

# Chapter 8: Class and Method Design



# Objectives

- Become familiar with coupling, cohesion, and connascence.
- Be able to specify, restructure, and optimize object designs.
- Be able to identify the reuse of predefined classes, libraries, frameworks, and components.
- Be able to specify constraints and contracts.
- Be able to create a method specification.

# Introduction

- Review the characteristics of object orientation
- Present useful criteria for evaluating a design
- Present design activities for classes and methods
- Present the concept of constraints & contracts to define object collaboration
- Discuss how to specify methods to augment method design
- Caution:
  - Class & method design must precede coding
  - While classes are specified in some detail, jumping into coding without first designing them may be disastrous

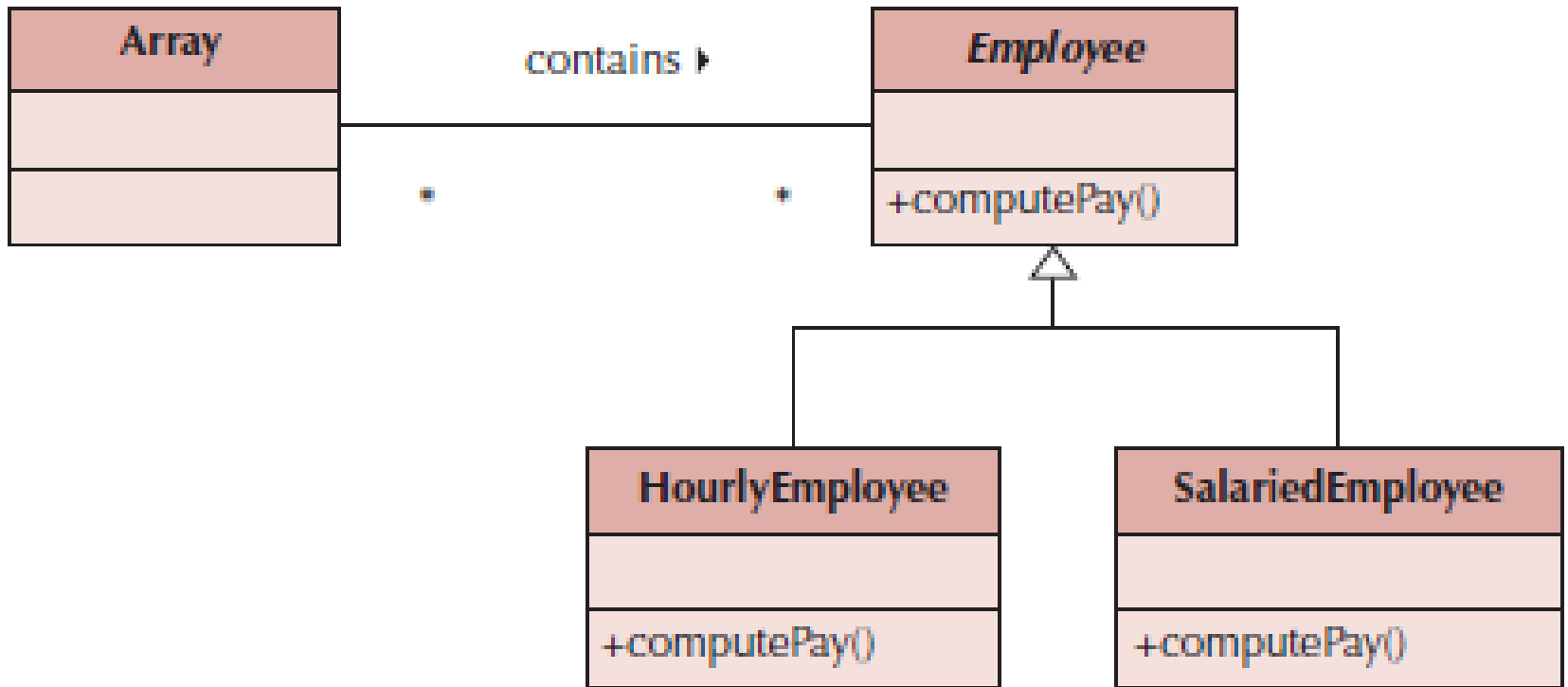
# Characteristics of OOSAD

- Classes
  - Instantiated classes are objects
  - Classes are defined with attributes, states & methods
  - Classes communicate through messages
- Encapsulation & information hiding
  - Combine data and operations into a single object
  - Reveal only how to make use of an object to other objects
  - Key to reusability
- Polymorphism & dynamic binding
- Inheritance

