Objects, UML, and Java

Abram Hindle
hindle1@ualberta.ca

Henry Tang
hktang@ualberta.ca

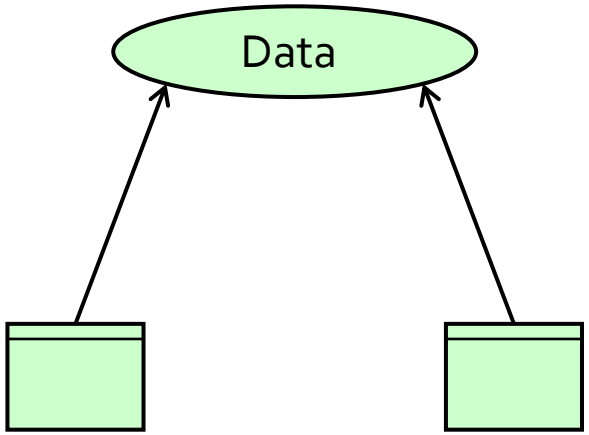
Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong

Modeling Principles

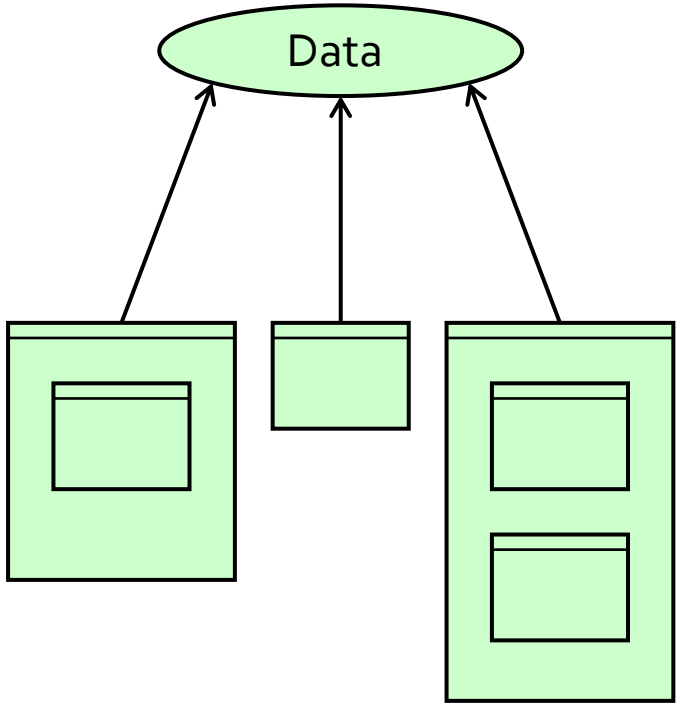
Language Evolution

- COBOL, Fortran:
  - Subprograms (subroutines) access global data
  - Break up system into subroutines



Language Evolution

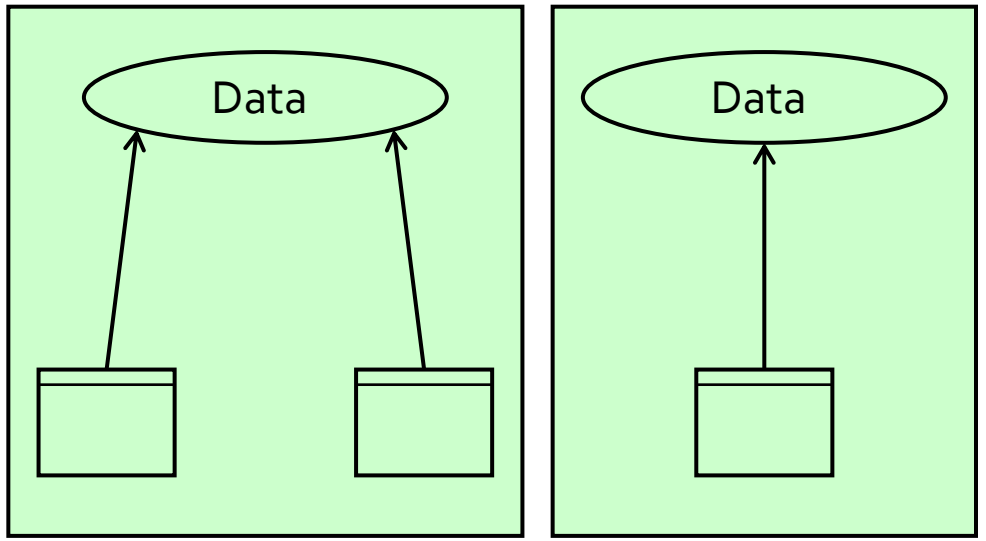
- Algol, Pascal:
  - (Nested) procedures with block structured scope
  - Break up system into nested procedures



Nested procedures

Language Evolution

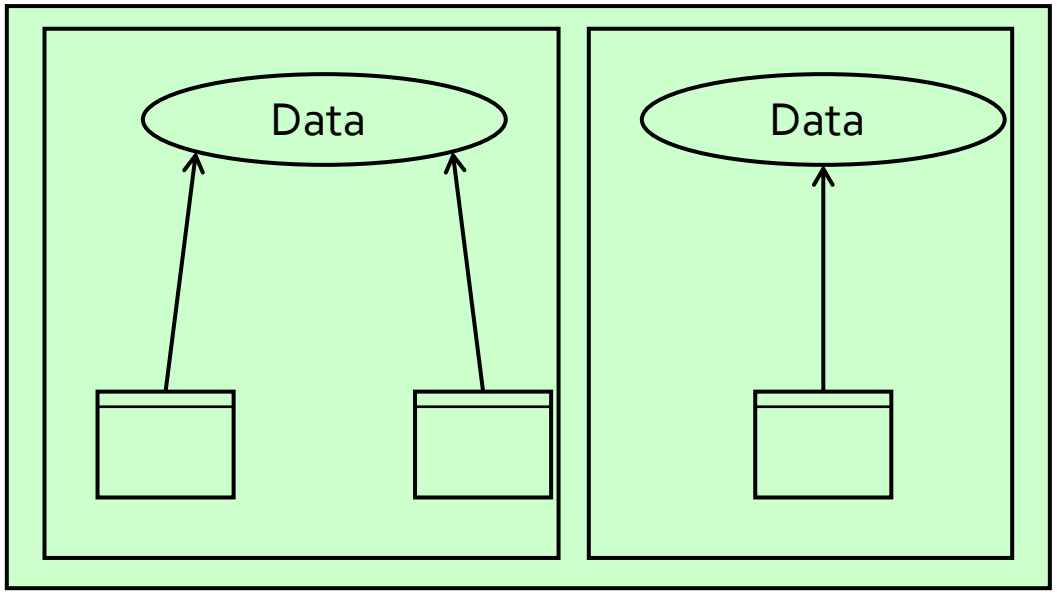
- Modula-2, C:
  - Modules (files) of related data and functions
  - Break up system into modules (e.g., abstract data types)



Modules

Language Evolution

- Smalltalk, C++, Java:
  - Classes with data and methods
  - Classes as “factories” for objects
  - Break up system into classes



Classes and packages

Discussion

- Question:
  - What software engineering design principles drove programming language evolution?
  - https://www.altexsoft.com/blog/pros-and-cons-of-java-programming/

