Vietnam and Japan
Joint ICT HRD Program

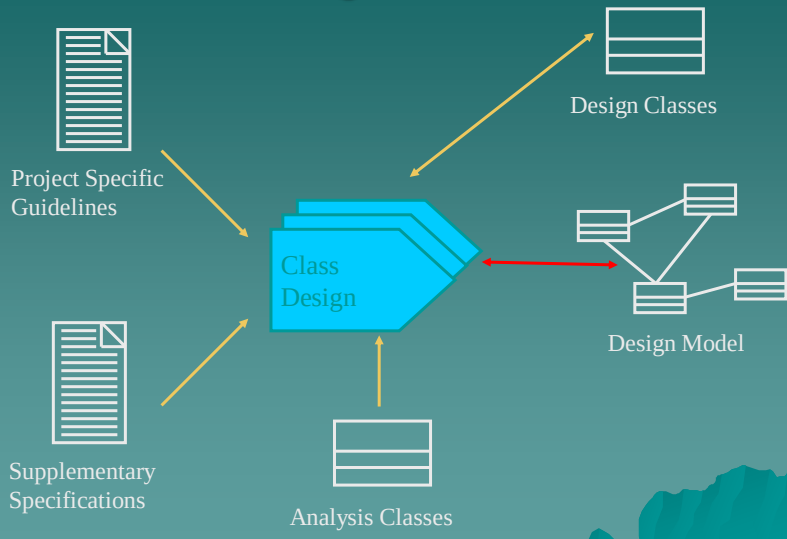
ITSS Software Development

# Chapter 6. Class design

Nguyen Thi Thu Trang
trangntt-fit@mail.hut.edu.vn



## Class Design Overview



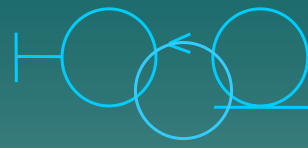
# Content

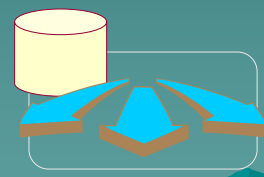
1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram



# Class Design Considerations

- Class stereotype
  - Boundary
  - Entity
  - Control
- Applicable design patterns





# 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!

5

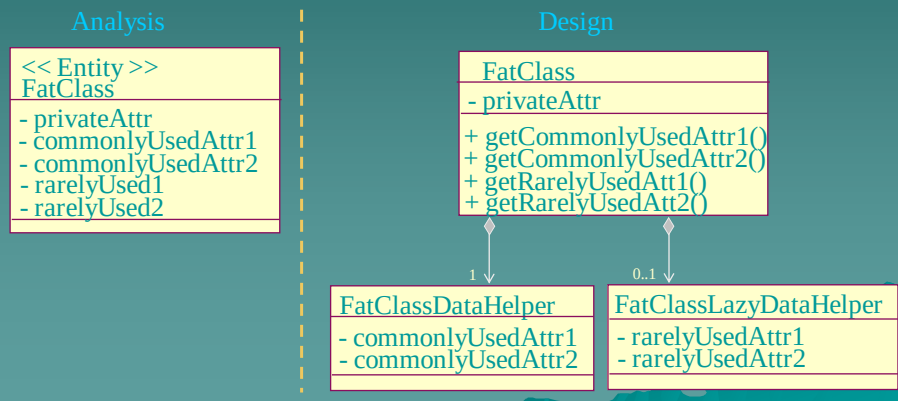
# Strategies for Designing Boundary Classes

- 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


6

# Strategies for Designing Entity Classes

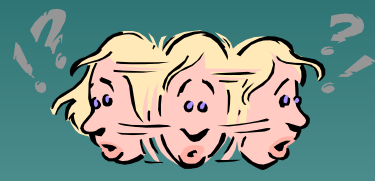
- Entity objects are often passive and persistent
- Performance requirements may force some re-factoring




