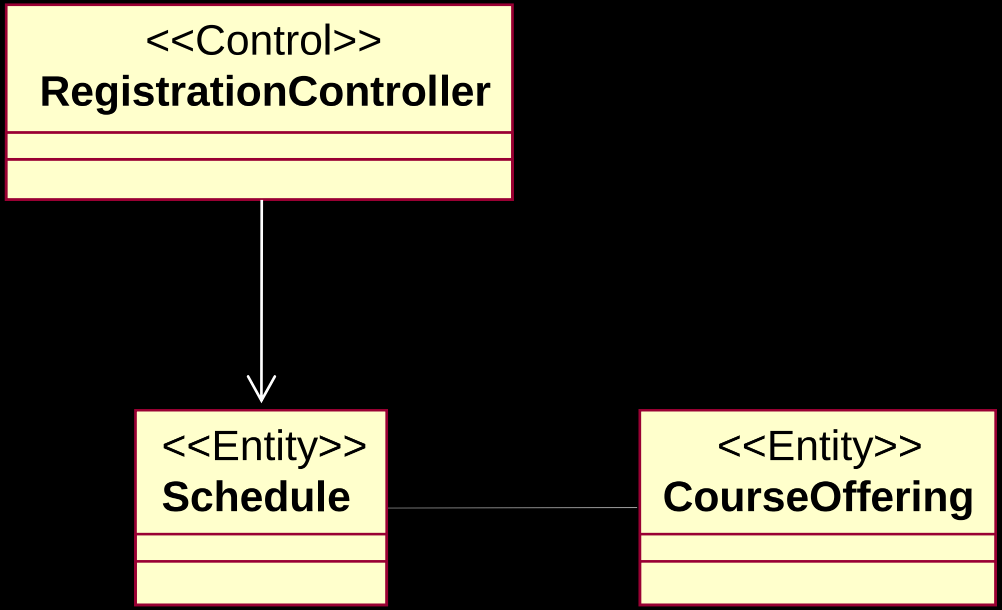
- Use composition when
  - Properties need independent identities
  - Multiple classes have the same properties
  - Properties have a complex structure and properties of their own
  - Properties have complex behavior of their own
  - Properties have relationships of their own
- Otherwise use attributes

Example: Attributes vs. Composition



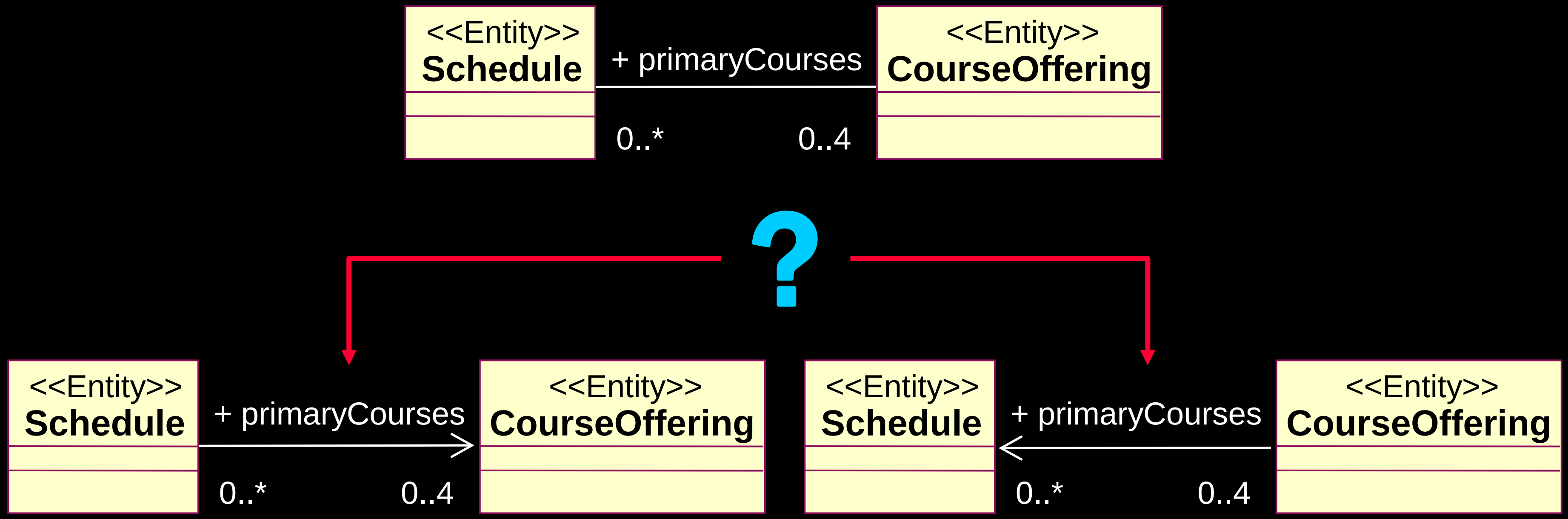
Review: What Is Navigability?

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



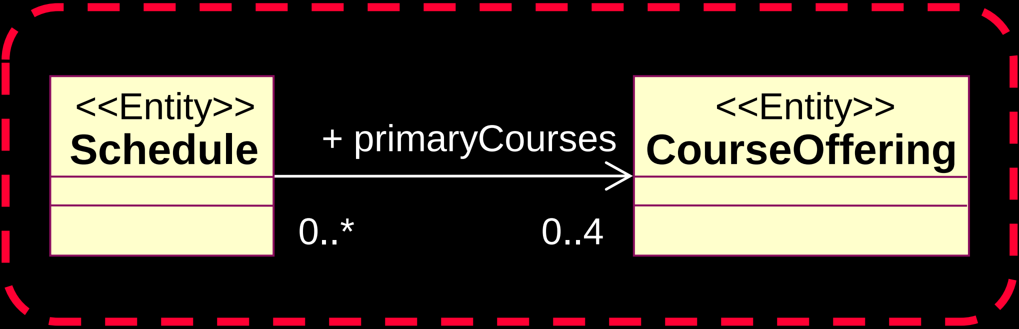
Navigability: Which Directions Are Really Needed?

- 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

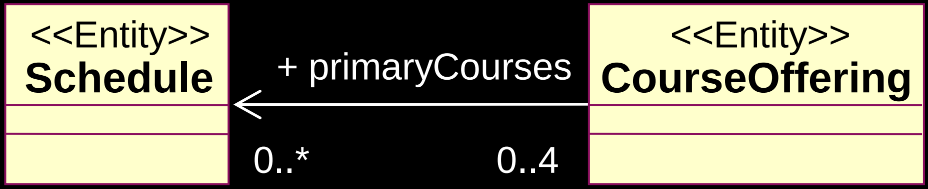


Example: Navigability Refinement

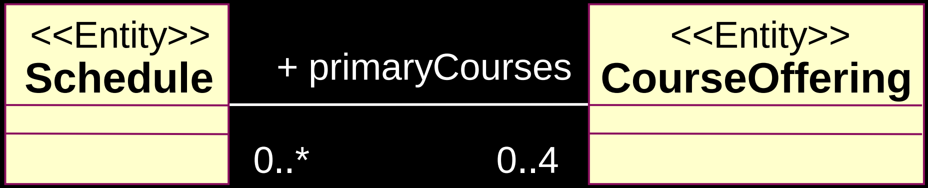
- Total number of Schedules is small, or
- Never need a list of the Schedules on which the CourseOffering appears



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

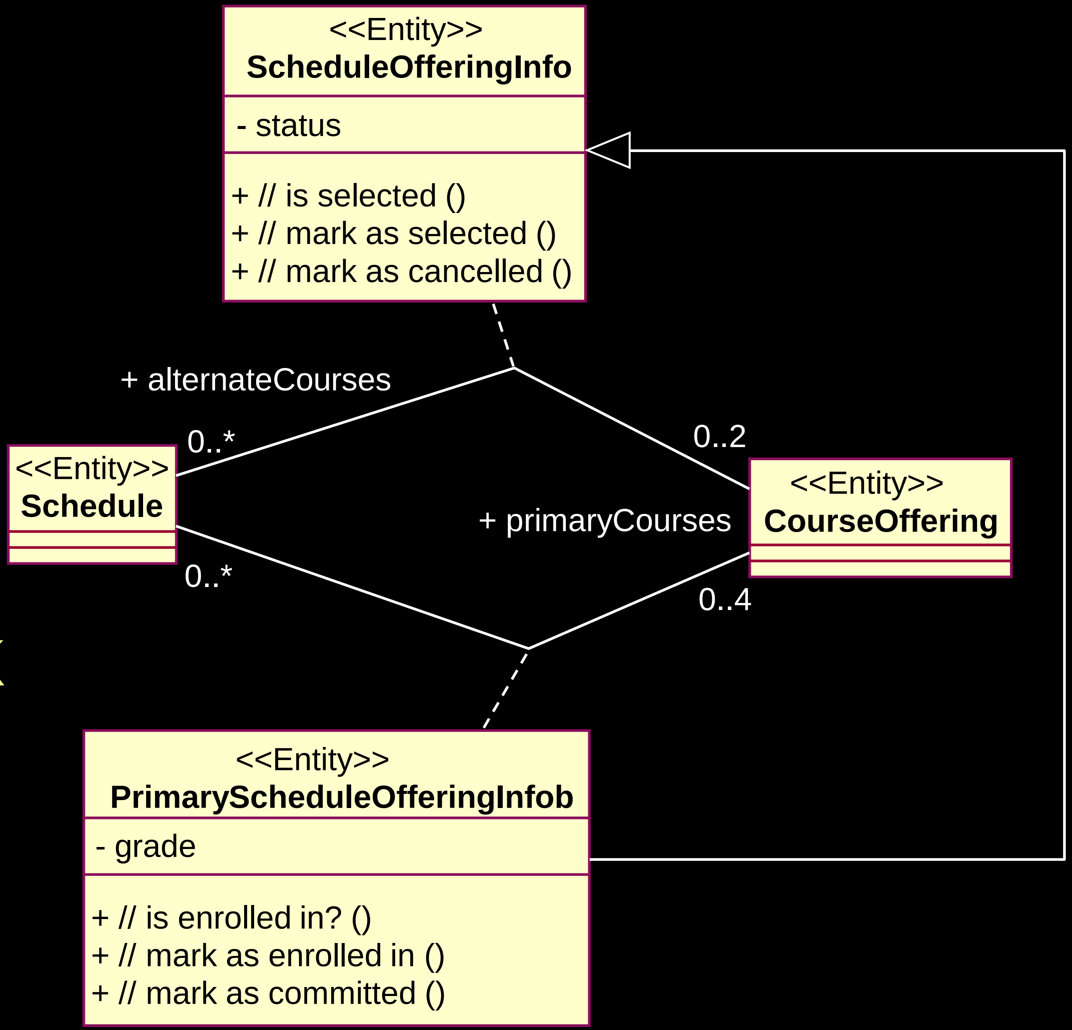


- Total number of CourseOfferings and Schedules are not small
- Must be able to navigate in both directions

