ITSS SOFTWARE DEVELOPMENT/SOFTWARE DESIGN AND CONSTRUCTION

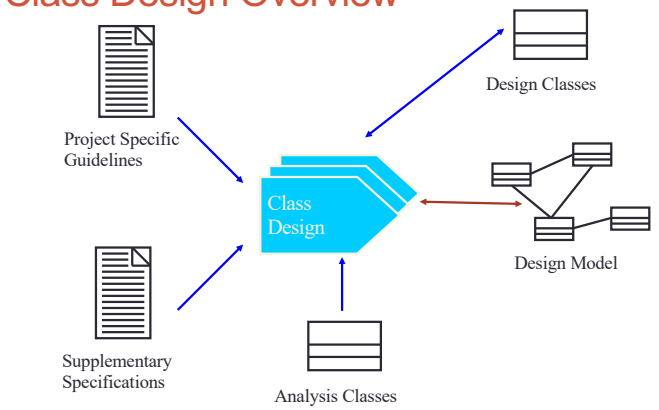
6. CLASS DESIGN

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

Some slides extracted from IBM coursewares

1

Class Design Overview

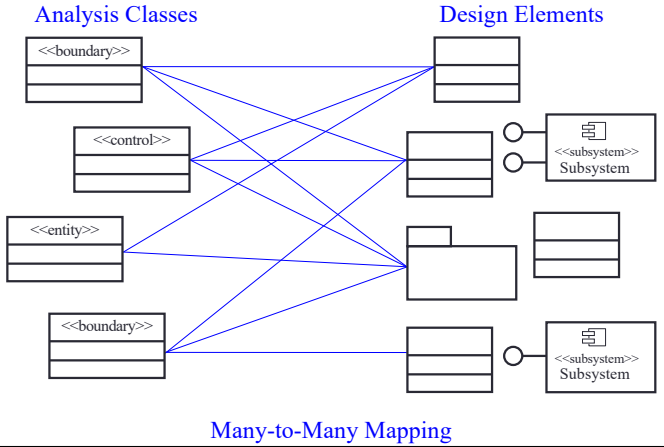


2

Content

3

From Analysis Classes to Design Elements



Many-to-Many Mapping

4

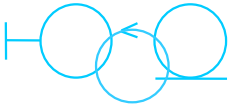
Identifying Design Classes

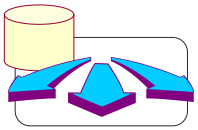
- An analysis class maps directly to a design class if:
  - It is a simple class
  - It represents a single logical abstraction
- More complex analysis classes may
  - Split into multiple classes
  - Become a package
  - Become a subsystem (discussed later)
  - Any combination …

5

Class Design Considerations

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





6

How Many Classes Are Needed?

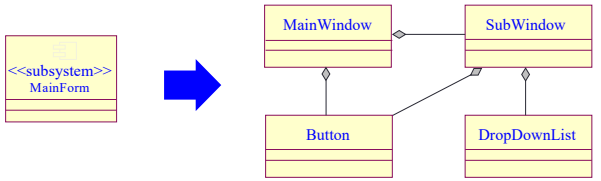
- Many, simple classes means that each class
  - Encapsulates less of the overall system intelligence
  - Is more reusable
  - Is easier to implement
- A few, complex classes means that each class
  - Encapsulates a large portion of the overall system intelligence
  - Is less likely to be reusable
 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

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

7

Strategies for Designing Boundary Classes

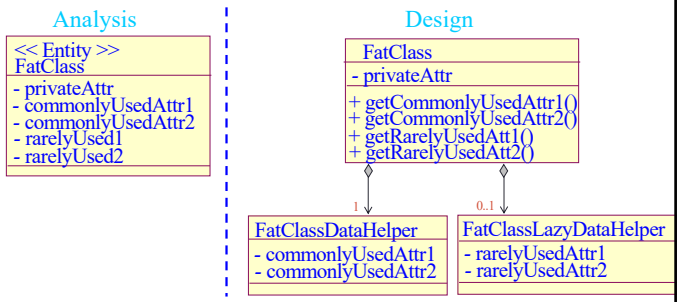
- User interface (UI) boundary classes
  - What user interface development tools will be used?
  - How much of the interface can be created by the development tool?
- External system interface boundary classes
  - Usually model as subsystem



