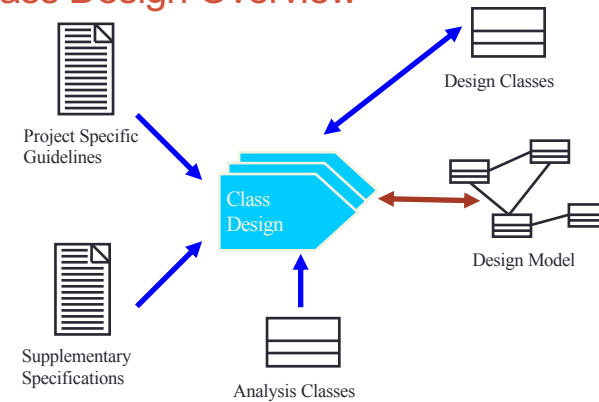SOFTWARE DESIGN AND CONSTRUCTION

6. CLASS DESIGN

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

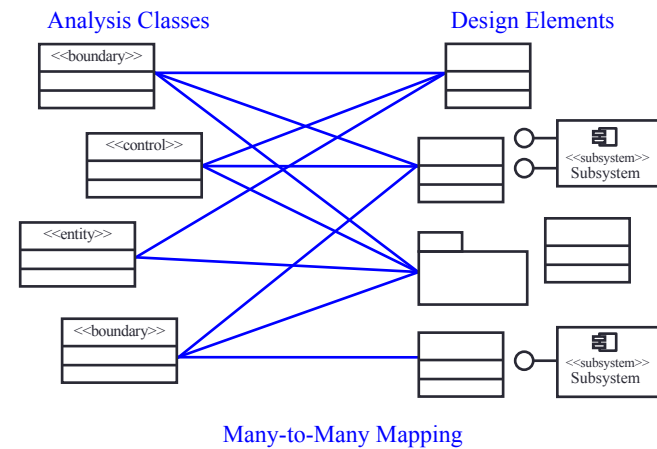*Some slides extracted from IBM coursewares*

Class Design Overview

Project Specific Guidelines

Supplementary Specifications

Analysis Classes

Class Design

Design Classes

Design Model

Content

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

From Analysis Classes to Design Elements

Analysis Classes

Design Elements

<<boundary>>

<<control>>

<<entity>>

<<boundary>>

<<subsystem>> Subsystem

<<subsystem>> Subsystem

Many-to-Many Mapping

Identifying Design Classes

- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
- More complex analysis classes may
  - Split into multiple classes
  - Become a package
  - Become a subsystem (discussed later)
  - Any combination …

Class Design Considerations

- Class stereotype
  - Boundary
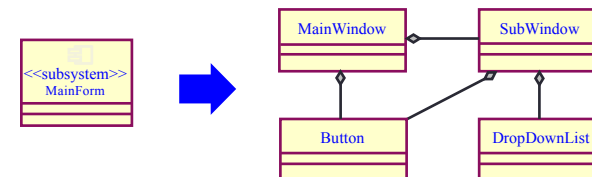 - Entity
  - Control
- Applicable design patterns
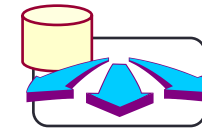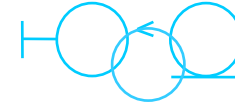
How Many Classes Are Needed?

