HY351 - Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
CS351 - Information Systems Analysis and Design

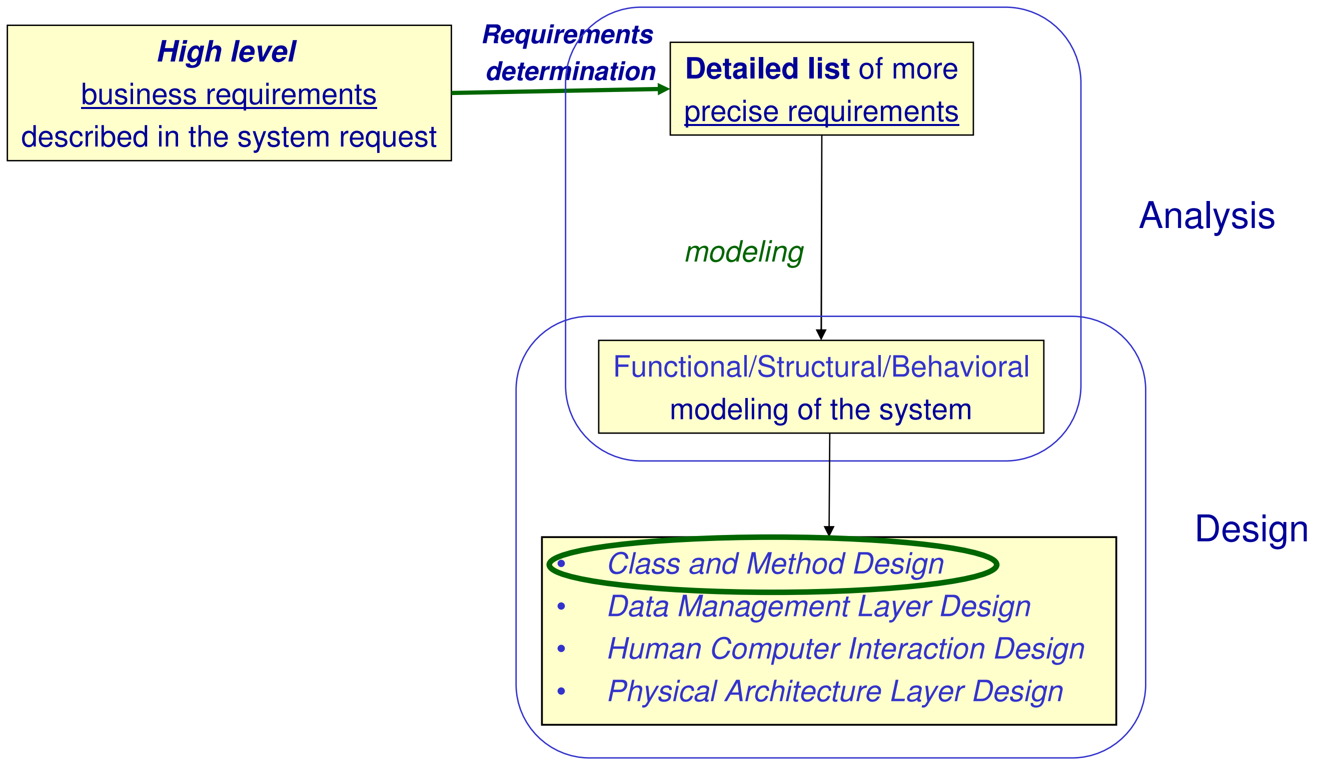
Class and Method Design

.

Lecture :
Date : 24-11-2005

Yannis Tzitzikas
University of Crete, Fall 2005

Design> Class and Method Design


High level
business requirements
described in the system request
Requirements determination
Detailed list of more
precise requirements
modeling
Functional/Structural/Behavioral
modeling of the system
Analysis
Design
• *Class and Method Design*
• *Data Management Layer Design*
• *Human Computer Interaction Design*
• *Physical Architecture Layer Design*

Outline

- Why to do detailed class and method design?
- Design criteria
  - coupling, cohesion
- Restructuring the design (Factoring and Optimizing)
- Mapping problem domain classes to implementation classes
- Method specification
- [Constraints and Contracts]
- Opportunities for Reuse
  - Design Patterns

Class and Method Design: Motivation

- One of the most important steps in the design phase is the design of classes and methods
- Analysts should create instructions and guidelines for programmers that clearly describe what the system must do

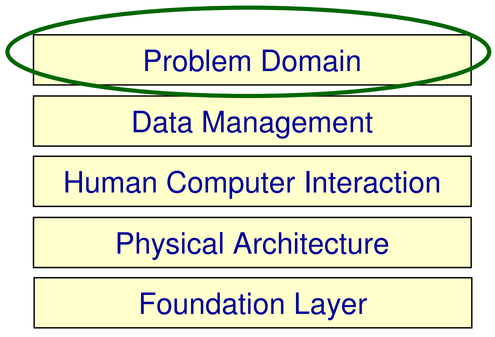
Why to design (in more detail) classes and methods?

- Some persons say that with reusable class libraries and off-the-shelf components, low level detailed design is a waste of time, so we should directly start coding.
- However, experience shows that detailed design is useful despite the reusable class libraries
  - even pre-existing classes and components need to be understood, organized and pieced together properly
  - the team will probably have to create its own classes for the application logic of the system

What could go wrong without careful design?

- Objects will not be able to communicate <u>correctly</u>, so the functioning of the system will not be correct
- A not careful use of layers may introduce communication <u>overhead</u> that make the system very inefficient.
- A change at a part of the system may requires <u>changing</u> too many other things

The importance of problem domain classes



- We have already designed the structure and the interaction of the classes of the domain model
- The classes of the other layers (system architecture, HCI, data management) will be dependent on the problem domain layer.
- So, it is important to design correctly the problem domain classes.

What is detailed class and method design?

- Structural modeling and behavioural modeling (that we discussed in the previous lectures) is indeed class and method design
- ***So what we should do more ?***
- We should ask ourselves questions of the form:
  - ***Are all of the classes necessary?***
  - ***Are there any missing classes?***
  - ***Are the classes fully defined?***
  - ***Are there any missing attributes or methods?***
  - ***Do the classes have any unnecessary attributes and methods?***
  - ***Is there any inheritance conflict?***
  - ***Is there any inefficiency in the design, and how we could fix it?***
  - ***Can we map the class diagram to the programming language that will be used in the project?***
  - ***How we can reuse code?***

Detailed Design Activities

Apart from the above we should:

- ***Check that nothing is missing from the domain model***
- ***Finalize the visibility of the attributes and methods in each class***
- ***Decide on the signature of every method in every class***
- ***Define constraints that must be preserved by the objects***

Refresher: Object Orientation, Class Diagrams

Encapsulation
- Hiding the content of the object from outside view
- Communication only through object's methods
- Key to reusability

Polymorphism
- Same message triggers different methods in different objects
- Dynamic binding means specific method is selected at run time
- Implementation of dynamic binding is language specific
- Need to be very careful about run time errors
- Need to ensure semantic consistency

Inheritance
- Single inheritance -- one parent class
- Multiple inheritance -- multiple parent classes
- Inheritance conflict

There are 3 perspectives for the design of a class diagram (of a conceptual model in general)
- ***Conceptual***
  - Independent of implementation. This is often called **domain model.**
- ***Specification***
  - Based on **interfaces of the software**, <u>not the implementation</u>
- ***Implementation***
  - Here we model the **implementation classes**.

