Software Tools & Methods Class 4

Lecturer: Greg Bolcer, greg@bolcer.org

Summer Session 2009

ELH 110 9am-11:50am

Overview

- HW3 due today; HW3 Lab this afternoon
- Reading for today was on UML, Larman
- Last Class
  - Revision Control
  - Programming Practices
- Up next
  - Programming Practices
  - UML
  - Associations
  - Design Patterns

Software Methods & Tools

PROGRAMMING PRACTICES

Program Intently and Expressively

- The PIE Principle
  - “Code you write must clearly communicate your intent and must be expressive. By doing so, your code will be readable and understandable. Since your code is not confusing you will also avoid some potential errors. Program Intently and Expressively.”
- Choose readability over convenience
- Clearly communicate your intent
  - int result = val <<1; //More difficult to understand.
  - int result = val * 2; //Much clearer – Pick this.
- Examples
  - Avoid magic numbers
  - Avoid unnecessary optimization

Avoid Magic Numbers

- Numerical constants (literals) should not be coded directly
  - Prevent the number from being changed inconsistently in different places, e.g. in previous example WEIGHT can be used in many places. If it is hard-coded and its value need to be changed, the change can be inconsistent.
  - Easier to understand, e.g. "OBJECT_HEIGHT * width * length" is more readable than "4 * width * length"

Compare these two snippets

```
int volume(int i, int j) {
int vol;
vol = i * j * k;
return vol;
}
```

- Only know that the function multiplies i, j, and k
- But for what purpose?
- And where did k come from?

```
int calculateVolume(int width, int
                              length) {
    int volume = 0;
    volume = OBJECT_HEIGHT
              * width * length;
    return volume;
}
```

- Instantly know that it calculate volume of a rectangular solid object with a fixed height

Code Conventions

This code is correctly indented, but ugly and hard to read. It also can go very far to the right if there are many tests

```
if (score < 35) {
    g.setColor(Color.MAGENTA);
} else {
    if (score < 50) {
        g.setColor(Color.RED);
    } else {
        if (score < 60) {
            g.setColor(Color.ORANGE);
        } else  {
            if (score < 80) {
                g.setColor(Color.YELLOW);
            } else {
                g.setColor(Color.GREEN);
            }
        }
    }
}
```

Here is the same example, using a style of writing the if immediately after the else. This is a common exception to the indenting rules, because it results in more readable programs. Note that it makes use of the rule that a single statement in one of the Java clauses doesn't need braces.

```
if (score < 35) {
    g.setColor(Color.MAGENTA);
} else if (score < 50) {
    g.setColor(Color.RED);
} else if (score < 60) {
    g.setColor(Color.ORANGE);
} else if (score < 80) {
    g.setColor(Color.YELLOW);
} else {
    g.setColor(Color.GREEN);
}
```

Code Conventions

```
package Shopping;
public class Product {

	private int store_keeping_unit; private float price;
    do not define variable on the same line
	private product_description description;

	/*
		Product(int, float) is the constructor for the Product class.
		It takes an int and a float as arguments.
	*/
	public Product(int keeping_unit,  float price) {
		store_keeping_unit = keeping_unit;
		price = price;
		description = null;
	}

	public int getStoreKeepingUnit() {
		return store_keeping_unit;
	}

	public float ItemPriceWithTax() {
		float tax = (float) (price * 0.078);
		return price + tax;
	}
}
```

Whiteboard exercise: There's 5 things that break Java code conventions in this code snippet.

Code Conventions

```
package Shopping;
public class Product {

        private int store_keeping_unit; private float price;

        private product_description description;

        /*
        Product(int, float) is the constructor for the Product class.
        It takes an int and a float as arguments.
        */
        public Product(int keeping_unit,  float price) {
                store_keeping_unit = keeping_unit;
                price = price;
                description = null;
        }

        public int getStoreKeepingUnit() {
                return store_keeping_unit;
        }

        public float ItemPriceWithTax() {
                float tax = (float) (price * 0.078);
                return price + tax;
        }
}
```

- Java packages typically lower-case, shopping
- Two declarations of different types on single line
- Capitalization of variables, Java uses leadingCaps
- Comments repetitive, unhelpful
- Method names typically start with lowercase
- Method names should be verbs
- No magic numbers

Software Tools & Methods

UNIFIED MODELING LANGUAGE

Unified Modeling Language

- Let’s look at each of the words in the name
- Unified
  - Two important methodologists Rumbaugh and Booch decided to merge their approaches in 1994.
    - They worked together at the Rational Software Corporation
  - In 1995, another methodologist, Jacobson, joined the team
    - His work focused on use cases
  - In 1997 the Object Management Group (OMG) started the process of UML standardization

Models

- Models are abstract representations
  - Contain essential characteristics and omit non-essential details
    - "Essential" depends on the problem domain
  - There are no perfect representations
- Models can be representations of the world
  - Domain models
  - Requirements
- Models can be representations of software
  - Specifications
  - Design
  - Systems

Why make models?