- 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!
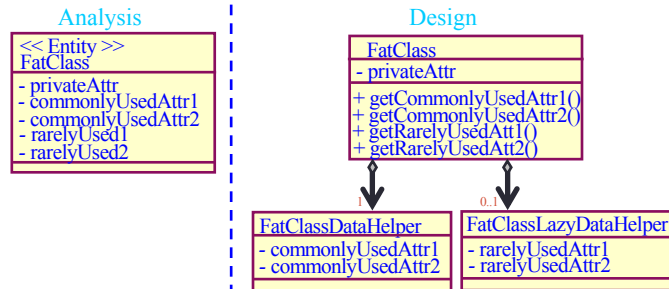
Strategies for Designing Boundary Classes

- 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

| <<subsystem>> MainForm | ⟹ | MainWindow | SubWindow |
| | | Button | DropDownList |

2

Strategies for Designing Entity Classes

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

Analysis

<< Entity >> FatClass
- privateAttr
- commonlyUsedAttr1 - commonlyUsedAttr2 - rarelyUsed1 - rarelyUsed2

Design

FatClass
- privateAttr
+ getCommonlyUsedAttr1() + getCommonlyUsedAttr2() + getRarelyUsedAtt1() + getRarelyUsedAtt2()

FatClassDataHelper
- commonlyUsedAttr1 - commonlyUsedAttr2

0..1

| FatClassLazyDataHelper |
| --- |
| - rarelyUsedAttr1
- rarelyUsedAttr2 |

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

---
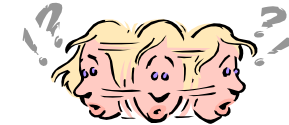
Review: Class and Package

- What is a class?
  - A description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics

Class Name

- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain other model elements
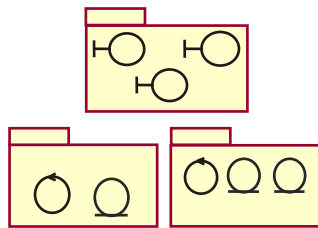
Package Name
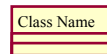
Group Design Classes in Packages

- You can base your packaging criteria on a number of different factors, including:
  - Configuration units
  - Allocation of resources among development teams
  - Reflect the user types
  - Represent the existing products and services the system uses
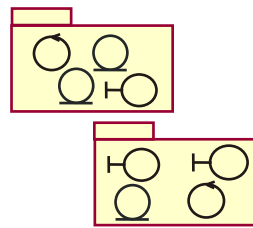
Package C

Package B

Package A

3
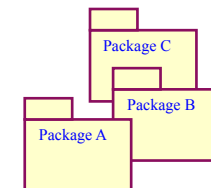
Packaging Tips: Boundary Classes

| If it is **likely** the system interface will undergo considerable changes	If it is **unlikely** the system interface will undergo considerable changes
Boundary classes placed in separate packages	Boundary classes packaged with functionally related classes

---

Packaging Tips: Functionally Related Classes

- Criteria for determining if classes are functionally related:
  - Changes in one class' behavior and/or structure necessitate changes in another class
  - Removal of one class impacts the other class
  - Two objects interact with a large number of messages or have a complex intercommunication
  - A boundary class can be functionally related to a particular entity class if the function of the boundary class is to present the entity class
  - Two classes interact with, or are affected by changes in the same actor

---

Packaging Tips: Functionally Related Classes (continued)

- Criteria for determining if classes are functionally related (continued):
  - Two classes have relationships between each other
  - One class creates instances of another class
- Criteria for determining when two classes should **NOT** be placed in the same package:
  - Two classes that are related to different actors should not be placed in the same package
  - An optional and a mandatory class should not be placed in the same package