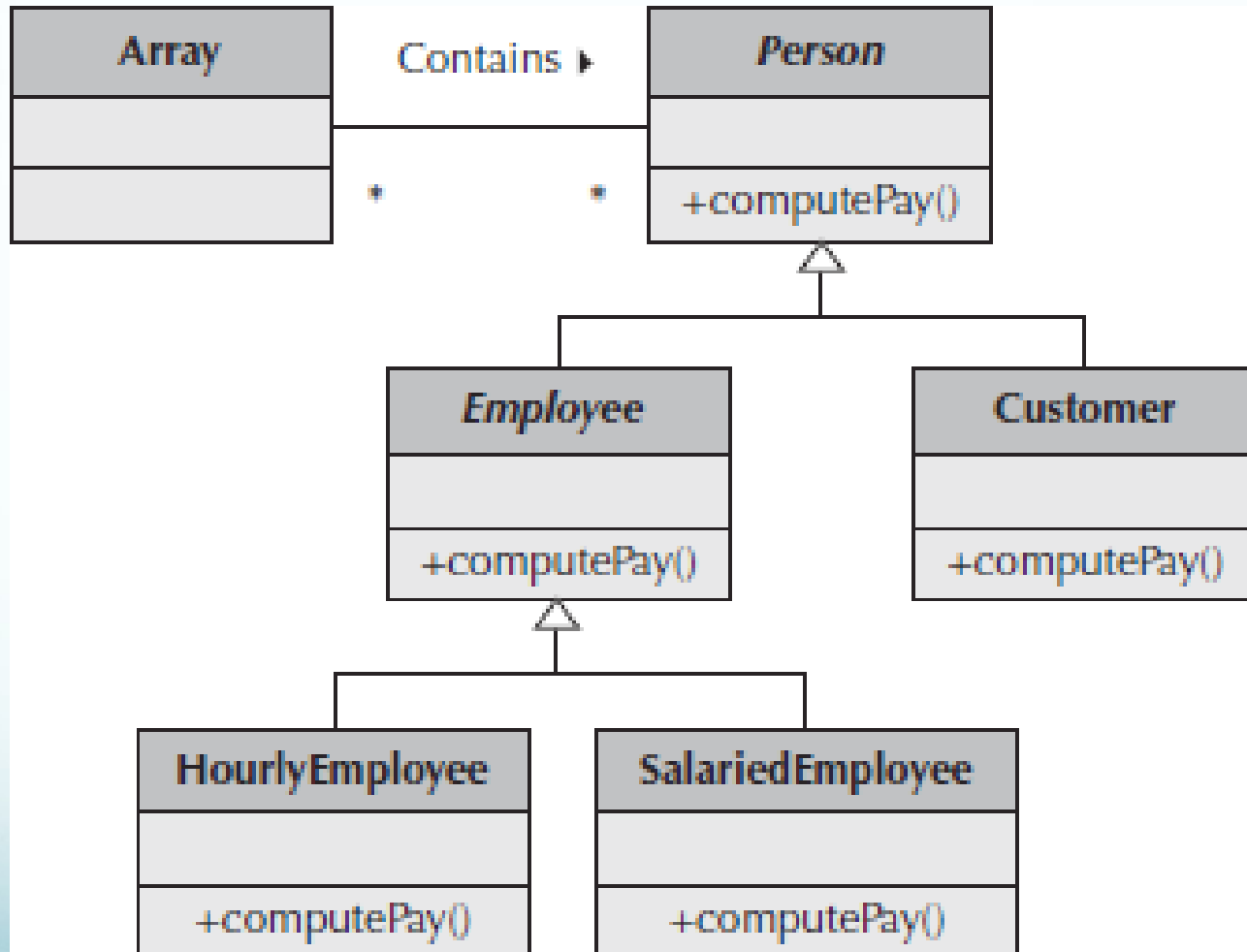
# Polymorphism & Dynamic Binding

- Polymorphism
  - The ability to take on several different forms
  - Same message triggers different methods in different objects
- Dynamic binding
  - Methods—the specific method used is selected at run time
  - Attributes—data type is chosen at run time
  - Implementation of dynamic binding is language specific
- Decisions made at run time may induce run-time errors
- Need to ensure semantic consistency
  - Yes the methods have different semantics
  - But they are logically the same thing.