Abstraction

- Simplifying to its essentials the description of a real-world entity or concept
  - Coping with complexity
  - “Selective ignorance”
- Modeling the problem space
  - E.g., a “Person” abstraction

Encapsulation

- Bundling data with access functions
  - Distinguishing “what” from “how”
  - “Need to know” restricted access
  - Maintaining integrity
- Information hiding criterion
  - Hide changeable internal details from the outside world, but reveal assumptions through interface
  - E.g., a “Person” abstract data type

Decomposition

- Dividing whole things into parts
  - Or composing whole things out of parts
  - “Separation of concerns”
- Data parts
  - Fixed or dynamic number
  - Sharing of parts
  - Lifetime of parts

Generalization

- From specific cases, looking for commonalities that can be factored out
  - Reusing common designs
  - Reducing redundant code
- Making systems flexible and extensible

Object-Oriented Models

Object-Oriented Models

- Implementing OO models:
  - OO programming languages
    - E.g., Java, C++
- Expressing OO models:
  - OO design notations
    - E.g., UML

Java

- Principal designer:
  - James Gosling, Sun Microsystems
- Language goals:
  - Simple, object-oriented
  - Robust, secure
  - Network and thread support
  - “Compile once, run anywhere”

Java

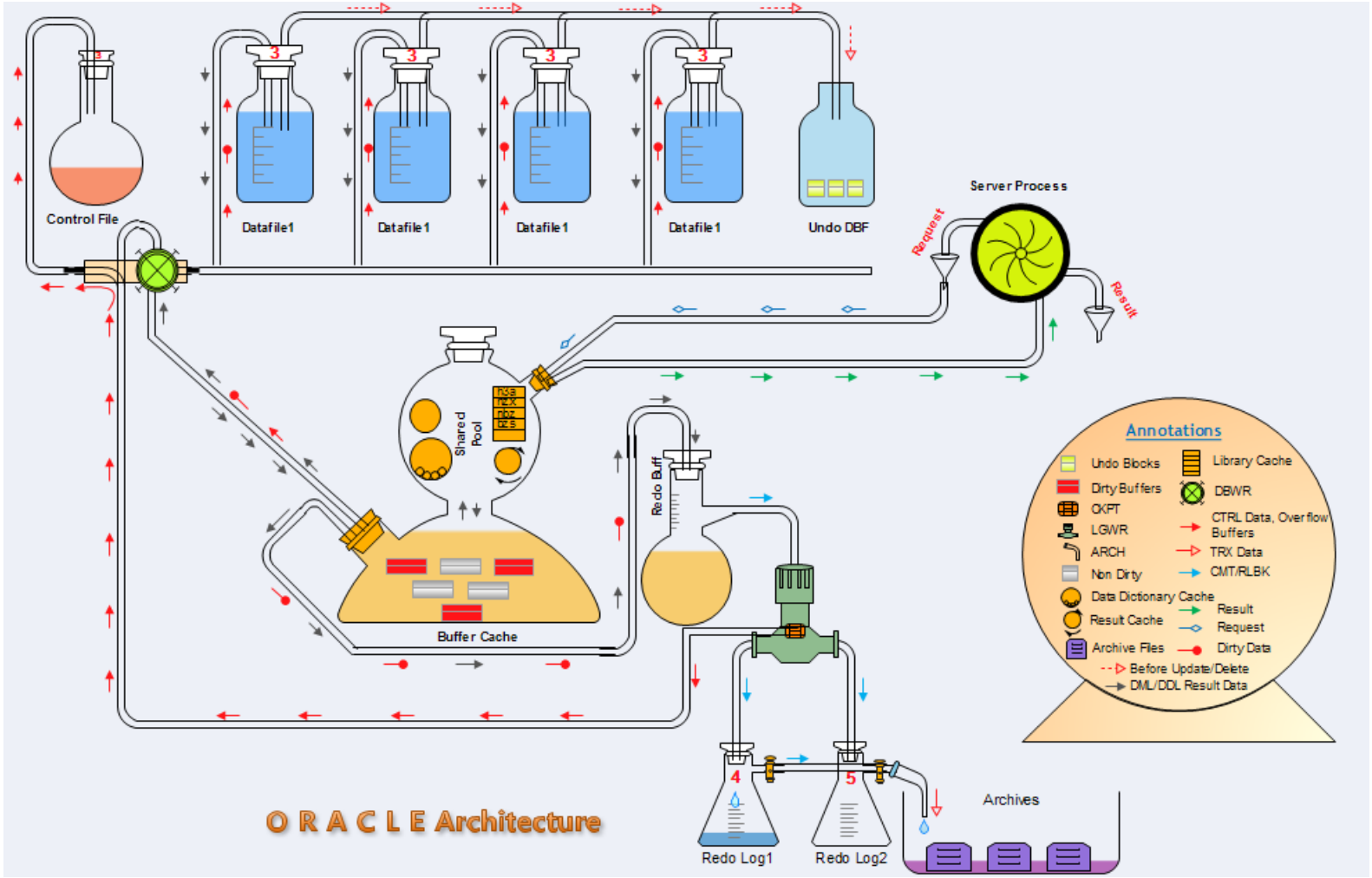
- Language design inspired by:

Language	Feature
Lisp	Garbage collection, reflection
Simula-67, C++	Classes
Algol-68	Overloading
Pascal, Modula-2	Strong type checking
C	Syntax
Ada	Exceptions
Objective C, Eiffel	Interfaces
Modula-3	Threads

Unified Modeling Language (UML)

- Principal designers:
  - Grady Booch, Ivar Jacobson, James Rumbaugh
- Language goals:
  - Express object-oriented designs visually
  - Programming language independent
  - Communicate, evaluate, and reuse designs
  - Make design intent more explicit
- Allows thought about design before coding

Why UML?


Control File
Datafile1
Datafile1
Datafile1
Datafile1
Undo DBF
Server Process
Request
Result
Shared Pool
Buffer Cache
Redo Buff
Redo Log1
Redo Log2
Archives
ORACLE Architecture
Annotations
Undo Blocks
Dirty Buffers
DBWR
OKPT
LGWR
ARCH
Non Dirty
Data Dictionary Cache
Result Cache
Archive Files
Library Cache
CTRL Data, Overflow Buffers
TRX Data
CMT/RLBK
Result
Request
Dirty Data
Before Update/Delete
DML/DDL Result Data

Abstraction

- Object:
  - An entity with specific attribute values (state), behavior, and identity
  - Typically instantiated from a class
- Class:
  - Associated type of an object
  - Defines attributes and methods

Java and UML Class

```
public class Frame { // version 0
    // represent a 'window'
    /* body of class definition goes here */
}
```

UML class notation

Encapsulation

- Class:
  - Access control for attributes and methods
    - E.g., public or private
  - Access is not the same as visibility
  - “Design by contract”
    - Public interface represents a contract between the developer who implements the class and the developer who uses the class

Java Class

```
• public class Frame { // version 1
      // private implementation

      private datatype variablename;

      // public interface

      public Frame( arguments ) {
          // implementation of constructor
      }

      public returntype methodname( arguments ) {
          // implementation of method
      }
  }
```

