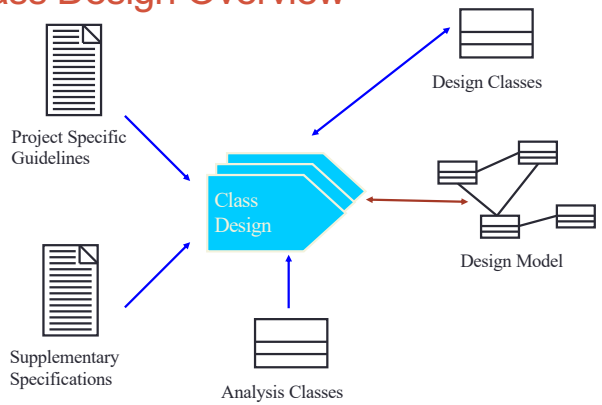
it4490 - Software DeSIGN AND CONSTRUCTION

8. CLASS DESIGN

Some slides extracted from IBM coursewares

1

Class Design Overview



2

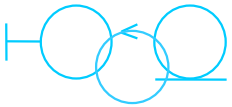
Content

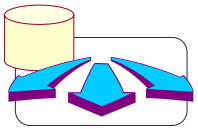
1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

3

Class Design Considerations

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





4

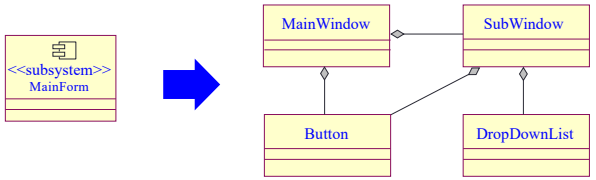
How Many Classes Are Needed?

- 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!

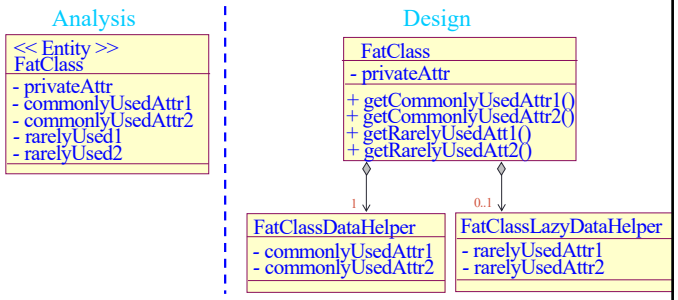
Strategies for Designing Boundary Classes

- 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

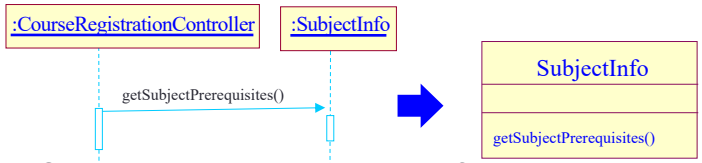
Content

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

9

2.1. Define Operations

- Messages displayed in interaction diagrams



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

10

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

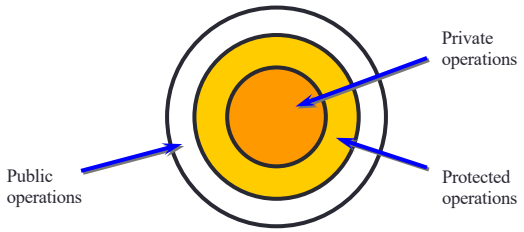
Guidelines: Designing Operation Signatures

- 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

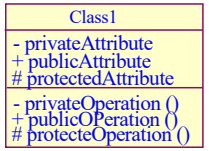
- Visibility is used to enforce encapsulation
- May be public, protected, or private



13

How Is Visibility Noted?