---

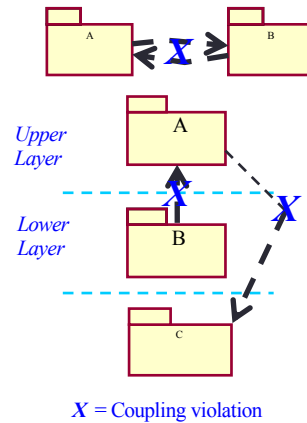## Package Dependencies: Package Element Visibility
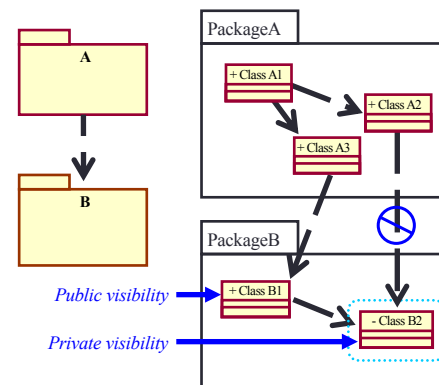
Only public classes can be referenced outside of the owning package
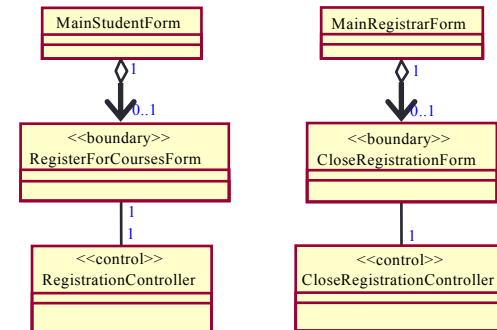
Public visibility →

Private visibility →

OO Principle: Encapsulation

Package Coupling: Tips

- Packages should not be cross-coupled

- Packages in lower layers should not be dependent upon packages in upper layers

- In general, dependencies should not skip layers

A   *X*   B

Upper Layer

A

Lower Layer

B

X   *X*

C

X = Coupling violation

Example: Registration Package

MainStudentForm — 1 — 0..1 — <<boundary>> RegisterForCoursesForm — 1 1 — <<control>> RegistrationController

MainRegistrarForm — 1 — 0..1 — <<boundary>> CloseRegistrationForm — 1 — <<control>> CloseRegistrationController

---

Example: University Artifacts Package: Generalization

<<entity>> Student — <<entity>> FulltimeStudent — <<entity>> ParttimeStudent

<<entity>> ScheduleOfferingInfo — <<entity>> PrimaryScheduleOfferingInfo

---

Example: University Artifacts Package: Associations

<<entity>> Student — <<entity>> Schedule

0..* — 0..*

primaryCourses 0..4 — alternateCourses ..2

<<entity>> Professor — instructor 0..1 — <<entity>> CourseOffering 0..* — <<entity>> Course 0..* — Prerequisites

0..* 1 — CourseOfferingList
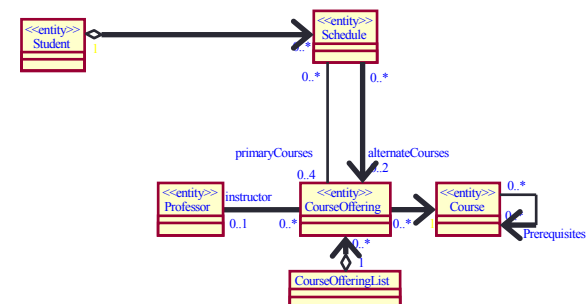
Example: External System Interfaces Package

<<Interface>> IBillingSystem

<<Interface>> ICourseCatalogSystem

Content

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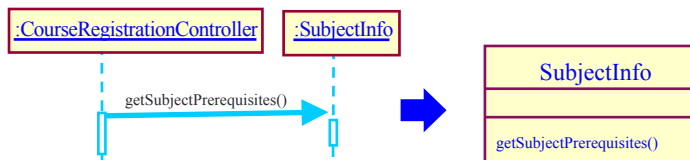2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

2.1. Define Operations

• Messages displayed in interaction diagrams

:CourseRegistrationController   :SubjectInfo

getSubjectPrerequisites()

SubjectInfo
getSubjectPrerequisites()

• 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

Program documentation

```java
/**
 * Parse XML data into a DOM representation, taking local resources and
Schemas into account.
 * @param inputData a string representation of the XML data to be parsed.
 * @param validating whether to Schema-validate the XML data
 * @return the DOM document resulting from the parse
 * @throws ParserConfigurationException if no parser could be created
 * @throws SAXException if there was a parse error
 * @throws IOException if there was a problem reading from the string
 */
public static Document parseDocument(String inputData, boolean validating)
throws ParserConfigurationException, SAXException, IOException {
        //Change to UnicodeReader for utf-8
        ...
}
```

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

Private operations

Public operations

Protected operations

How Is Visibility Noted?