Java Class

```
public class Frame { // version 2
    private int x;
    private int y;
    …
    public Frame ( String name,
        int x, int y, int height, int width ) { … }

    public void resize(
        int newHeight, int newWidth ) { … }

    public void moveTo(
        int newX, int newY ) { … }

    public void drawText( String text,
        int x, int y ) { … }
}
```

```
Frame f = new Frame( "Untitled", 0, 0, 480, 640 );
```

UML Class

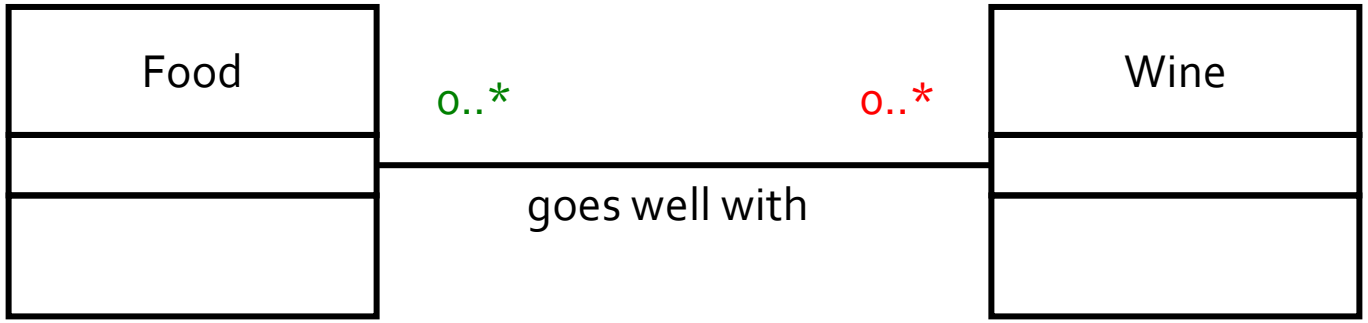
Frame
-x : Integer -y : Integer
+Frame(name: String, x: Integer, y: Integer, height: Integer, width: Integer) +resize(height: Integer, width: Integer): void +moveTo(x: Integer, y: Integer): void +drawText(text: String, x: Integer, y: Integer): void

- – Private
- \+ Public

Decomposition

- Association relationship:
  - "Some" relationship between classes
    - E.g., between Book and Patron

UML Association



- Read class diagram using "objects"
  - A Food *object* goes well with a Wine *object*
  - A Food *object* is associated with o or more Wine *objects*
  - A Wine *object* is associated with o or more Food *objects*

Decomposition

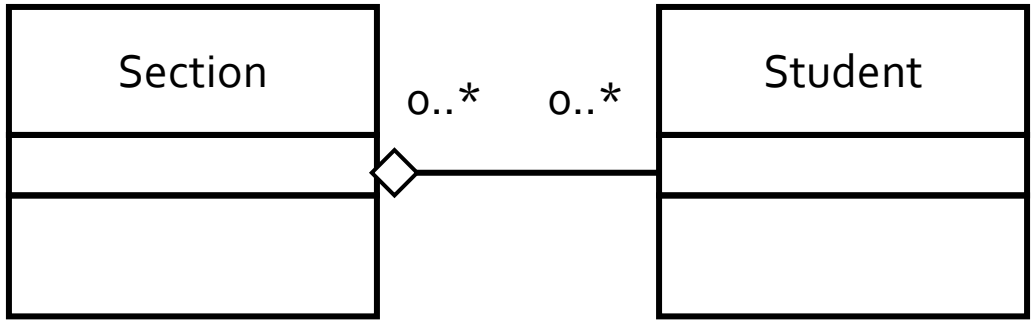
- Aggregation relationship:
  - Weak “has-a” relationship
  - Whole “has-a” part
  - A part may belong to (be shared with) other wholes
    - E.g., a Section and a Student

Java and UML Aggregation

- Dynamic number of aggregated objects:

```
public class Section {
    private ArrayList<Student> roster;
    …

    public Section() {
        roster = new ArrayList<Student>();
        …
    }
    public void add( Student s ) { … }
}
```



Java and UML Aggregation

- Fixed number of aggregated objects:

```
public class Frame {

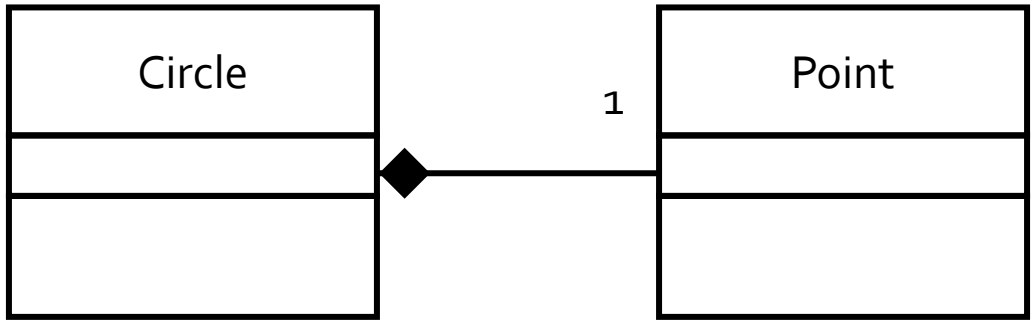
    private Location defaultLocation; // shared
    private Size defaultSize; // shared
    …
}
```

Decomposition

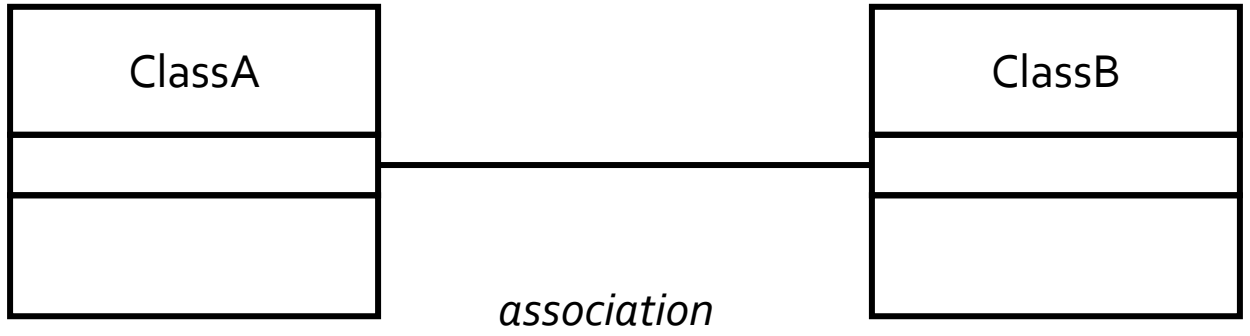
- Composition relationship:
  - Strong “has-a” relationship
  - Exclusive containment of parts
  - Related object lifetimes
    - The whole cannot exist without having the parts; if the whole is destroyed, the parts should also be destroyed
  - Often access the parts through the whole