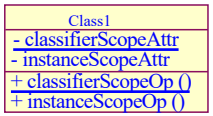
- The following symbols are used to specify export control:
  - \+ Public access
  - # Protected access
  - \- Private access



14

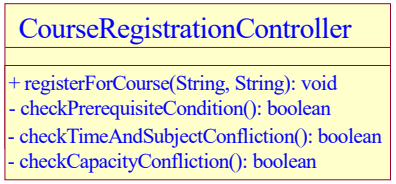
Scope

- 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 CourseOffering. and CourseRegistrationController



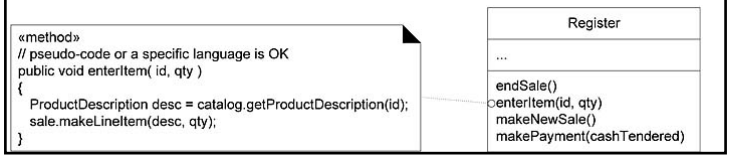
16

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

Operation vs. Method

- An operation is *not* a method. A UML **operation** is a *declaration*, with a name, parameters, return type, exceptions list, and possibly a set of *constraints* of pre and post-conditions. But, it isn't an implementation — rather, methods are implementations
- A method may be illustrated several ways, including:
  - in interaction diagrams, by the details and sequence of messages
  - in class diagrams, with a UML note symbol stereotyped with «method»

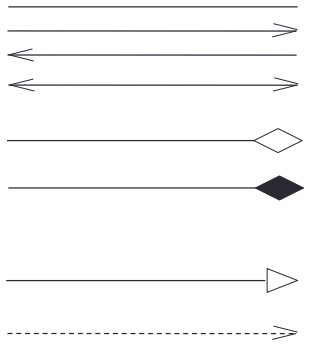


Content

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

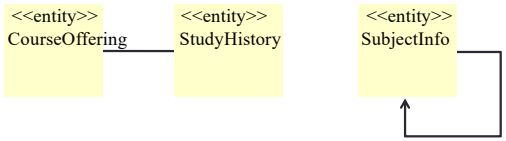
Class Relationships

- Association
  - Aggregation
    - Composition
- Inheritance
- Dependency



3.1. What is an Association?

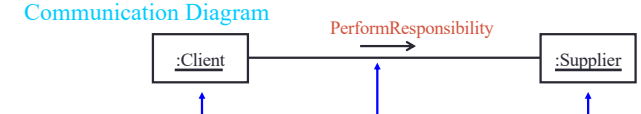
- The semantic relationship between two or more classifiers that specifies connections among their instances
- A structural relationship, specifying that objects of one thing are connected to objects of another

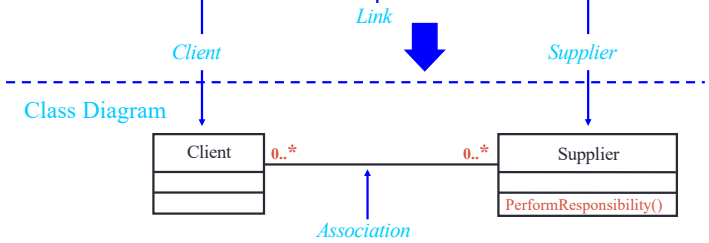


21

Finding Association

Communication Diagram





Relationship for every link!

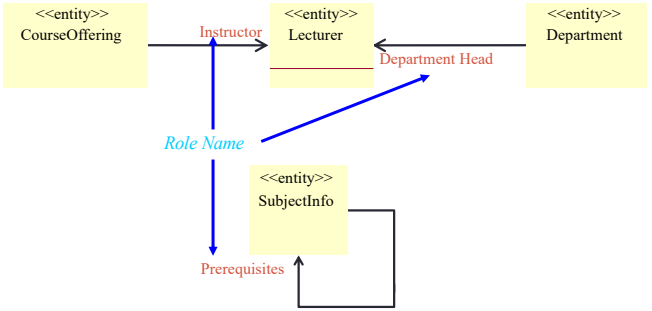
22

Name and direction of association

23

3.1.1. What Are Roles?

- The "face" that a class plays in the association



24

3.1.2. What Is Multiplicity?

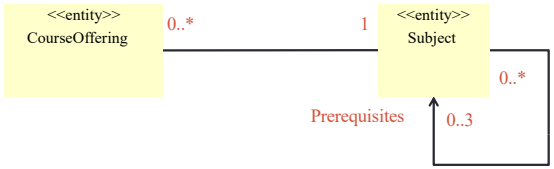
- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course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

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

What Does Multiplicity Mean?

