

# Chapter 12 - Object-Oriented Design

---

# Relationships Between Classes

---

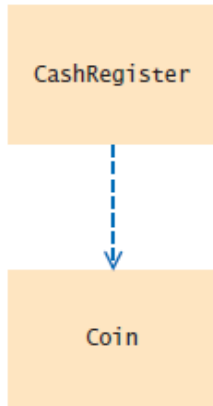
The most common types of relationships:

- Dependency
- Aggregation
- Inheritance

# Dependency

---

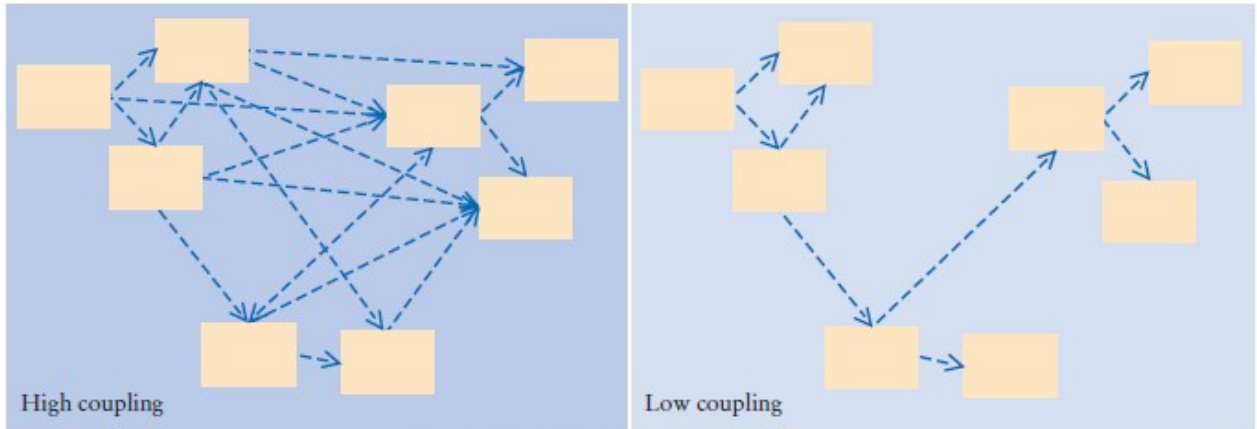
- A class depends on another class if it uses objects of that class.
  - *The “knows about” relationship.*
- Example: CashRegister depends on Coin



**Figure 3** Dependency Relationship Between the CashRegister and Coin Classes

# Dependency

- It is a good practice to minimize the coupling (i.e., dependency) between classes.



**Figure 4** High and Low Coupling Between Classes

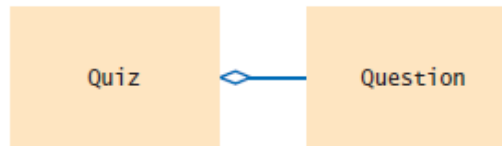
- When a class changes, coupled classes may also need updating.

# Aggregation

---

- A class aggregates another if its objects contain objects of the other class.
  - *Has-a* relationship
- Example: a `Quiz` class aggregates a `Question` class.
- The UML for aggregation:

**Figure 5**  
Class Diagram  
Showing Aggregation



- Aggregation is a stronger form of dependency.
- Use aggregation to remember another object between method calls.

- Use an instance variable

```
public class Quiz
{
    private ArrayList<Question> questions;
    . . .
}
```

A class may use the `Scanner` class without ever declaring an instance variable of class `Scanner`.

This is dependency NOT aggregation

# Aggregation

---



© bojan fatur/iStockphoto.

A car has a motor and tires. In object-oriented design, this “has-a” relationship is called aggregation.

# Inheritance

---

- Inheritance is a relationship between a more general class (the superclass) and a more specialized class (the subclass).
  - The “is-a” relationship.
  - Example: Every truck is a vehicle.
- Inheritance is sometimes inappropriately used when the has-a relationship would be more appropriate.
  - Should the class `Tire` be a subclass of a class `Circle`? No
    - A tire has a circle as its boundary
    - Use aggregation

```
public class Tire
{
    private String rating;
    private Circle boundary;
    . . .
}
```



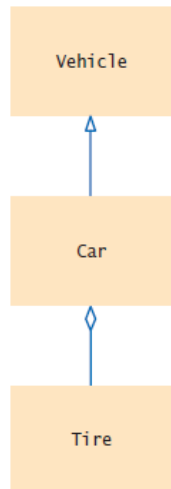
# Inheritance

---

- Every car is a vehicle. (Inheritance)
- Every car has a tire (or four). (Aggregation)

```
class Car extends Vehicle
{
    private Tire[] tires;
    . . .
}
```





- Aggregation denotes that objects of one class contain references to objects of another class.