- Systems are complex and hard to understand
  - The world, organizations, relationships, software
- Models can make certain aspects more clearly visible than in the real system

- What can you do with models?
  - Express your ideas and communicate with other engineers
  - Reason about the system: detect errors, predict qualities
  - Generate parts of the real system: code, schemas
  - Reverse engineer the real system to make a model

Brooks on Invisibility of Software

“Software is invisible and unvisualizable. Geometric abstractions are powerful tools.”

“As soon as we attempt to diagram software structure, we find it to constitute not one, but several general directed graphs, superimposed on upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These are usually not even planar, much less hierarchical. Indeed, one of the ways of establishing conceptual control over such structure is to enforce link cutting until one or more of the graphs becomes hierarchical.”

What constitutes a good model?

- A model should…
  - Provide abstraction
  - Render the problem in a format amenable to reasoning
    - use a standard notation
    - be understandable by clients and users
    - lead software engineers to have insights about the system
    - make the problem solvable computationally
  - Be good enough

Remember: It’s only a model

- There will always be:
  - Phenomena in the application domain that are not in the model
  - Details in the application that are not in the model

- A model is never perfect
  - “If the map and the terrain disagree, believe the terrain”
  - Perfecting the model is not always a good use of your time...

Modeling Languages

- Natural language
  - Extremely expressive and flexible
  - Very poor at capturing the semantics of the model
  - Better used for elicitation, and to annotate models for communication
- Semi-formal notation
  - Captures structure and some semantics
  - Can perform (some) reasoning, consistency checking, animation, etc.
    - Examples: diagrams, tables, structured English, etc.
  - Mostly visual - for rapid communication with a variety of stakeholders
- Formal notation
  - very precise semantics, extensive reasoning possible
  - Every detailed models (may be more detailed than we need)

Visual Languages

- Words = symbols
- Syntax = rules for combining symbols, drawing and layout of language
  - Example: Sheet music, tic-tac-toe
  - Example: Visual Basic is a visual programming language

UML

- UML is a semi-formal visual modeling language
  - Semantics are not completely specified by standard
  - It has *extension* mechanisms
  - It has an associated textual language
    - *Object Constraint Language* (OCL)
  - Well suited for object-oriented designs

Types of UML Diagrams

Structure

- Class diagrams
- Object diagram
- Package diagram
- Composite structure diagram
- Component diagram
- Deployment Diagram

Behavior

- Activity diagram
- Use case diagram
- State machine diagram
- Interaction diagrams
  - Sequence diagram
  - Communication diagram
  - Interaction overview diagram
  - Timing diagram