Design Criteria

Design Criteria

A good design is one that balances trade-offs to minimize the total cost of the system over its lifetime [Yourdon'91]

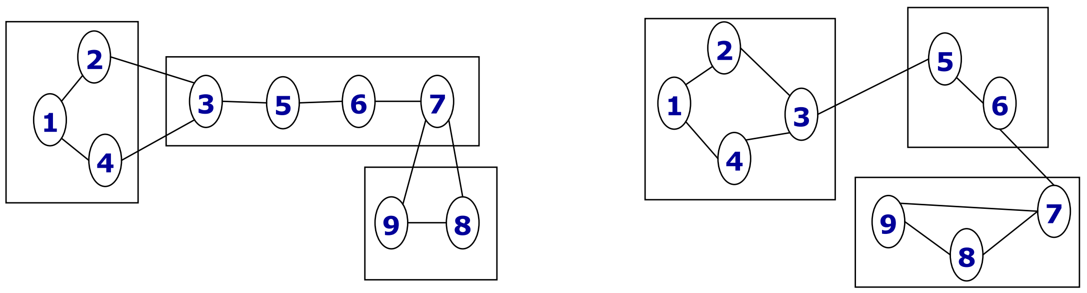
General Design Criteria

[A] Coupling (σύζευξη)

[B] Cohesion (συνεκτικότητα / συνοχή)

…. [C] Connascence

We have already discussed coupling and cohesion in the context of layering/packaging (Lecture 12)



Coupling and Cohesion

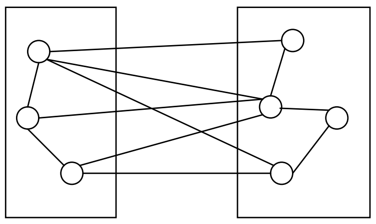
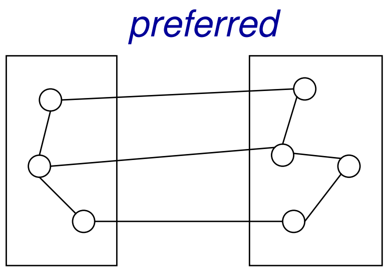
- **Coupling**: measures how *interdependent* are the modules (classes, objects, methods) of the system
- **Cohesion**: measures how single-minded is a module (class, object, method) within a system
  - (single-minded ~ προσηλωμένος σε στόχο)

Aspects of coupling and cohesion:

Coupling	Cohesion
Interaction	method
Inheritance	class
	generalization

[A] Coupling

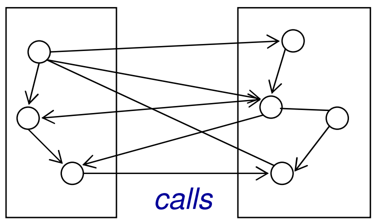
- The higher the interdependence the more likely changes in a part of the design will require changing other parts of the design
- So we would like to minimize it

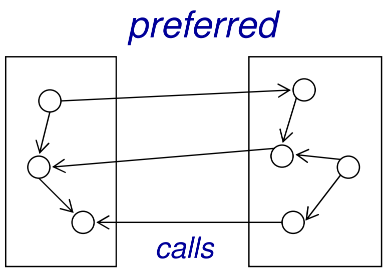



- Types of coupling
  - Interaction coupling
  - Inheritance coupling

Coupling> Interaction

Interaction coupling concerns message passing





Law of Demeter [Lieberherr & Holland 89]

- **minimize the number of objects that can receive messages from a given object**
- an object should send a message to:
  - itself
  - an object that is contained in an attribute of the object (or one of its superclasses)
  - an object that is passed as a parameter to a method
  - an object that is created by the method
  - an object that is stored in a global variable

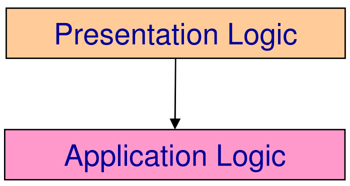
6 types of Interaction Coupling

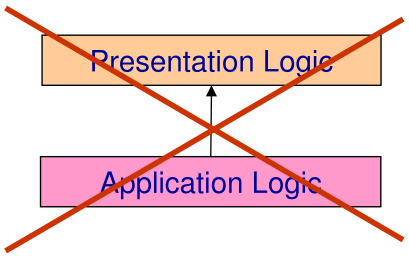
• **No direct coupling**	• The methods do not call one another
• **Data**	• The calling method passes a variable to the called method. If the variable is an object, the entire object is used by the called method to perform its function
• **Stamp**	• The calling method passes a composite variable to the called method, but the called method only uses a <u>portion of the object</u> to perform its function
• **Control**	• The calling method passes a control variable whose value will control the execution of the called method
• **Common or Global**	• The methods refer to a "<u>global data area</u>" that is outside the individual objects
• **Content or Pathological**	• A method of one object <u>refers to the inside</u> (hidden parts) of another object. This violates the principles of encapsulation (C++ allows this with "friends")

Adapted from Dennis et al. 2005

Coupling> <u>Interaction</u>: *Guidelines*

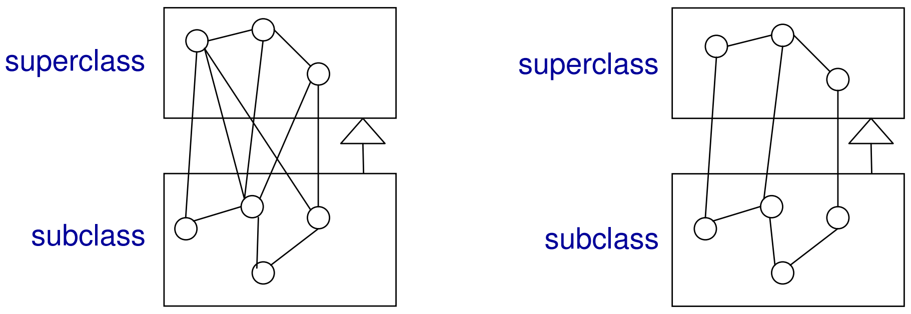
- We should try to minimize it
- Exception:
  - sometimes non-problem domain classes must be coupled with domain classes
    - e.g. UIPerson can be coupled to Person
      - for optimization the UIPerson could be pathologically coupled to Person class
- However, problem domain classes must never be coupled to non-problem domain classes





Coupling>Inheritance

Concerns the classes within one generalization/ specialization hierarchy

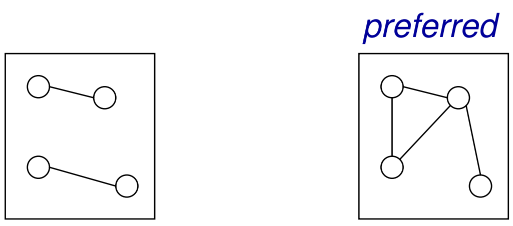


- Some developers believe that high coupling is not a bad thing. Others believe that high coupling is not good (due to the various inheritance conflicts)
- Related questions
  - *should a method defined in a subclass be able to call a method of the superclass?*
  - *should a method defined in a subclass refer to an attribute of the superclass?*
- This of course depends on the PL
- *Guideline*:
  - ensure inheritance is used only to support generalization/specialization semantics (i.e. subset) and the principle of substitutability

[B] Cohesion

- **Cohesion**: measures how single-minded a module (class, object, method) is within a system

We should try to maximize it



Types of Cohesion
- Method cohesion
- Class cohesion
- Generalization/specialization cohesion

Cohesion>Method

A method should solve a **single task**

– A method performing multiple functions is more difficult to understand and reuse

7 types of method cohesion:

Functional	A method performs one single task
Sequential	The method combines two functions: the output from the first is used as input to the second
Communicational	The method combines two functions that use the same attributes to execute
Procedural	The method supports multiple weakly related functions
Temporal or Classical	The method supports multiple related functions in time (e.g. initialize all attributes)
Logical	The method supports multiple related functions but the choice of the specific function is chosen based on a control variable that is passed as parameter
Coincidental	The method supports multiple unrelated functions

Adapted from Dennis et al. 2005

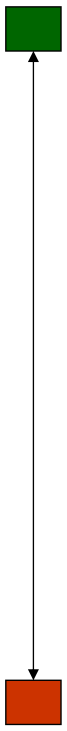
Cohesion>Class

- A class should represent one thing (e.g. person, car, department)
- All attributes and methods of a class should be required for the class to represent one thing
- No redundant attributes should exist
- The Cohesion of a class is the degree of cohesion between the attributes and the methods of a class

Guidelines [G. Meyers 78]

- a class should contain multiple methods that are visible outside of the class and that each visible method only performs a single function (I.e. functional cohesion)
- a class should have methods that only refer to attributes or other methods defined with the class or its superclasses
- a class should not have any control-flow couplings between its methods

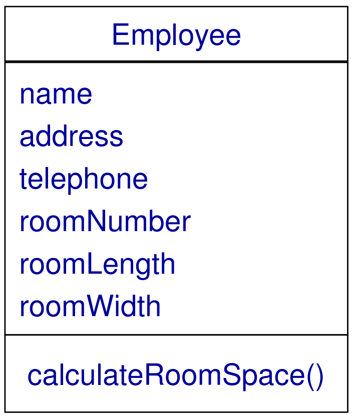
4 types of Class Cohesion



• **Ideal**	• The class has no mixed cohesions
• **Mixed-Role**	• The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g. the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class
• **Mixed-Domain**	• The class has one or more attributes that relate objects of the class to other objects on a different layer. So these attributes have nothing to do with the underlying semantics of the thing that the class represents.
• **Mixed-Instance**	• The class represents different types of objects meaning that different instances only use a portion of the full definition of the class. The class should be decomposed into separate classes.