7

# 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



8

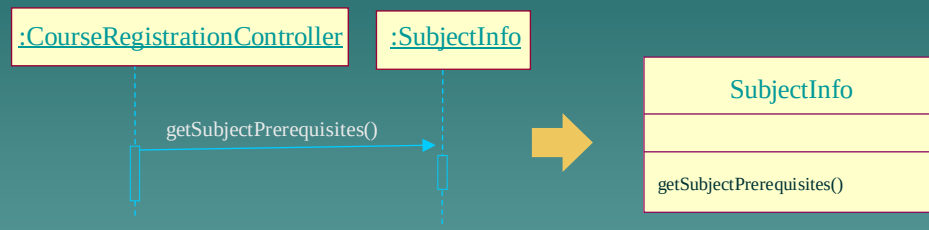
# Content

1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram



# 2.1. Define Operations

- 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

11

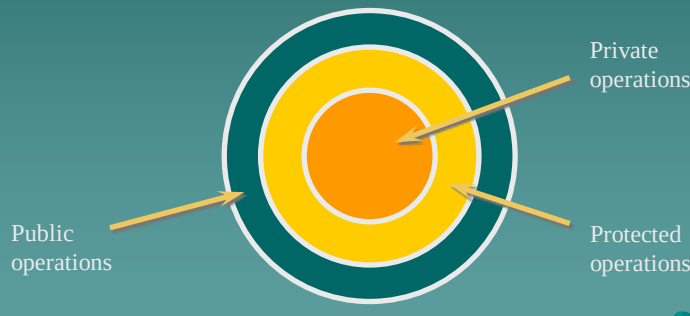
## Guidelines: Designing Operation Signatures

- 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”

12

# Operation Visibility

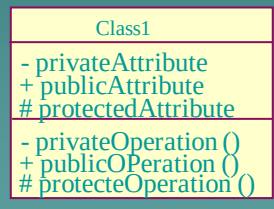
- Visibility is used to enforce encapsulation
- May be public, protected, or private




13

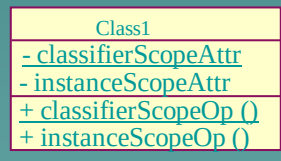
# How Is Visibility Noted?

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



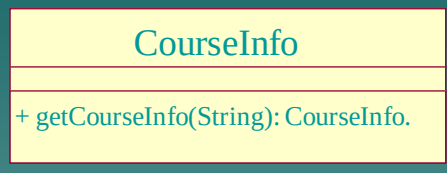
# Scope

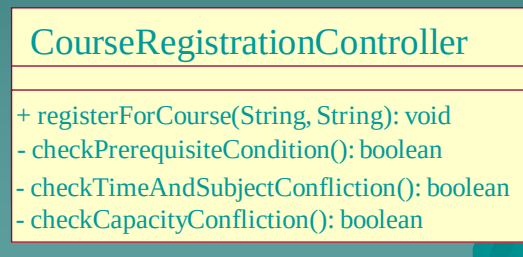
- Determines number of instances of the attribute/operation
  - Instance: one instance for each class instance
  - Classifier: one instance for all class instances
- Classifier scope is denoted by underlining the attribute/operation name




15

# Course Registration CS: Operations for CourseInfo. and CourseRegistrationController




16

## 2.2. 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

17

## Content

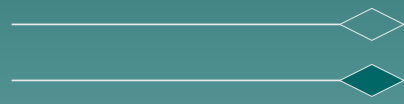
1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