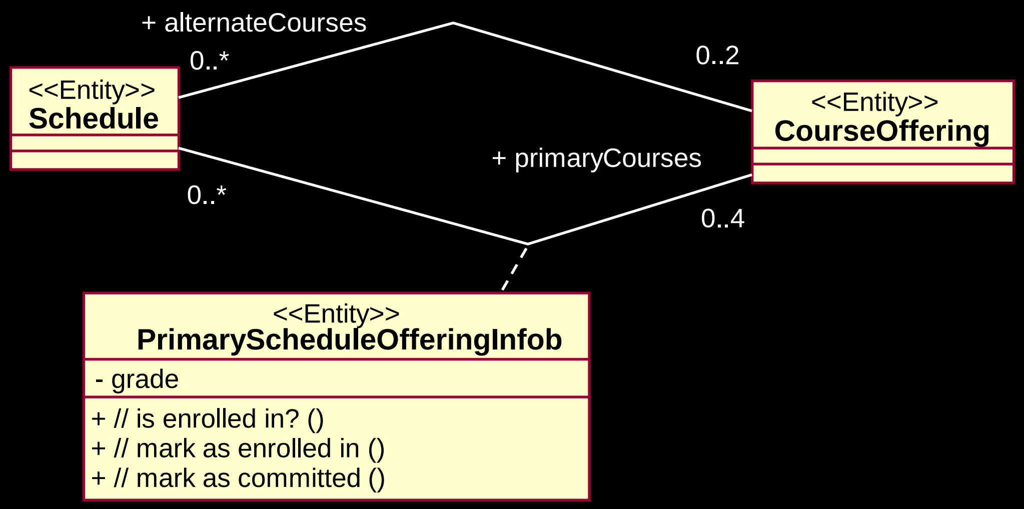


Association Class

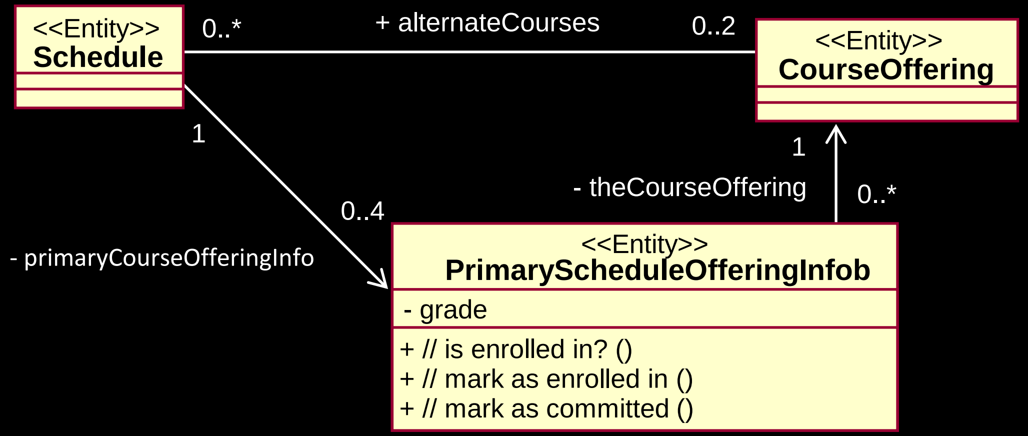
- A class is "attached" to an association
- Contains properties of a relationship
- Has one instance per link



Example: Association Class Design

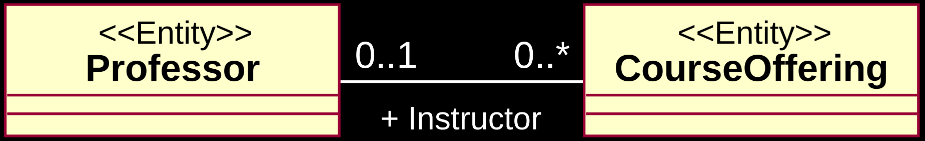


Design Decisions



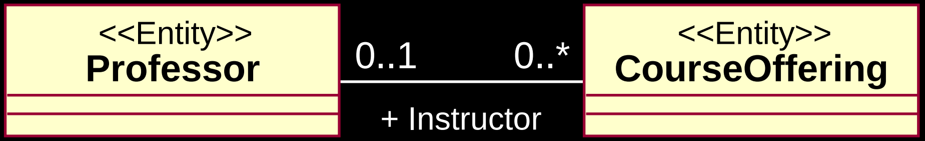
Multiplicity Design

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

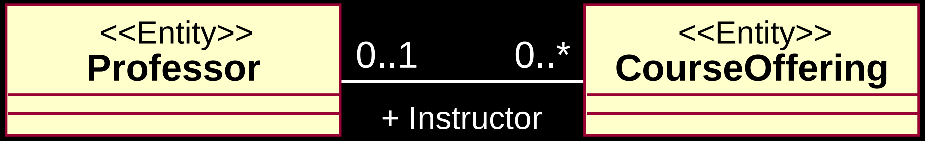


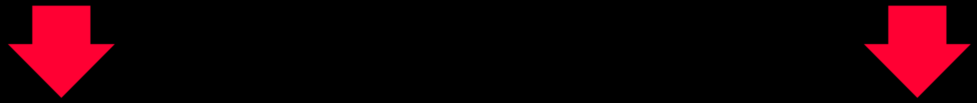
- Multiplicity > 1
  - Cannot use a simple value or pointer
  - Further "design" may be required

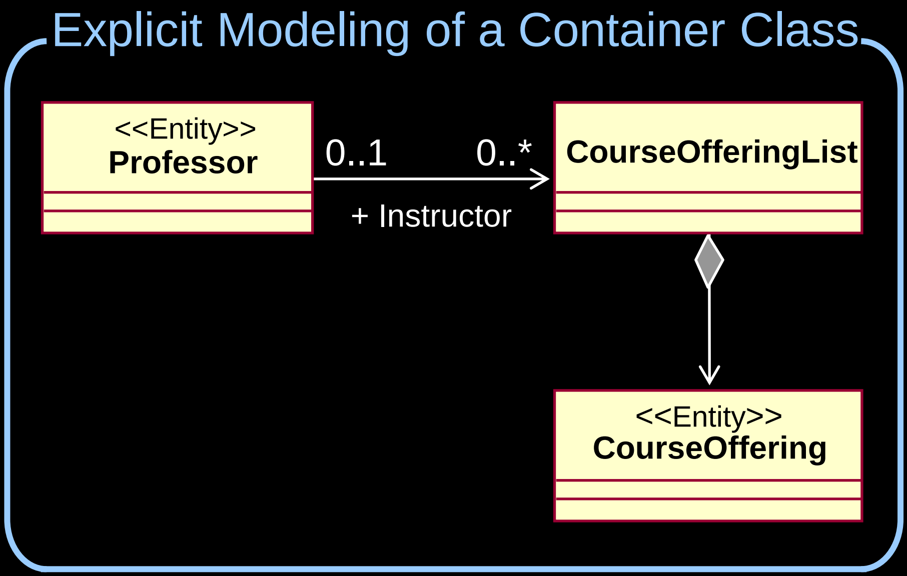
Needs a container for CourseOfferings

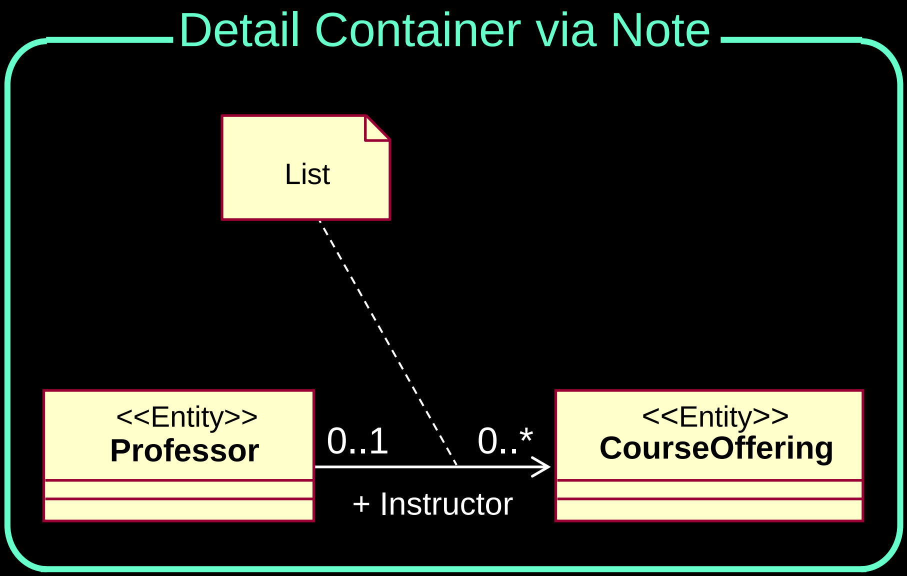


Multiplicity Design Options



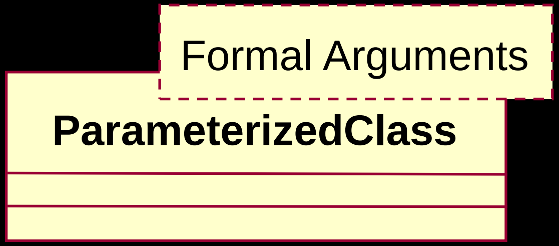


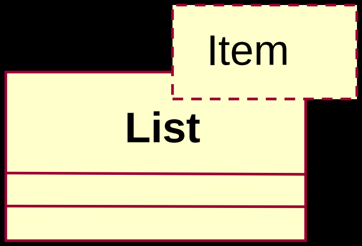




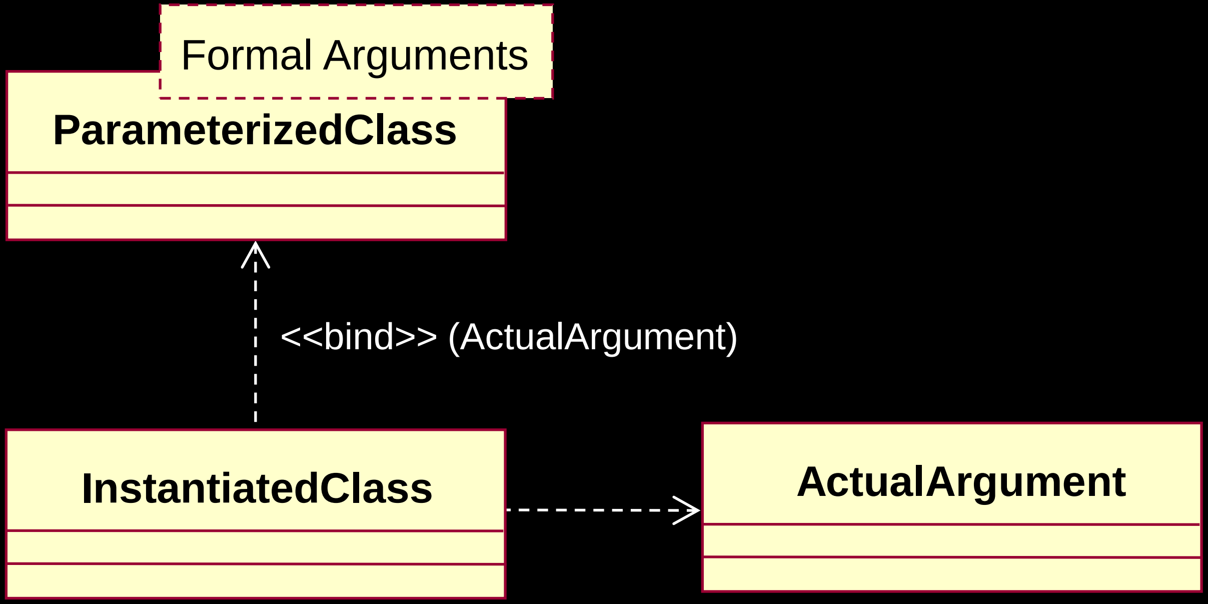
What is a Parameterized Class (template)?