UML Composition

- Contained *objects* are exclusive to the container

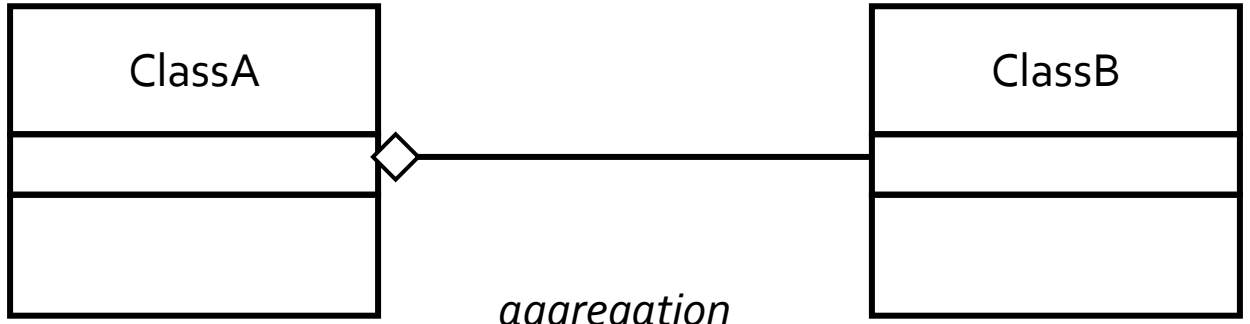


- A Circle object has a Point object that is exclusive to it (however, other objects may contain Point objects, just not this one)



association

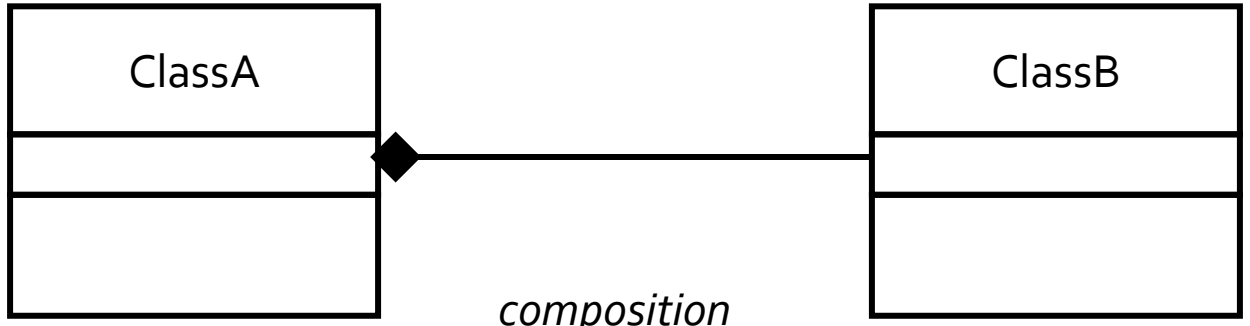
whole *part*



aggregation

ClassA and ClassB can created/destroyed independently

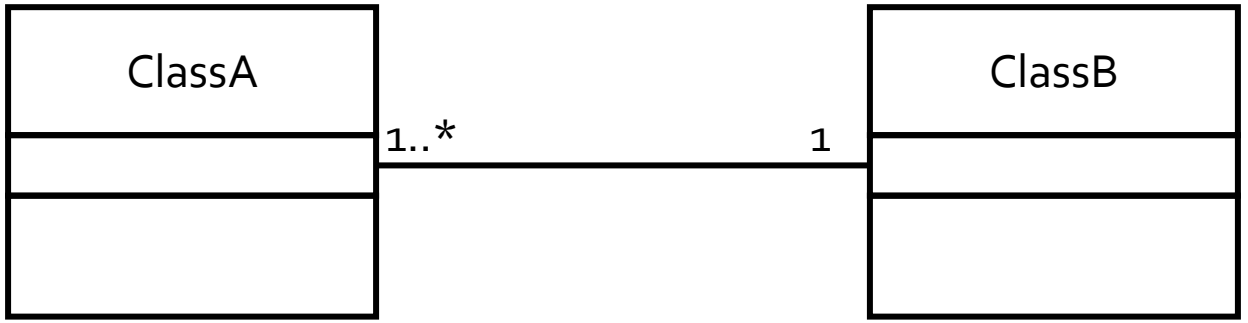
whole *part*



composition

If ClassA instance is deleted, all of its ClassB instances get deleted too

Navigability

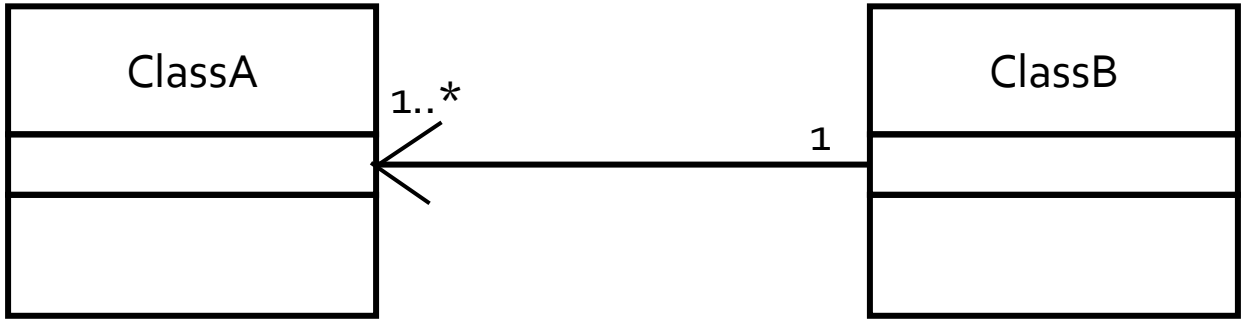


Missing navigability

Usually implies:

