

# **Systems Analysis and Design with UML Version 2.0, Second Edition**

---

Alan Dennis, Barbara Haley Wixom, and David Tegarden

## **Chapter 10: Class and Method Design**

John Wiley & Sons, Inc.

Copyright 2005

# Copyright © 2005 John Wiley & Sons, Inc.

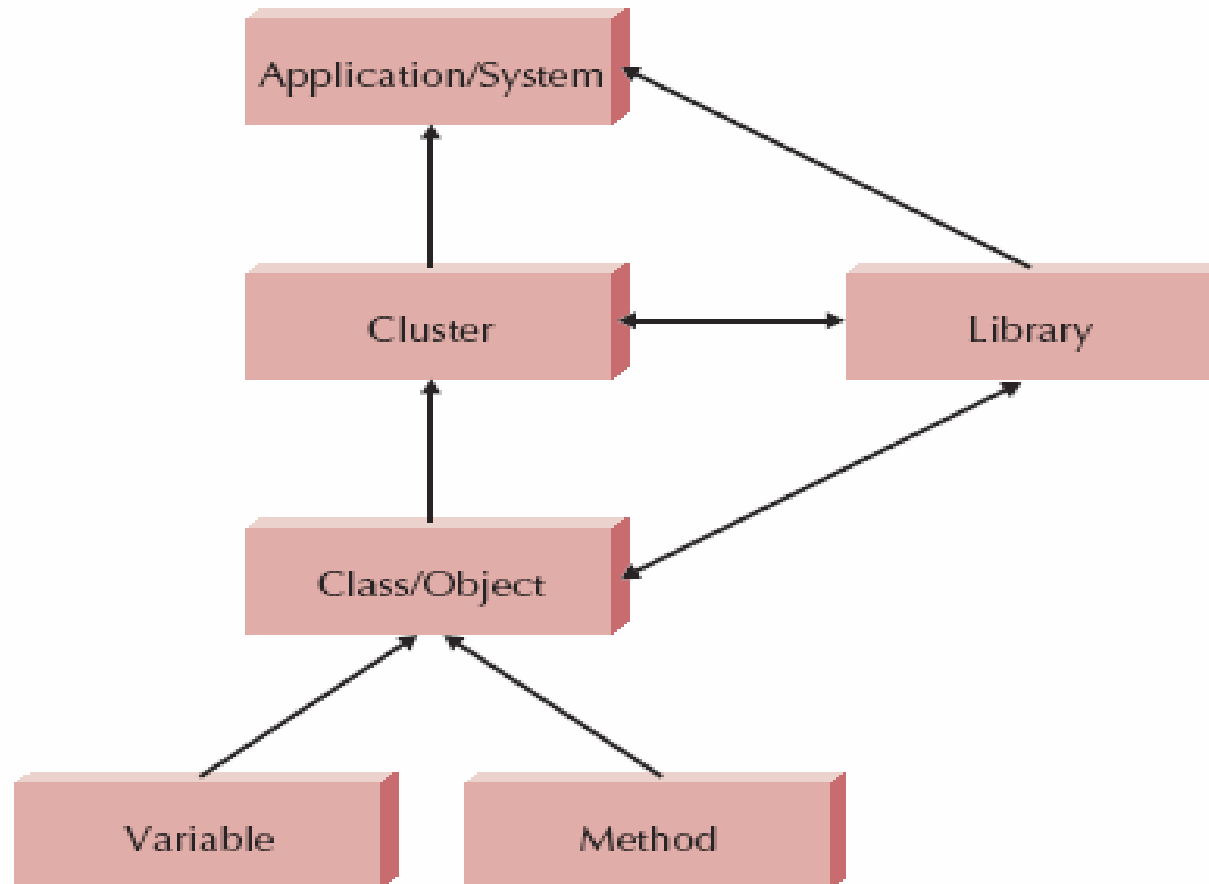
---

- All rights reserved. Reproduction or translation of this work beyond that permitted in Section 117 of the 1976 United States Copyright Act without the express written permission of the copyright owner is unlawful.
- Request for further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.
- The purchaser may make back-up copies for his/her own use only and not for redistribution or resale.
- The Publisher assumes no responsibility for errors, omissions, or damages, caused by the use of these programs or from the use of the information contained herein.