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



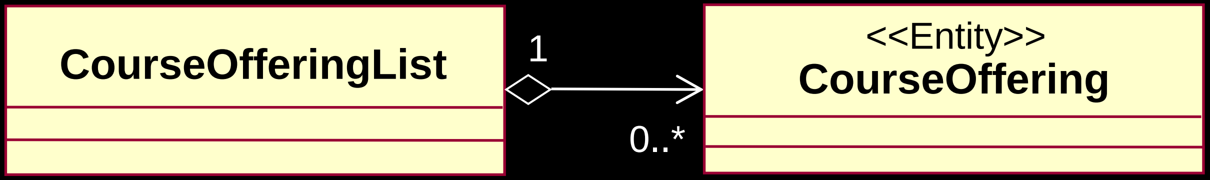


Instantiating a Parameterized Class

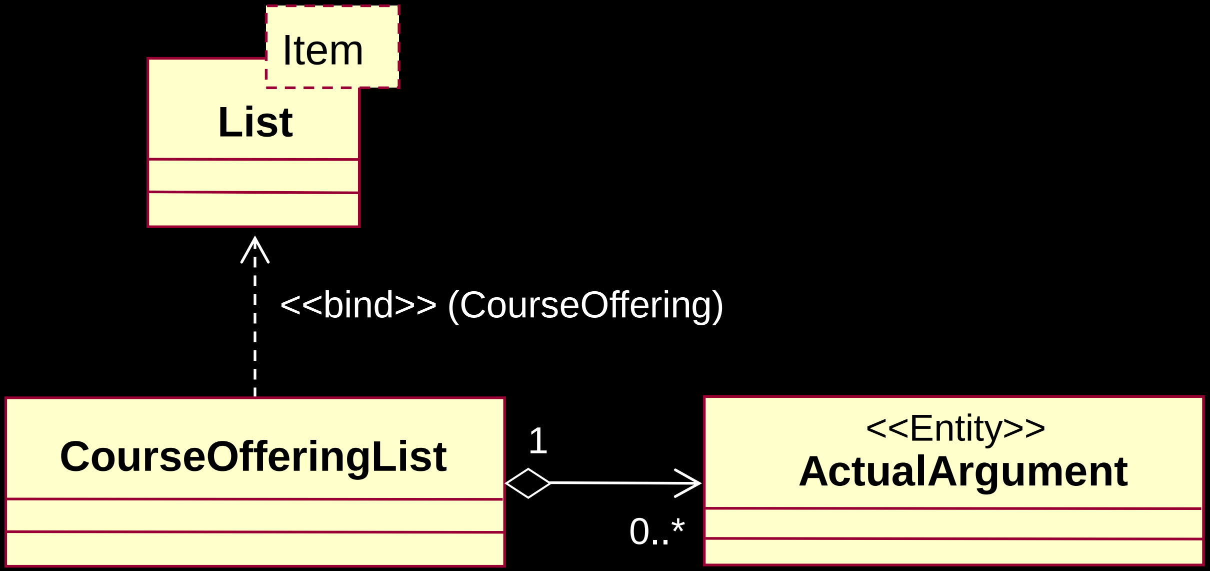


Example: Instantiating a Parameterized Class

Before

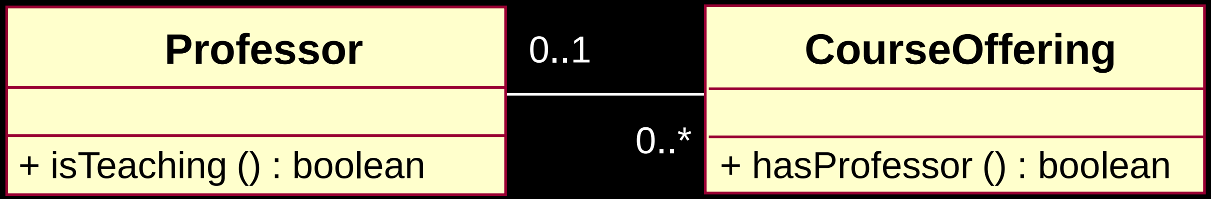


After



Multiplicity Design: Optionality

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

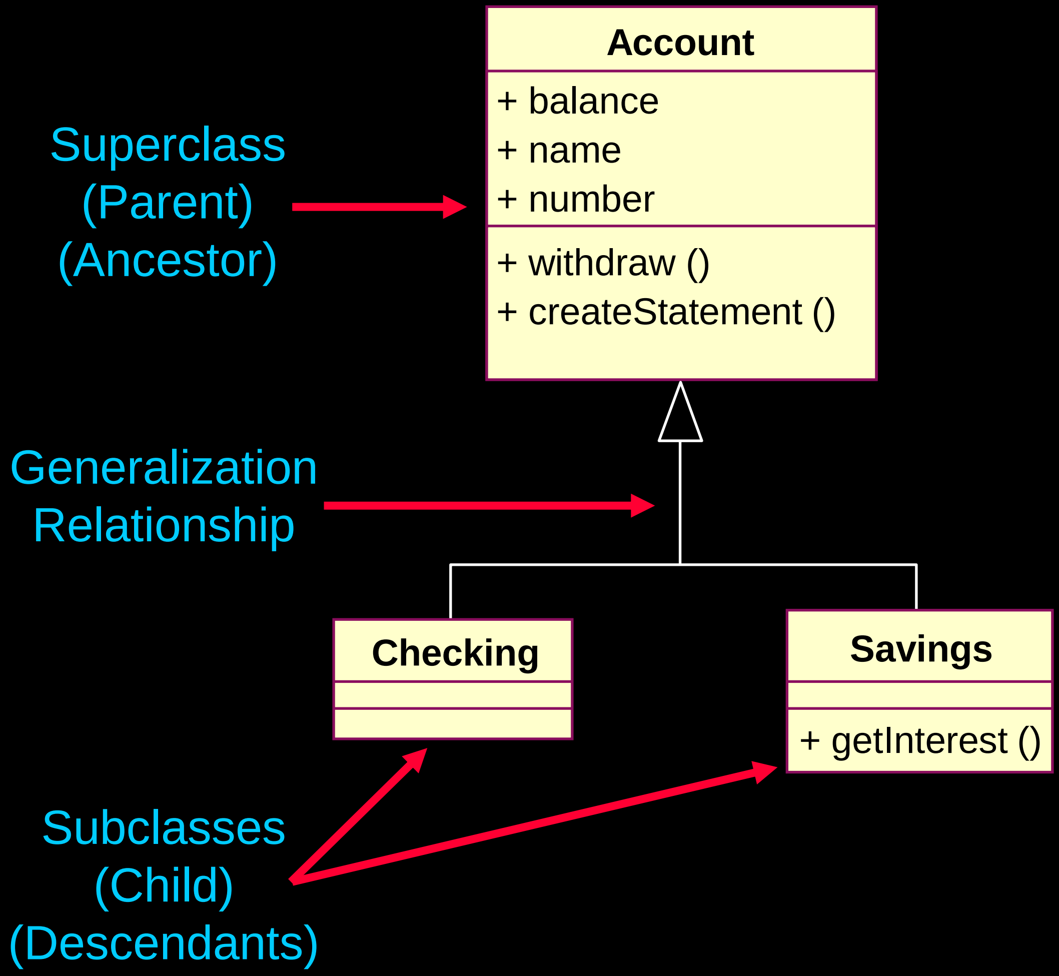


Class Design Steps

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

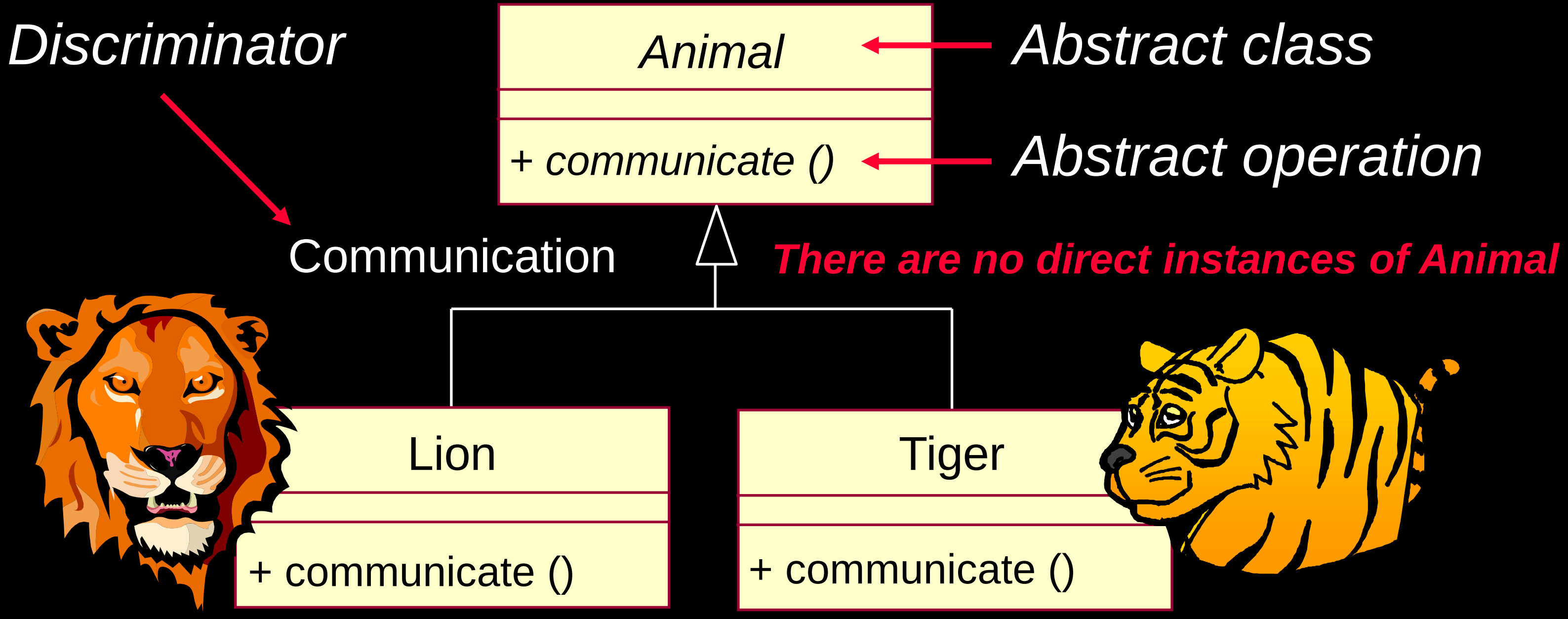
Review: Generalization

- 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

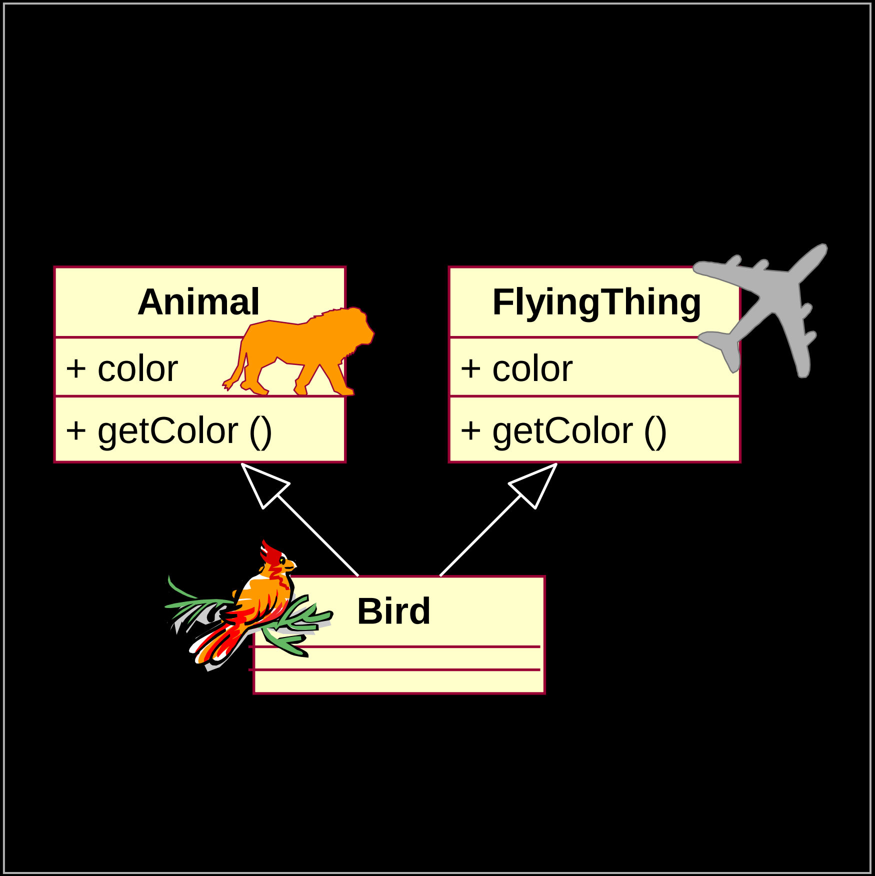
- Abstract classes cannot have any objects
- Concrete classes can have objects