18

# Class Relationships

- Association

  - Aggregation
    - Composition



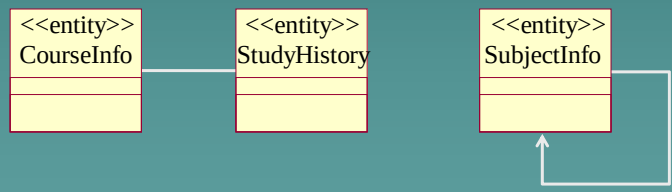
- Inheritance

- Dependency

19

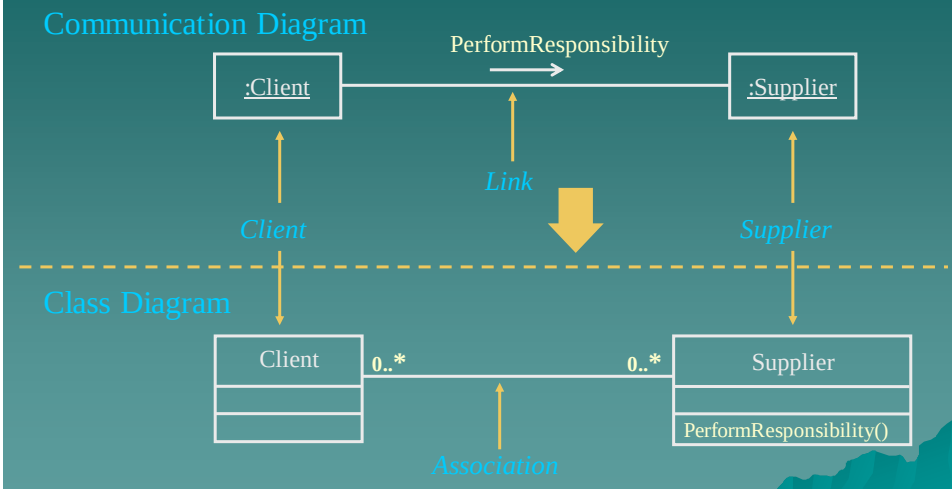
# 3.1. What is an Association?

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




20

# Finding Association

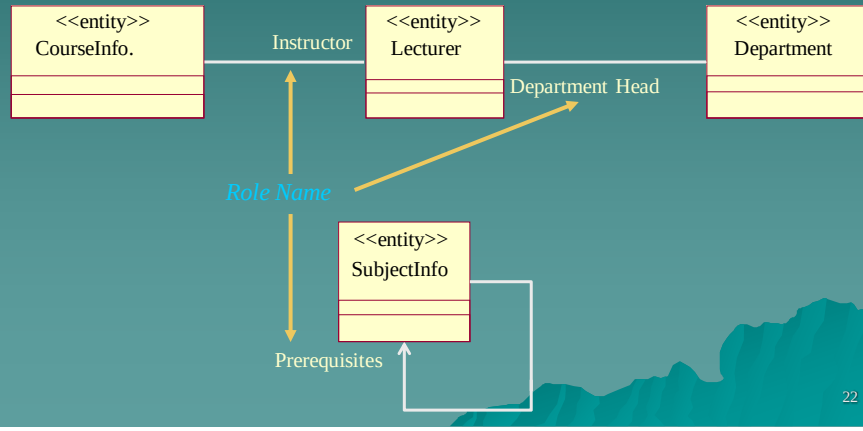



Relationship for every link!

21

# 3.1.1. What Are Roles?

- The “face” that a class plays in the association




22

# 3.1.2. What Is Multiplicity?

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


23

# Multiplicity Indicators

| | |
|---|---|
| Unspecified | |
| Exactly One | 1 |
| Zero or More | 0..* |
| Zero or More | * |
| One or More | 1..* |
| Zero or One (optional value) | 0..1 |
| Specified Range | 2..4 |
| Multiple, Disjoint Ranges | 2, 4..6 |

24

# What Does Multiplicity Mean?

- Multiplicity answers two questions:
  - Is the association mandatory or optional?
  - What is the minimum and maximum number of instances that can be linked to one instance?


25

# 3.1.3. Association Types

- Association
  - use-a
  - Objects of one class are associated with objects of another class

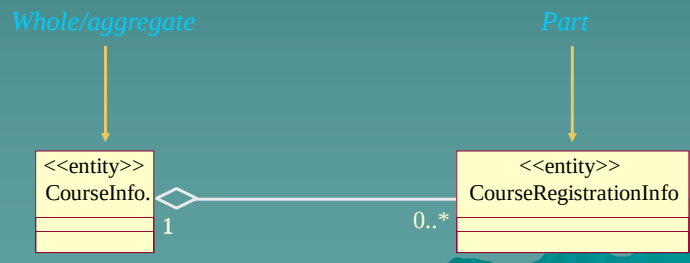

- Aggregation
  - has-a/is-a-part
  - Strong association, an instance of one class is made up of instances of another class


- Composition
  - Strong aggregation, the composed object can't be shared by other objects and dies with its composer
  - Share life-time


26

# Review: What Is Aggregation?

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts
  - An aggregation is an "is a part-of" relationship.
- Multiplicity is represented like other associations.