# Polymorphism Example



# Polymorphism Misuse



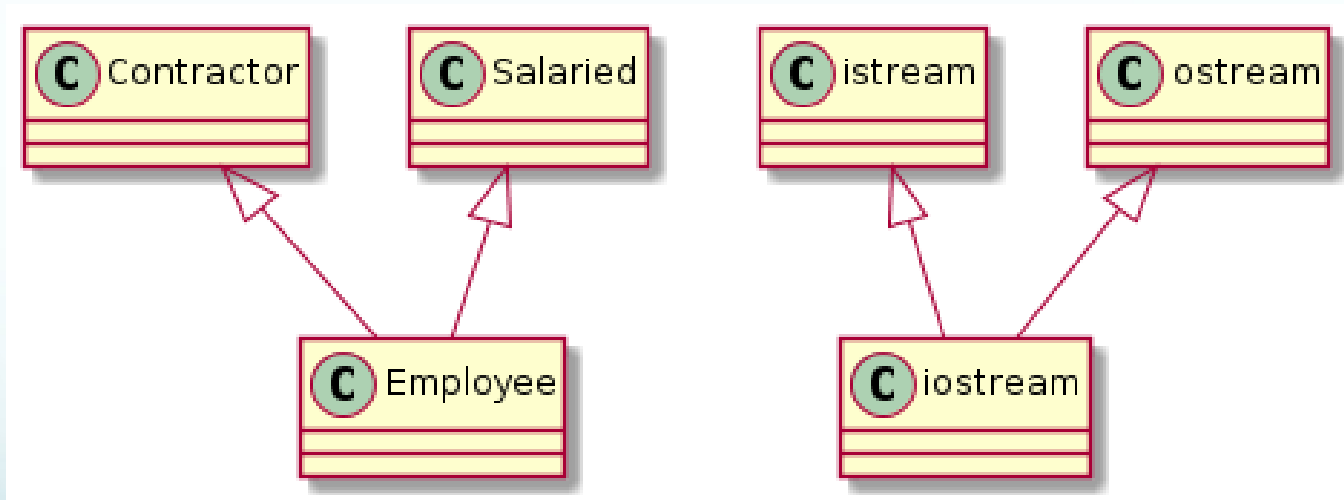
# Inheritance

- Permits reuse of existing classes with extensions for new attributes or operations
- Types
  - Single inheritance -- one parent class
  - Multiple inheritance -- multiple parent classes (not supported by all programming languages)
  - Redefinition of methods and/or attributes
    - Not supported by all programming languages
    - May cause inheritance conflict
- Designers must know what the chosen programming language supports



# Multiple Inheritance

- Supported in C++
- Java interfaces are a limited version.
- Useful when objects are characterized on different dimensions.



# Multiple Inheritance

- Actually does make sense for some things.
- Increases opportunities to make errors.
- Few languages support it.
- Many development shops forbid its use.



# Inheritance Conflicts

- An attribute or method in a derived class with the same name as an attribute or method in the super class.
  - Generally, the derived declaration simply overrides the inherited one.
  - An error, if this was not what was intended.
- Complicated by multiple inheritance, since the same name can be inherited from multiple base classes.
  - Generally, any direct reference must be qualified.

# Design Criteria

- A set of metrics to evaluate the design
- Coupling—refers to the degree of the closeness of the relationship between classes
- Cohesion—refers to the degree to which attributes and methods of a class support a single object
- Connascence—refers to the degree of interdependency between objects



# Coupling

- Close coupling means that changes in one part of the design may require changes in another part
- Types
  - Interaction coupling measured through message passing
  - Inheritance coupling deals with the inheritance hierarchy of classes
- Minimize interaction coupling by restricting messages (Law of Demeter)
- Minimize inheritance coupling by using inheritance to support only generalization/specialization and the principle of substitutability

# Law of Demeter

Messages should be sent only by an object:

to itself


to objects contained in attributes of itself or a superclass

to an object that is passed as a parameter to the method

to an object that is created by the method

to an object that is stored in a global variable

# Types of Interaction Coupling

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object 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 "global data area" that is outside the individual objects.
	Bad	Content or Pathological

Source: These types were adapted from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed. (Englewood Cliffs, NJ: Yardon Press, 1988); and Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).

# Cohesion


- A cohesive class, object or method refers to a single thing
- Types
  - Method cohesion
    - Does a method perform more than one operation?
    - Performing more than one operation is more difficult to understand and implement
  - Class cohesion
    - Do the attributes and methods represent a single object?
    - Classes should not mix class roles, domains or objects
  - Generalization/specialization cohesion
    - Classes in a hierarchy should show “a-kind-of” relationship, not associations or aggregations



# Types of Method Cohesion


Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.