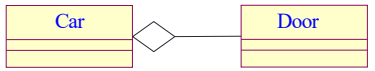
- Multiplicity answers two questions:
  - Is the association mandatory or optional?
  - What is the minimum and maximum number of instances that can be linked to one instance?



3.1.3. Association Types

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

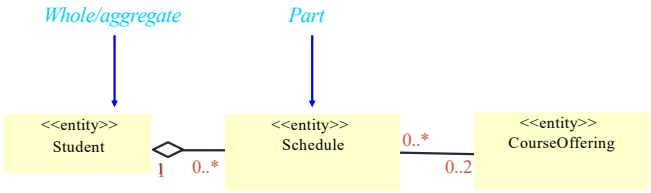
- Aggregation
  - has-a/is-a-part
  - Strong association, an instance of one class is **made up** of instances of another class



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

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.



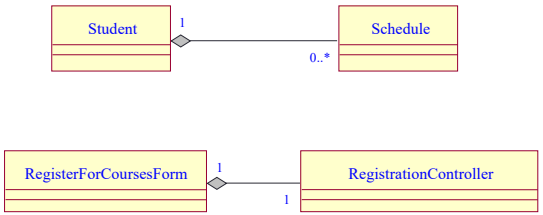
29

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.

30

"Register for course" Use case



31

Association or Aggregation?

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

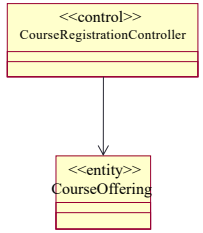
- If two objects are usually considered as independent, although they are often linked
  - The relationship is an association.

When in doubt, use association.

32

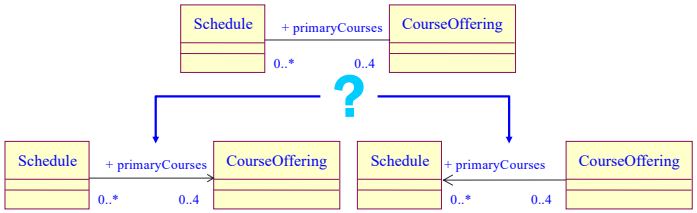
3.1.4. Navigability

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



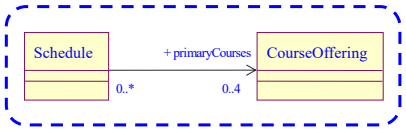
Navigability: Which Directions Are Really Needed?

- 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

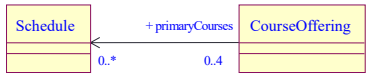


Example: Navigability Refinement

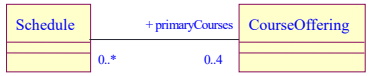
- Total number of Schedules is small, or
- Never need a list of the Schedules on which the CourseOffering appears



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



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



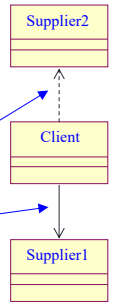
3.2. Dependency

- 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

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



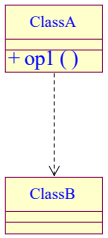
Associations vs. Dependencies in Collaborations

- 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.

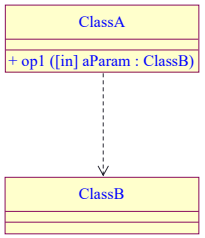
3.2.1. Local Variable Visibility

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



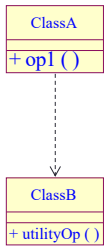
3.2.2. Parameter Visibility

- The ClassB instance is passed to the ClassA instance



3.2.3. Global Visibility

- The ClassUtility instance is visible because it is global



41

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

42

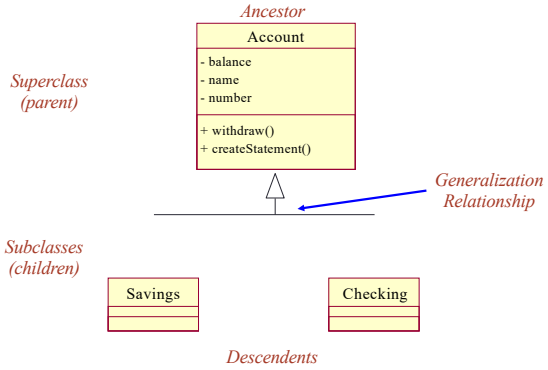
3.3. Generalization

- 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.