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

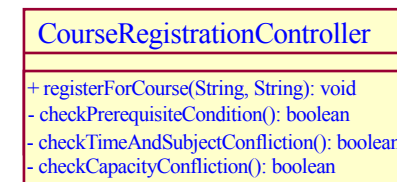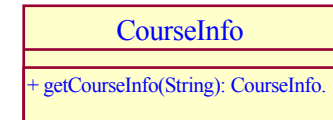| Class1 |
| --- |
| - privateAttribute
+ publicAttribute
# protectedAttribute |
| - privateOperation ()
+ publicOPeration ()
# protecteOperation () |

7

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

Class1
- classifierScopeAttr
- instanceScopeAttr
+ classifierScopeOp ()
+ instanceScopeOp ()
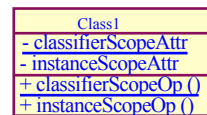
Course Registration CS: Operations for CourseInfo. and CourseRegistrationController

CourseInfo
+ getCourseInfo(String): CourseInfo.

CourseRegistrationController
+ registerForCourse(String, String): void
- checkPrerequisiteCondition(): boolean
- checkTimeAndSubjectConfliction(): boolean
- checkCapacityConfliction(): boolean

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

Content

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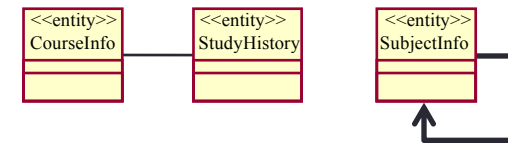4. Define States
5. Define Attributes
6. Class Diagram

Class Relationships

- Association

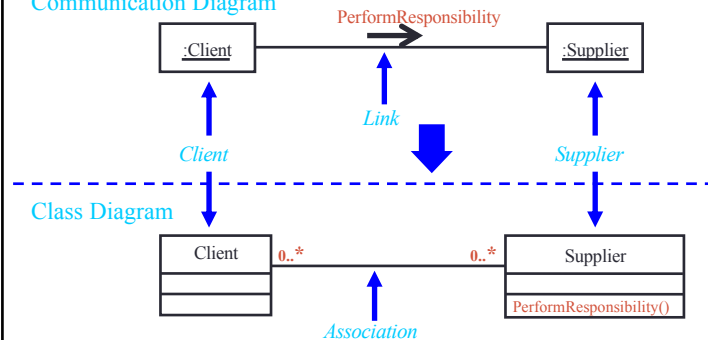- Aggregation
  - Composition

- 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

<<entity>> CourseInfo — <<entity>> StudyHistory   <<entity>> SubjectInfo

Finding Association

Communication Diagram

PerformResponsibility

:Client → :Supplier

Link

Client   *Supplier*

Class Diagram

Client   0..*   0..*   Supplier

PerformResponsibility()

Association

Relationship for every link!

3.1.1. What Are Roles?

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

<<entity>> CourseInfo.   Instructor   <<entity>> Lecturer   Department Head   <<entity>> Department

Role Name

<<entity>> SubjectInfo

Prerequisites
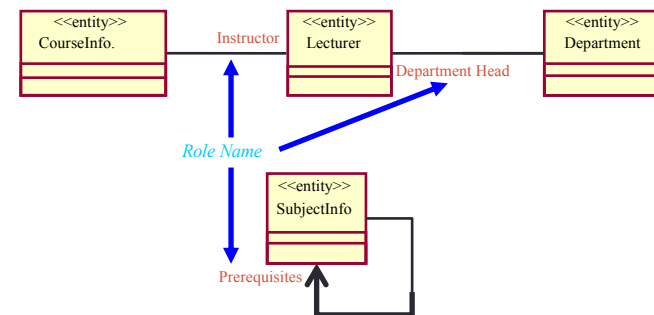
3.1.2. What Is Multiplicity?