# Types of Method Cohesion

Level	Type	Description
 Bad	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 into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its calling method.
	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

*Source:* These types are based on material from Page-Jones, *The Practical Guide to Structured Systems*; Myers, *Composite/Structured Design*; Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design* (Englewood Cliffs, NJ: Prentice-Hall, 1979).

# Types of Class Cohesion

Level	Type	Description
Good	Ideal	The class has none of the 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. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

Source: Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

# Connascence

- Classes are so interdependent that a change in one necessitates a change in the other
- Good programming practice should:
  - Minimize overall connascence; however, when combined with encapsulation boundaries, you should:
    - Minimize across encapsulation boundaries (less interdependence between or among classes)
    - Maximize within encapsulation boundary (greater interdependence within a class)
  - A sub-class should never directly access any hidden attribute or method of a super class

# Types of Connascence

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.
<i>Source: Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, <i>Fundamentals of Object-Oriented Design in UML</i>.</i>	

# Object Design Activities

- An extension of analysis & evolution activities
- Expand the descriptions of partitions, layers & classes by:
  - Adding specifications to the current model
  - Identifying opportunities to reuse classes that already exist
  - Restructuring the design
  - Optimize the design
  - Map the problem domain classes into a programming language



# Adding Specifications

- Review the current set of analysis models
  - All classes included are both sufficient and necessary to solve the problem
  - No missing attributes or methods
  - No extra or unused attributes or methods
  - No missing or extra classes
- Examine the visibility of classes
  - Private—not visible
  - Public—visible to other classes
  - Protected: visible to descendants classes.
    - In Java, also visible to all classes in the same package.

# Adding Specifications (cont.)

- Decide on method signatures:
  - Name of the method
  - Parameters or arguments to pass
  - Type of value(s) to be returned
- Define constraints that must be preserved by the objects
  - Preconditions, post-conditions, & invariants
  - Decide how to handle constraint violations



# Preconditions and Postconditions

- Apply to methods
- What must be true before calling, and what must be true after return
- For `Stack::pop`
  - Precondition: stack is not empty
  - Postcondition: stack contents is the same, except top has been removed.
- For `Stack::push`
  - Postcondition: stack is the same, with arg. added to the top.
  - If the stack is implemented in a fixed array, precondition that stack is not full.



# Invariants

- Apply to all objects of some class.
- Established by the constructor.
- Maintained by each method, though may be temporarily false during execution of a method; true before and true after.

# Example Invariants

- Head pointer points to the top node of the stack, and tail pointer points to the bottom node.
- For BST structures: all values of the left subtree are less than the root, and all values in the right subtree are greater or equal.
- The attribute size is equal to the number of items in the linked list.

# Identify Opportunities for Reuse

- Design patterns—groupings of classes that help solve a commonly occurring problem
- Framework—a set of implemented classes that form the basis of an application
- Class libraries—also a set of implemented classes, but more general in nature than a framework
- Components—self-contained classes used as plug-ins to provide specific functionality
- Choice of approaches depends on the layer



# Restructuring the Design

- Factoring—separating aspects from a class to simplify the design
- Normalization—aids in identifying missing classes
- Assure all inheritance relationships support only generalization/specialization semantics



# Optimizing the Design

- Balance clarity with efficiency
- Methods:
  - Review access paths between objects
    - If a frequent message is passed through many objects to reach its destination, maybe you need to go direct.
    - Add an attribute in the caller to refer directly to the callee.
  - Review all attributes of each class
    - Suppose  $x$ ,  $y$  and  $z$  belong to class  $Q$ , and are only read and set.
    - Suppose the read and set are only called from class  $P$ .
    - Maybe those attributes belong in class  $P$ .