A instances have references to one B

B instances have references to one or more A

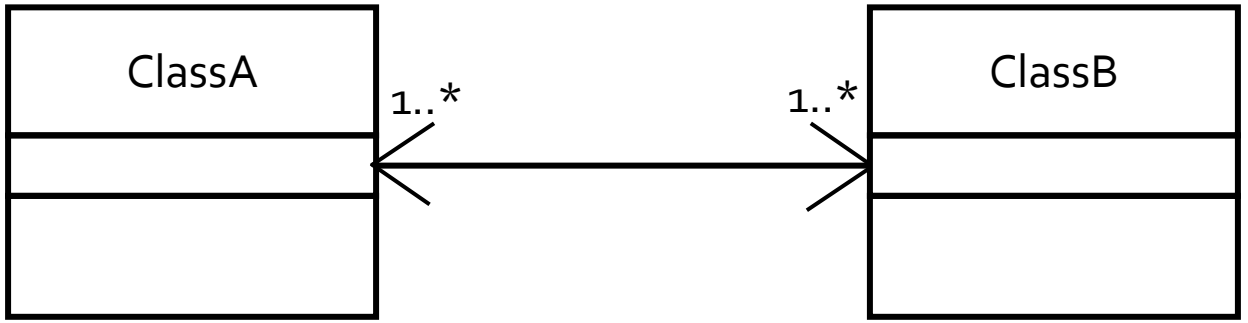


Navigability

← Arrow

B instances have references to one or more A

A instances DO NOT have reference(s) to B



Explicit two-way navigability

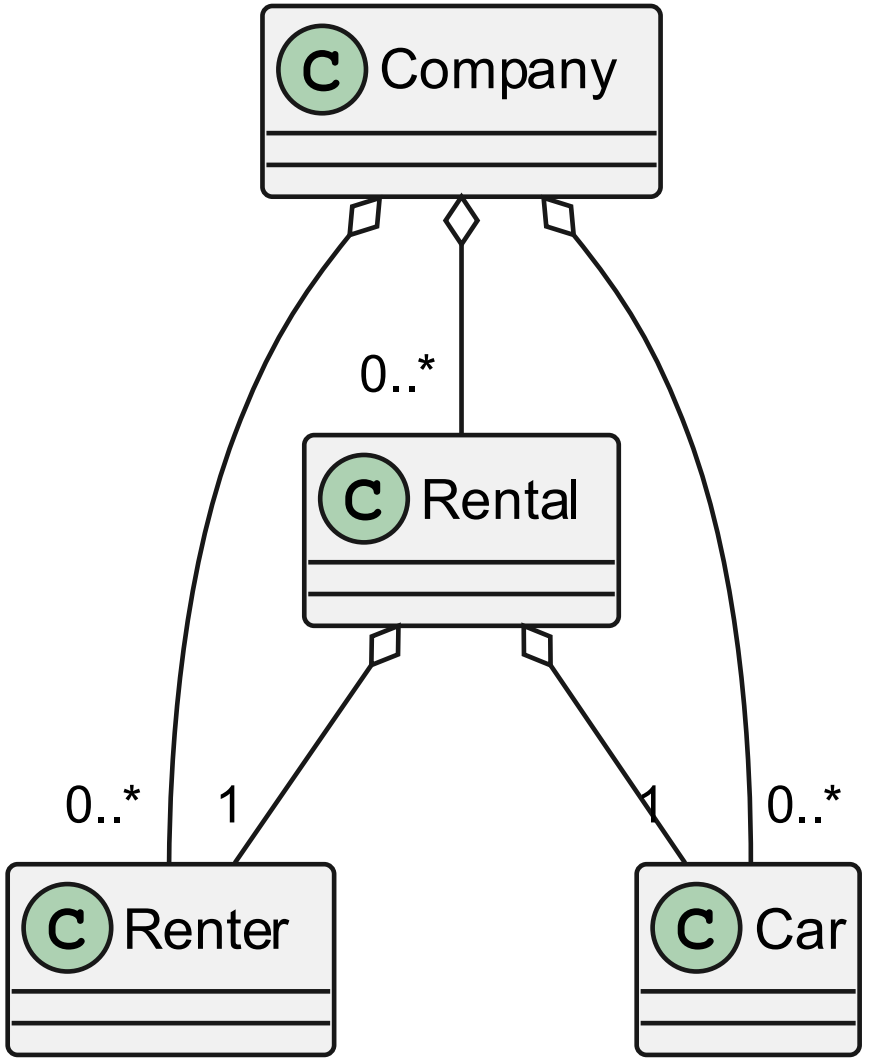
(rare)

A instances have references to one or more B

B instances have references to one or more A

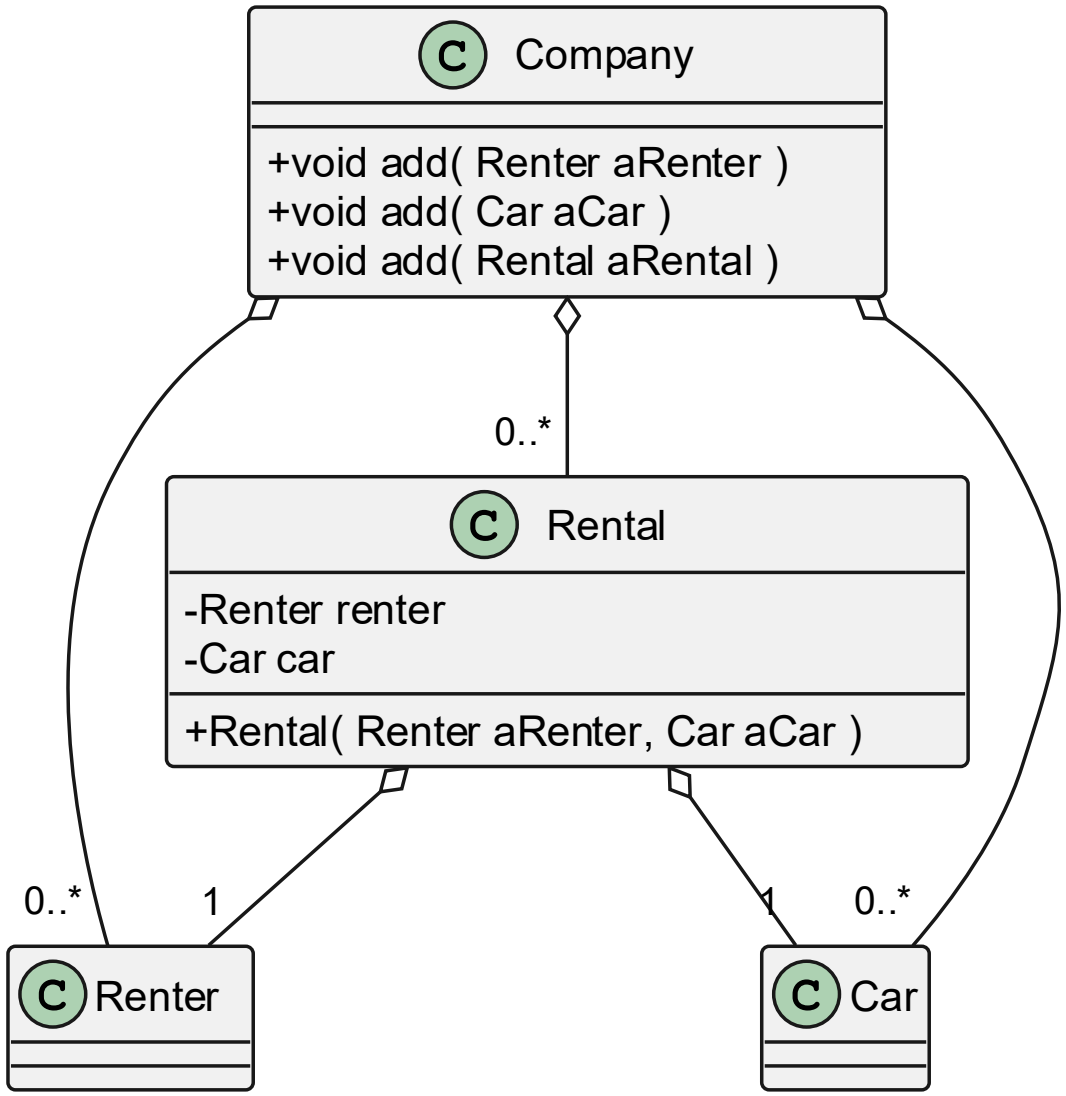
Exercise

- Analyze a UML class model for a car rental company that keeps track of cars, renters, and renters renting cars.


Company
0..*
Rental
0..*
1
1
0..*
Renter
Car

Exercise

- Implement the corresponding Java code.