- 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.
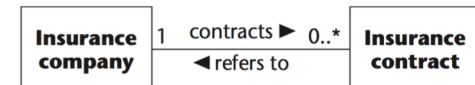
Lecturer	instructor	CourseInfo.
	0..1 —————————— 0..*	

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?

```
<<entity>>           0..*        1    <<entity>>
CourseInfo.  ——————————————————————  Subject
                                                  0..*
                        Prerequisites      0..3
```

Java implementation

```
Insurance        1   contracts ▶  0..*   Insurance
company              ◀ refers to         contract
```

```java
//InsuranceCompany.java file
public class InsuranceCompany
{
    // Many multiplicity can be implemented using Collection
    private List<InsuranceContract> contracts;


    /* Methods */
}
// InsuranceContract.java file
public class InsuranceContract
{
    private InsuranceCompany refers_to;

    /* Methods */
}
```
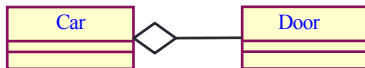
3.1.3. Association Types

- Association

Car	Driver

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

Car	◇	Door

  - 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

Car	◆	Body

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.

Whole/aggregate   *Part*

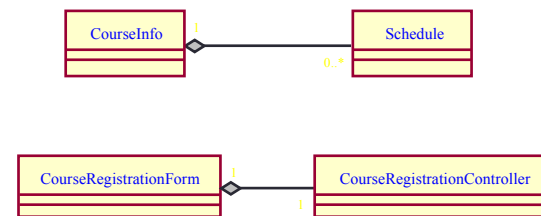| <<entity>> CourseInfo.	◇ ————	<<entity>> CourseRegistrationInfo

1   0..*
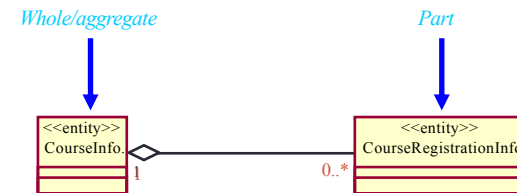
Review: What is Composition?

- 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.

Whole	◆ ————	Part

"Register for course" Use case

CourseInfo	◇ ————	Schedule

0..*

CourseRegistrationForm	◇ ————	CourseRegistrationController

Slide 45

Association or Aggregation?

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

```
    ┌──────┐                    ┌──────┐
    │ Car  │◇───────────────────│ Door │
    ├──────┤                    ├──────┤
    └──────┘                    └──────┘
      1                  0..2,4
```

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

```
    ┌──────┐                    ┌──────┐
    │ Car  │────────────────────│ Door │
    ├──────┤                    ├──────┤
    └──────┘                    └──────┘
      1                  0..2,4
```

When in doubt, use association.

Slide 46

Aggregation – Java implementation

```java
class Car {
    private List<Door> doors;
    Car(String name, List<Door> doors) {
        this.doors = doors;
    }

    public List<Door> getDoors() {
        return doors;
    }
}
```

Slide 47

Composition – Java implementation

```java
final class Car {
    // For a car to move, it need to have a engine.
    private final Engine engine; // Composition
    //private Engine engine;     // Aggregation

    Car(Engine engine) {
        this.engine = engine;
    }

    // car start moving by starting engine
    public void move() {
        //if(engine != null)
        {
            engine.work();
            System.out.println("Car is moving ");
        }
    }
}
```
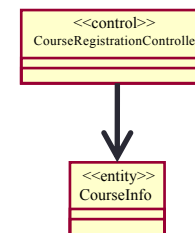
```java
class Engine {
    // starting an engine
    public void work() {
        System.out.println("Engine of car has been started ");
    }
}
```