**Figure 6** UML Notation for Inheritance and Aggregation

# UML Relationship Symbols

---

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

## Self Check 12.7

---

Consider the `Question` and `ChoiceQuestion` objects. How are they related?

**Answer:** The `ChoiceQuestion` class inherits from the `Question` class.

## Self Check 12.9

---

Why should coupling or dependency be minimized between classes?

**Answer:** If a class doesn't depend on another, it is not affected by interface changes in the other class.

## Self Check 12.10

---

In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.

**Answer:**

## Self Check 12.10

---

In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.

**Answer:**



## Self Check 12.11

---

You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the `Book` class aggregate `Patron` or the other way around?

**Answer:** Typically, a library system wants to track which books a patron has checked out, so it makes more sense to have `Patron` aggregate `Book`. However, there is not always one true answer in design.

If you feel strongly that it is important to identify the patron who checked out a particular book (perhaps to notify the patron to return it because it was requested by someone else), then you can argue that the aggregation should go the other way around.

Please note, implementation of this behavior is also possible when `Patron` aggregate `Book`.

## Self Check 12.12

---

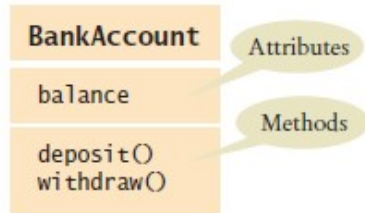
In a library management system, what would be the relationship between classes Patron and Author?

**Answer:** There would be no relationship.



# Attributes and Methods in UML Diagrams

---



# Multiplicities

---

- any number (zero or more): \*
- one or more: 1..\*
- zero or one: 0..1
- exactly one: 1



*An Aggregation Relationship with Multiplicities*

# Aggregation and Association, and Composition

---

- Association: More general relationship between classes.
- Use early in the design phase.
- A class is associated with another if you can navigate from objects of one class to objects of the other.
- Given a Bank object, you can navigate to Customer objects.



*An Association Relationship*

- Composition: one of the classes can not exist without the other.



*A Composition Relationship*

# Association

---

## Association

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).

Association can be represented by a line between these classes with an arrow indicating the navigation direction. In case an arrow is on both sides, the association is known as a bidirectional association.

A single student can associate with multiple teachers:



The example indicates that every Instructor has one or more Students:



# Aggregation vs Composition

---

Aggregation and Composition are subsets of association meaning they are specific cases of association. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

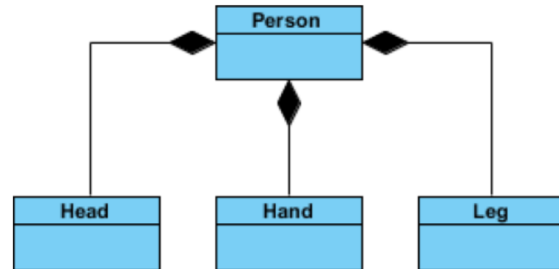
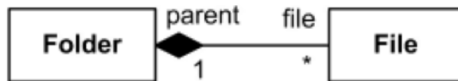
Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

# Composition

Composite aggregation is a subtype of aggregation relation with characteristics as:

- It is a two-way association between the objects.
- It is a whole/part relationship.
- If a composite is deleted, all other parts associated with it are deleted.

Composite aggregation is described as a binary association decorated with a filled black diamond at the aggregate (whole) end.