# REVISITING THE BASIC CHARACTERISTICS OF OBJECT-ORIENTATION



# Levels of Abstraction



# Elements



- ▣ Classes
- ▣ Objects
- ▣ Attributes
- ▣ States
- ▣ Methods
- ▣ Messages

# 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
- ❑ Redefinition and inheritance conflict
- ❑ Most inheritance conflicts are due to poor classification



# Rumbaugh's Rules



Query operations should not be redefined

Methods that redefine inherited ones should only restrict the semantics of the inherited ones

The underlying semantics of the inherited method should never be changed

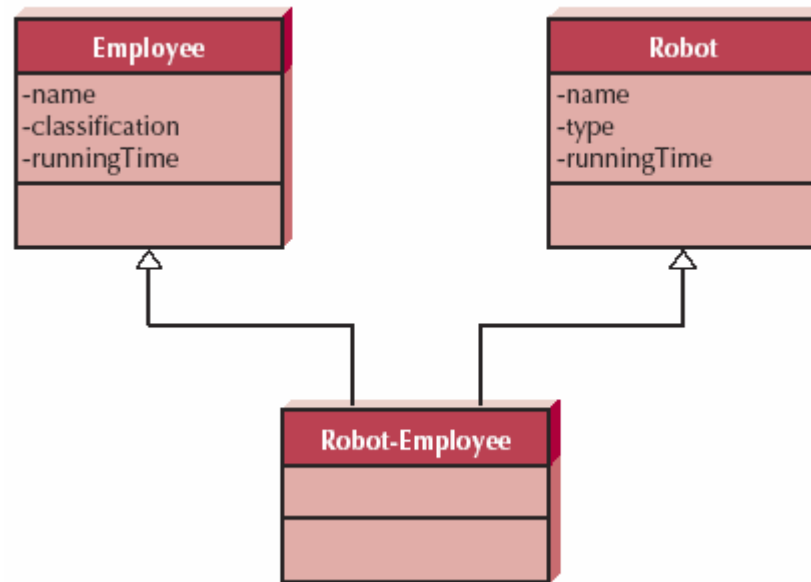
The signature (argument list) of the inherited method should never be changed

# Additional Inheritance Conflicts

---

- Two inherited attributes or methods have same name and semantics
- Two inherited attributes or methods have different name, but same semantics
- Two inherited attributes or methods have same name and different semantics

# Inheritance Conflicts with Multiple Inheritance



# Design Criteria




## Chapter 10

# Coupling




- ▣ Indicates the interdependence or interrelationships of the modules
- ▣ Interaction coupling
  - ▣ Relationships with methods and objects through message passage

# 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



# 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.
	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 send by its calling method.
Bad	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.




# Ideal Class Cohesion

---

- Contain multiple methods that are visible outside the class
- Have methods that refer to attributes or other methods defined with the class or its superclass
- Not have any control-flow coupling between its methods

# 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.

# Connascence



- Two modules (classes or methods) are so intertwined, that if you make a change in one, it is likely that a change in the other will be required

# Connascence and Encapsulation Levels

---

- ☒ Minimize overall connascence by eliminating any unnecessary connascence throughout the system,
- ☒ Minimize connascence across any encapsulation boundaries, such as method boundaries and class boundaries,
- ☒ Maximize connascence within any encapsulation boundary.

# Object Design Activities



# 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.

Source: Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Page-Jones, *Fundamentals of Object-Oriented Design in UML*.