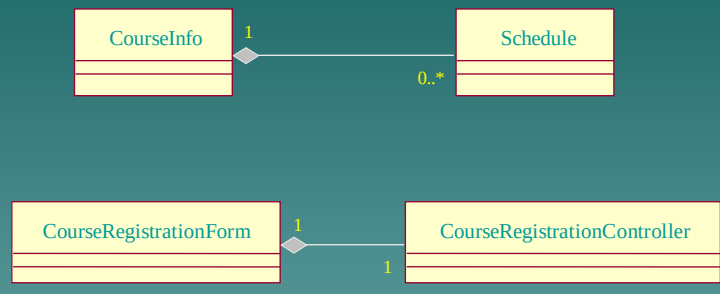

27

# Review: What is Composition?

- A special form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate.
- The whole "owns" the part and is responsible for the creation and destruction of the part.
  - The part is removed when the whole is removed.
  - The part may be removed (by the whole) before the whole is removed.


28

# “Register for course” Use case




29

# Association or Aggregation?

- If two objects are tightly bound by a whole-part relationship
  - The relationship is an aggregation.

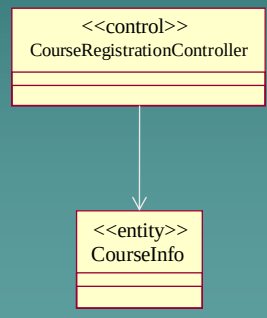

- If two objects are usually considered as independent, although they are often linked
  - The relationship is an association.


When in doubt, use association.

30

# 3.1.4. Navigability

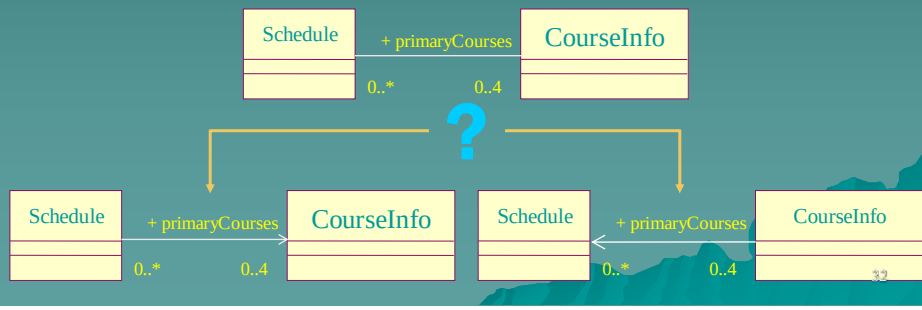
- Indicates that it is possible to navigate from an associating class to the target class using the association




31

# Navigability: Which Directions Are Really Needed?

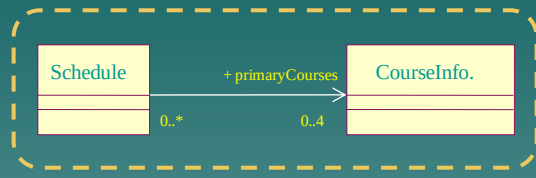
- Explore interaction diagrams
- Even when both directions seem required, one may work
  - Navigability in one direction is infrequent
  - Number of instances of one class is small



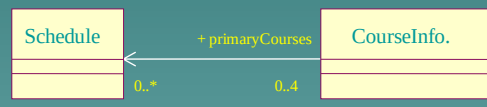

32

# 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

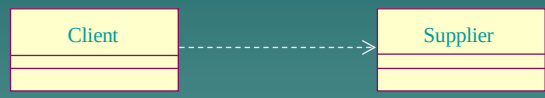



- Total number of CourseOfferings and Schedules are not small
- Must be able to navigate in both directions


33

# 3.2. 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

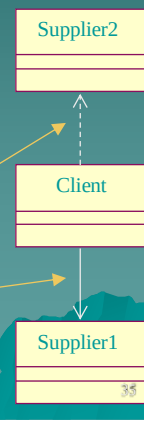
34

# 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

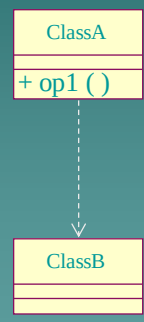
- An instance of an association is a link
  - All links become associations unless they have global, local, or parameter visibility
  - Relationships are context-dependent
- Dependencies are transient links with:
  - A limited duration
  - A context-independent relationship
  - A summary relationship

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.