Adapted from Dennis et al. 2005

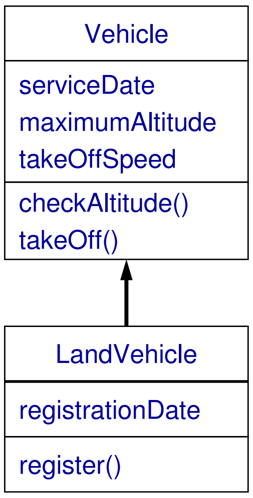
Example: Method vs Class Cohesion



High method cohesion but
low class cohesion

Cohesion>Generalization/Specialization

- *How are the classes in the inheritance hierarchy related?*
- *Are they related through a generalization/specialization semantics?*
- *Or they are related for simple reuse purposes?*
- *To what degree a subclass actually needs the features it inherits?*

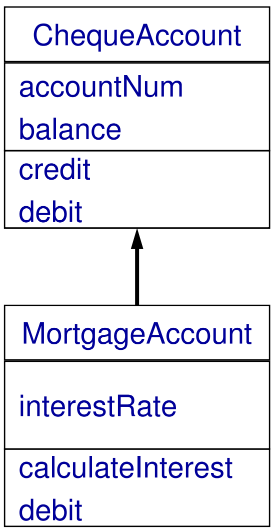


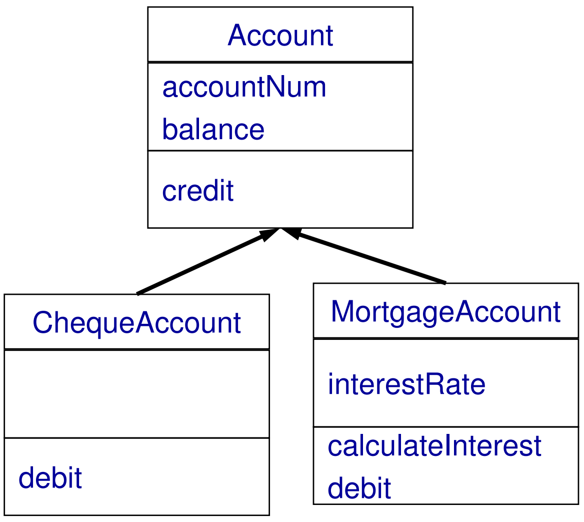
Low Inheritance Cohesion

Example: Restructuring for satisfying LSP

Liskov Substitution Principle (LSP)

– In a class hierarchy it should be possible to treat a specialized object as if it were a base object





Example (1/3)

```
class Rectangle {
public:
   void setWidth(double w) {itsWidth=w;}
   void setHeight(double h){itsHeight=h;}
   double getArea() {return itsHeight * itsWidth;}
private:
  double itsWidth;
  double itsHeight;
};
```

```
class Square: public Rectangle {
…
};
```

```
void Square::setWidth(double w)
{
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
};
void Square::setHeight(double w)
{
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
};
```

Example (2/3)

```
void f(Rectangle* r)
{
    r->setWidth(32);
}
```

Violation of LSP:
Function f will not function correctly if r is a square

We could fix this problem by allowing polymorphism

```
class Rectangle {
public:
   virtual void setWidth(double w) {itsWidth=w;}
   virtual void setHeight(double h){itsHeight=h;}
   double getArea() {return itsHeight * itsWidth;}
private:
  double itsWidth;
  double itsHeight;
};
```

Example (3/3)

```
void g(Rectangle* r)
{
   r->setWidth(5);
   r->setHeight(4);
  assert(r->getArea()==20);
}
```

It will function correctly if r is a rectangle.
It will not function correctly if r is square

So g() is fragile and it violates LSP

The class Rectangle actually violates an “invariant” of the class Rectangle, specifically the width-height independence.

This could be expressed using OCL (will be described in the next lecture)
It will expressed as a post condition of setWidth:
i.e. the height is the old value of height

Restructuring the Design

Factoring
Optimizing
Translate to Implementation Language

Factoring

Factoring is the process of separating out aspects of a method or class into a new method or class to simplify the overall design.

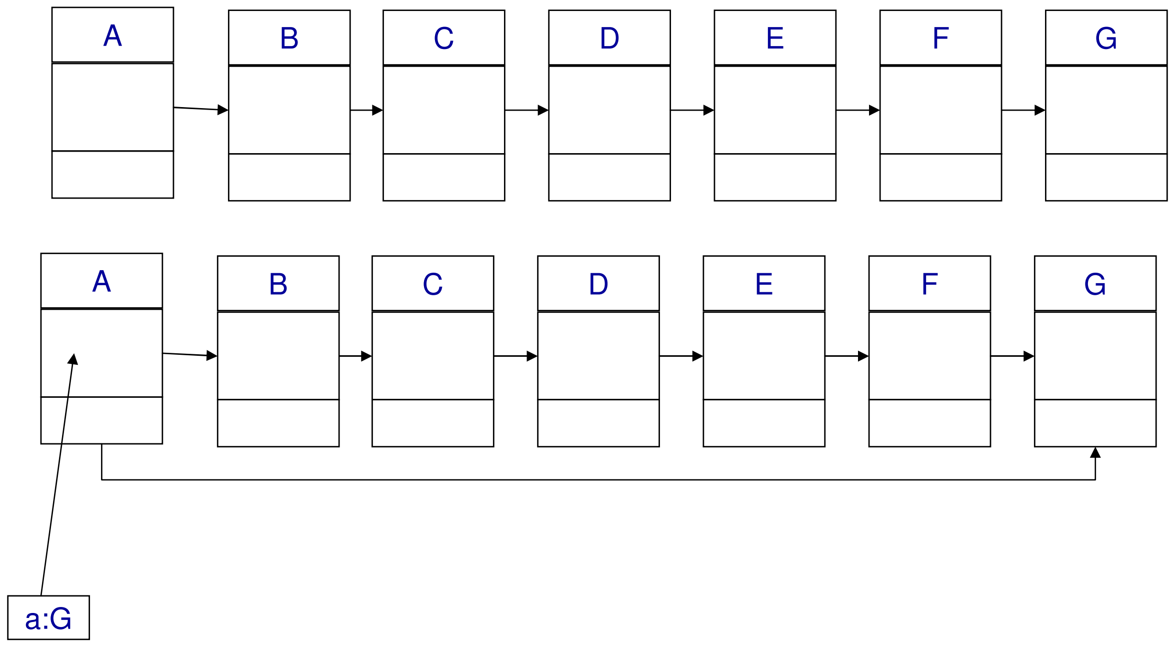
For example we may realize that some classes of the design share a similar definition. In this case we can factor out the similarities and define a new class. The new class is then related with the existing classes through generalization, aggregation or association.