# Additional Specification

---

- ☒ Ensure the classes are both necessary and sufficient for the problem
- ☒ Finalize the visibility of the attributes and methods of each class
- ☒ Determine the signature of every method of each class
- ☒ Define constraints to be preserved by objects

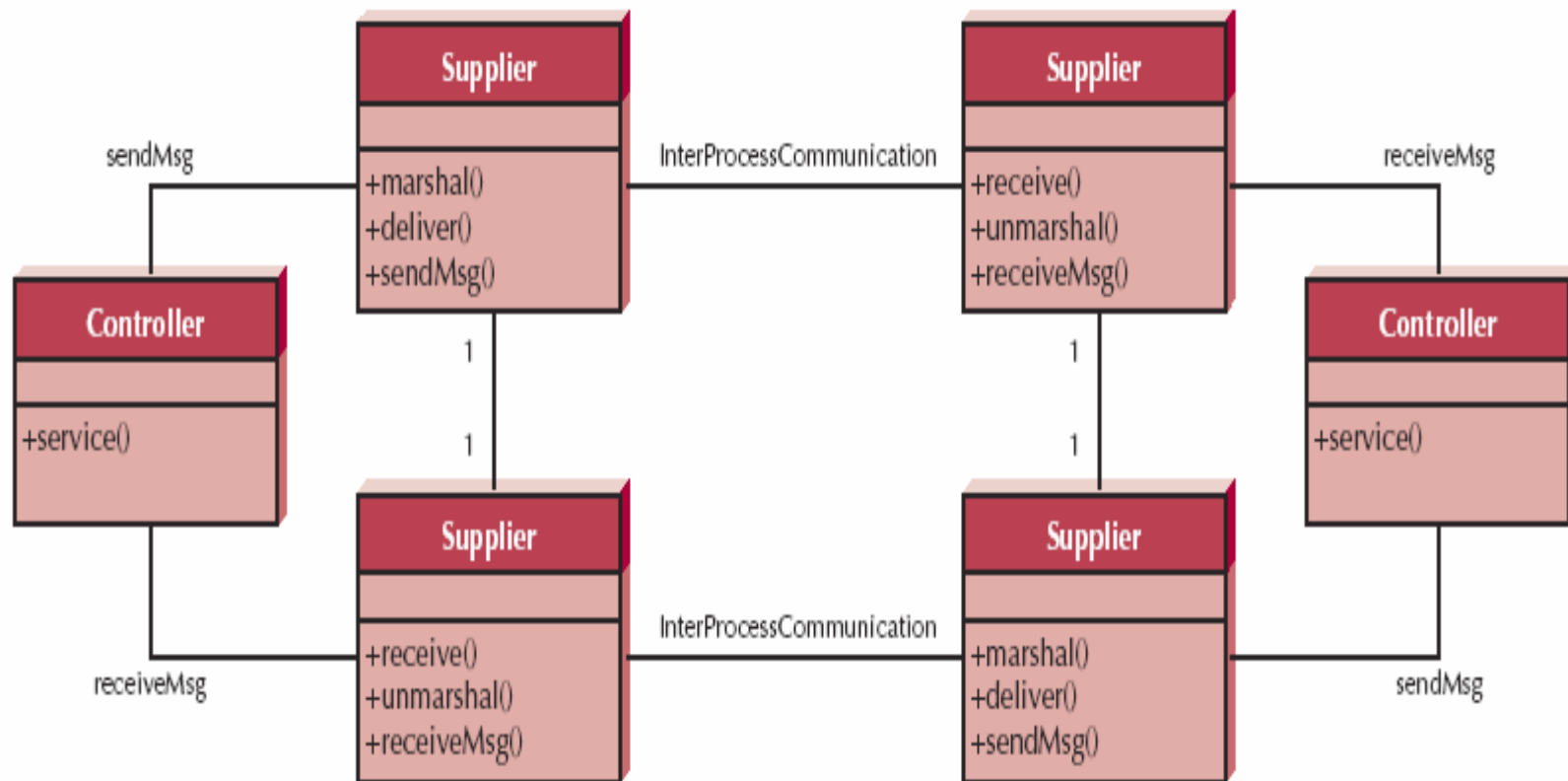


# Identifying Opportunities for Reuse

---

- ▣ Analysis patterns
- ▣ Design patterns
- ▣ Frameworks
- ▣ Libraries
- ▣ components