36

# 3.2.1. Local Variable Visibility

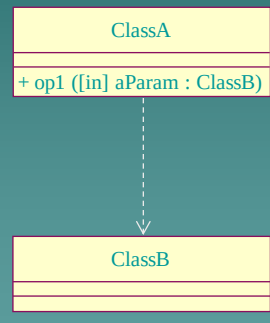
- The op1() operation contains a local variable of type ClassB




37

# 3.2.2. Parameter Visibility

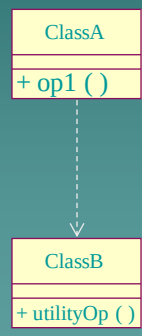
- The ClassB instance is passed to the ClassA instance




38

# 3.2.3. Global Visibility

- The ClassUtility instance is visible because it is global




39

# 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

40

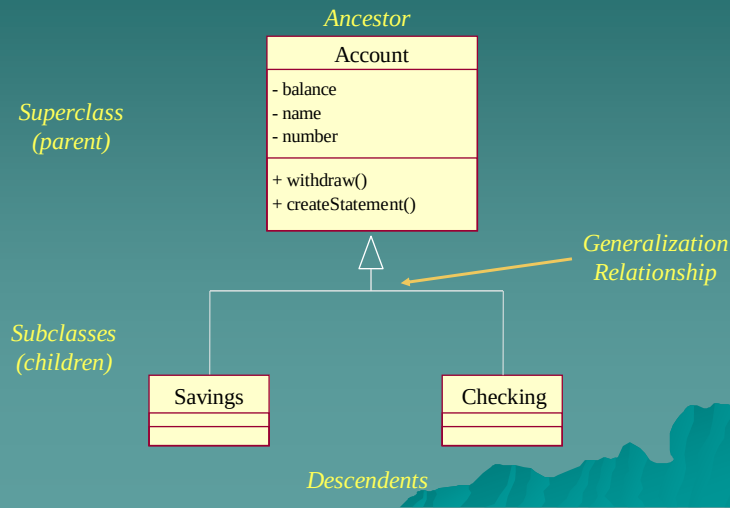
# 3.3. Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes.
- Defines a hierarchy of abstractions where a subclass inherits from one or more superclasses.
  - Single inheritance
  - Multiple inheritance
- Is an "is a kind of" relationship.

41

# Example: Single Inheritance

- One class inherits from another




42

# Content

1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

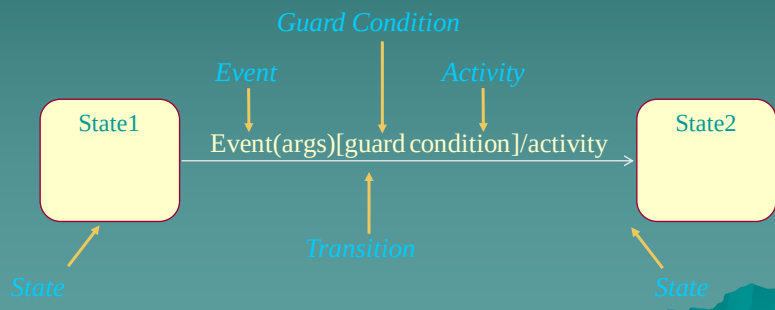


# 4. Define States

- 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?

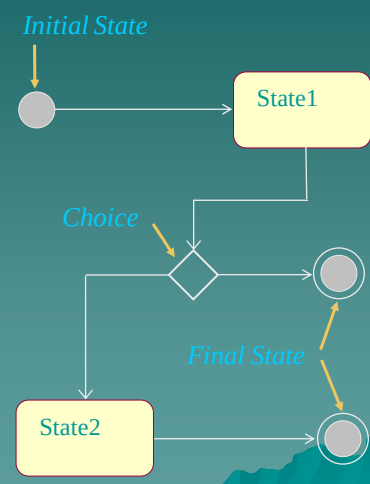
- A directed graph of states (nodes) connected by transitions (directed arcs)
- Describes the life history of a reactive object