43

Example: Single Inheritance

- One class inherits from another



44

Content

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

45

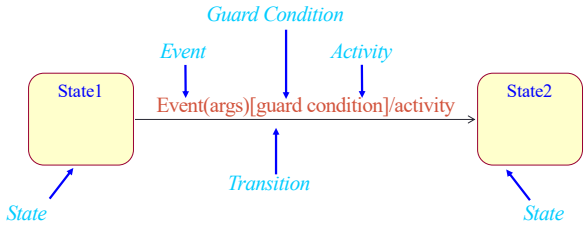
4. Define States

- 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?

46

What is a State Machine?

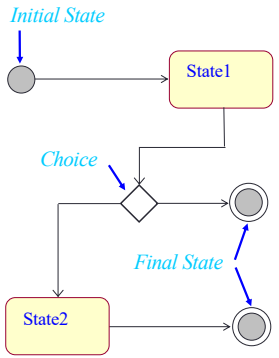
- A directed graph of states (nodes) connected by transitions (directed arcs)
- Describes the life history of a reactive object



47

Pseudo States

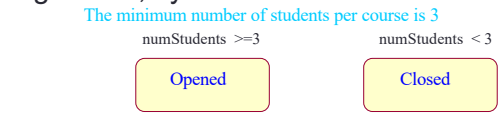
- Initial state
  - The state entered when an object is created
  - Mandatory, can only have one initial state
- Choice
  - Dynamic evaluation of subsequent guard conditions
  - Only first segment has a trigger
- Final state
  - Indicates the object's end of life
  - Optional, may have more than one



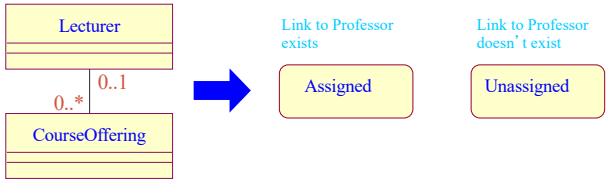
48

Identify and Define the States

- Significant, dynamic attributes

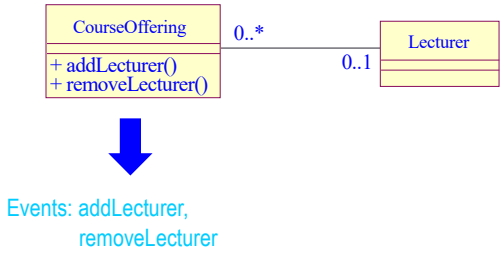


- Existence and non-existence of certain links



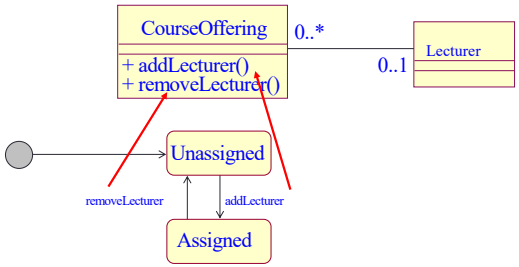
Identify the Events

- Look at the class interface operations



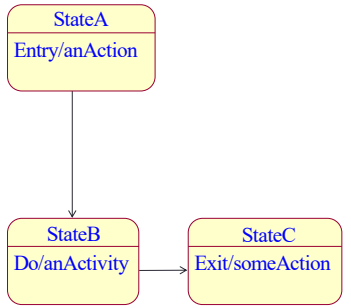
Identify the Transitions

- 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



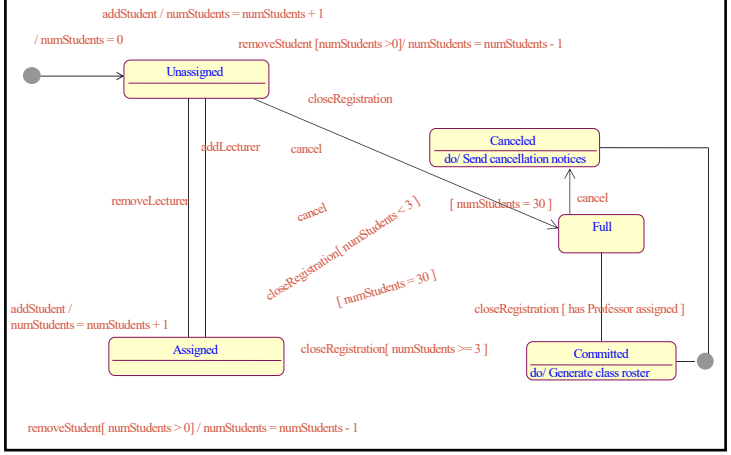
Add Activities

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



53

Example: State Machines



53

54

Which Objects Have Significant State?