# Sample Design Pattern



# Restructuring the Design



- ☒ Factoring
  - ☐ Separate aspects of a method or class into a new method or class
- ☒ Normalization
  - ☐ Identifies classes missing from the design
- ☒ Challenge inheritance relationships to ensure they only support a generalization/specialization semantics

# Optimizing the Design



- ▣ Review access paths between objects
- ▣ Review each attribute of each class
- ▣ Review fan-out of each method
- ▣ Examine execution order of statements
- ▣ Create derived activities

# Map Problem Domain Classes to Implementation Languages

---

- Single-Inheritance Language
  - Convert relationships to association relationships
  - Flatten inheritance hierarchy by copying attributes and methods of additional superclass(es)

# Implement in Object-Based Language

---

- Factor out all uses of inheritance from the problem domain class design

# Your Turn

---

- Dentist office appointment system
  - Assume that you now know that the system must be implemented in Visual Basic 6, which does not support implementation inheritance.
  - As such, redraw the class diagram factoring out the use of inheritance in the design by applying the above rules.



# Implement in a Traditional Language



- Stay away from this!
- But if necessary, factor out all uses of
  - Polymorphism
  - Dynamic binding
  - Encapsulation
  - Information hiding

# Constraints and Contracts

# Types of Constraints



- ☒ Pre-Conditions
  - ☐ A constraint to be met to allow a method to execute
- ☒ Post-condition
  - ☐ A constraint to be met after a method executes
- ☒ Invariants
  - ☐ Constraints that must be true for all instances of a class