Types of UML Diagrams

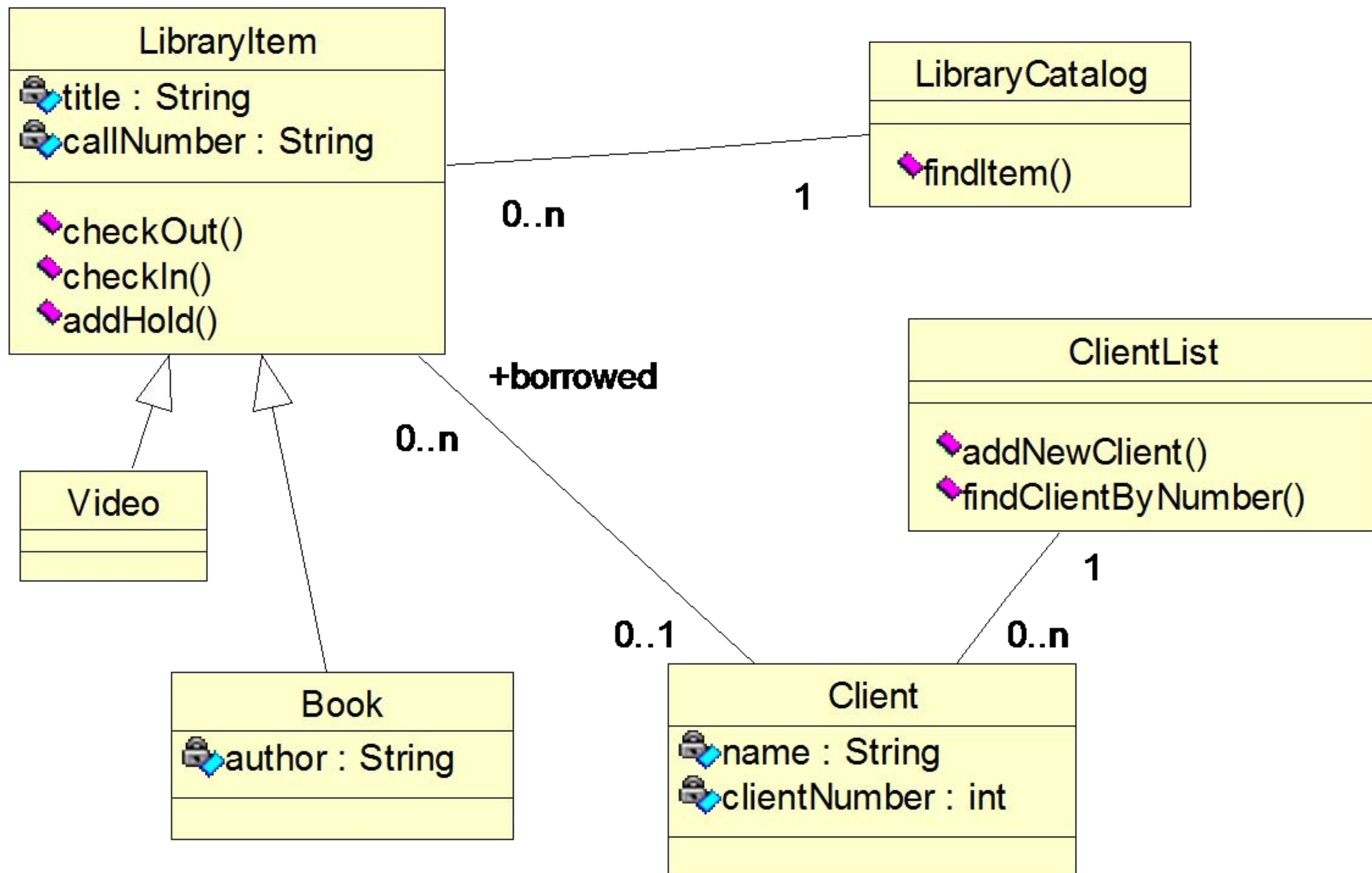
Structure

- Class diagram
- Object diagram
- Package diagram
- Composite structure diagram
- Component diagram
- Deployment diagram

Behavior

- Activity diagram
- Use case diagram
- State machine diagram
- Interaction diagrams
  - Sequence diagram
  - Communication diagram
  - Interaction overview diagram
  - Timing diagram

Class Diagram



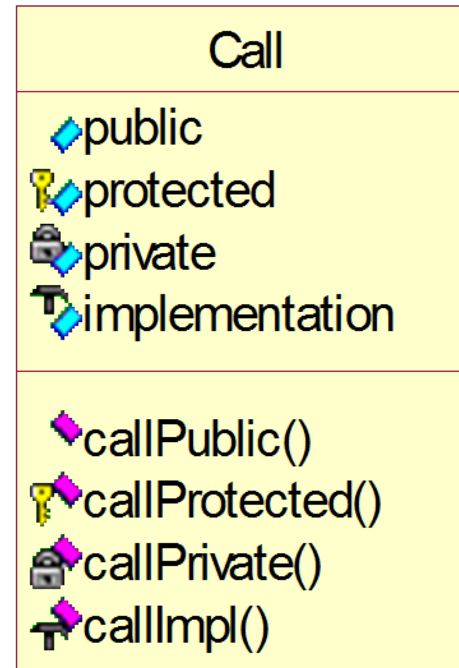
- A UML class corresponds to a Java class.

Different Uses for Class Diagrams

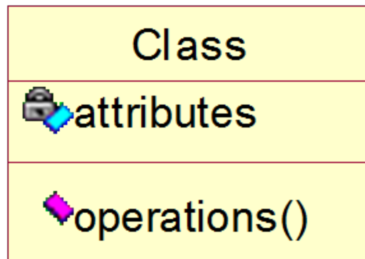
- Domain
  - Things and relationships out in the world
- Specification
  - High-level design
- Implementation
  - Prescriptions for code

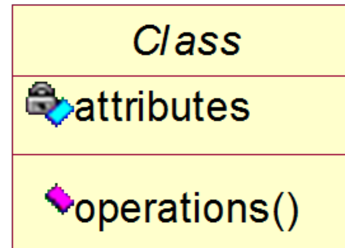
Classes, Attributes, and Operations

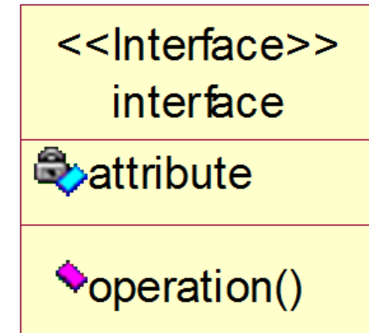
- Public (nothing or +)
- Protected (#)
- Private (-)



Class Diagrams







- Name
  - `Name:type`
- Attributes
  - `visibility name: type multiplicity = default {property-string}`
- Operations
  - `visibility name (parameter-list) : return-type {property-string}`
  - `direction name: type = default`

Attribute Syntax

```
visibility name: type multiplicity = default {property-string}
```

- optional visibility: + public, - private, # protected
- name: the name of this attribute
- optional type: data type of this attribute
- optional default: initial value of attribute
- optional property string: OCL, e.g. ordered, readonly

- Examples:

  firstName

  -middleName

  lastName: String

  -age: int = 0

  birthSign: String = "Gemini"

Property String

- Any information that cannot be expressed in the diagram notation can be included as text
  - Example: constraints
- Example:

  customerNumber: int { >=0 }
- All diagram elements can be annotated with constraints
- Can be:
  - Natural language text
  - Object Constraint Language
  - Predefined properties
    - Examples on next two slides
  - Any other text