Slide 48

48

3.1.4. Navigability

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

```
    ┌────────────────────────────┐
    │        <<control>>         │
    │ CourseRegistrationController│
    ├────────────────────────────┤
    └────────────────────────────┘
                  │
                  ▼
         ┌──────────────┐
         │  <<entity>>  │
         │  CourseInfo  │
         ├──────────────┤
         └──────────────┘
```

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
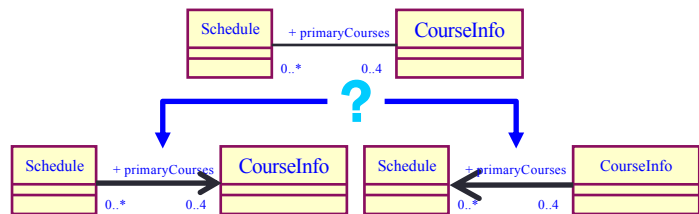
Example: Navigability Refinement

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

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

- Total number of CourseInfo and Schedules are not small
- Must be able to navigate in both directions
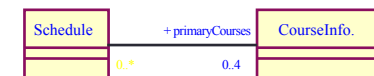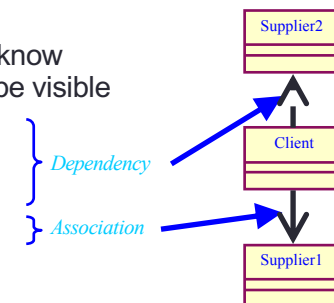
3.2. Dependency

- 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
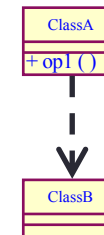
Dependency

Association

13

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.
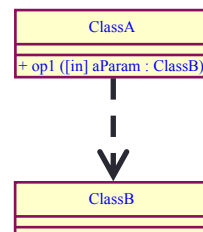
3.2.1. Local Variable Visibility

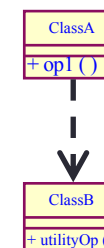- The op1() operation contains a local variable of type ClassB

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

14

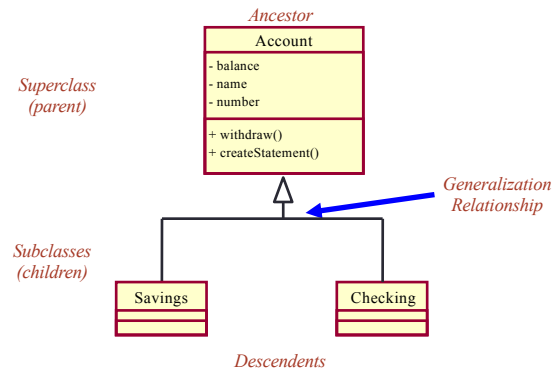## Identifying Dependencies: Considerations

- 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

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.

Example: Single Inheritance

- One class inherits from another

Ancestor

Superclass (parent)

Account
- balance
- name
- number
+ withdraw()
+ createStatement()

Generalization Relationship

Subclasses (children)

Savings   Checking

Descendents

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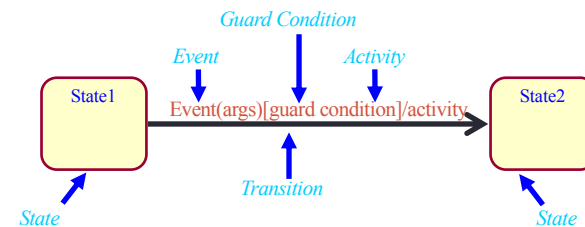
1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