45

# 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



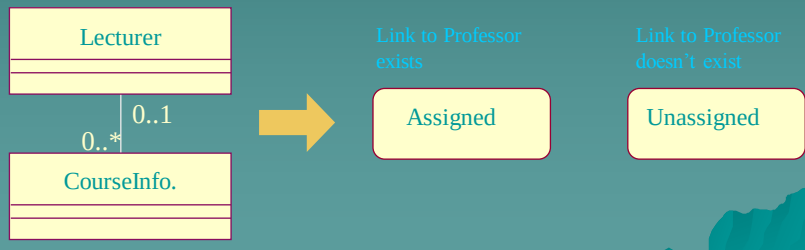

46

# Identify and Define the States

- Significant, dynamic attributes

The minimum number of students per course is 3

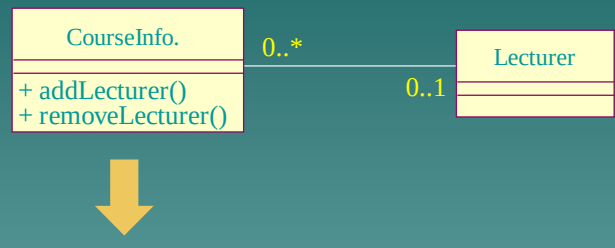

- Existence and non-existence of certain links




47

# Identify the Events

- Look at the class interface operations

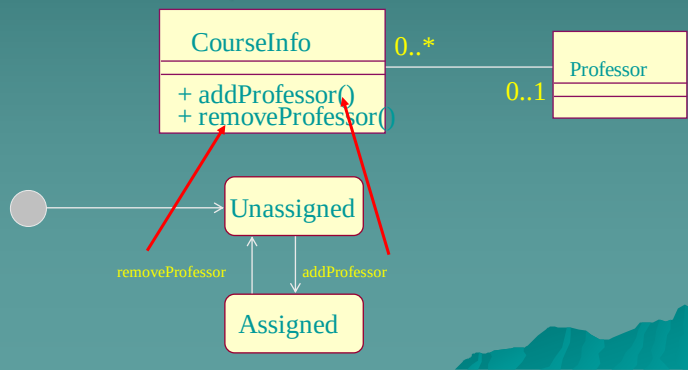



Events: addLecturer, removeLecturer

48

# Identify the Transitions

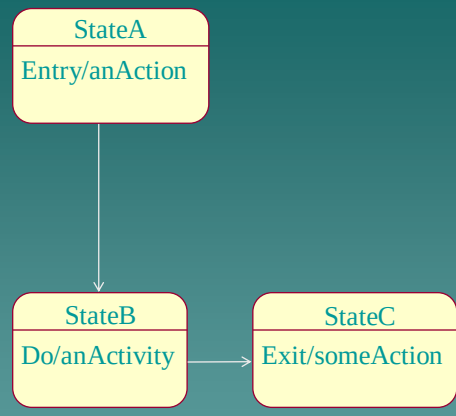
- For each state, determine what events cause transitions to what states, including guard conditions, when needed
- Transitions describe what happens in response to the receipt of an event