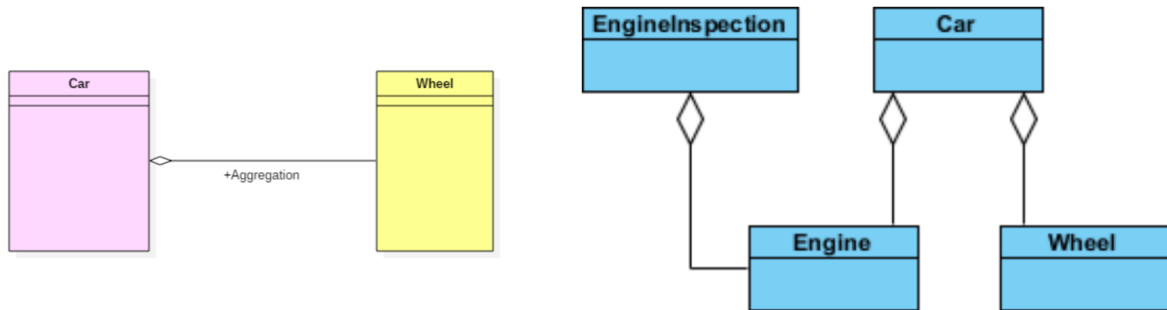


# Aggregation

An aggregation is a subtype of an association relationship in UML.

Aggregation and composition are both the types of association relationship in UML. An aggregation relationship can be described in simple words as “an object of one class can own or access the objects of another class.”

In an aggregation relationship, the dependent object remains in the scope of a relationship even when the source object is destroyed.



# Application: Printing an Invoice

---

## Five-part program development process

1. Gather requirements
2. Use CRC cards to find classes, responsibilities, and collaborators
3. Use UML diagrams to record class relationships
4. Use javadoc to document method behavior
5. Implement your program



# Application: Printing an Invoice – Requirements

---

- Start the development process by gathering and documenting program requirements.
- Task: Print out an invoice
- Invoice: Describes the charges for a set of products in certain quantities.
- Omit complexities
  - Dates, taxes, and invoice and customer numbers
- Print invoice
  - Billing address, all line items, amount due
- Line item
  - Description, unit price, quantity ordered, total price
- Test program: Adds line items to the invoice and then prints it.

# Application: Printing an Invoice

- Sample Invoice

I N V O I C E			
Sam's Small Appliances 100 Main Street Anytown, CA 98765			
Description	Price	Qty	Total
Toaster	29.95	3	89.85
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98
AMOUNT DUE: \$154.78			

- An invoice lists the charges for each item and the amount due.



© Scott Cramer/iStockphoto.

# Application: Printing an Invoice – CRC Cards

---

- Use CRC cards to find classes, responsibilities, and collaborators.
- Discover classes
- Nouns are possible classes:

```
Invoice  
Address  
LineItem  
Product  
Description  
Price  
Quantity  
Total  
Amount Due
```

# Application: Printing an Invoice – CRC Cards

---

- Analyze classes:

```
Invoice
Address
LineItem    // Records the product and the quantity
Product
Description // Field of the Product class
Price       // Field of the Product class
Quantity    // Not an attribute of a Product
Total       // Computed – not stored anywhere
Amount Due  // Computed – not stored anywhere
```

- Classes after a process of elimination:

```
Invoice
Address
LineItem
Product
```



# CRC Cards for Printing Invoice

---

Add collaborators to Invoice card:

Invoice	
<i>format the invoice</i>	Address
	LineItem

[illegible]

# CRC Cards for Printing Invoice

---

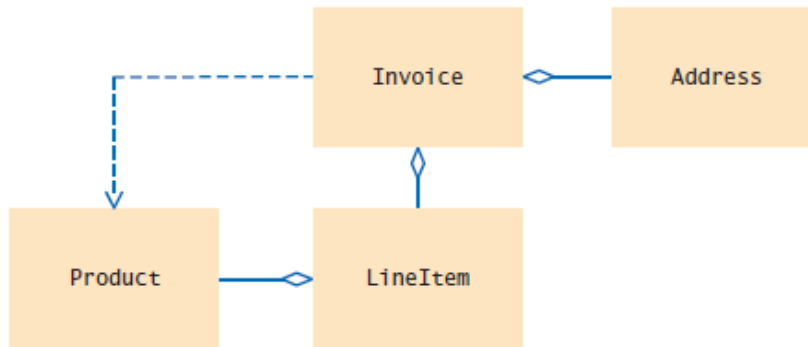
Invoice must be populated with products and quantities:

Invoice	
<i>format the invoice</i>	Address
<i>add a product and quantity</i>	LineItem
	Product