Property Strings on Attributes

- changeable (default)
  - Value of attribute can be changed
  - May want to list legal/possible values
- addOnly
  - Can add possible values, but can't change existing ones
- frozen
  - Can't add or change

Operation Syntax

```
visibility name (parameter-list) : return-type {property-
   string}
```

- optional visibility: + public, - private, # protected
- name: the name of this operation
- optional parameters-list: parameters to this operation

```
direction name: type = default
```

  - optional direction: in, out, inout
  - name
  - optional type
  - optional default
- optional type: return type of operation
- optional property string

Operation Syntax

- Examples:

  getFirstName()

  +getMiddleName(): String

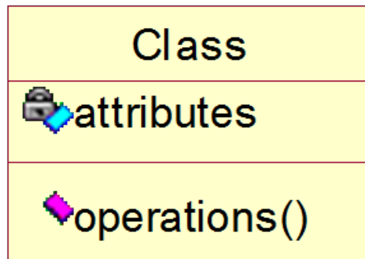
  -setLastName(name: String)

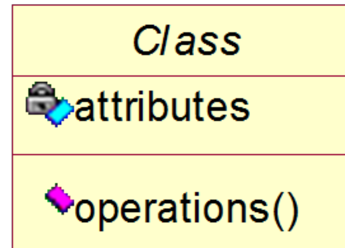
  +paintPortrait(inout c: Canvas, subject: Person)

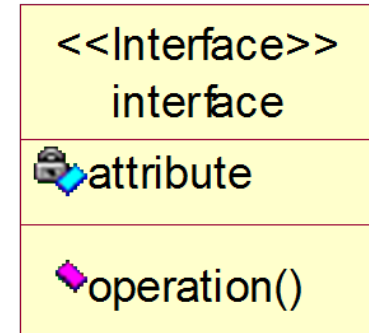
Property Strings on Operations

- sequential
  - Only one call to a method within an instance
- concurrent
  - Multiple simultaneous calls to a method within an instance may occur
- guarded
  - Multiple simultaneous calls to a method within an instance may occur but only one at a time will be executed
- isQuery
  - Operation doesn’t change the value of any attributes

Class Diagrams







- Name
  - `Name [:type]`
- Attributes
  - `[visibility] name [multiplicity] [:type] [=default] [{property-string}]`
- Operations
  - `[visibility] name [(parameter-list)] [:return-type] [{property-string}]`
  - `[direction] name [:type] [=default]`

Multiplicities

- Descriptive
  - Optional (0 or more)
  - Mandatory (at least 1)
  - Single-valued (upper bound 1)
  - Multi-valued (upper bound of >1, usually *)

- Symbolic
  - 1 (or another number, exactly the number specified)
  - 0..1 (zero or one, i.e. optional)
  - 2..4 (range)
  - * (zero or more, no upper limit; n in Rose)