Optimizing the Design

- There is a ***trade-off*** between <u>understandability</u> and <u>efficiency</u>
  - increasing the understandability of a design usually results in inefficiencies
  - focusing only on efficiency usually results in design that is difficult to understand by someone else
- Some ways to improve the efficiency of a design
  - ***Review Access Paths***
  - ***Review Attributes***
  - ***Use <u>derived attributes</u> when necessary to cache values***
  - ***Review the order of execution of the statements in frequently used methods (this is "Method Design" described later)***

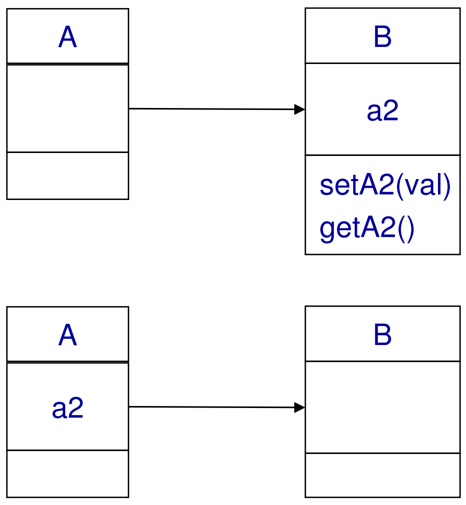
Optimizing the Design
Reviewing the Access Paths

If a message has to traverse a long path and such calls occur very frequently, it is better to create a redundant direct connection

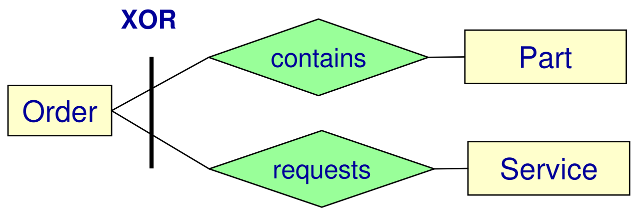


Optimizing the Design
Reviewing the Attributes

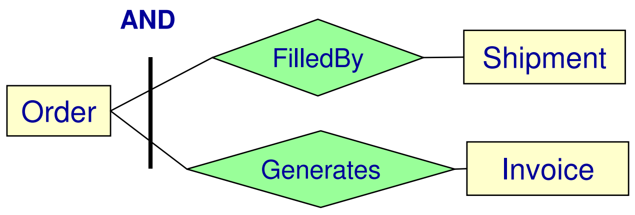
If attribute a2 of B is used only by the methods setA2 and getA2 and if only class A uses these 2 methods, then a2 should be probably placed at A.



Parenthesis: AND/XOR for Relationships (also for associations in class diagrams)



- Orders either order a part, or request a service.Not both



- For any given order, whenever there is at least one invoice there is also at least one shipment and vice versa.

Mapping Problem Domain Classes to Implementation Languages

Mapping Problem Domain Classes to Implementation Languages

- Multiple Inheritance Conflicts
- Eliminating Multiple Inheritance
  - If the implementation will be done in a PL that does not support multiple inheritance, then the multiple inheritance must be factored out from our class diagrams
- Eliminating Inheritance
  - If the implementation will be done in a PL that does not support inheritance, we have to map our class diagrams to constructs that can be implemented within that PL. For example, Visual Basic 6 does not support inheritance
- Handling Multiple and Dynamic Classification

Inheritance in PLs

- Different PLs address inheritance differently
  - so we should know the PL that is going to be used

Feature	C++	Eiffel	Java
multiple inheritance	yes	yes	no

- The accessibility of inherited properties also depends on the PL.

- In UML, visibility (private, public, protected) applies to methods and attributes
- Let A be a class with some private and public attributes and methods.
- Let B be a class defined as a subclass of class A.
- What B inherits?
- In Java, we can answer this question right away.
- In C++ we should see how B has been declared as subclass of A
  - C++ allows visibility at the class level.

Inheritance and Visibility

In C++, class B may have been defined as:

class B: public A

class B: protected A

class B: private A

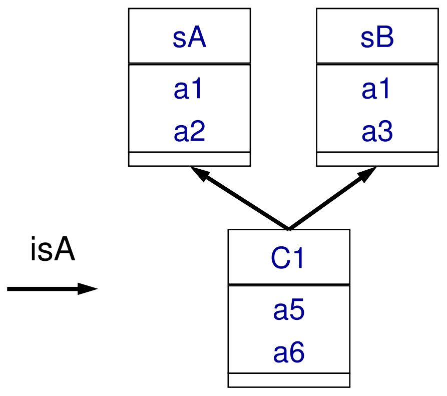
- Accessibility rules (C++)
  - The private properties of A are not visible to class B objects (in every case)
  - If base class A is defined as public, the visibility of inherited properties does not change in derived class B (public are still public and protected are still protected)
  - If base class A is defined as protected, the visibility of inherited public properties changes in derived class B to protected
  - If base class A is defined as private, the visibility of inherited public and protected properties changes in derived class B to private.

Inheritance Conflicts: Multiple Inheritance

[1] Two (or more) attributes (or methods) have the <u>same name and semantics</u>

[2] Two (or more) attributes (or methods) have the <u>same name but different semantics</u> (homonyms)

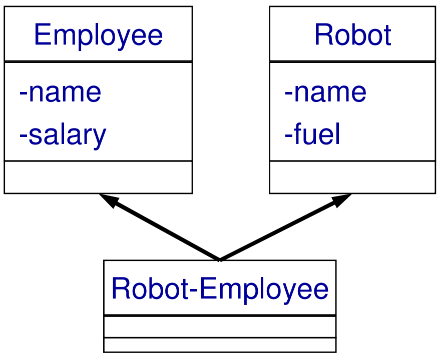
[3] Two (or more) attributes (or methods) have <u>different names but identical semantics</u> (synonyms)



Here we could be in case [1] or [2]

Inheritance Conflicts: Multiple Inheritance

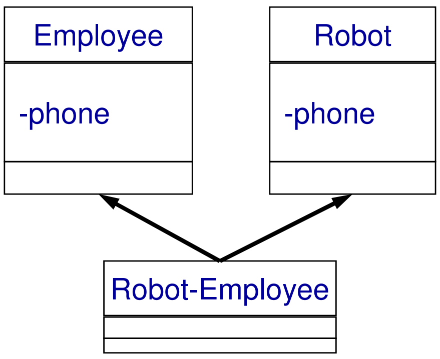
[1] Two (or more) attributes (or methods) inherited from different superclasses have the same name and semantics

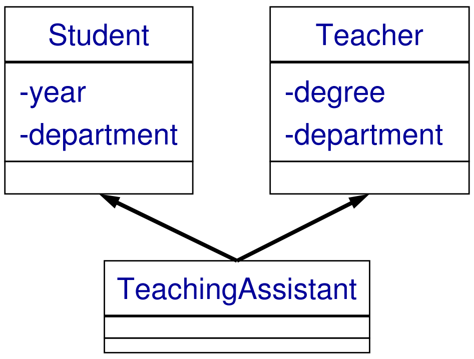


name

Inheritance Conflicts: Multiple Inheritance

[2] Two (or more) attributes (or methods) have the same name but different semantics (homonyms)



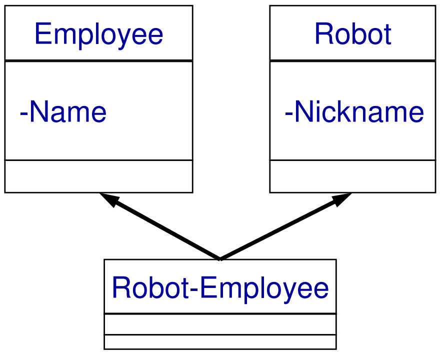


Phone ≠ Phone (of technician)

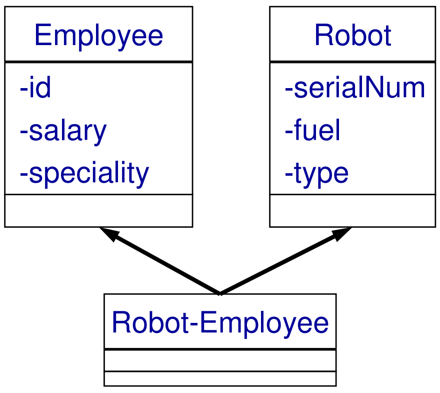
A student of CSD may be assistant to a course of the Math. Dep.