8

Strategies for Designing Entity Classes

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



9

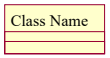
Strategies for Designing Control Classes

- 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

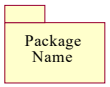
10

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



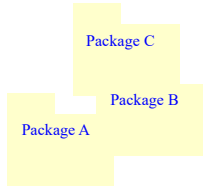
- What is a package?
  - A general purpose mechanism for organizing elements into groups
  - A model element which can contain other model elements



11

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



12

Packaging Tips: Boundary Classes

If it is **likely** the system interface will undergo considerable changes

Boundary classes placed in separate packages

If it is **unlikely** the system interface will undergo considerable changes

Boundary classes packaged with functionally related classes

13

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

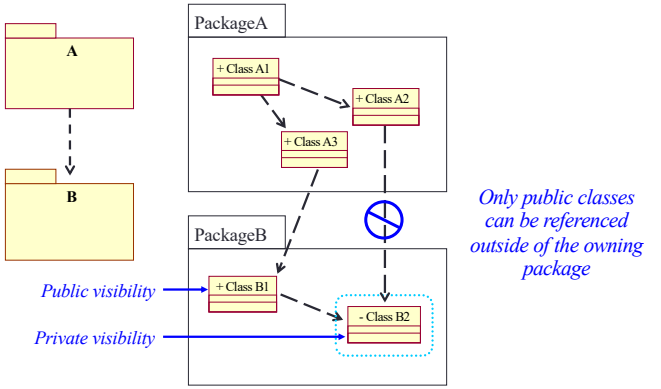
14

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

15

Package Dependencies: Package Element Visibility

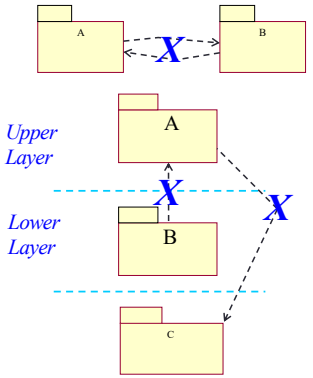


OO Principle: Encapsulation

16

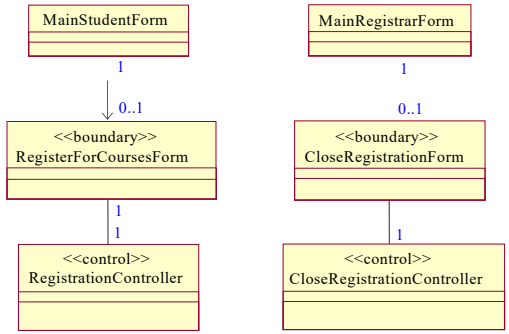
Package Coupling: Tips

- 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



17

Example: Registration Package

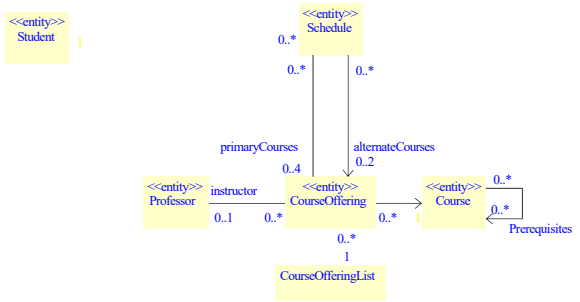


18

Example: University Artifacts Package: Generalization

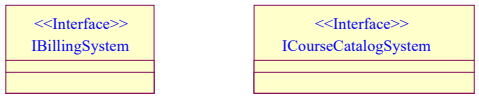
19

Example: University Artifacts Package: Associations



20

Example: External System Interfaces Package



21

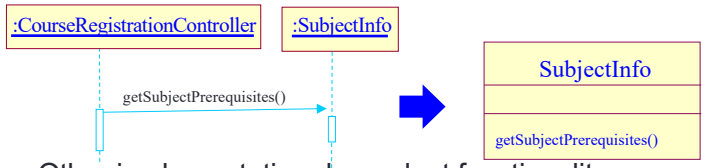
Content

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

22

2.1. Define Operations

- Messages displayed in interaction diagrams



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

23

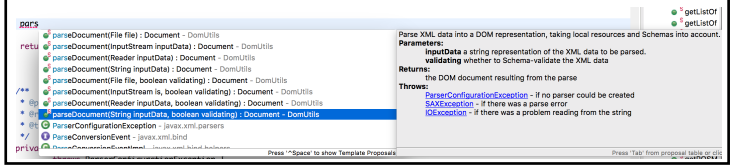
Name and Describe the Operations