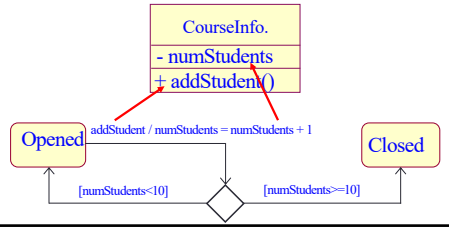
- Objects whose role is clarified by state transitions
- Complex use cases that are state-controlled
- It is not necessary to model objects such as:
  - Objects with straightforward mapping to implementation
  - Objects that are not state-controlled
  - Objects with only one computational state

54

55

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



55

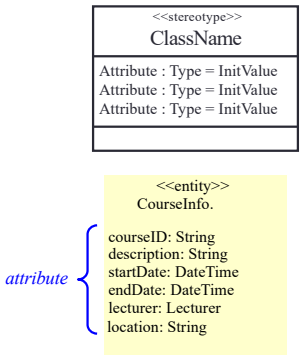
56

Content

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

56

Review: What Is an Attribute?



57

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

58

5.1. Finding Attributes (2)

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

59

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: #

60

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

Example: Define Attributes

<<entity>>
Schedule

- scheduleID: int
- day: String
- teachingPeriod: int

<<boundary>>
CourseRegistrationForm

- courseID: String
- studentID: String

<<controll>>
CourseRegistrationController

- courseID: String
- studentID: String

<<entity>>
StudyHistory

- historyNo: int
- pass: boolean
- result: int
- studentID: String
- studentName: String

<<entity>>
CourseOffering

- courseID: String
- description: String
- startDate: DateTime
- endDate: DateTime
- lecturer: Lecturer
- location: String

<<entity>>
SubjectInfo.

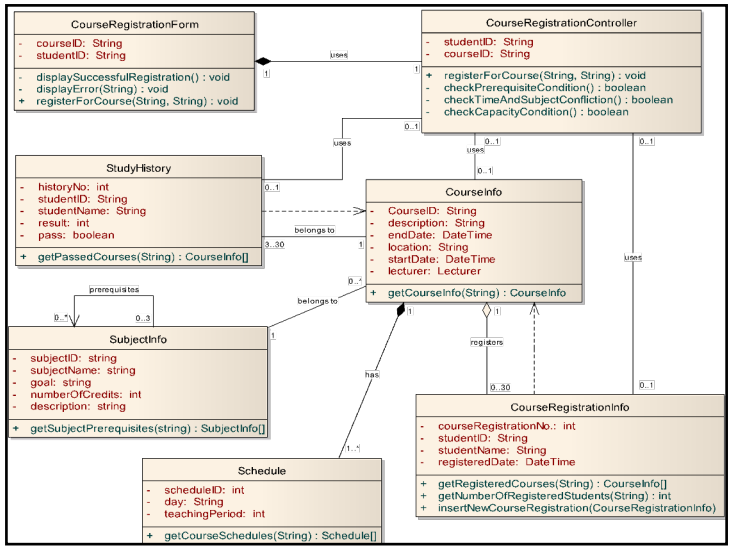
- subjectID: String
- subjectName: String
- goal: String
- description: String
- numberOfCredits: int

Content

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

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



65

Review: What Is a Package?

- A general purpose mechanism for organizing elements into groups.
- A model element that can contain other model elements.
- A package can be used:
  - To organize the model under development
  - As a unit of configuration management

66

Review points: 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

67

Review points: 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

68

Review points: Attributes

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

69

Review points: Relationships

- Descriptive role names
- Correct multiplicities

70

Question?

71