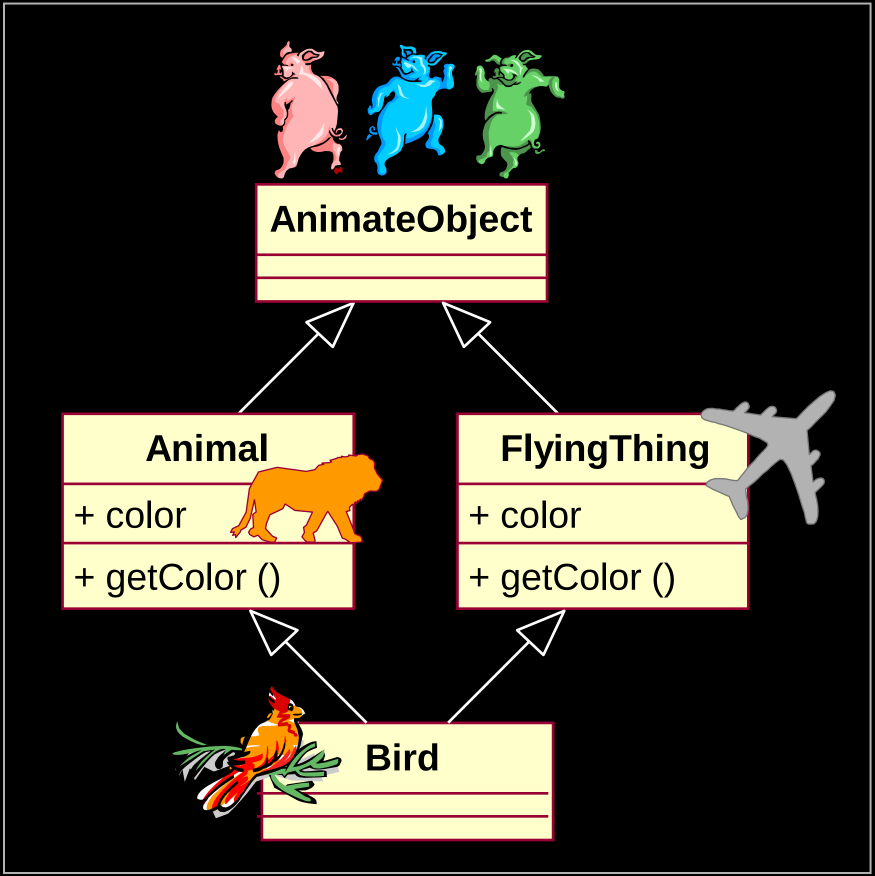
All objects are either lions or tigers

Multiple Inheritance: Problems

Name clashes on attributes or operations



Repeated inheritance

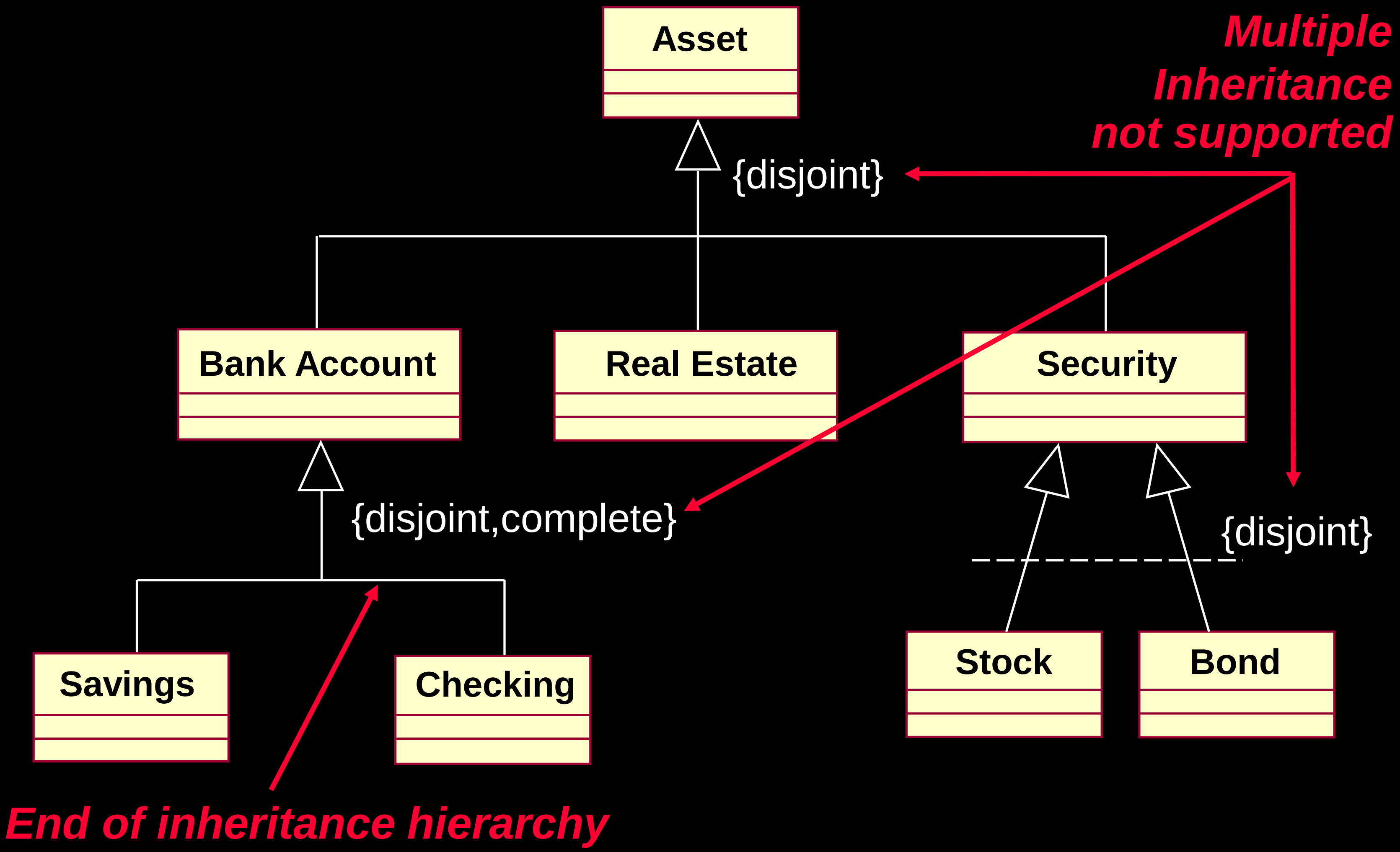


Resolution of these problems is implementation-dependent

Generalization Constraints

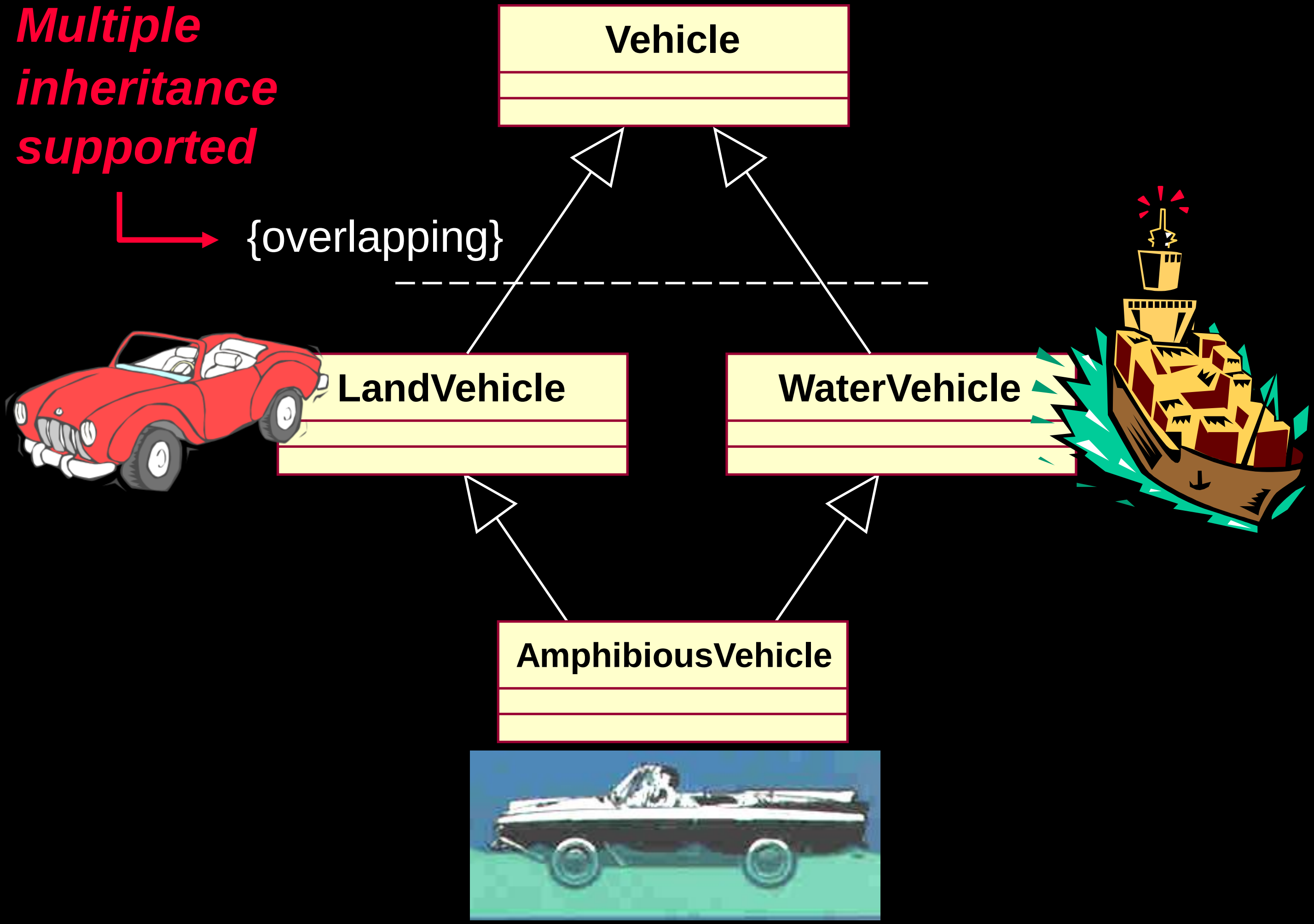
- Complete
  - End of the inheritance tree
- Incomplete
  - Inheritance tree may be extended
- Disjoint
  - Subclasses mutually exclusive
  - Doesn’t support multiple inheritance
- Overlapping
  - Subclasses are not mutually exclusive
  - Supports multiple inheritance

Example: Generalization Constraints



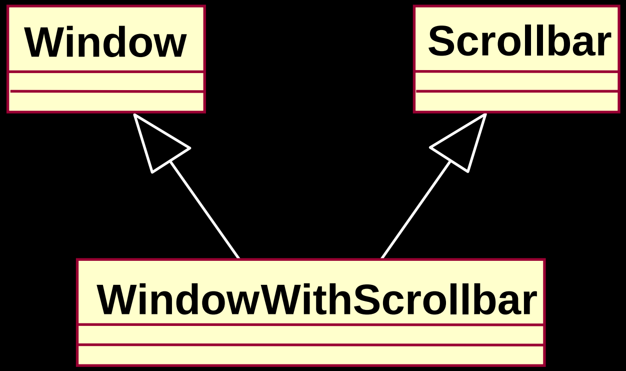
Example: Generalization Constraints (cont.)

Multiple inheritance supported