- 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

24

Program documentation

```
/**
  * 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
        ...
 }
```

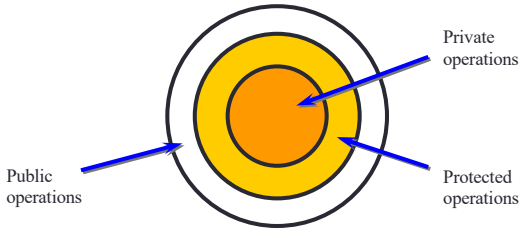


Guidelines: Designing Operation Signatures

- 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”

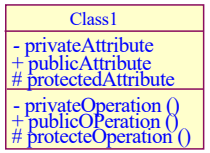
Operation Visibility

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



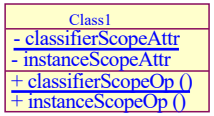
How Is Visibility Noted?

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



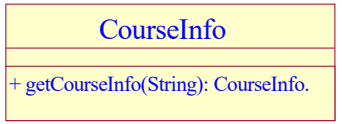
Scope

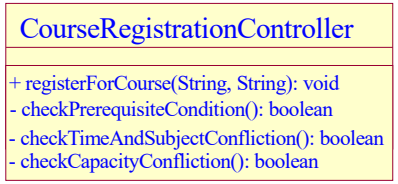
- Determines number of instances of the attribute/operation
  - Instance: one instance for each class instance
  - Classifier: one instance for all class instances
- Classifier scope is denoted by underlining the attribute/operation name



29

Course Registration CS: Operations for CourseInfo. and CourseRegistrationController

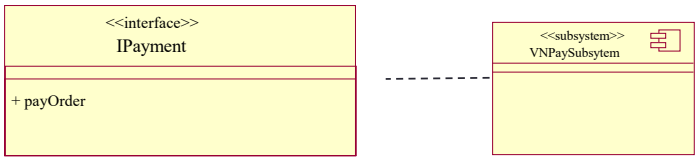




30

Quiz: Design operation payOrder()

- Please design operation payOrder() of the interface IPayment



31

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

32

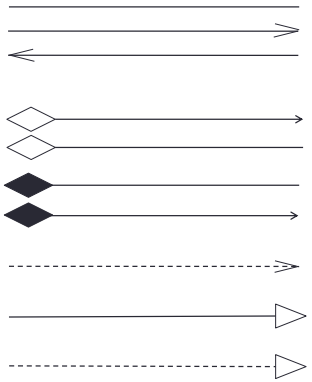
Content

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

33

Class Relationships

- Association
  - Aggregation
    - Composition
- Dependency
- Generalization/Inheritance
- Realization



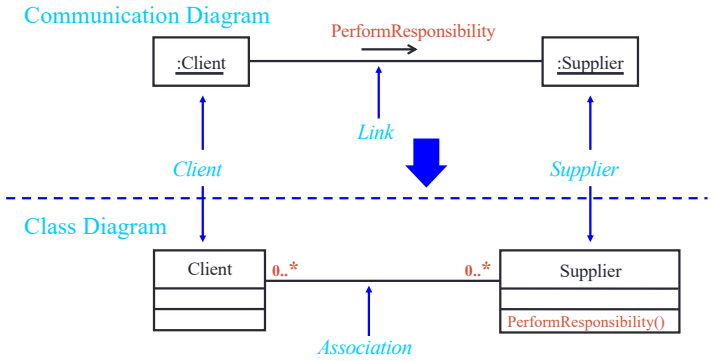
34

Association / Inheritance / Realization

- Association
  - "use"
  - Remove duplication → Reuse code through object
- Inheritance
  - "is a kind of" "is a"
  - Sub-class inherits from super-class
  - Remove duplication → Reuse code through class
- Realization
  - Class/sub-interface realizes interface
  - Communication

35

Finding Association

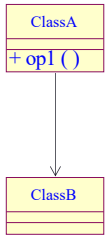


Relationship for every link!