49

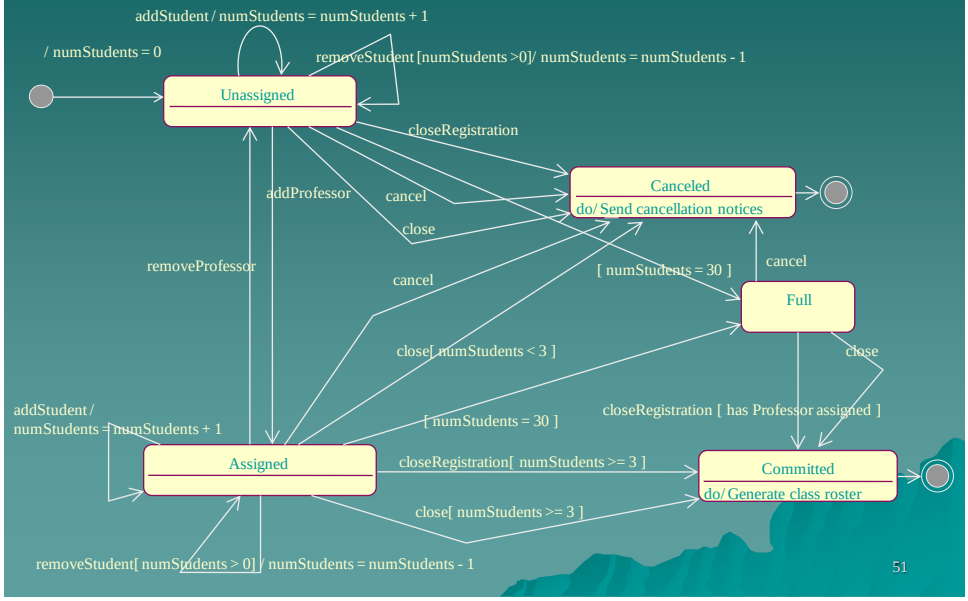
# Add Activities

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



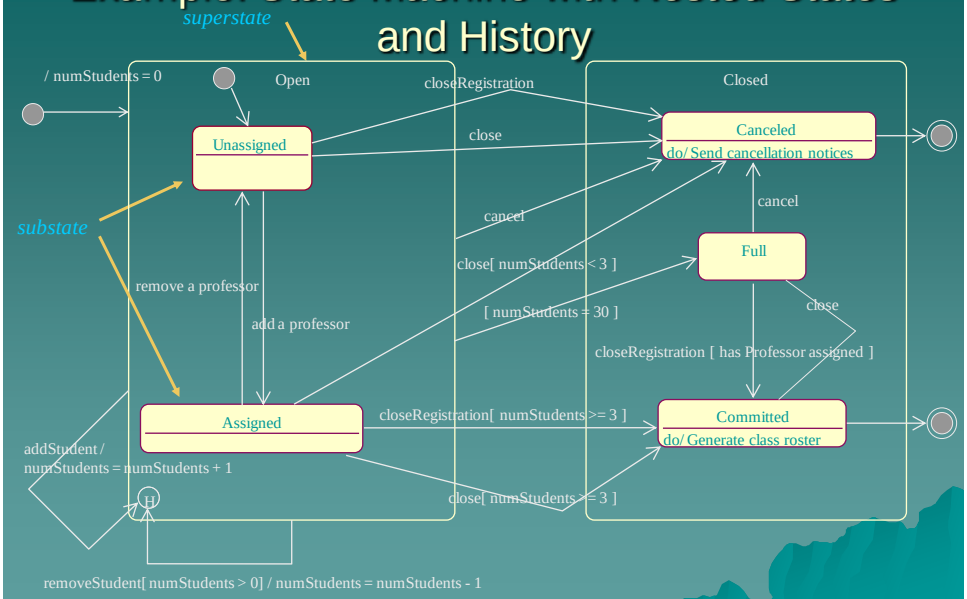

50

# Example: State Machines




51

# Example: State Machine with Nested States and History




52

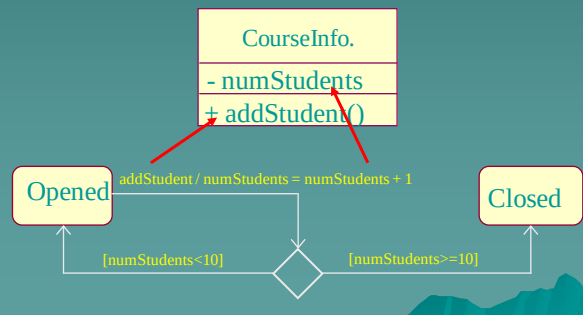
## Which Objects Have Significant State?

- 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

53

## 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



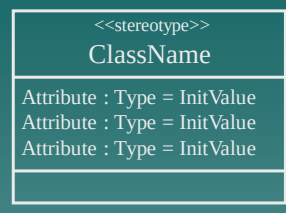

54

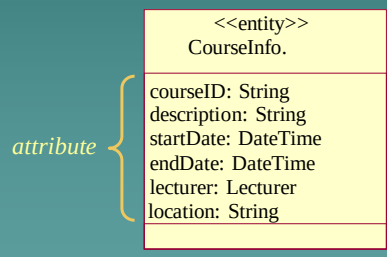
# Content

1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram



# Review: What Is an Attribute?



# 5.1. Finding Attributes

- Properties/characteristics of identified classes
- Information retained by identified classes
- “Nouns” that did not become classes
  - Information whose value is the important thing
  - Information that is uniquely "owned” by an object
  - Information that has no behavior