# Invariants

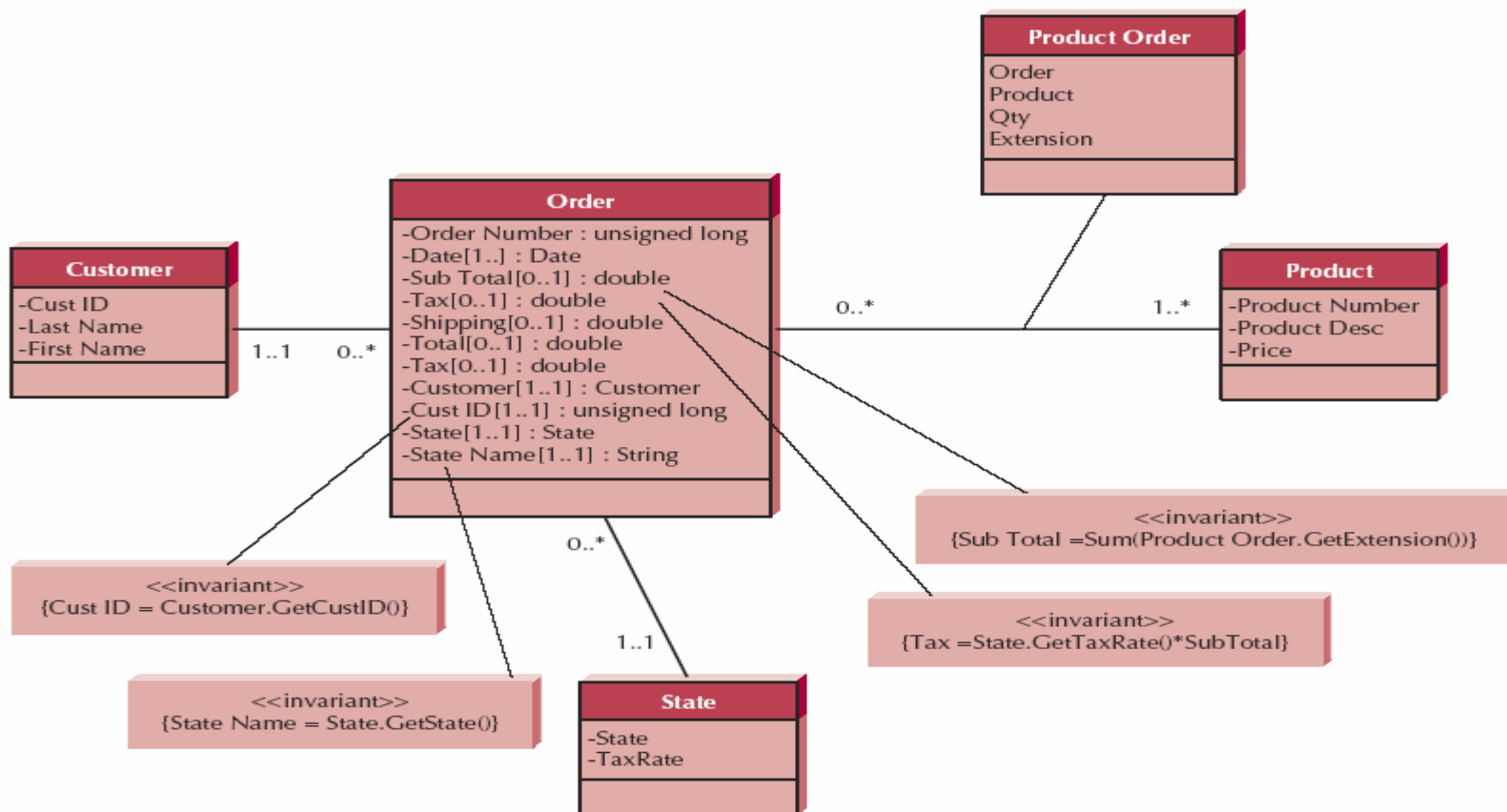


FIGURE 10-12 Invariants on a Class Diagram

# Elements of a Contract

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

# Your Turn

---

- **Using** the CRC card in Figure 10-11, the class diagram in Figure 10-12, and the sample contract format in Figure 10-13 as guides, create contracts for the Calculate subtotal, Calculate tax, Calculate shipping, and Calculate total methods.

# Method Specification





# Method Specification



- ▣ General information
- ▣ Events
- ▣ Message passing
- ▣ Algorithm specification
  - ▣ Structured English
  - ▣ Pseudocode
  - ▣ UML activity diagram