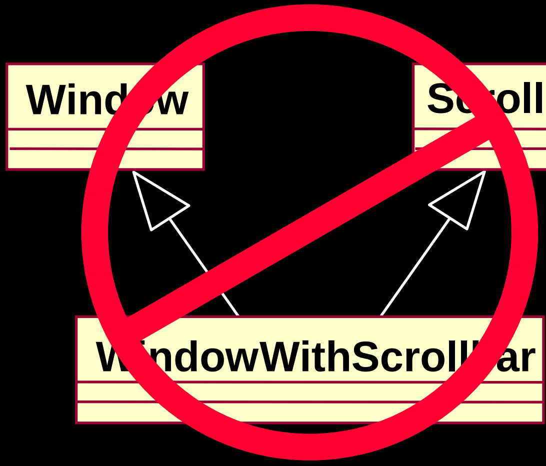
Generalization vs. Aggregation

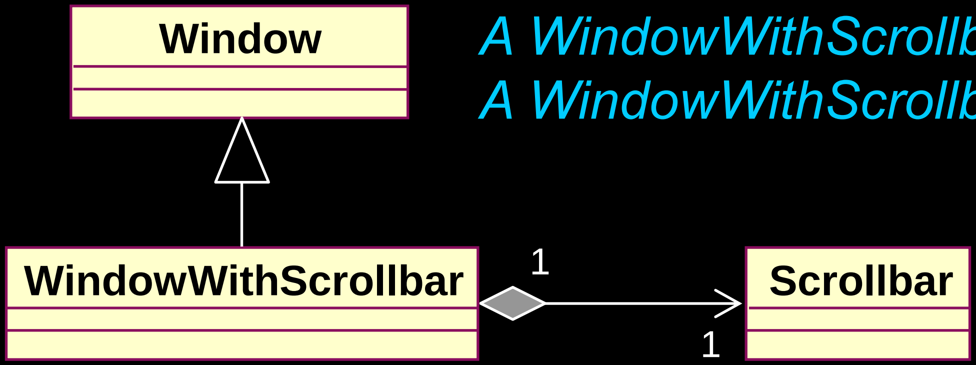
- Generalization and aggregation are often confused
  - Generalization represents an “is a” or “kind-of” relationship
  - Aggregation represents a “part-of” relationship



Is this correct?

Generalization vs. Aggregation

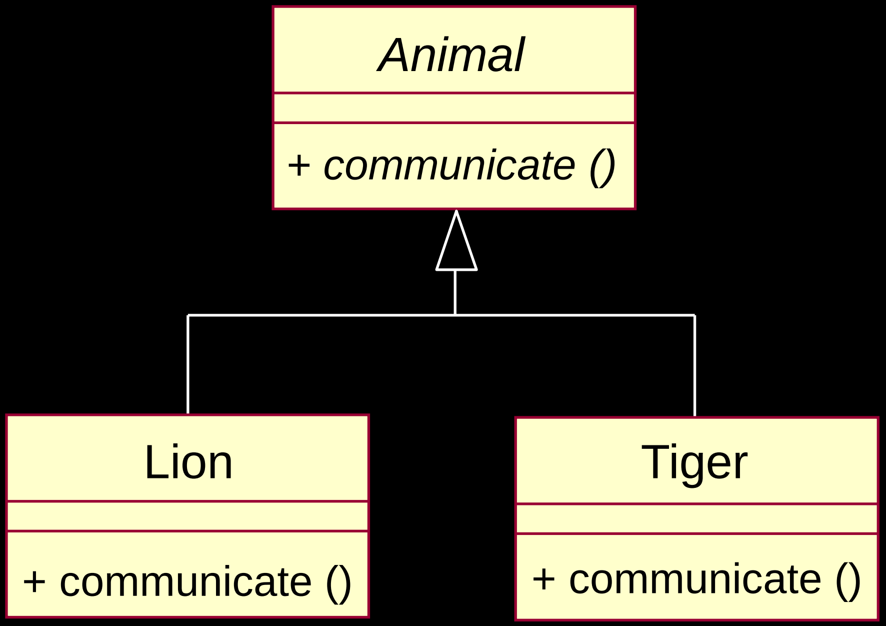


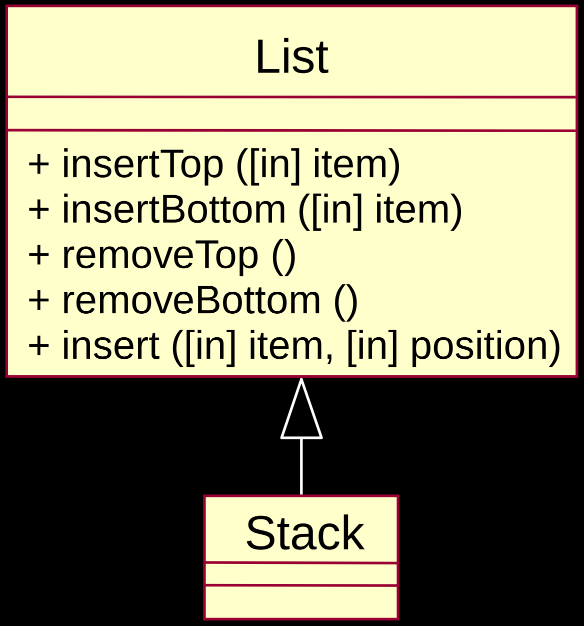


A WindowWithScrollbar "is a" Window
A WindowWithScrollbar "contains a" Scrollbar

Generalization: Share Common Properties and Behavior

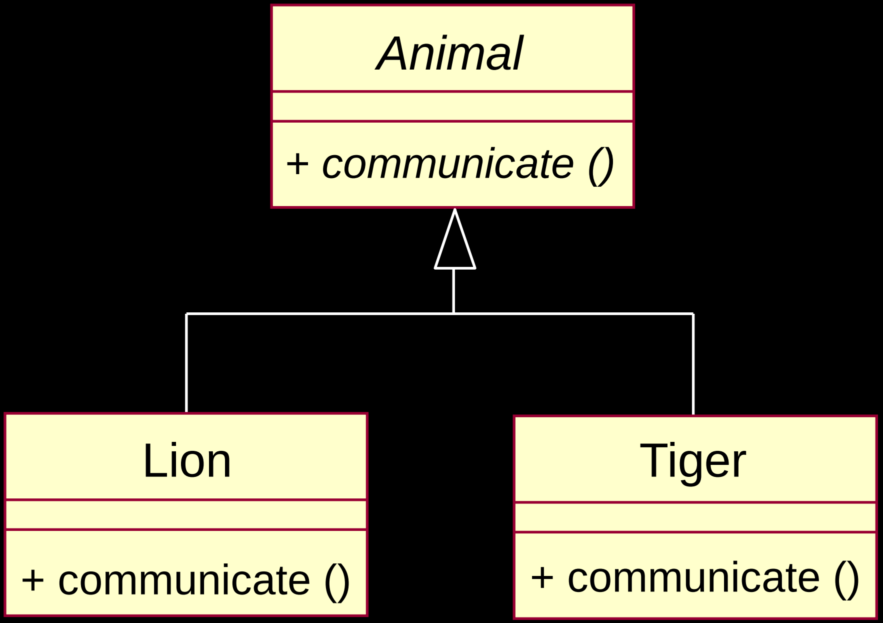
- Follows the "is a" style of programming
- Class substitutability

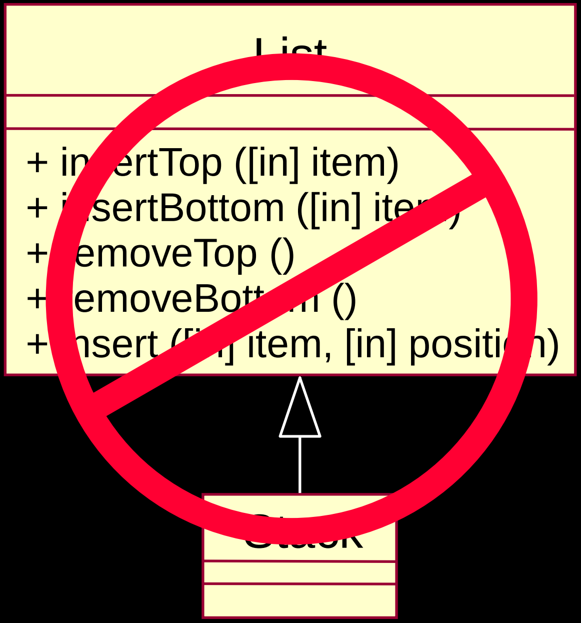




Do these classes follow the "is a" style of programming?