Generalization

Generalization

- Look for commonalities:
  - Common attributes
    - E.g., all vehicles have…
  - Common methods (behaviour)
    - E.g., all vehicles can…
- Generalize:
  - Find what is common and factor it out into a more general “base” abstraction

Generalization

- Implementation inheritance:
  - Generalize about method signatures, method implementations, and/or attributes
    - I.e., classes having these in common

Implementation Inheritance

- General part:
  - A base class (or "superclass") defines the attributes and methods to be shared
- Specific part:
  - A derived class (or "subclass") is endowed with the attributes and methods of its base class
  - A subclass may "extend" a superclass by adding attributes and methods, or overriding an existing method

Java Implementation Inheritance

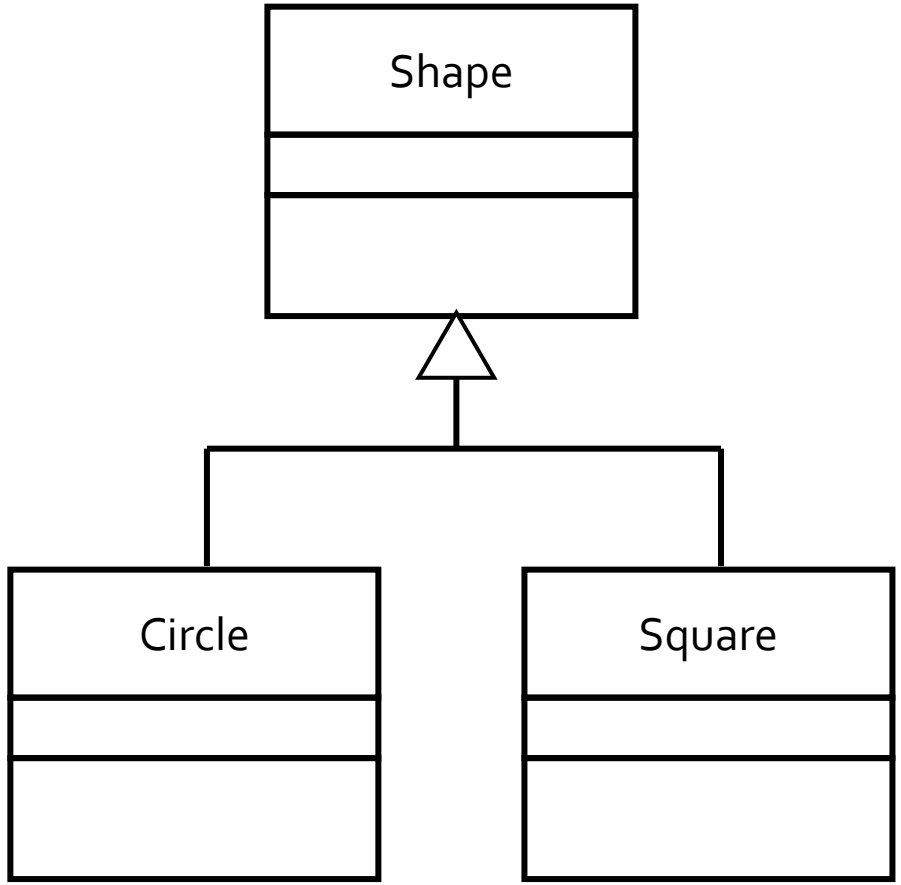
- ```
  public class Shape { // superclass
      protected Location myLocation;
      public Shape() { … }
      public void setLocation( Location p ) { … }
      public Location getLocation() { … }
  }
  ```

- ```
  public class Circle extends Shape { // subclass
      private int diameter;
      public Circle() { … }
      public void setDiameter( int d ) { … }
      …
  }
  ```

- ```
  public class Square extends Shape { // subclass
      private int side;
      public Square() { … }
      public void setSide( int s ) { … }
  }
  ```
  ```

UML Inheritance

- Implementation inheritance relationship:
  - "Is-a" relationship between classes
  - I.e., subclass "is-a" kind of superclass
  - I.e., subclass "extends" superclass
- E.g., Circle "is-a" kind of Shape



Generalization Principles

- Inappropriate inheritance:
  - Subclass inherits from superclass but "is-a" (is a kind of) relationship *does not* exist
  - If "is-a" test fails:
    - Likely not appropriate
  - If "is-a" test succeeds:
    - *May* or *may not* be appropriate

Generalization Principles

- Liskov substitution principle:
  - An instance of the subclass should be substitutable anywhere a reference to a superclass object is used
  - ```
    Shape s;
    s = new Circle(); // instance of subclass
    …
    Location l = s.getLocation(); // superclass method
    ```
    ```

Inheritance Example

- Suppose:
  - class Dog
    - Provides bark(), fetch()
  - class Cat extends Dog
    - “Hides” bark(), “hides” fetch(), and adds purr()
- Question:
  - Cat “is a” Dog?

Inheritance Example

- Suppose:
  - class Window
    - Provides show(), move(), resize()
  - class FixedSizeWindow extends Window
    - “Hides” resize()
- Question:
  - FixedSizeWindow “is a” Window?

Inheritance Example

- Suppose:
  - class ArrayList
    - Provides add(), get(), remove(), …
  - class ProjectTeam extends ArrayList
- Question:
  - ProjectTeam “is a” ArrayList?

Inheritance Issue

- Problem:
  - Superclass method is inherited, but it is not appropriate
  - What to do?

Inheritance Issue

- ```
  public class Rectangle {
      …
      public Rectangle( Size s ) { … }
      public void setLocation( Location p ) { … }
      public void setSize( Size s ) { … }
      public void draw() { … }
      public void clear() { … }
      public void rotate() { … }
      …
  }
  ```

- ```
  public class Square extends Rectangle {

      // inherits setSize(), but want to "hide" it
  }
  // Square 'is a' Rectangle?
  // Square specializes Rectangle?
  ```

Override the Method Approach

- ```
  public class Square extends Rectangle {
      @Override
      public void setSize( Size s ) {
          // should not implement
      }

  }
  ```
  ```

Aggregation Approach

```
public class Square {
    private Rectangle rect;
    // Square 'has a' Rectangle,
    // not 'is a' Rectangle

    public Square( int side ) {
        rect = new Rectangle(
            new Size( side, side ) );
    }
    …
    public void setSide( int newSide ) {
        rect.setSize(
            new Size( newSide, newSide ) );
    }

    public void draw() {
        rect.draw();
    }
    …
}
```

Restructuring Approach

- ```
  public class Quadrilateral {
      …
      public Quadrilateral() { … }
      public void setLocation( Location p ) { … }
      public void draw() { … }
      public void clear() { … }
      public void rotate() { … }
  }
  ```

- ```
  public class Rectangle extends Quadrilateral {
      …
      public Rectangle( Size s ) { … }
      public void setSize( Size s ) { … }
  }
  ```

- ```
  public class Square extends Quadrilateral {
      …
      public Square( int side ) { … }
      public void setSide( int side ) { … }
  }
  ```
  ```

Inheritance

- Java abstract class:
  - Declares one or more abstract methods
  - Cannot be instantiated; must be subclassed and have abstract methods overridden

- ```
  public abstract class Shape {
      public abstract double area();
      public abstract double perimeter();
      // there may be other instance data and methods
  }
  ```
- ```
  public class Circle extends Shape {
      public Circle() { … }
      public double area() { … }
      public double perimeter() { … }
  }
  ```

Interface Inheritance

- Java interface:
  - Declares method signatures
  - Classes implement the interface by providing all the method bodies

- ```
  public interface Bordered {
      public double area();
      public double perimeter();
  }
  ```
- ```
  public class Circle implements Bordered {
      public Circle() { … }
      public double area() { … }
      public double perimeter() { … }
  }
  ```