Inheritance Conflicts: Multiple Inheritance

[3] Two (or more) attributes (or methods) inherited from different superclasses have different names but they have identical semantics (synonyms)



Name <=> Nickname

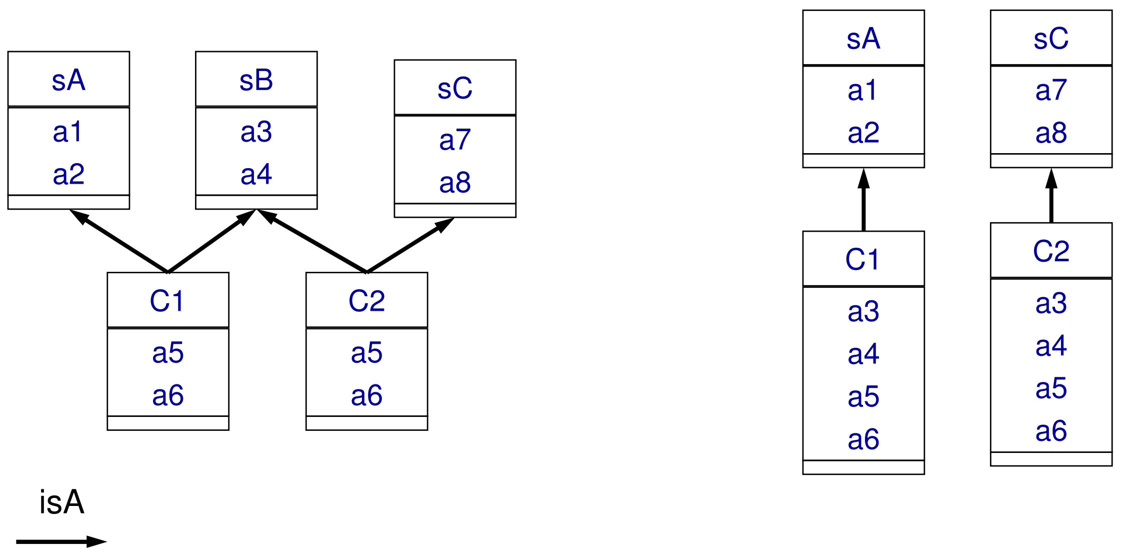


Id <=> serialNum

speciality <=> type

Eliminating Multiple Inheritance

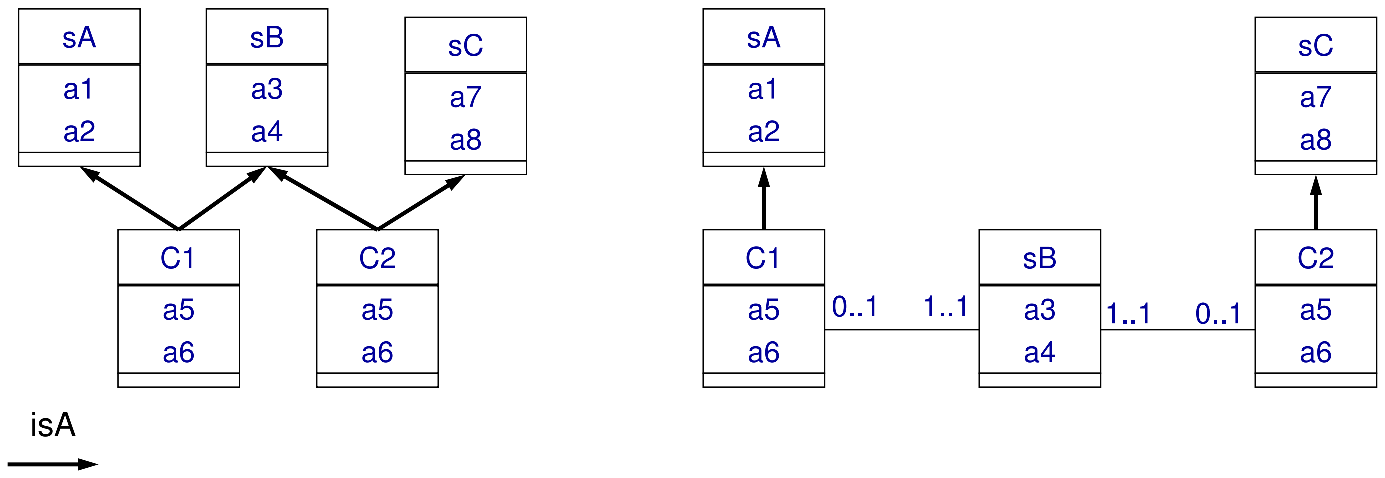
Eliminating Multiple Inheritance: Method 1: Flatten the Extra Superclasses



Flattening the inheritance by copying the attributes and methods of the additional superclass(es) to all of the subclasses and remove the additional superclasses from the design

Eliminating Multiple Inheritance: Method 2: Convert the Extra Superclasses to Associations

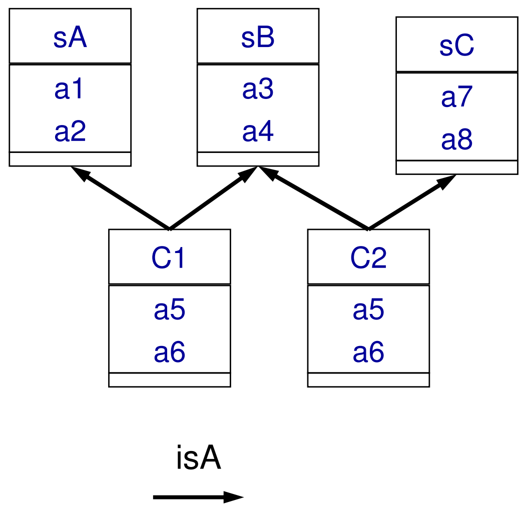
Convert the extra superclass(es) to associations with the appropriate multiplicity.

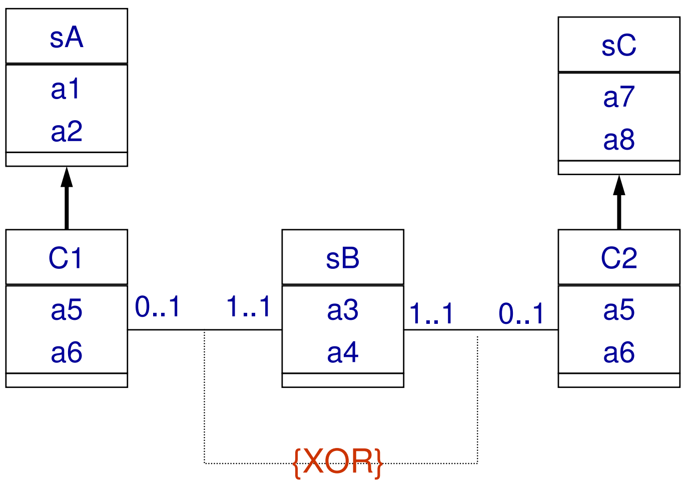


Assumption: sB can be either concrete or abstract.

Is everything ok?

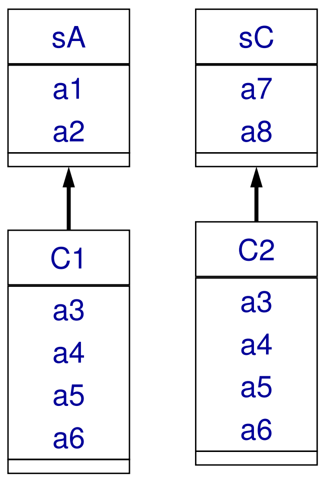
Eliminating Multiple Inheritance: Method 2: Convert the Extra Superclasses to Associations (II)

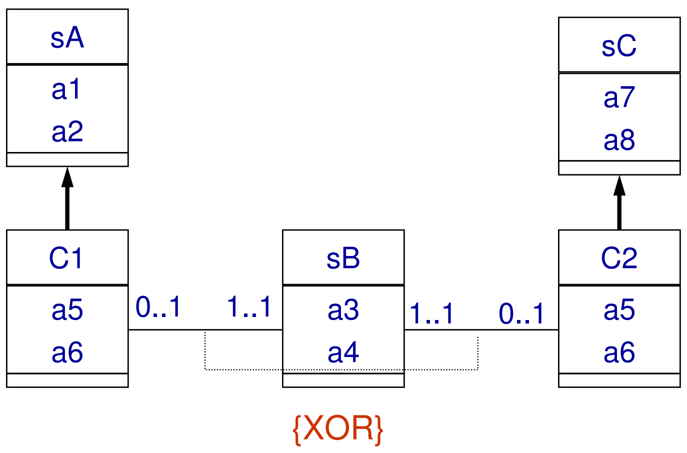




Assumption: sB can be either concrete or abstract.