57

# 5.1. Finding Attributes (2)

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

58

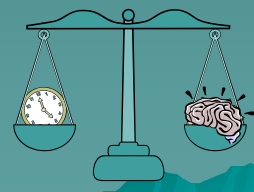
# 5.2. Attribute Representations

- 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: #

59

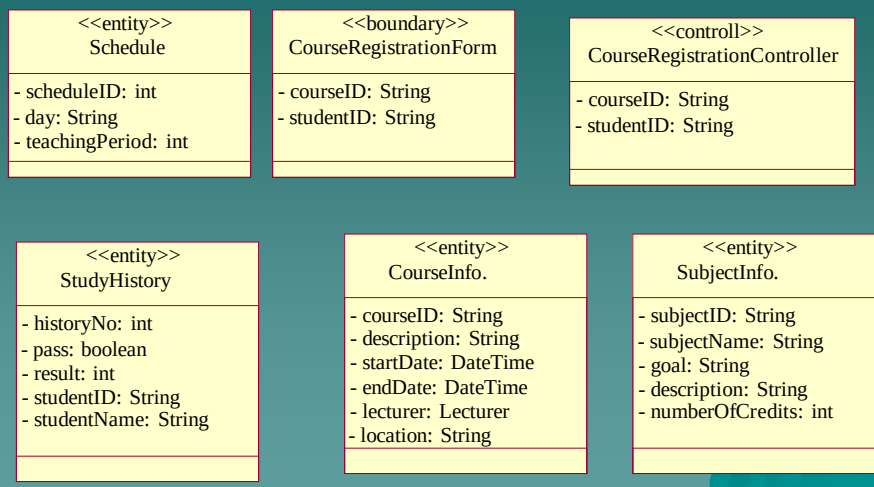
# 5.3. Derived Attributes

- 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



60

# Example: Define Attributes




61

# Content

1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

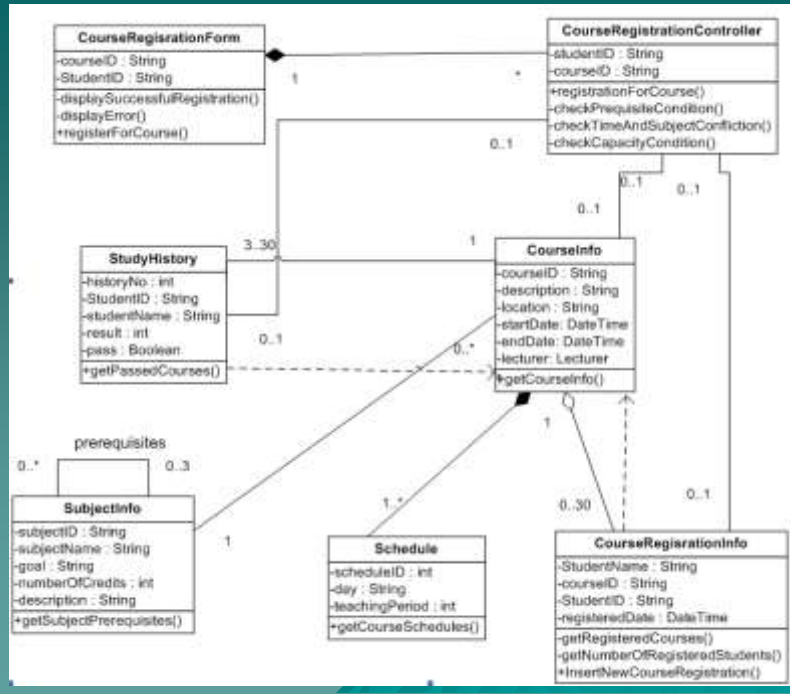
62

# 6. Class diagram

- Static view of a system
- When modeling the static view of a system, class diagrams are typically used in one of three ways, to model:
  - The vocabulary of a system
  - Collaborations
  - A logical database schema

63

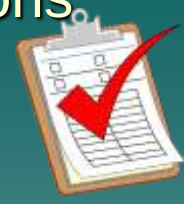
Class diagram for "Register for course" UC



## 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

# Question?