# Applying the Concepts at CD Selections



# Revised CD Selections Class Diagram

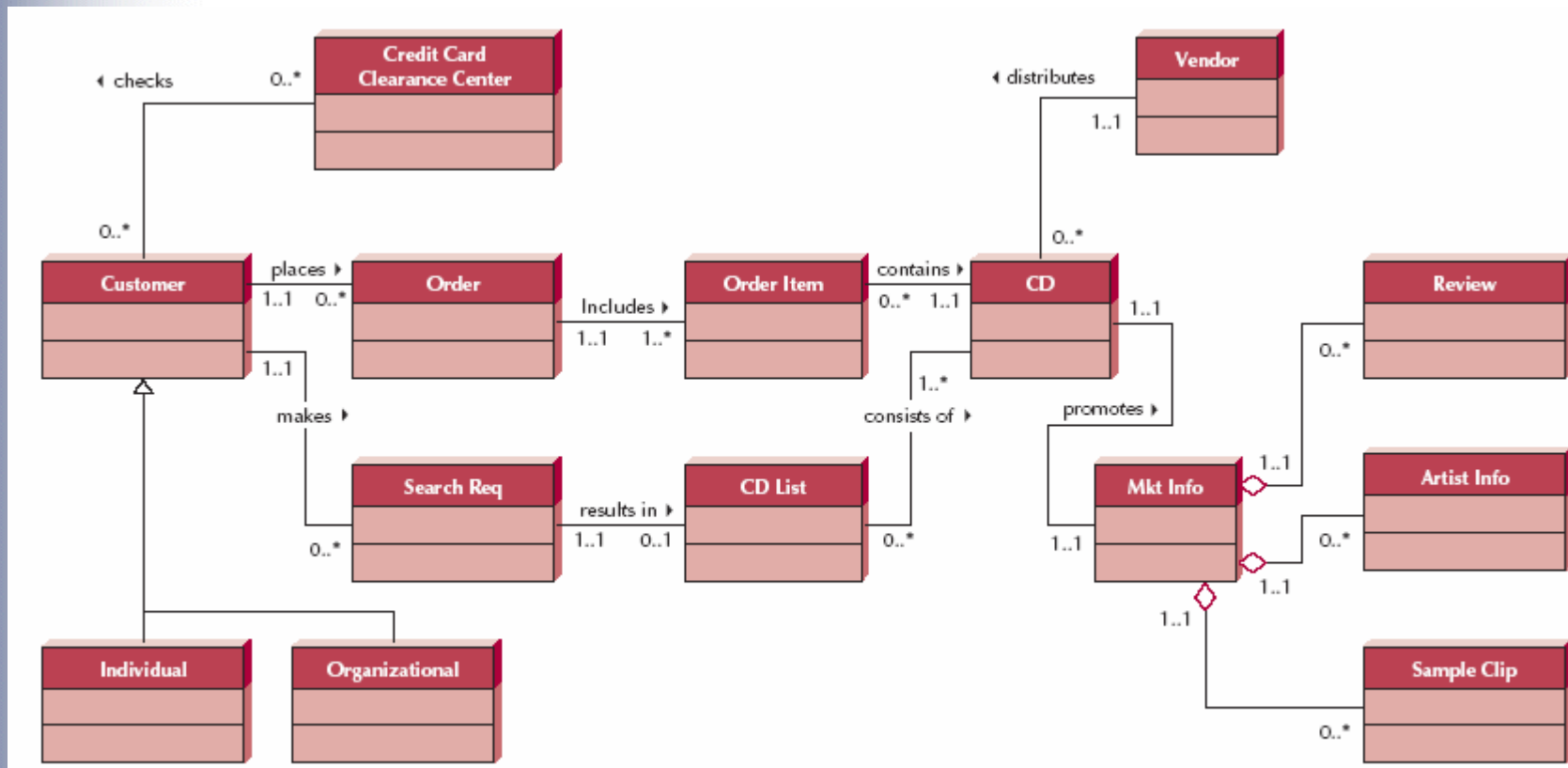


FIGURE 10-17 Revised CD Selections Internet Sales System Class Diagram (Places Order Use Case View)

# Back of CD CRC Card

## Back:

### Attributes:

CD Number	(1..1)	(unsigned long)	_____
CD Name	(1..1)	(String)	_____
Pub Date	(1..1)	(Date)	_____
Artist Name	(1..1)	(String)	_____
Artist Number	(1..1)	(unsigned long)	_____
Vendor	(1..1)	(Vendor)	_____
Vendor ID	(1..1)	(unsigned long) {Vendor ID = Vendor.GetVendorID()}	_____
			_____
			_____
			_____

### Relationships:

**Generalization (a-kind-of):** \_\_\_\_\_

**Aggregation (has-parts):** \_\_\_\_\_

**Other Associations:**

Order Item {0..\*} CD List {0..\*} Vendor {1..1} Mkt Info {0..1}

# Get Review Method Contract

<b>Method Name:</b> GetReview()	<b>Class Name:</b>	<b>ID:</b>
<b>Clients (Consumers):</b> CD Detailed Report		
<b>Associated Use Cases:</b> Places Order		
<b>Description of Responsibilities:</b> Return review objects for the Detailed Report Screen to display		
<b>Arguments Received:</b>		
<b>Type of Value Returned:</b> List of Review objects		
<b>Pre-Conditions:</b> Review attribute not Null		
<b>Post-Conditions:</b>		

# Revised Package Diagram