Interface Inheritance

- Java interface:
  - A "contract", specifying a *capability* that an implementing classes must provide
  - Gives method signatures, but typically no implementation
  - Cannot be instantiated
  - May extend other (sub)interfaces

- ```
  public interface Transformable extends Scalable, Translatable, Rotatable {
      …
  }
  ```
  ```

Java Interface

- ```
  public interface Cloneable {
      public Cloneable clone();
  }
  ```

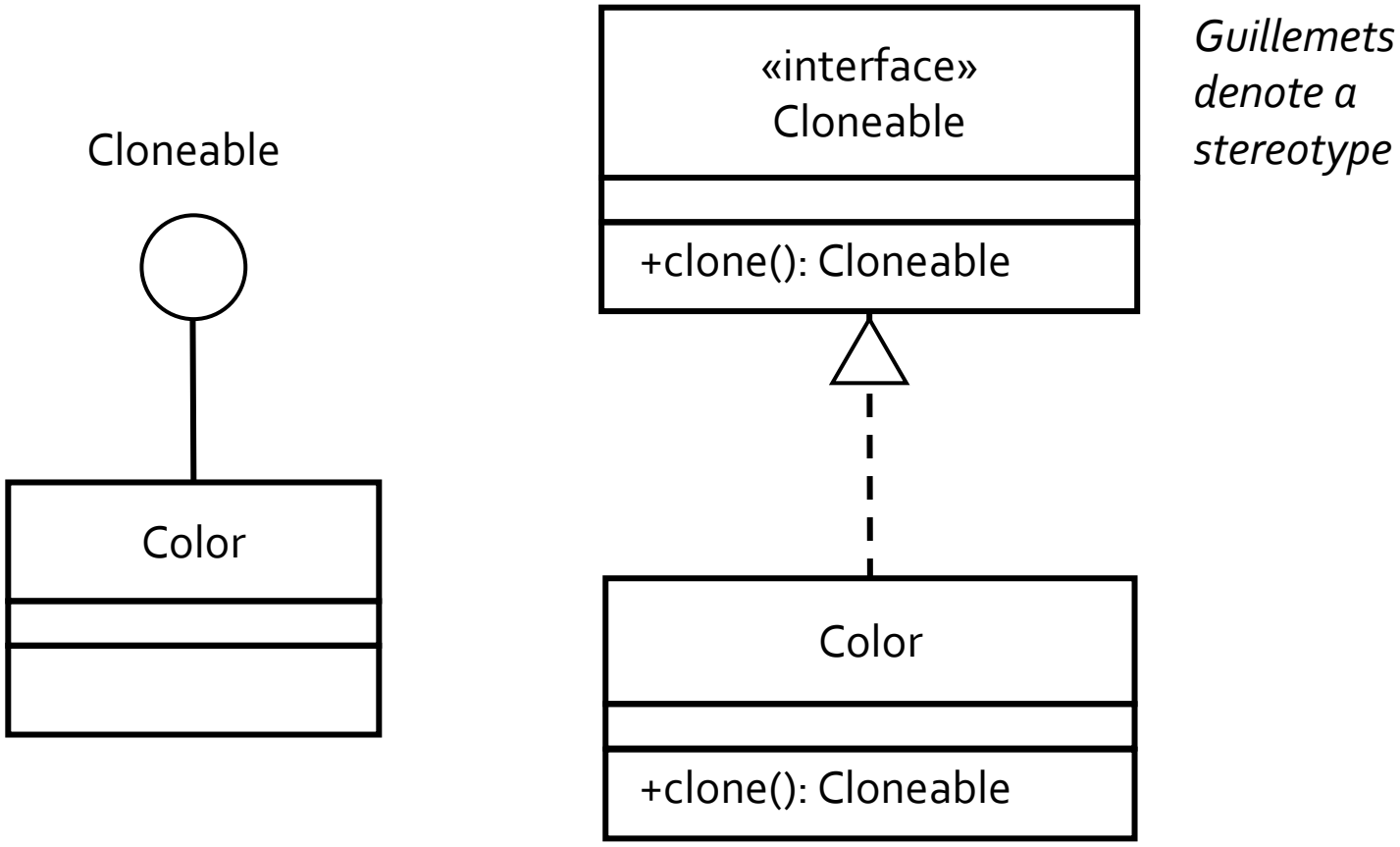
- ```
  public class Color implements Cloneable {
      private int red;
      private int green;
      private int blue;

      public Color( int r, int g, int b ) { … }

      public Cloneable clone() {
          return new Color( red, green, blue );
      }
  }
  ```

- ```
  Color red = new Color( 255, 0, 0 );
  Cloneable redClone = red.clone();
  Cloneable redClone2 = redClone.clone();
  ```
  ```

UML Interface

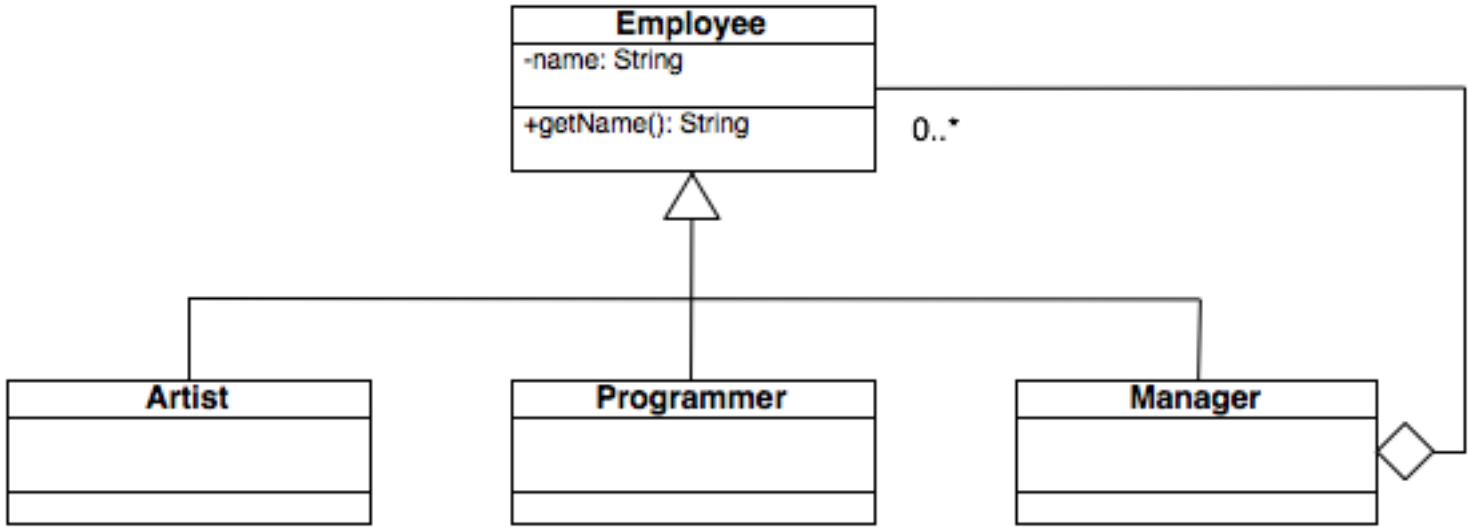


Guillemets denote a stereotype

Abstract Class versus Interface

- Differences:
  - An abstract class may provide a partial implementation
  - A class may implement any number of interfaces, but only extend one superclass
  - Adding a method to an interface will “break” any class that previously implemented it

Exercise



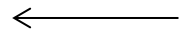
- Implement the corresponding Java code.