# Optimizing the Design, Cont

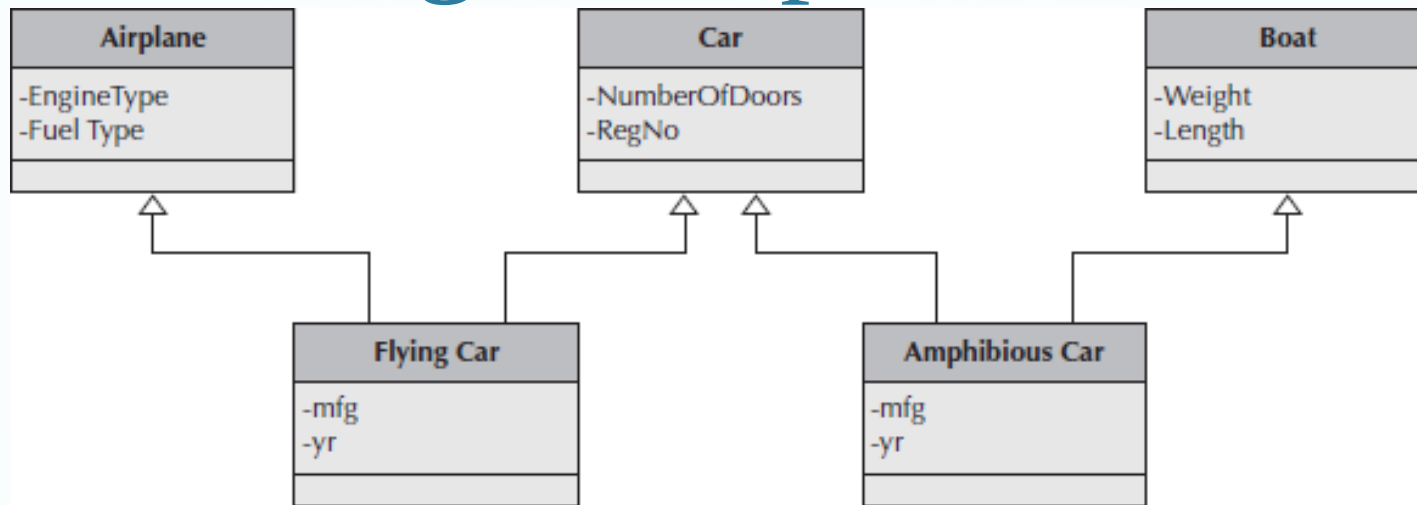
- Methods:
  - Consider execution order of statements in often-used methods: for instance, search the rare condition first.
  - If a value is recomputed often, consider caching it.
    - Create a derived attribute.
    - Recompute when any input changes
    - Just record the fact when an input changes and recompute when out of date.
  - Consider combining classes that form a one-to-one association

# Mapping Problem-Domain Classes

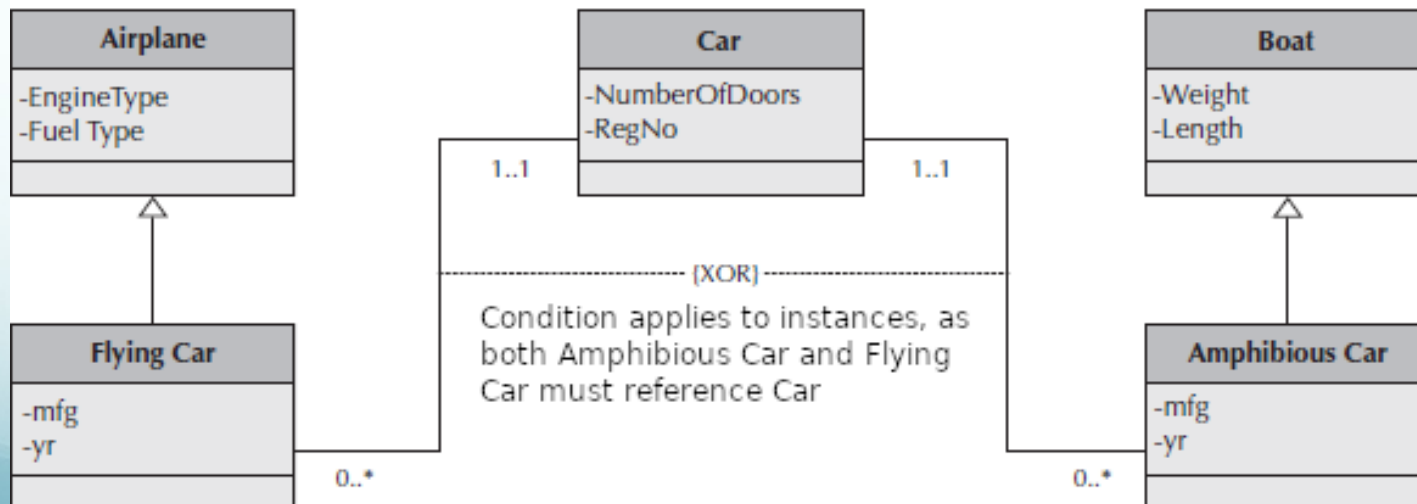
- Factor out multiple inheritance if using a language that supports only single inheritance
- Factor out all inheritance if the language does not support inheritance
- Avoid implementing an object-oriented design in non-object languages
  - Objects can be faked by combining structs and top-level functions.
  - Inheritance can be faked using method pointers.
  - Assuming the language does support these things, and you're willing to go to a lot of trouble.