Generalization: Share Common Properties and Behavior (cont.)

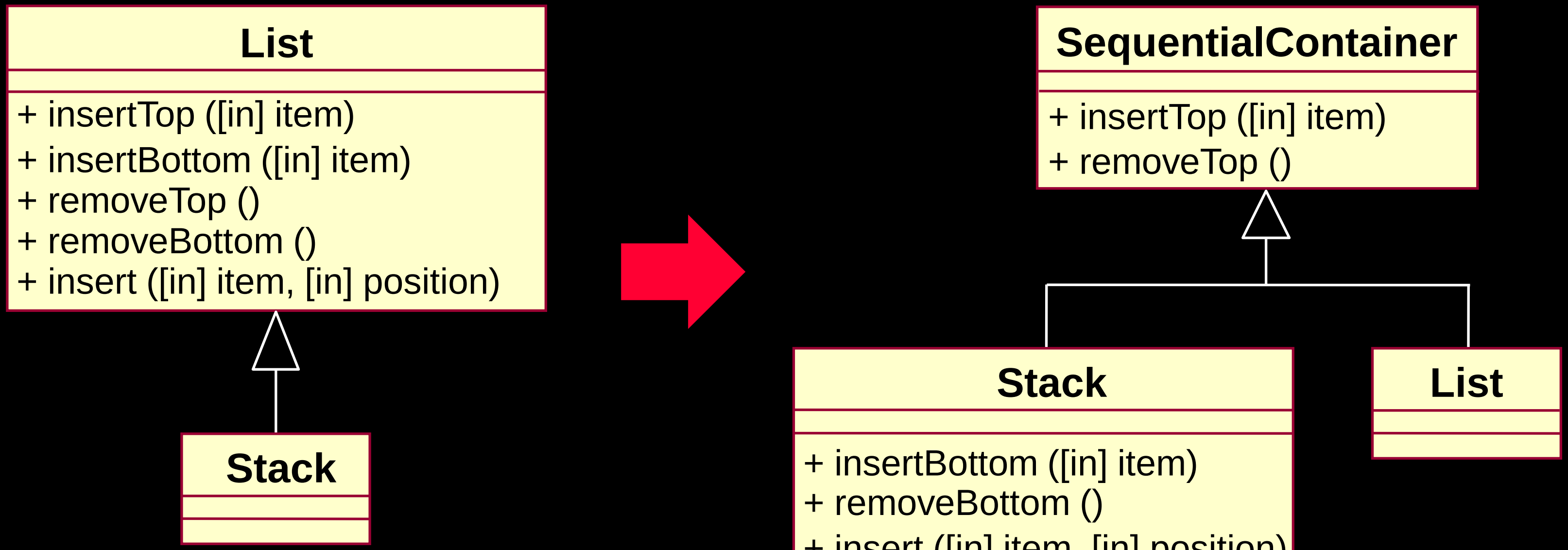




Rational
the software development company

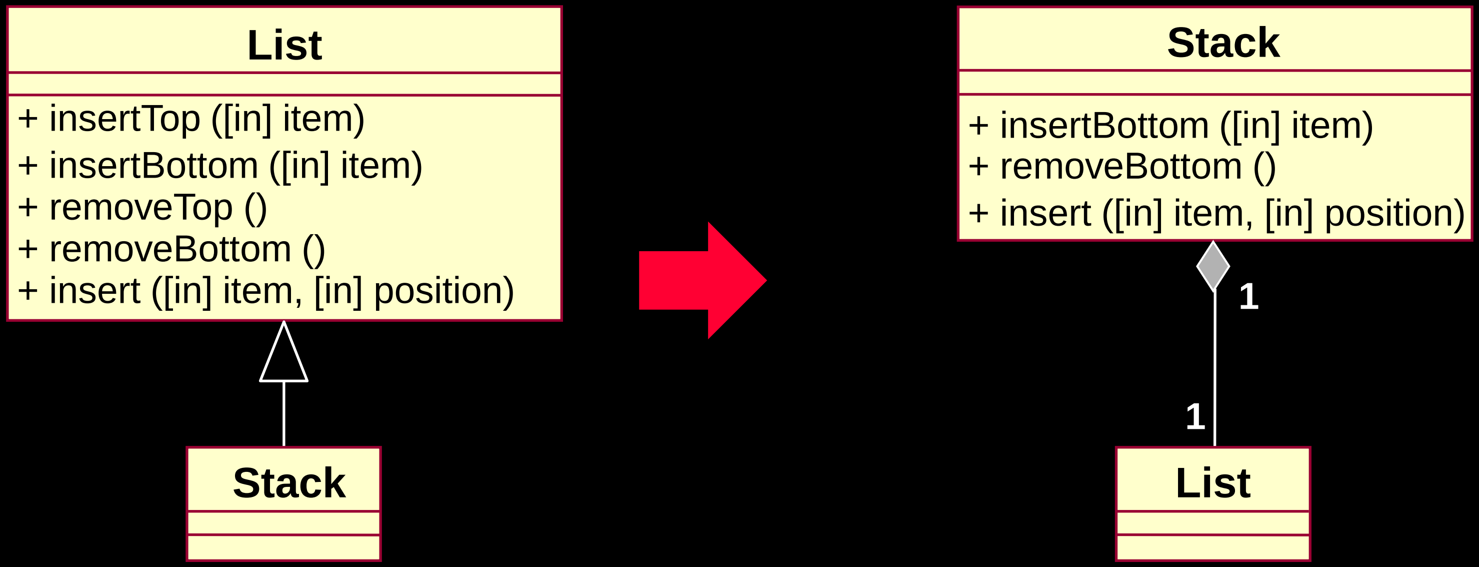
Generalization: Share Implementation: Factoring

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



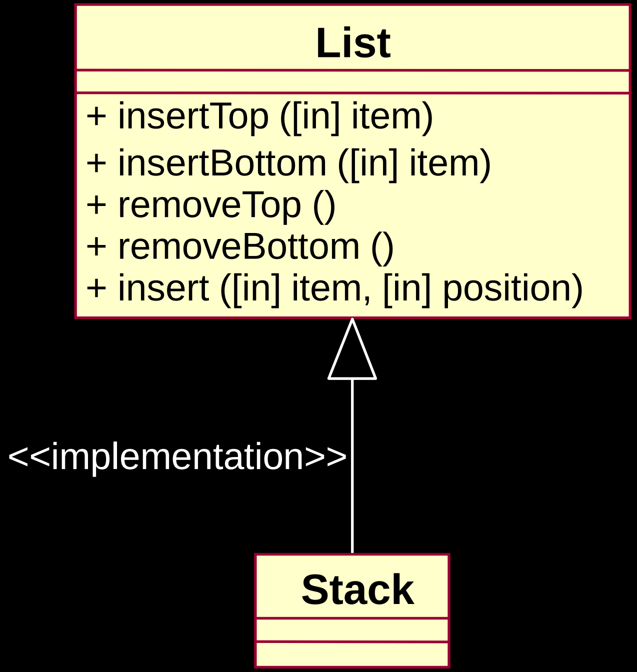
Generalization Alternative: Share Implementation: Delegation

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


List
+ insertTop ([in] item)
+ insertBottom ([in] item)
+ removeTop ()
+ removeBottom ()
+ insert ([in] item, [in] position)
Stack
Stack
+ insertBottom ([in] item)
+ removeBottom ()
+ insert ([in] item, [in] position)
1
1
List

Implementation Inheritance

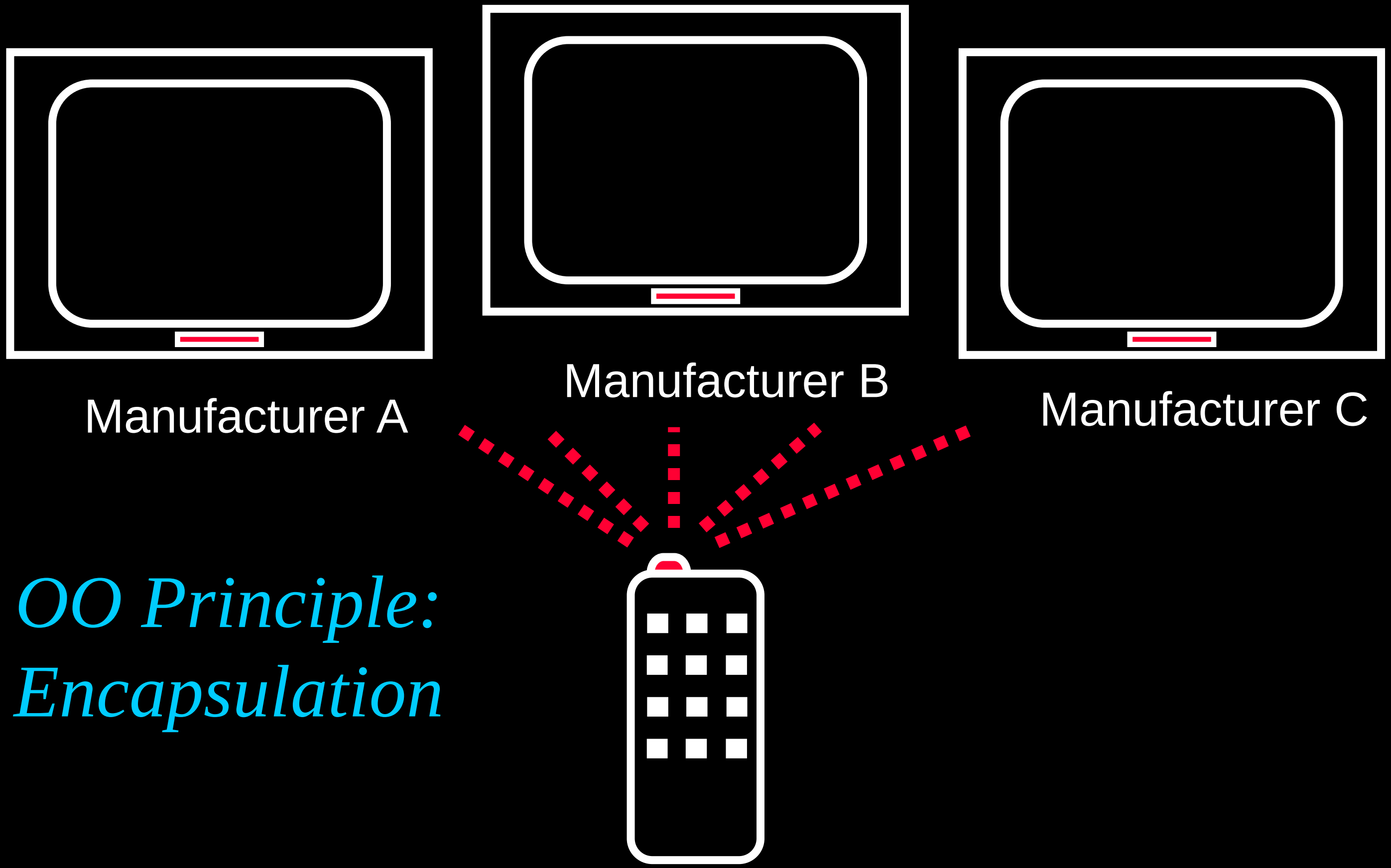
- Ancestor public operations, attributes, and relationships are NOT visible to clients of descendent class instances
- Descendent class must define all access to ancestor operations, attributes, and relationships