Method 1 versus Method 2





Method 2

+: the domain class (sB) is preserved

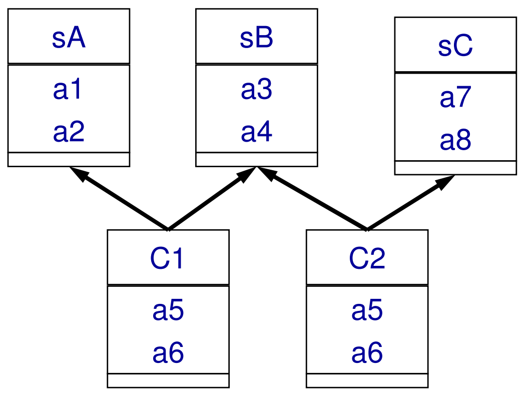
-: increases message passing and we have to keep the XOR constraint (computationally more expensive)

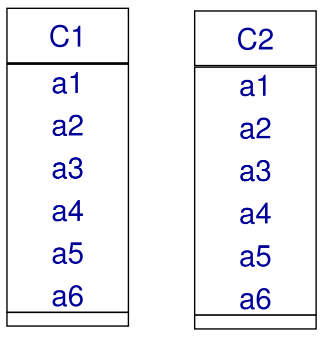
Hint: Use method 2 only if the extra superclass (sB) is concrete (not abstract). Otherwise use method 1

Eliminating Inheritance

Eliminating Inheritance
Method 1: Flattening

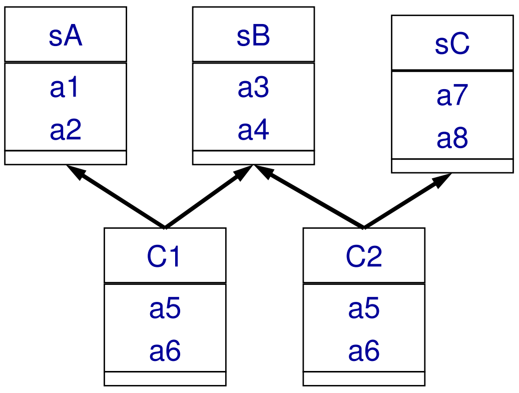
Assuming sA, sB and sC are abstract

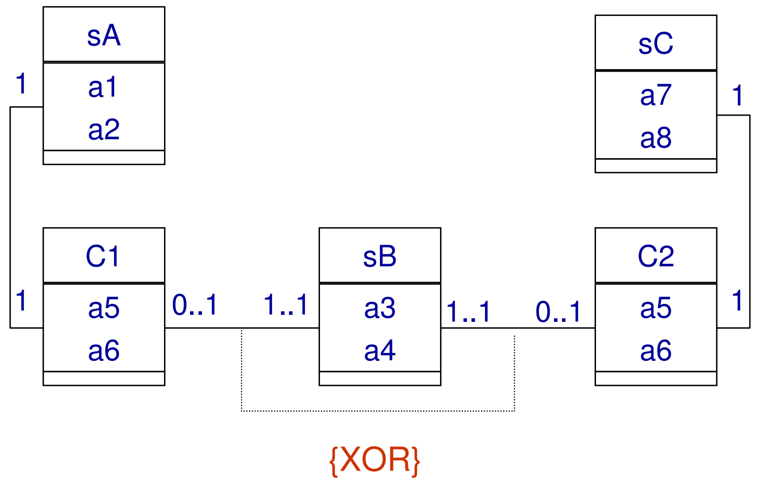




Eliminating Inheritance: Method 2: Convert all Superclasses to Associations

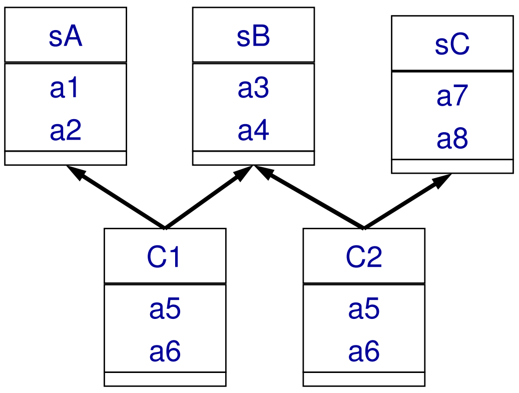
Assuming sA, sB and sC are abstract

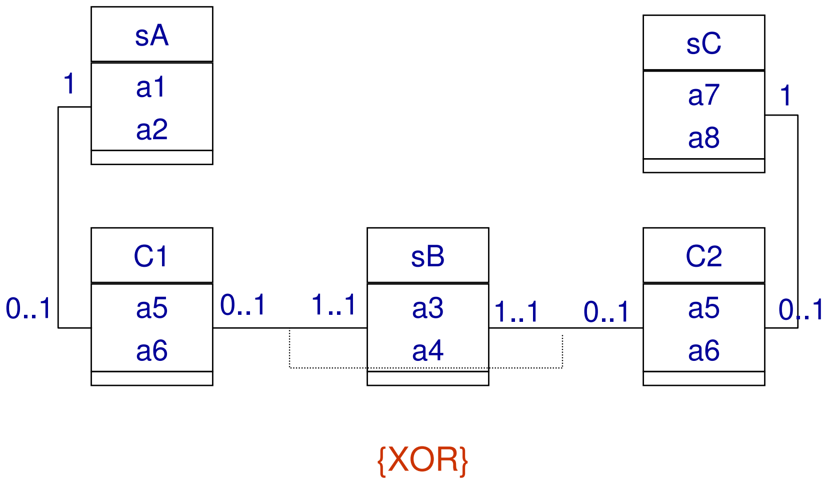




Eliminating Inheritance: Method 2: Convert all Superclasses to Associations (2)

Assuming sA, sB and sC are concrete

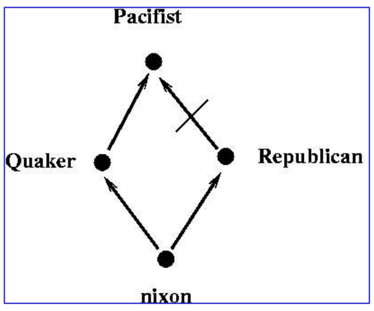




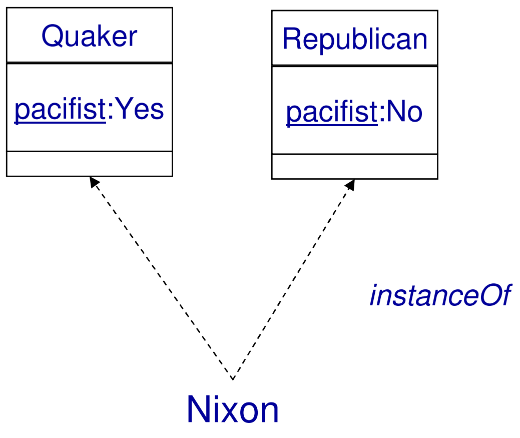
Multiple Classification

Multiple classification and conflicts

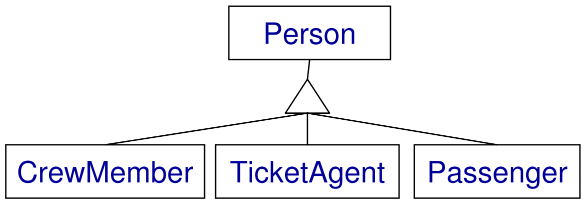
Nixon Diamond



- Nixon is a Quaker.
- Quakers are typically pacifists.
- Nixon is a Republican.
- Republicans are typically not pacifists.
- What to conclude?
  * "Nixon is a pacifist." or
  * "Nixon is not a pacifist."?



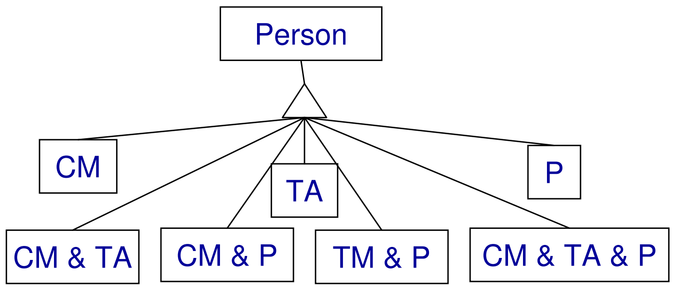
Multiple Classification



Suppose that according to the domain model:
A person part of the flight crew can also be a passenger.
Occasionally the flight crew can work as ticket agent.

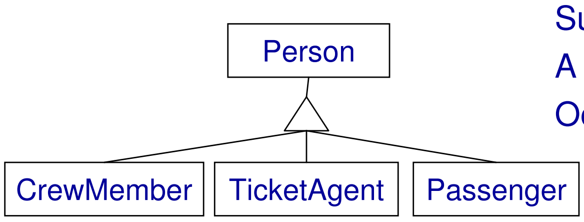
To implement this in Java we would need multiple and dynamic classification (which is not supported).