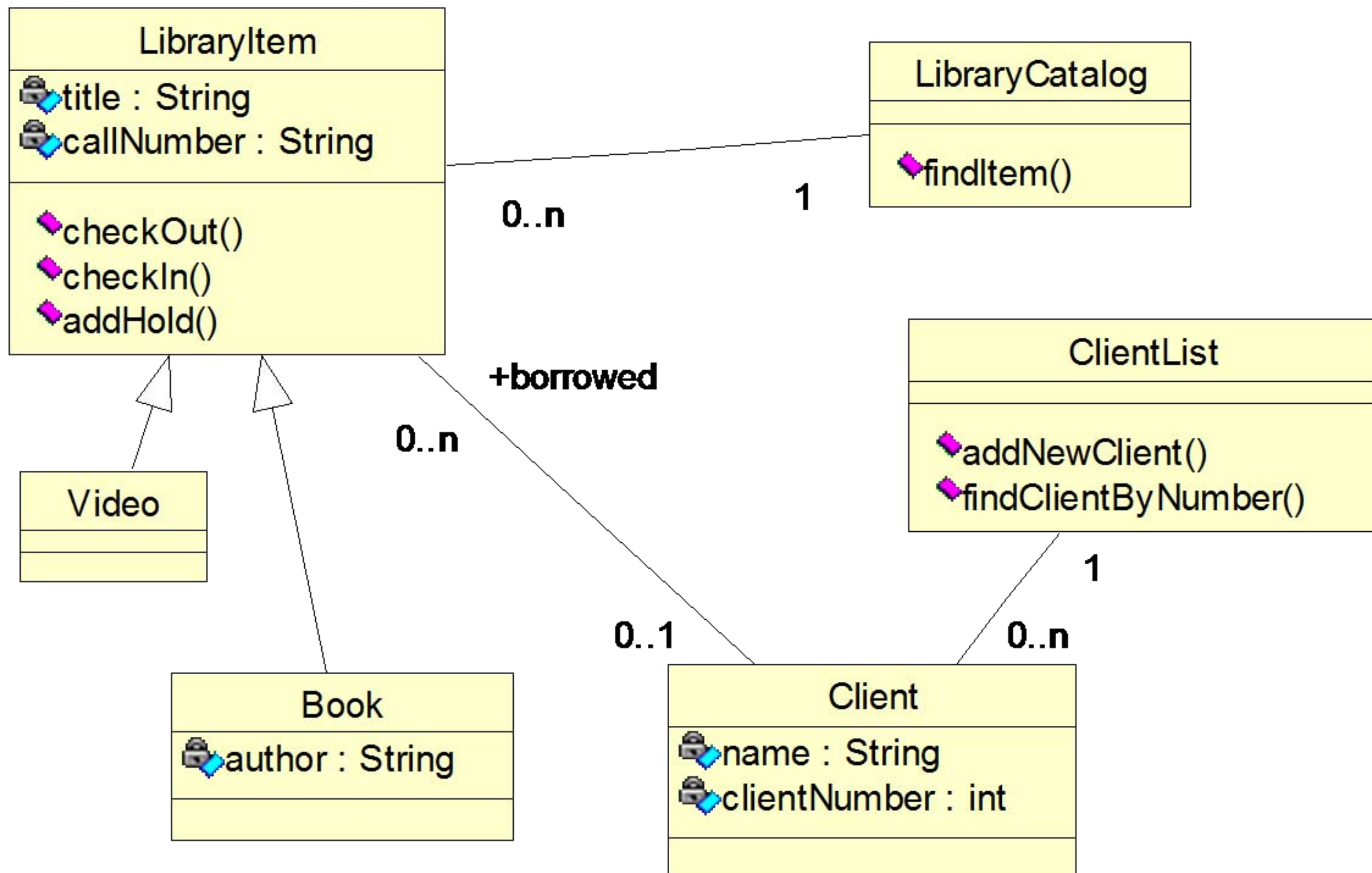
Inheritance

- Generalization correspond to the Java keyword "extends"
  - Generalizations are drawn with a solid line and a white triangular arrow touching the superclass.
- Realization correspond to the Java keyword "implements"
  - Realizations are drawn with a dashed line and a white triangular arrow touching the interface.
- UML itself is not restricted to single inheritance. However, you would not use multiple inheritance if you plan to implement in Java.

Inheritance

- It is common practice to arrange the diagram so that:
  - Generalization and Realization arrows point upward
  - If one class has many subclasses, the Generalization arrows overlap

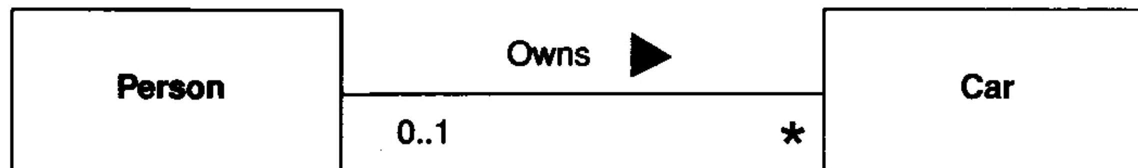
Class Diagram



Associations

- Relations between classes
- Roles
  - analogous to names of instance variables
- Multiplicities
  - 0, 1, *, 0..1, 1..*, 5..6, and so on
  - says how many objects each object knows
  - would be realized through arrays, Sets, Lists, and so on
- Navigability
  - bidirectional: each class references the other
  - unidirectional: A knows B, but B doesn't know A
  - no arrow heads: means either "bidirectional" or "not specified

Association Name vs. Role Names

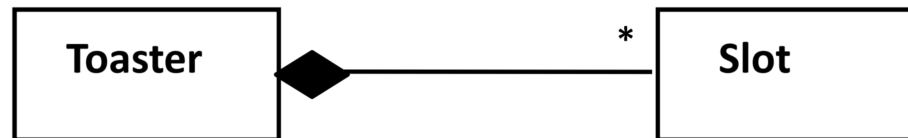


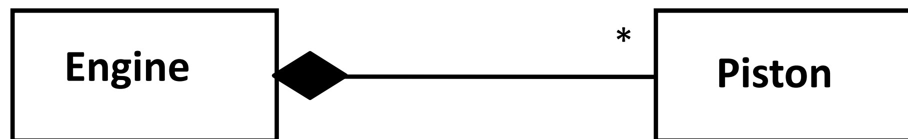
Types of Associations

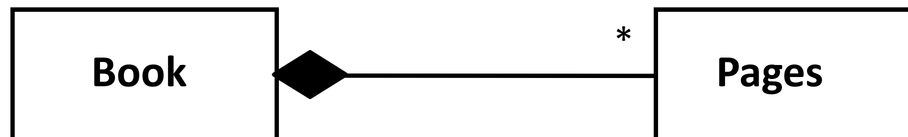
- Composition = Black diamond: parts are created with the whole and stay with exactly one whole until both are destroyed together.
- Aggregation = White diamond: parts can join the whole and later leave; one part could be part of more than one whole.
  - Some operations will be recursively applied to the parts of a whole