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

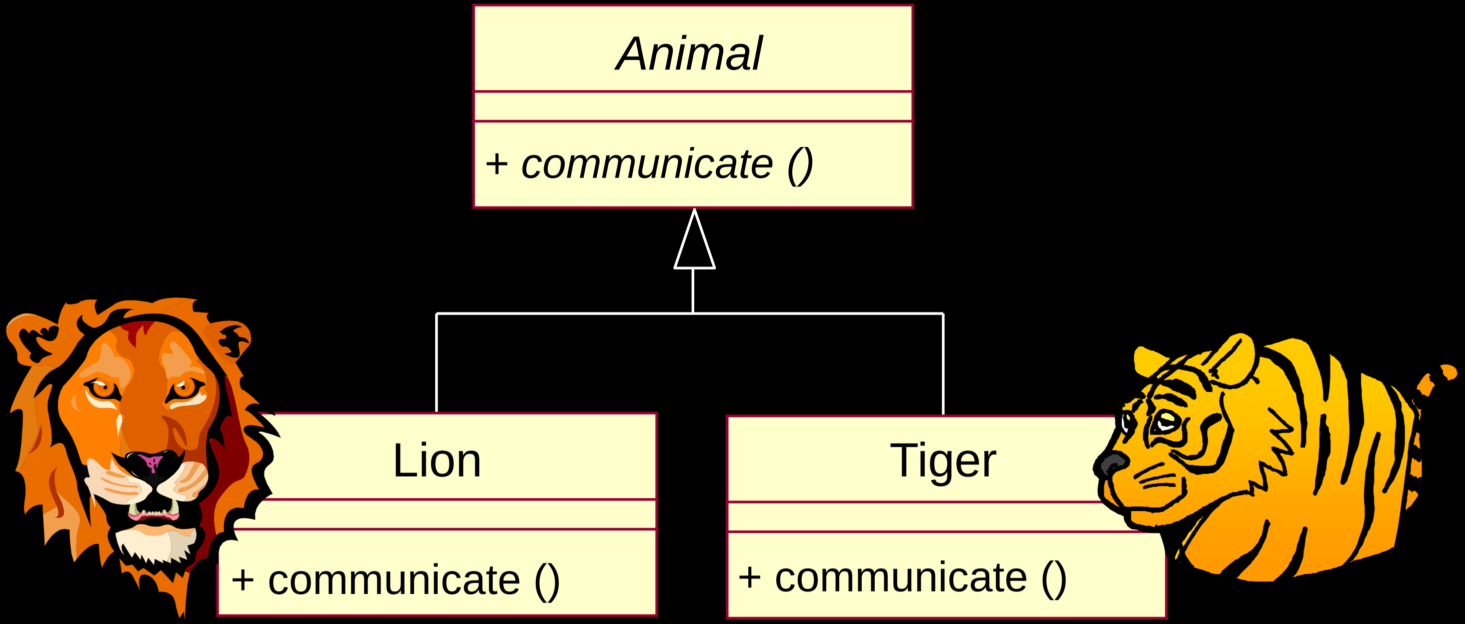
Review: What Is Polymorphism?

- 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?

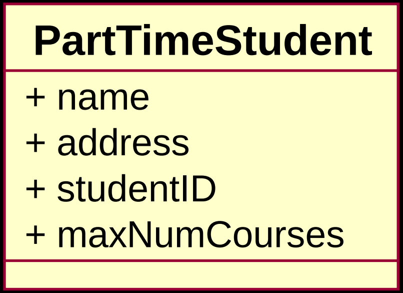
- Metamorphosis
  - 1. A change in form, structure, or function; specifically the physical change undergone by some animals, as of the tadpole to the frog.
  - 2. Any marked change, as in character, appearance, or condition.

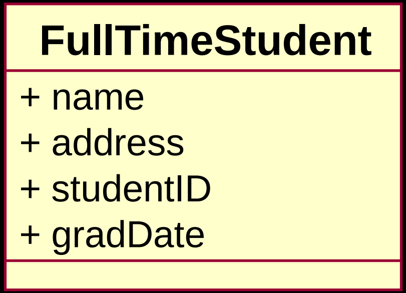
~ Webster's New World Dictionary, Simon & Schuster, Inc., 1979

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

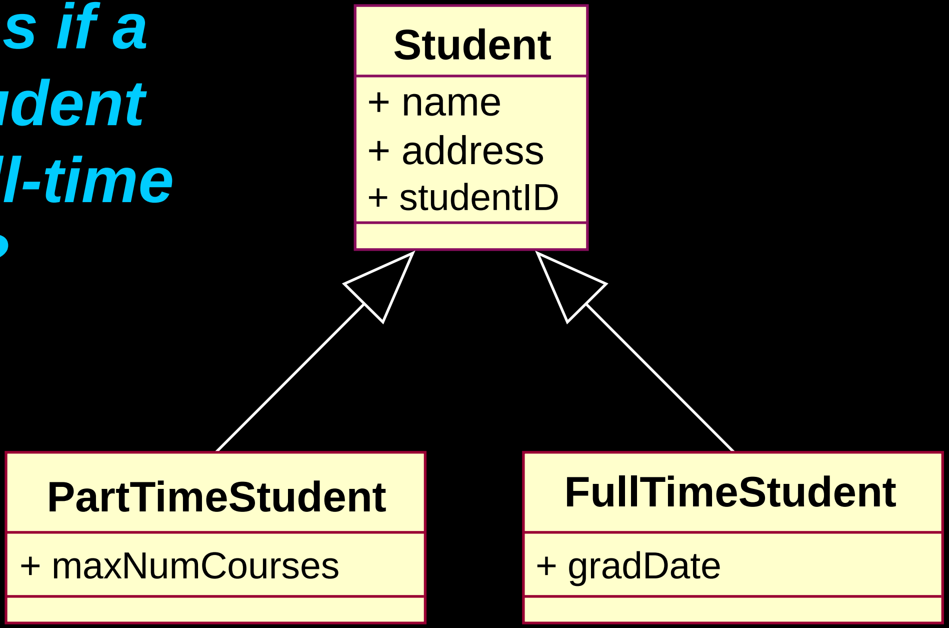




Modeling Metamorphosis: One Approach

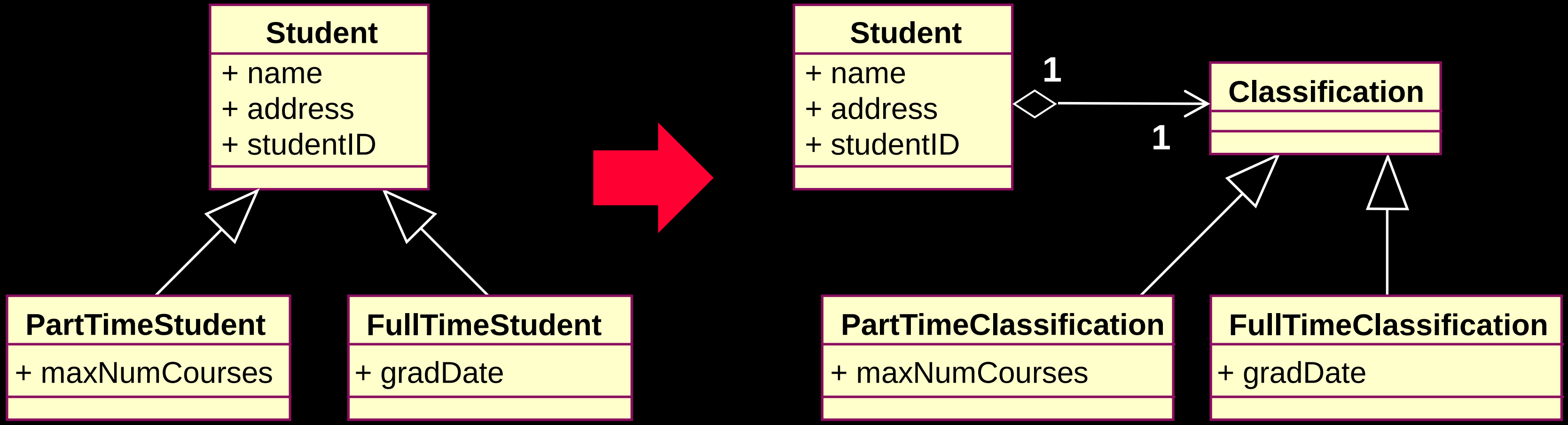
- A generalization relationship may be created