Solution 1: To overcome multiple classification, create “join” classes



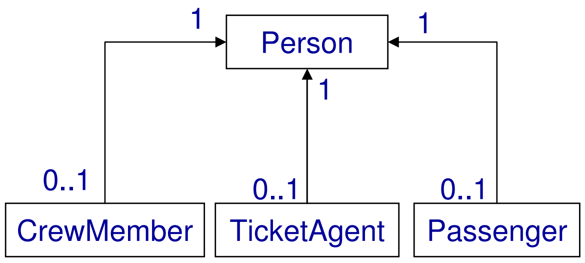
We need to define an exponential number of classes.
And still we will not be able to support *dynamic* classification

Multiple Classification (II)



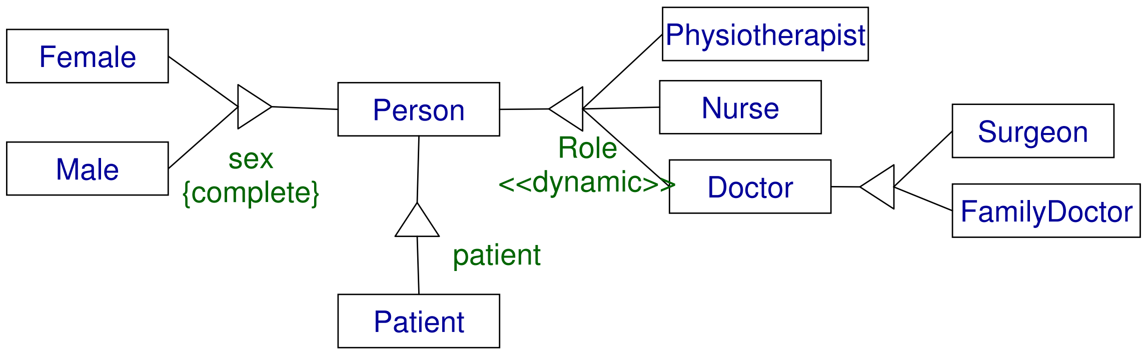
Suppose that according to the domain model:
A person part of the flight crew can also be a passenger.
Occasionally the flight crew can work as ticket agent.

Solution 2: **Delegation**



- Delegation is a way to extend a class’s behavior without using inheritance.
- It is useful when multiple and dynamic classification is not possible in the PL.

For homework



Method Specification

Method Specification

- Objective: give enough detail for the programmers
- There is no standard format for this
- Some organizations use forms of the following format:

Method Name:	Class Name:	ID:
Programmer:	Date due:	
Programming Language:		
Triggers/Events:		
Arguments Received:		
Data Type:	Notes:	
Messages Sent and Arguments Passed:		
ClassName.MethodName:	DataType:	Notes:
Arguments Returned:		
Data Type:	Notes:	
Algorithm Specification:		
Notes:		

Method Specification

Method Name:	Class Name:	ID:
Programmer:	Date due:	
Programming Language:		
Triggers/Events:		
Arguments Received:		
Data Type:	Notes:	
Messages Sent and Arguments Passed:		
ClassName.MethodName:	DataType:	Notes:
Arguments Returned:		
Data Type:	Notes:	
Algorithm Specification:		
Notes:		

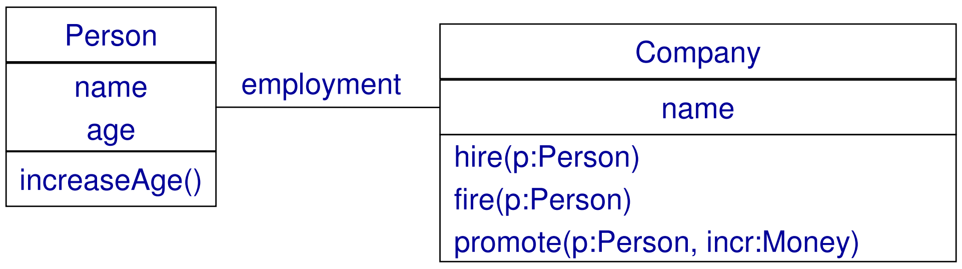
E.g. event-driven programming

Recall behavioural modeling

Pseudocode
Structured English
Activity Diagram

Constraints and Contracts

Constraints and Contracts



- *Can a minor (underage) work for a company ?*
- *Can a company hire a person already hired ?*
- *Can a promotion lower the salary of an employee?*

- A set of constraints and guarantees for classes and method
- We could express constraints using
  - natural language,
  - structured English,
  - pseudocode,
  - or a formal language.

Constraints and Contracts

- The designer should decide how to handle a violation of a constraint
  - abort, undo, let user handle it?
  - The designer must design the errors that the system is expected to handle. It is best not to leave these types of problems for the programmer to solve
  - Violations of a constraint are known as exceptions in languages like C++/Java.
- *In the next lecture we will see a formal language (called OCL) for expressing constraints*

Opportunities to Reuse

Identifying Opportunities for Reuse

We could exploit:

- Frameworks
- Class libraries
  - frameworks tend to be more domain specific. Frameworks may be implemented using class libraries
- Design Patterns

Identifying Opportunities for Reuse
Frameworks

Frameworks

- Is a set of implemented classes that can be used as the basis for implementing the system
- Most frameworks allow you to create subclasses
- Frameworks like CORBA and DCOM can be used to specify the physical architecture layer of the system
- Object-persistence frameworks can be used to add persistence to the problem domain classes and thus specify the data management layer of the system.

Design Patterns

Identifying Opportunities for Reuse

Design patterns

A pattern is a commonly occurring reusable piece in software system that provides a certain set of functionality.

- Using patterns in modeling of systems helps in keeping design standardized and more importantly, minimizes the reinventing of the wheel in the system design.
- *How design patterns relate to UML ?*
  - The patterns need to be captured and documented in a sufficiently descriptive manner so that they can be referred for future use.
  - UML provides the perfect tools to do just this. The class diagram in UML can be used to capture the patterns identified in a system. In addition, UML has a sufficiently extensive and expressive vocabulary to capture the details of patterns.

Categorizing Patterns

Based on how they are to be used, patterns are primarily categorized as:

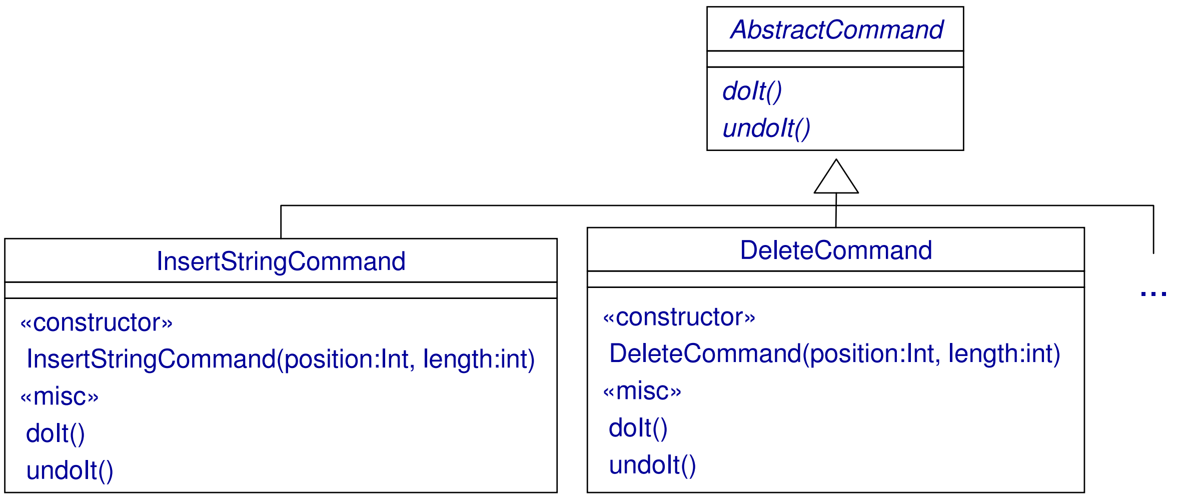
- **Creational**
  - They define mechanisms for instantiating objects. The implementation of the creational pattern is responsible for managing the lifecycle of the instantiated object.
  - Examples: Factory, Singleton
- **Structural**
  - They define compositions of objects and their organization to obtain new and varied functionality.
  - Examples: Adapter, Proxy.
- **Behavioral**
  - They define interaction between different objects.
  - Examples: Command, Iterator