# Removing Multiple Inheritance



(a)



(b)

# Constraints and Contracts

- A contract is a set of constraints & guarantees
  - If the requestor (client) meets the constraints, the responder (server) will guarantee certain behavior
    - Constraints must therefore be unambiguous
  - Contracts document message passing between objects
  - A contract is created for each visible method in a class
  - Should contain enough information for the programmer to understand what the method is supposed to do
- Constraint types
  - Precondition—must be true *before* the method executes
  - Post-condition—must be true *after* the method finishes
  - Invariant—must *always* be true for all instances of a class

# Constraints

- Constraints can be expressed in UML's Object Constraint Language (OCL).
- Comparison and boolean operators.
- Basic spreadsheet-like operations.
- Refer to object attributes.

## **Order class invariants:**

**Cust ID = Customer.GetCustID()**

**State Name = Sate.GetState()**

**Sub Total = ProductOrder.sum(GetExtension())**

**Tax = State.GetTaxRate() \* Sub Total**

# OCL

Operator Type	Operator	Example
<b>Comparison</b>	=	a = 5
	<	a < 100
	<=	a <= 100
	>	a > 100
	>=	a >= 100
	<>	a <> 100
<b>Logical</b>	and	a and b
	or	a or b
	xor	a xor b
	not	not a
<b>Math</b>	+	a + b
	-	a - b
	*	a * b
	/	a / b
<b>String</b>	concat	a = b.concat(c)
<b>Relationship Traversal</b>	.	relationshipAttributeName.b
	::	superclassName::propertyName
<b>Collection</b>	size	a.size
	count(object)	a.count(b)
	includes(object)	a.includes(b)
	isEmpty	a.isEmpty
	sum()	a.sum(b,c,d)
	select(expression)	a.select(b > d)

# Sample Contract Form

<b>Method Name:</b>	<b>Class Name:</b>	<b>ID:</b>
<b>Clients (Consumers):</b>		
<b>Associated Use Cases:</b>		
<b>Description of Responsibilities:</b>		
<b>Arguments Received:</b>		
<b>Type of Value Returned:</b>		
<b>Pre-Conditions:</b>		
<b>Post-Conditions:</b>		

# Method Specification

- Documentation details for each method
  - Allows programmers to code each method
- Must be explicit and clear
- No formal standards exist, but information should include:
  - General information (e.g., method name, class name, etc.)
  - Events—anything that triggers a method (e.g., mouse click)
  - Message passing including values passed into a method and those returned from the method
  - Algorithm specifications
    - Pseudocode
    - Activity diagram (flowchart).
  - Other applicable information (e.g., calculations, procedure calls)

# Method Specification Form

<b>Method Name:</b> insertOrder	<b>Class Name:</b> OrderList	<b>ID:</b> 100
<b>Contract ID:</b> 123	<b>Programmer:</b> J. Doe	<b>Date Due:</b> 1/1/12
<b>Programming Language:</b>		
<input type="checkbox"/> Visual Basic <input type="checkbox"/> Smalltalk <input type="checkbox"/> C++ <input type="checkbox"/> Java		
<b>Triggers/Events:</b>		
Customer places an order		
<b>Arguments Received:</b>	<b>Notes:</b>	
<b>Data Type:</b>		
Order	The new customer's new order.	
<b>Messages Sent &amp; Arguments Passed:</b>		
<b>ClassName.MethodName:</b>	<b>Data Type:</b>	<b>Notes:</b>
OrderNode.new()	Order	
OrderNode.getOrder()		
Order.getOrderNumber()		
OrderNode.setNextNode()	OrderNode	
self.middleListInsert()	OrderNode	
<b>Arguments Returned:</b>	<b>Notes:</b>	
<b>Data Type:</b>		
void		
<b>Algorithm Specification:</b>		
See Figures 8-30 and 8-31.		
<b>Misc. Notes:</b>		
None.		

# Summary

- Basic Characteristics of Object Orientation (review)
- Design Criteria—coupling, cohesion & connascence
- Object Design Activities
- Constraints and Contracts
- Method Specification