Association Type: Composition

- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



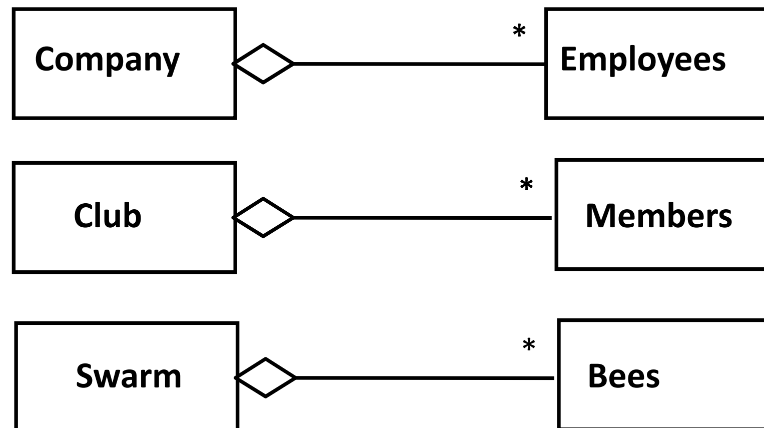




Association Type: Aggregation

Aggregations are special associations that represent 'part-whole' relationships.

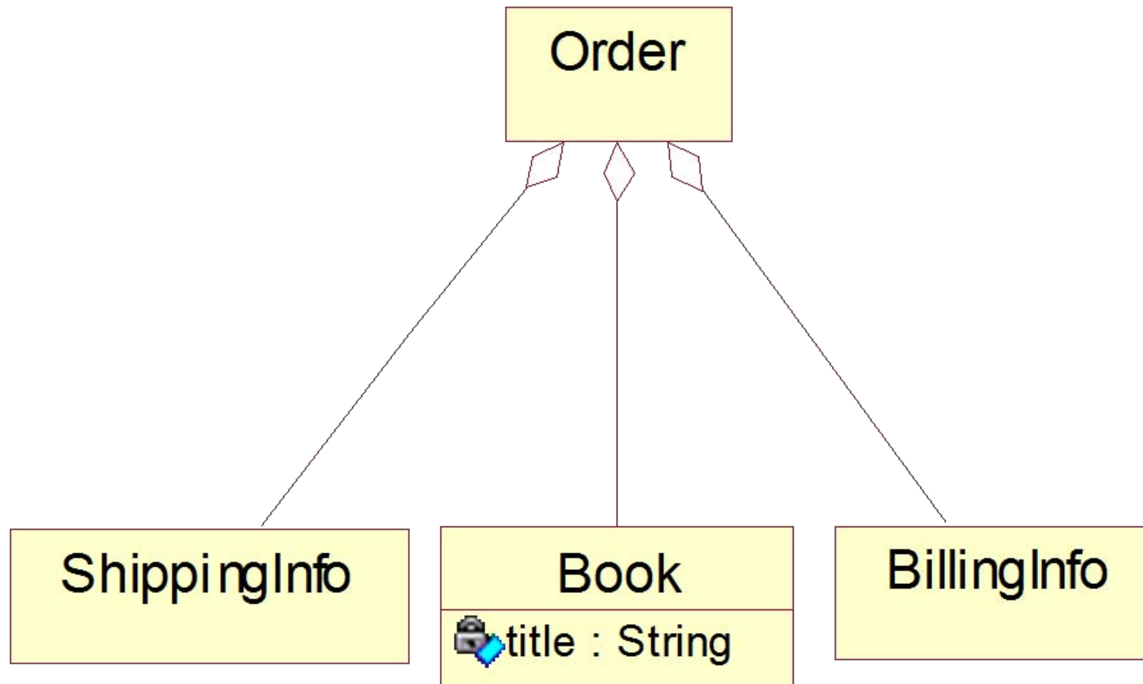
- The 'whole' side is often called the *assembly* or the *aggregate*
- This symbol is a shorthand notation association named `isPartOf`