Behavioral Patterns
Command

Encapsulate commands in objects so that you control their selection and sequencing, queue them, and othewise manipulate them

Example:

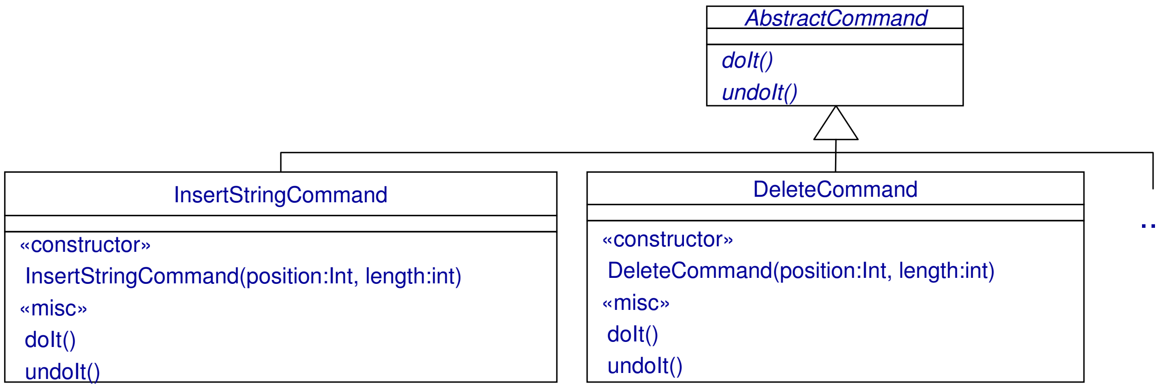
Design of a Word Processor that supports undoing and redoing commands



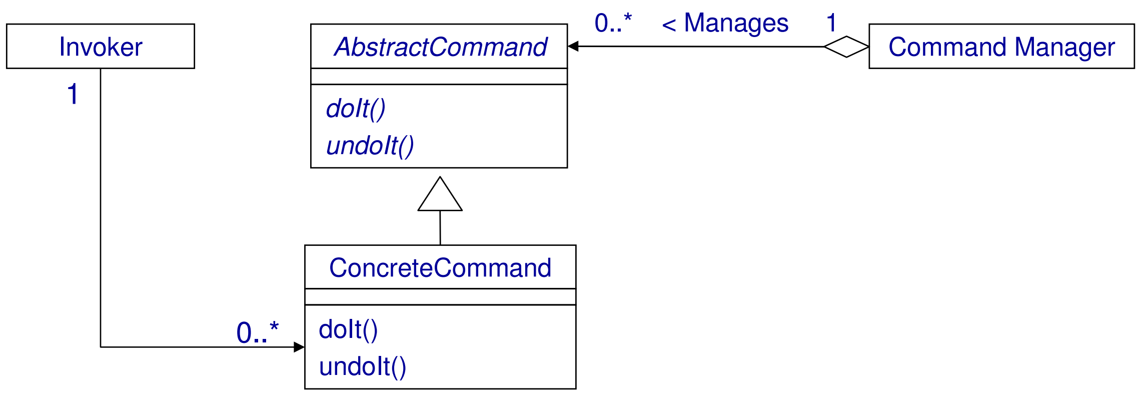
Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo

- materialize each command as an object with do and undo methods.
- when the user tells the WP to do something instead of performing the command,
  - it creates a new object using the appropriate constructor (e.g, an InsertStringCommand object)
  - it then calls the object's doIt method to execute the command

•The WP also puts the command object in a data structure that allow the WP to maintain a history of what commands have been executed. This allows the WP to undo commands in the reverse order that they were issued by calling their undo methods.

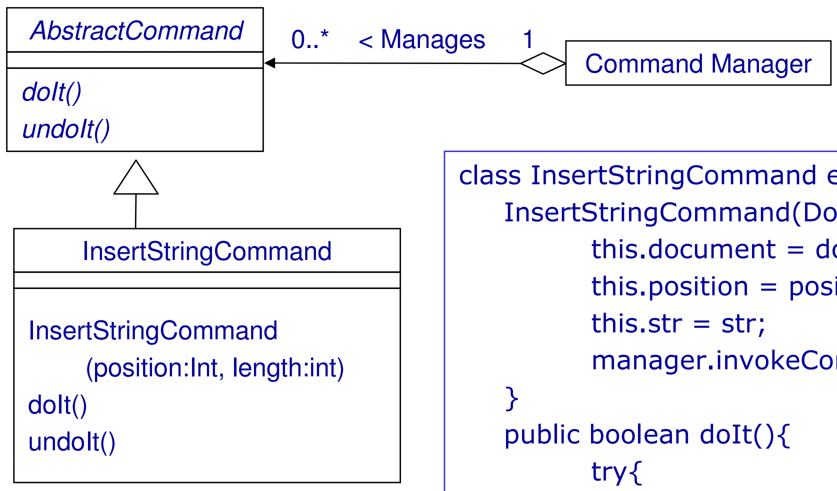


Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo



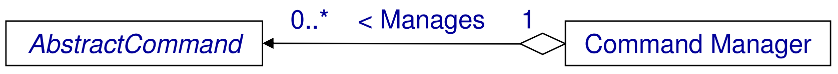
```
public abstract class AbstractCommand{
            public final static CommandManager manager
                        =new CommandManager();
            public abstract boolean doIt();  // returns True if successful
            public abstract boolean undoIt();
}
```

Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo (2)



```
class InsertStringCommand extends AbstractCommand{
    InsertStringCommand(Document doc, int position, String str){
            this.document = document;
            this.position = position;
            this.str = str;
            manager.invokeCommand(this);
    }
    public boolean doIt(){
            try{
                document.insertStringCommand(position, str);
            } catch (Exception e){
               return false;
            }
            return true;
   }
```

Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo (3)



```
class CommandManager{
    private int maxHistoryLength = 20;
    private LinkedList history = new LinkedList();
    private LinkedList redoList = new LinkedList();
    public void invokeCommand(AbstractCommand command){
            if (command instanceOf Undo){
                undo(); return;
            }
            if (command instanceOf Redo){
                redo(); return;
            }
             if (command.doIt()) {
                addToHistory(command);
            } else {
                history.clear()
            }
            if (redoList.size()>0)
                redoList.clear();
  }
```

```
interface Undo {
}
interface Redo{
}
```

```
Private void addToHistory(AbstractCommand command){
    history.addFirst(command);
    if (history.size() > maxHistoryLength) {
            history.removeLast();
    }
}
```

Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo (4)

Command Manager

```
Private void undo(){
    if (history.size() >0) {
            AbstractCommand undoCommand;
            undoCommand = (AbstractCommand) history.removeFirst();
            undoCommand.undoIt();
            redoList.addFirst(undoCommand);
    }
}
```

```
Private void redo(){
    if (redoList.size() >0) {
            AbstractCommand redoCommand;
            redoCommand = (AbstractCommand) redoList.removeFirst();
            redoCommand. doIt();
            history.addFirst (redoCommand);
    }
}
```

Design Patterns

- Design patterns is a useful mechanism to document and learn about common reusable design approaches.
- Design patterns can reduce the designing time for building systems and ensure that the system is consistent and stable in terms of architecture and design.
- The UML class diagrams provide an easy way to capture and document Design patterns.

- Some UML tools support design patterns.
  - They have a pre-built catalog of well-known design patterns. The design patterns can be easily pulled in into your design as templates and then customized for your application design.

More on patterns at CS352 - Software Engineering

Reading and References

- **Systems Analysis and Design with UML Version 2.0** (2nd edition) by A. Dennis, B. Haley Wixom, D. Tegarden, Wiley, 2005. Chapter 10
- **Requirements Analysis and System Design** (2nd edition) by Leszek A. Maciaszek, Addison Wesley, 2005, Chapter 5 and 6
- **Inheritance Hierarchies in KR and Programming Languages**, Lenzerini, Nardi, Simi, 1991
- **Patterns in Java,** Mark Grand, Wiley, 1998
- **Using Design Patterns in UML,** Mandar Chitnis, Pravin Tiwari, & Lakshmi Ananthamurthy
- **Slides of John Mylopoulos, University of Toronto**
- **Αντικειμενοστρεφής Σχεδίαση: UML, Αρχές, Πρότυπα και Ευρετικοί Κανόνες,** Α. Χατζηγεωργίου, Κλειδάριθμος 2005