36

Field Reference - Association

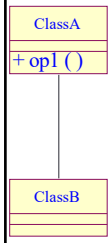
```
class ClassA {
      ClassB b;
      op1(){
            ...
            b.m1();
            ...
      }
      op2(){
            ...
            b.m2();
            ...
      }
}
```



37

Field Reference - Association

```
class ClassA {
      ClassB b;
      op1(){
            ...
            b.m1();
            ...
      }
      op2(){
            ...
            b.m2();
            ...
      }
}
```

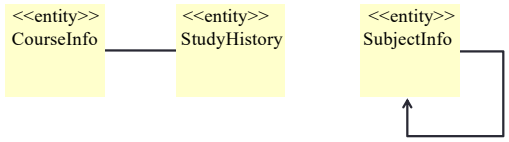



```
class ClassB {
      ClassA a;
      m1(){
            ...
            a.op1();
            ...
      }
      m2(){
            ...
            a.op2();
            ...
      }
}
```

38

3.1. What is an Association?

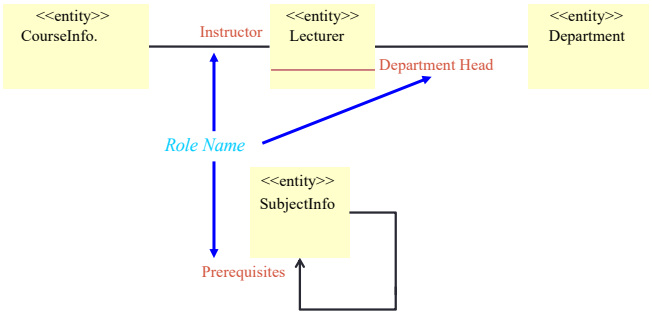
- The semantic relationship between two or more classifiers that specifies connections among their instances
- A structural relationship, specifying that objects of one thing are connected to objects of another



39

3.1.1. What Are Roles?

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



40

3.1.2. What Is Multiplicity?

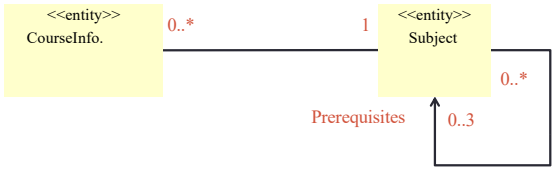
- Multiplicity is the number of instances one class relates to ONE instance of another class.
- For each association, there are two multiplicity decisions to make, one for each end of the association.
  - For each instance of Professor, many Course Offerings may be taught.
  - For each instance of Course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

Multiplicity Indicators

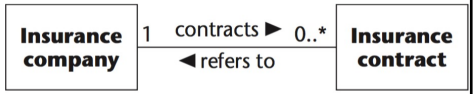
Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional value)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6

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?



Java implementation

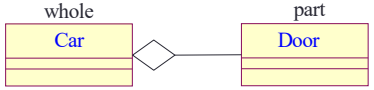



```
//InsuranceCompany.java file
public class InsuranceCompany
{
   // Many multiplicity can be implemented using Collection
   private List<InsuranceContract> contracts;


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

   /* Methods */
}
```

3.1.3. Association Types

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

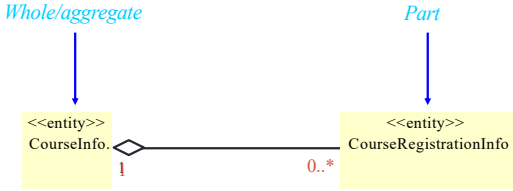
- Aggregation
  - is-a-part
  - Strong association, an instance of one class is made up of instances of another class



- 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.