When to use an aggregation

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts
- Use with care

Example Aggregation



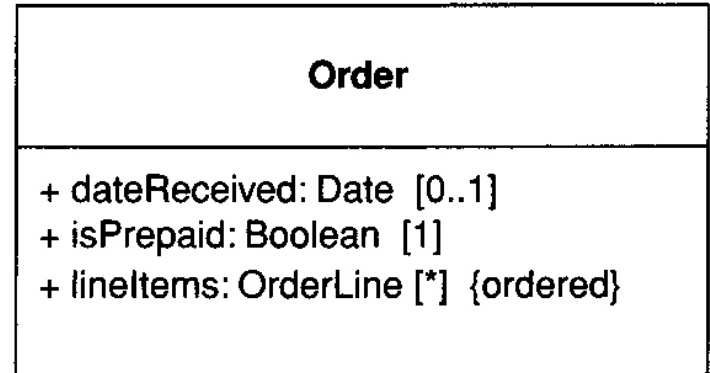
Some Mnemonics

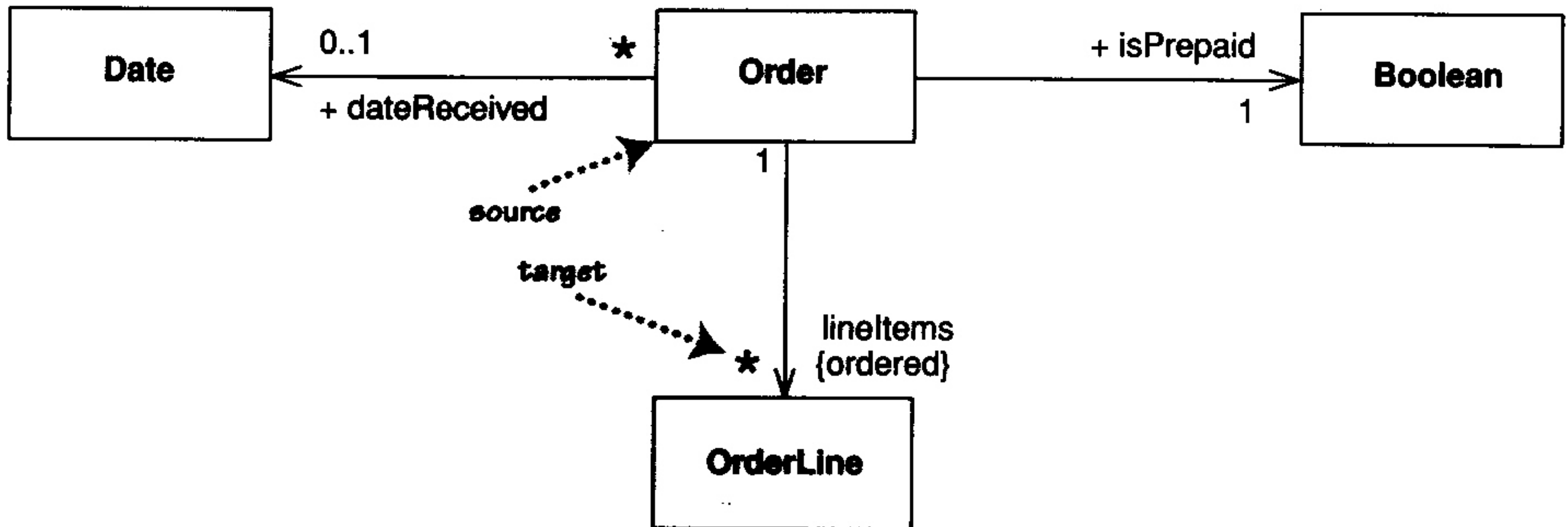
- Generalization = is-a
- Composition = has-a
- Aggregation = part-of

- Examples:
  - Toaster is-a Appliance
  - Toaster has-a slot
  - Member part-of club

Associations and Properties

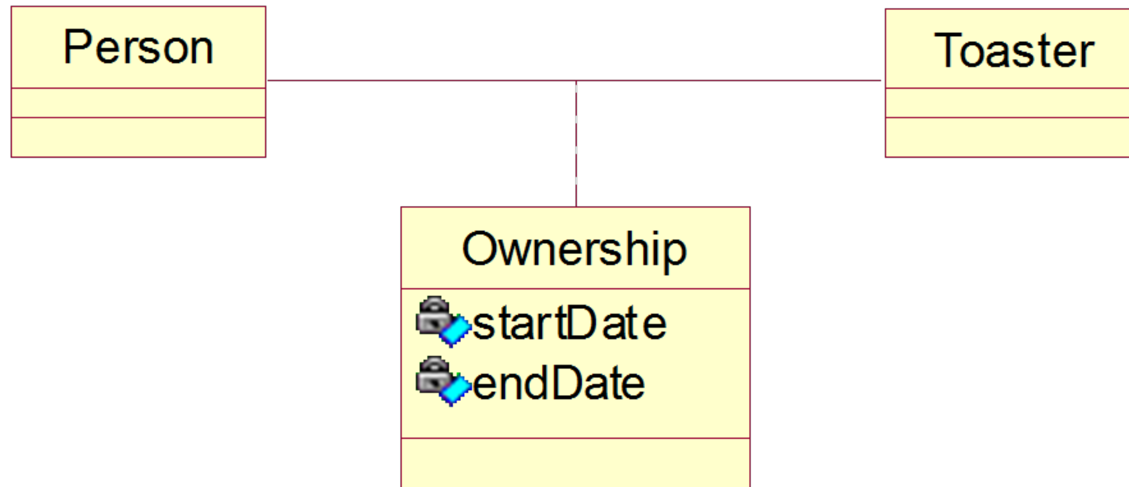
- Associations are another way of representing properties

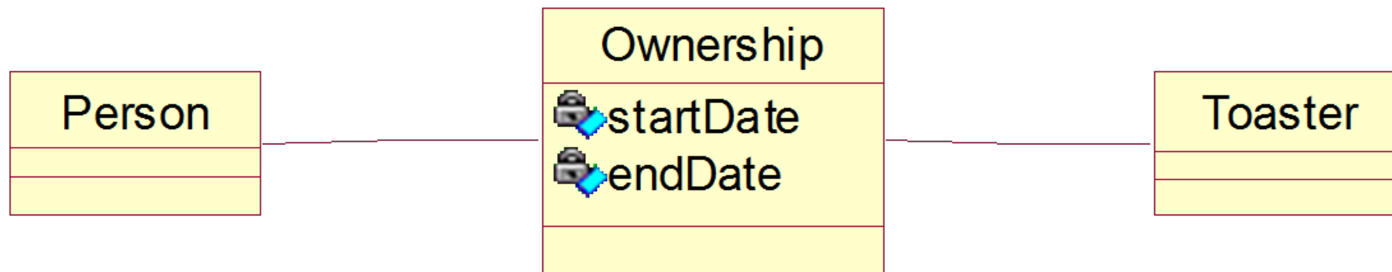




Association Classes

- When you want to add properties or operations to an association, use an association class.
  - Frequently simpler to promote associate class to full class