15

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

Guard Condition
Event
Activity

State1
Event(args)[guard condition]/activity
State2

State
Transition
State

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

Initial State

State1

Choice

Final State
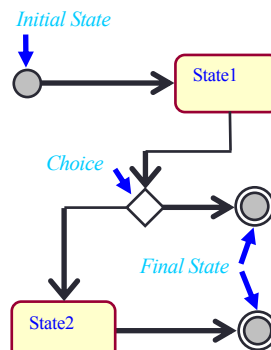
State2

Identify and Define the States

- Significant, dynamic attributes

The minimum number of students per course is 3

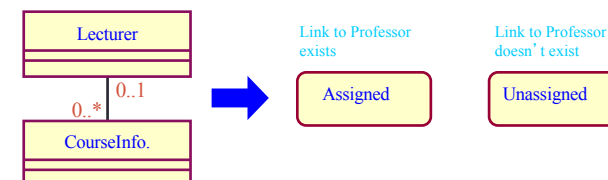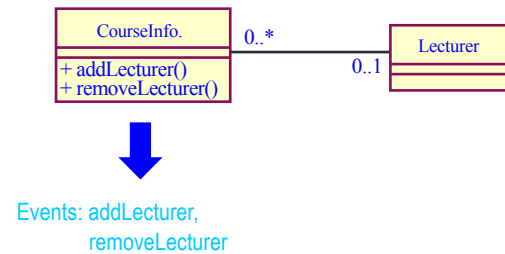numStudents >=3   numStudents < 3

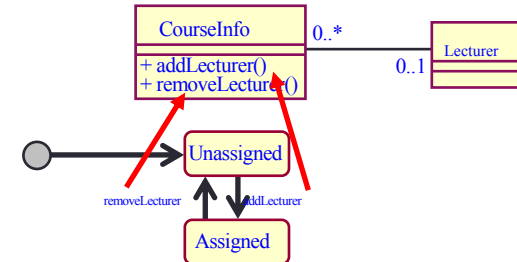Opened   Closed

- Existence and non-existence of certain links

Lecturer

0..1

0..*

CourseInfo.

Link to Professor exists

Assigned

Link to Professor doesn't exist

Unassigned

Identify the Events

- Look at the class interface operations

CourseInfo.
+ addLecturer() + removeLecturer()

0..*   0..1   Lecturer

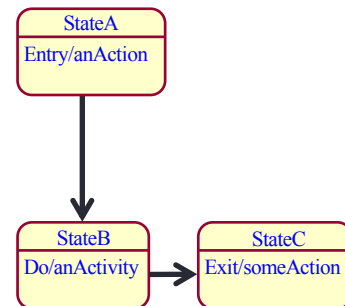Events: addLecturer,
　　　　 removeLecturer

Identify the Transitions

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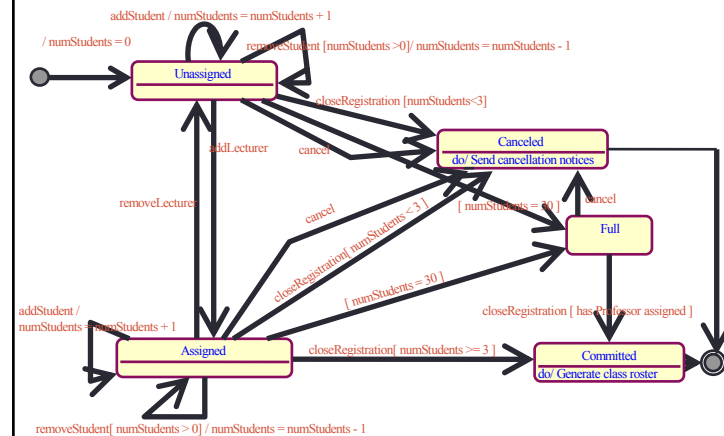- Transitions describe what happens in response to the receipt of an event

CourseInfo
+ addLecturer() + removeLecturer()

0..*   0..1   Lecturer

Unassigned

removeLecturer   addLecturer

Assigned

Add Activities

- Entry
  - Executed when the state is entered

- Do
  - Ongoing execution

- Exit
  - Executed when the state is exited

StateA
Entry/anAction

StateB
Do/anActivity

→

StateC
Exit/someAction

Example: State Machines

addStudent / numStudents = numStudents + 1

/ numStudents = 0

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

Unassigned

closeRegistration [numStudents<3]

Canceled
do/ Send cancellation notices

addLecturer   cancel

removeLecturer

cancel   [numStudents = 0]   cancel

Full

closeRegistration[numStudents < 3]

[numStudents = 30]

addStudent /
numStudents = numStudents + 1

closeRegistration [has Professor assigned]

Assigned   closeRegistration[numStudents >= 3]

Committed
do/ Generate class roster

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

17

Which Objects Have Significant State?