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.

"Register for course" Use case

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.

Aggregation – Java implementation

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

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

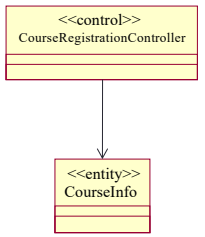

Composition – Java implementation

```
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 ");
        }
    }
}
```

```
class Engine {
    // starting an engine
    public void work() {
        System.out.println("Engine of car has been started ");
    }
}
```

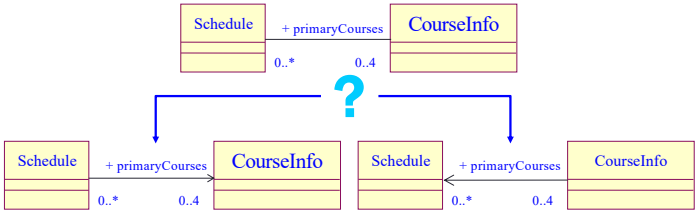

3.1.4. Navigability

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



Navigability: Which Directions Are Really Needed?

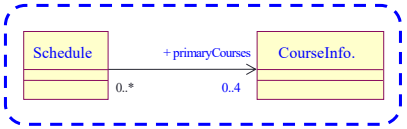
- Explore interaction diagrams
- Even when both directions seem required, one may work
  - Navigability in one direction is infrequent
  - Number of instances of one class is small



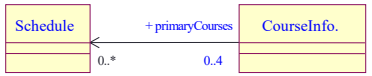
53

Example: Navigability Refinement

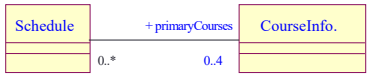
- Total number of Schedules is small, or
- Never need a list of the Schedules on which the CourseInfo appears



- 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



54

3.2. Dependency

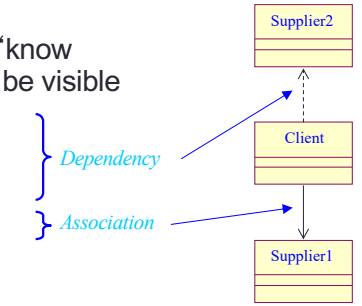
- What Is a Dependency?
  - A relationship between two objects

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

55

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



56

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

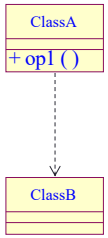
A dependency is a secondary type of relationship in that it doesn't tell you much about the relationship. For details you need to consult the collaborations.

57

3.2.1. Local Variable Visibility

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

```
class ClassA {
      op1(){
            ClassB b = new ClassB();
            ...
            b.m();
            ...
      }
}
```

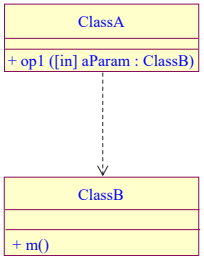


58

3.2.2. Parameter Visibility

- The ClassB instance is passed to the ClassA instance

```
class ClassA {
      op1(ClassB b){
            ...
            b.m();
            ...
      }
}
```

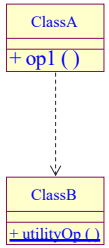


59

3.2.3. Global Visibility

- The ClassUtility instance is visible because it is global

```
class ClassA {
      op1(){
            ...
            ClassB.utilityOp();
            ...
      }
}
```



60

Identifying Dependencies: Considerations

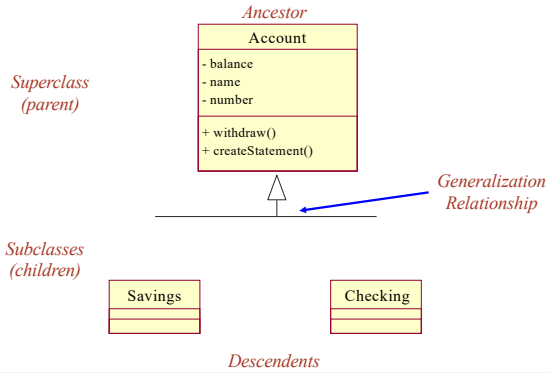
- Permanent relationships — Association (field visibility)
- Transient relationships — Dependency
  - Multiple objects share the same instance
    - Pass instance as a parameter (parameter visibility)
    - Make instance a managed global (global visibility)
  - Multiple objects don't share the same instance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