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



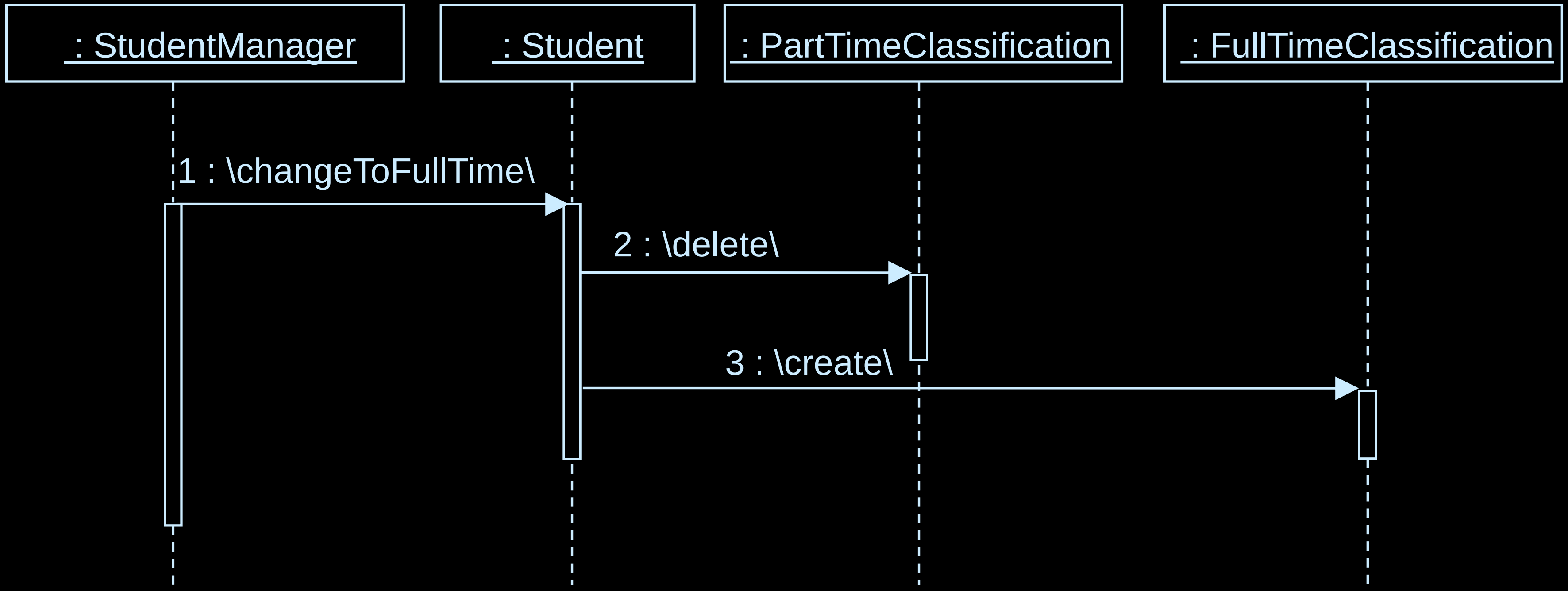
Modeling Metamorphosis: Another Approach

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



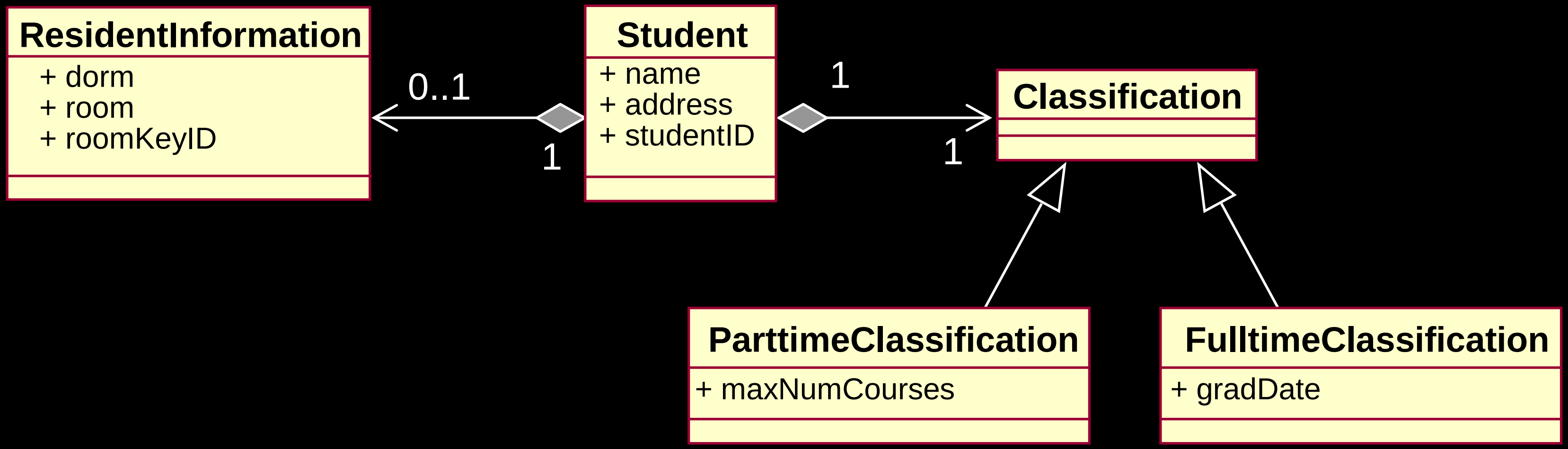
Modeling Metamorphosis: Another Approach (cont.)

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



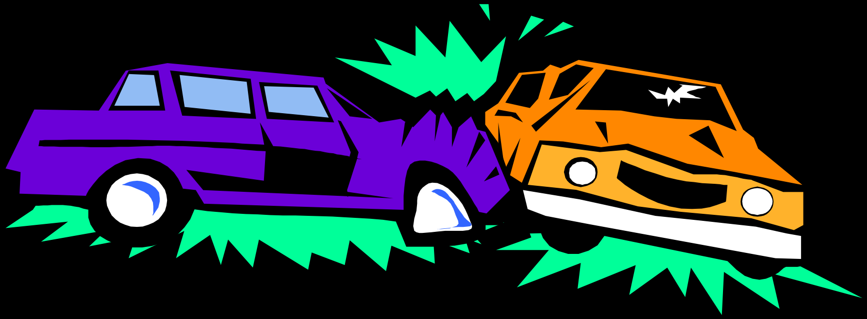
Metamorphosis and Flexibility

- 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 Generalizations
- ★ Resolve Use-Case Collisions
- Handle Non-Functional Requirements in General
- Checkpoints



Resolve Use-Case Collisions

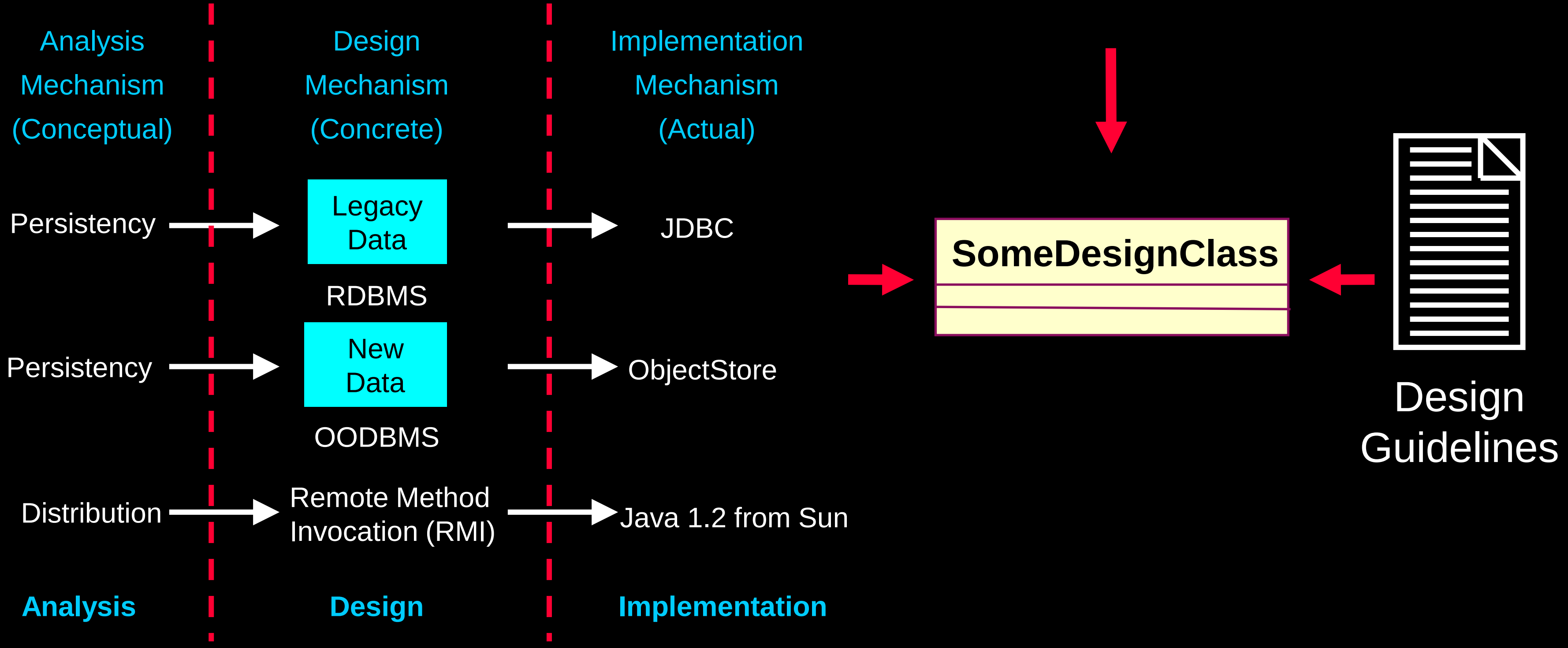
- Multiple use cases may simultaneously access design objects
- Options
  - Use synchronous messaging => first-come first-serve order processing
  - Identify operations (or code) to protect
  - Apply access control mechanisms
    - Message queuing
    - Semaphores (or “tokens”)
    - Other locking mechanism
- 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 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 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 statecharts.
- 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 statecharts created for every class?
- What are the major components of a statechart? Provide a brief description of each.
- What kind of relationship refinements occur?
- What is the difference between an association and a dependency?
- What is done with operations and attributes?

Exercise 2: Class Design

- Given the following:
  - The Use-Case Realization for a use case and/or the detailed design of a subsystem
  - The design of all participating design elements

(continued)

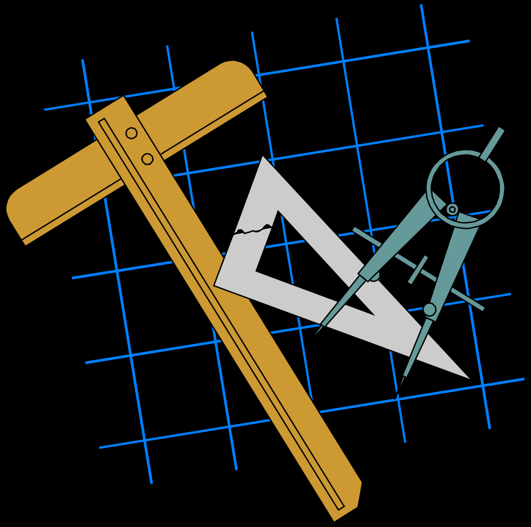
Exercise 2: Class Design (cont.)

- 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

(continued)

Exercise 2: Class Design (cont.)

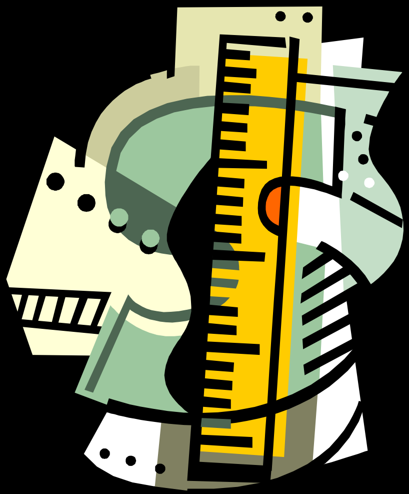
- Produce the following:
  - An updated VOPC, including the relationship refinements (generalization, dependency, association)



(continued)

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