# Application: Printing an Invoice – UML Diagrams

---



**Figure 7** The Relationships Between the Invoice Classes

# Printing an Invoice – Method Documentation

---

- Use javadoc comments (with the method bodies left blank) to record the behavior of the classes.
- Write a Java source file for each class:
  - Write the method comments for those methods that you have discovered,
  - Leave the body of the methods blank
- Run javadoc to obtain formatted version of documentation in HTML format.
- Advantages:
  - Share HTML documentation with other team members
  - Format is immediately useful: Java source files
  - Supply the comments of the key methods

# Method Documentation – Invoice Class

---

```
/**
 * Describes an invoice for a set of purchased products.
 */
public class Invoice
{
    /**
     * Adds a charge for a product to this invoice.
     * @param aProduct the product that the customer ordered
     * @param quantity the quantity of the product
     */
    public void add(Product aProduct, int quantity)
    {
    }

    /**
     * Formats the invoice.
     * @return the formatted invoice
     */
    public String format()
    {
    }
}
```

# Method Documentation – LineItem Class

---

```
/**
 * Describes a quantity of an article to purchase and its price.
 */
public class LineItem
{
    /**
     * Computes the total cost of this line item.
     * @return the total price
     */
    public double getTotalPrice()
    {
    }
    /**
     * Formats this item.
     * @return a formatted string of this line item
     */
    public String format()
    {
    }
}
```

# Method Documentation – Product Class

---

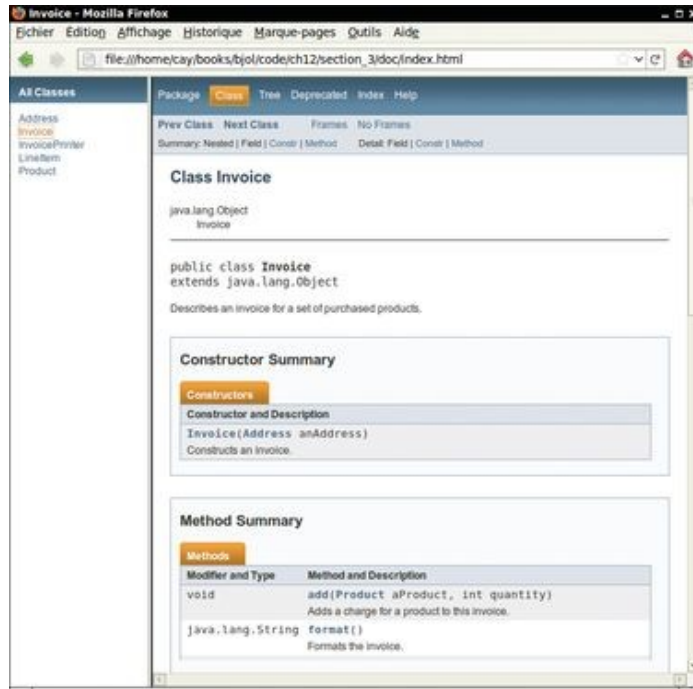
```
/**
 * Describes a product with a description and a price.
 */
public class Product
{
    /**
     * Gets the product description.
     * @return the description
     */
    public String getDescription()
    {
    }
    /**
     * Gets the product price.
     * @return the unit price
     */
    public double getPrice()
    {
    }
}
```

# Method Documentation – Address Class

---

```
/**
    Describes a mailing address.
 */
public class Address
{
    /**
        Formats the address.
        @return the address as a string with three lines
    */
    public String format()
    {
    }
}
```

# The Class Documentation in the HTML Format



**Figure 8** Class Documentation in HTML Format

# Printing an Invoice – Implementation

---

- After completing the design, implement your classes.
- The UML diagram will give instance variables:

Look for aggregated classes

They yield instance variables



# Implementation

---

- Invoice aggregates Address and LineItem.
- Every invoice has one billing address.
- An invoice can have many line items:

```
public class Invoice
{
    . . .
    private Address billingAddress;
    private ArrayList<LineItem> items;
}
```

# Implementation

---

A line item needs to store a Product object and quantity:

```
public class LineItem
{
    . . .
    private int quantity;
    private Product theProduct;
}
```

# Implementation

---

- The methods themselves are now very easy.
- Example:

`getTotalPrice` of `LineItem` gets the unit price of the product and multiplies it with the quantity

```
/**
 * Computes the total cost of this line item.
 * @return the total price
 */
public double getTotalPrice()
{
    return theProduct.getPrice() * quantity;
}
```

- Also supply constructors