Content

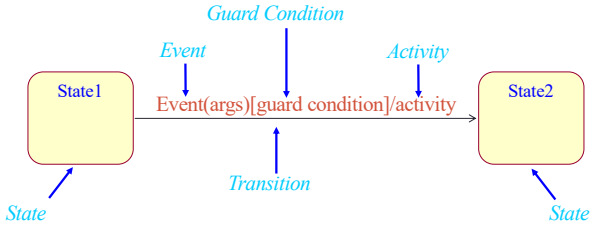
1. Create Initial Design Classes
2. Define Operations/Methods
3. Define Relationships Between Classes
4. Define States
5. Define Attributes
6. Class Diagram

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?

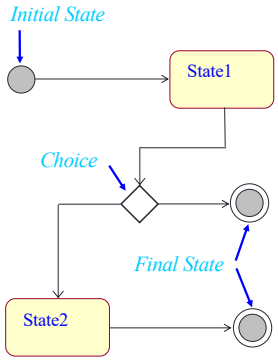
What is a State Machine?

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



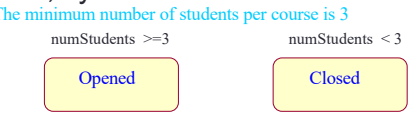
Pseudo States

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

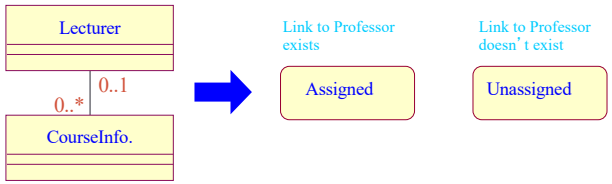


Identify and Define the States

- Significant, dynamic attributes



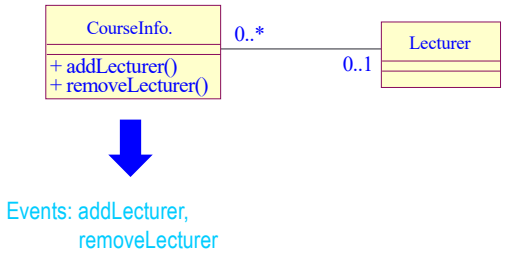
- Existence and non-existence of certain links