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

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

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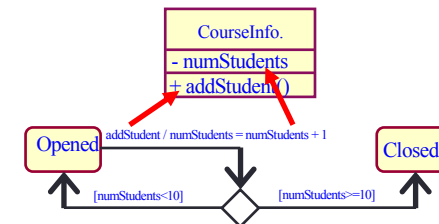
Content

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?

18
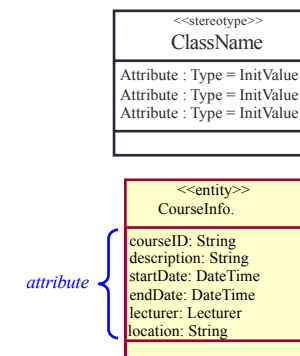
5.1. Finding Attributes

- 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

5.1. Finding Attributes (2)

- Examine method descriptions
- Examine states
- Examine any information the class itself needs to maintain
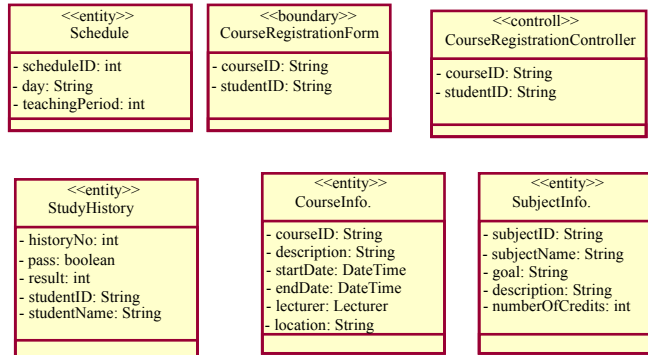
5.2. Attribute Representations

- 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: #

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>> CourseInfo.
- 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
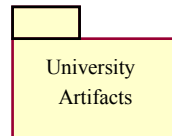4. Define States
5. Define Attributes
6. Class Diagram

---

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
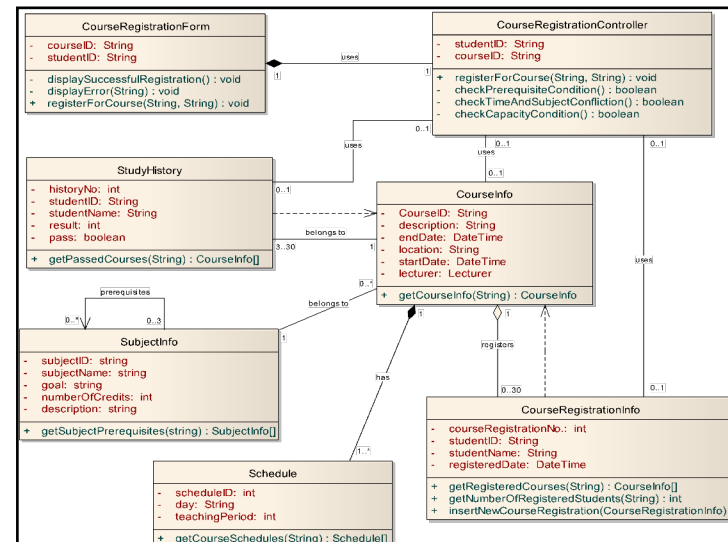
Review: What Is a Package?

- 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

University
Artifacts

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

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

Review points: Attributes

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

Review points: Relationships

- Descriptive role names
- Correct multiplicities

Question?