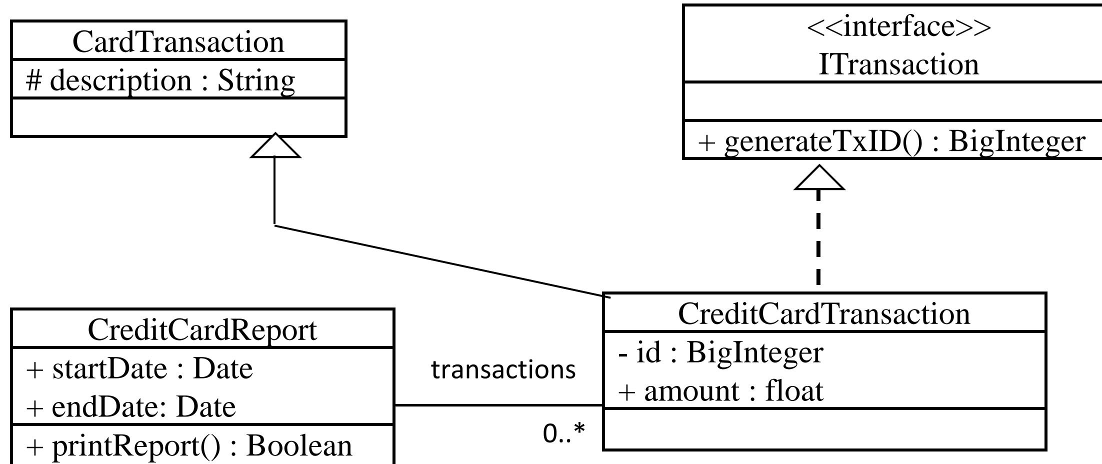


Software Tools & Methods

UML EXAMPLE

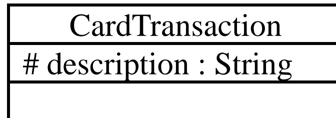
UML Class Diagram

How would you represent this in Java?



UML Class Diagram

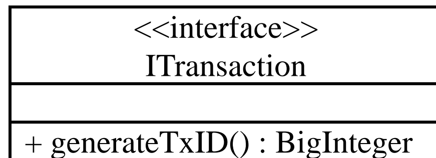
- Class CardTransaction
- Protected description




```
public class CardTransaction {
    protected String
    description;
}
```

UML Class Diagram

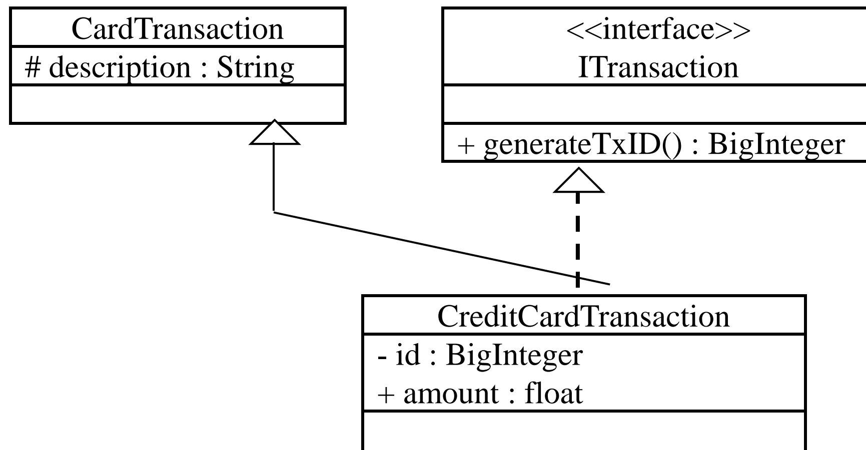
- Public interface ITransaction
- Method generateTxID returns a BigInteger




```
public interface ITransaction {
    public BigInteger
    generateTxID();
}
```

UML Class Diagram

- Public class Credit Card Transaction
  - Implements ITransaction
  - Extends Card Transaction




```
public class CreditCardTransaction
    extends CardTransaction implements
    ITransaction {
    private BigInteger id;
    public float amount;

//This method is needed
//implement all methods in the interface
public BigInteger generateTxID(){
     return new BigInteger(something);
}

} //CreditCardTransaction
```

UML Class Diagram

- Public class Credit Card Report has a Zero to many associations with Credit Card Transactions

```
public class CreditCardReport {
    public Date startDate;
    public Date endDate;
    public
ArrayList<CreditCardTransactions> transactions
    = new ArrayList<CreditCardTransaction>();
    public Boolean printReport();

}
```