69

Identify the Events

- Look at the class interface operations

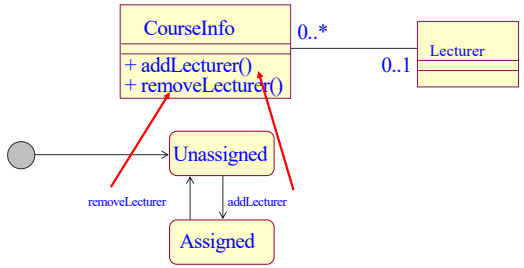


69

70

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

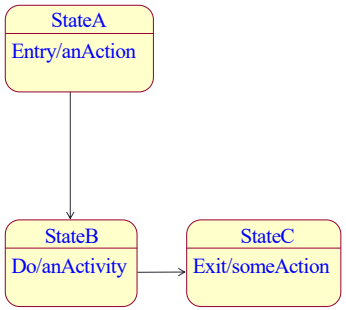


70

71

Add Activities

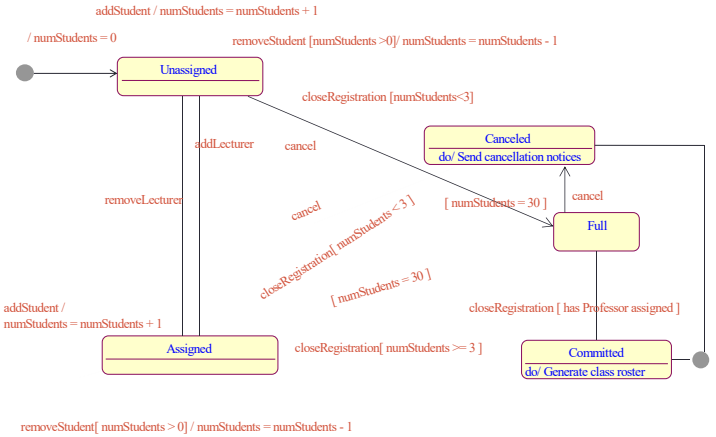
- Entry
  - Executed when the state is entered
- Do
  - Ongoing execution
- Exit
  - Executed when the state is exited



71

72

Example: State Machines



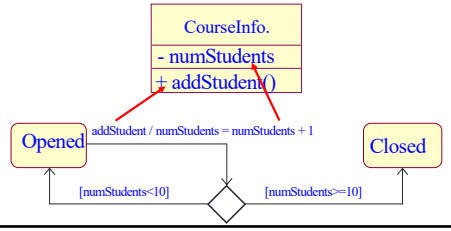
72

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?

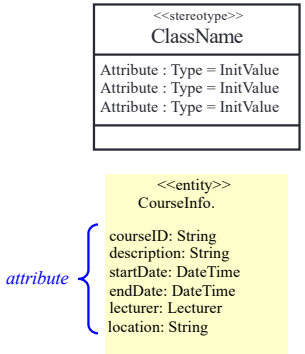
- Events may map to operations
- Methods should be updated with state-specific information
- States are often represented using attributes
  - This serves as input into the "*Define Attributes*" step



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?



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

81

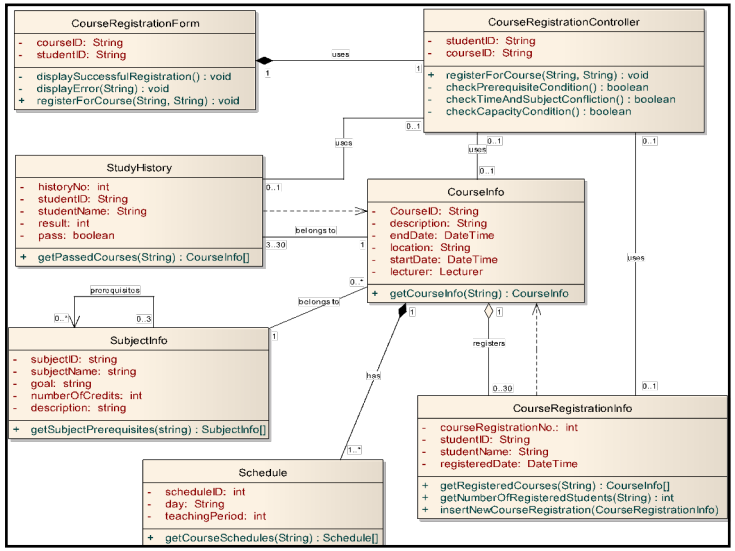
Content

82

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

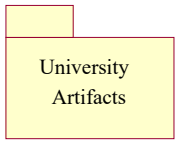
83



84

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



85

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

86

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

87

Review points: Attributes

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

88

Review points: Relationships

- Descriptive role names
- Correct multiplicities

89

Question?

90

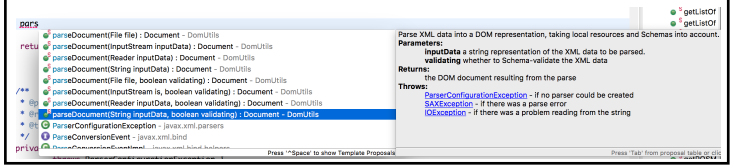
Class design

- Attribute design
  - Type, description
- Operation design
  - Operation Signature
  - Purpose/description of operation
  - Purpose /description of each parameter
  - Description of return value
  - Error/Exception (when)
- Method design
  - Special algorithm
  - How to use parameters

91

Program documentation

```
/**
  * 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
        ...
 }
```



92