Java Subtleties

Java Call-by-Value

- ```
  public class Sender {
      …
      public void send() {
          Receiver r = new Receiver();
          Info argRef = new Info();

          r.receive( argRef );
          argRef.doSomeMore();
      }
  }
  ```



- ```
  public class Receiver {
      …
      public void receive( Info infoRef ) {
          infoRef.doSomething();
          infoRef = null;
      }
  }
  ```

Java Constructors

- ```
  public class Base {
      protected int value;
      public Base() {

          value = -1;
      }
  }
  ```

- ```
  public class Derived extends Base {
      public Derived() {

      }
  }
  ```

- ```
  Derived d = new Derived();
  ```
  ```

Java Constructors

- ```
  public class Base {
      protected int value;
      public Base() {
          // implicitly inserted call to super()
          value = -1;
      }
  }
  ```

- ```
  public class Derived extends Base {
      public Derived() {
          // implicitly inserted call to super()
      }
  }
  ```

- ```
  Derived d = new Derived();
  ```
  ```

Java Constructors

- ```
  public class Base {
      protected int value;
      public Base( int initValue ) {
          // implicitly inserted call to super()
          value = initValue;
      }
  }
  ```

- ```
  public class Derived extends Base {
      public Derived( int initValue ) {
          super( initValue );
          // explicit call to super( … );
          // super( … ) if used, must come first
      }
  }
  ```

- ```
  Derived d = new Derived( -1 );
  ```
  ```

Java Constructors

- ```
  public class Base {
      protected int value;
      public Base( int initValue ) {
          // implicitly inserted call to super()
          value = initValue;
      }
      public Base() {
          this( -1 );
          // this( … ) if used, must come first
      }
  }
  ```
- ```
  public class Derived extends Base {
      public Derived( int initValue ) {
          super( initValue );
      }
      public Derived() {
          // implicitly inserted call to super()
      }
  }
  ```
- ```
  Derived d = new Derived();
  ```
  ```

Java Shadowing Data

- ```
  public class Base {
      protected int value; // 2, 3
  }
  ```

- ```
  public class Derived extends Base {
      private int value; // 0, 1

      public void test() {
          value = 0;
          this.value = 1;
          super.value = 2;
          ((Base)this).value = 3;
      }
  }
  ```

Java Dynamic Binding

```
public class Base {
    // default implementation
    public void op() { … }
}
```

```
public class Derived1 extends Base {
    // does not override op()
}
```

```
public class Derived2 extends Base {
    // override ...
    @Override
    public void op() { … }
}
```

Selection of method to be run is made at run time, depending on type of receiving object

```
Base base;
base = new Derived1();  // implicit upcast
base.op();              // calls op() in Base
base = new Derived2();  // implicit upcast
base.op();              // calls op() in Derived2
```

Receiving object does the "right thing", even if the calling code does not show its actual type

Java Dynamic Binding

- Upcast:
  - "Widening" cast is safe due to the principle of substitutability
  - ```
    Base base = new Derived2(); // implicit upcast
    base.op(); // calls op() in Derived2
    ```

- Downcast:
  - "Narrowing" cast must be explicit
  - ```
    Base base = new Derived2(); // implicit upcast
    Derived2 derived = (Derived2)base; // downcast
    derived.op(); // calls op() in Derived2
    ```

Overriding Is Not Shadowing

- ```
  public class Base {
      public int i = 1;
      public int f() { return i; }
  }
  ```
- ```
  public class Derived extends Base {
      public int i = 2;                // shadowing
      public int f() { return -i; }    // overriding
  }
  ```
- ```
  public class Test {
      public static void main( String args[] ) {
          Derived d = new Derived();
          // d.i is 2
          // d.f() returns -2
          Base b = (Base)d;
          // b.i is 1
          // b.f() returns -2, 'dynamic binding'
      }
  }
  ```
  ```

Object-Oriented Analysis and Design

UML and OOA&D

- Analysis:
  - Requirements specification activity
    - Create UML use cases and class diagrams
- Design:
  - Architectural design activity
    - Refine UML class diagrams
  - Detailed design activity
    - Refine UML class diagrams
    - Create UML sequence and state diagrams

Object-Oriented Analysis

- Steps:
  - Discover objects from problem domain
    - Nouns may lead to classes and attributes
    - Verbs may lead to relationships and methods
  - Use CRC cards to note the analysis
  - Evaluate

Problem Description

- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Nouns

- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Verbs

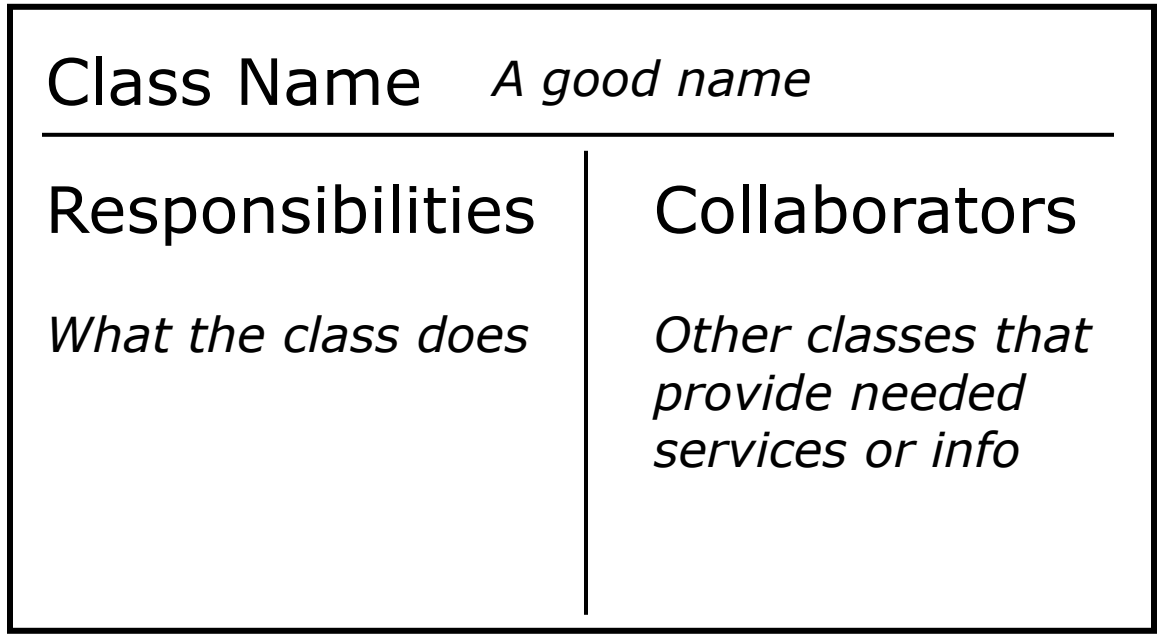
- The library has books and magazines. Books may be borrowed by any patron for four weeks while magazines may only be borrowed for two days. Up to six items at a time may be borrowed. The system tracks when books and magazines are borrowed.

Discover Objects

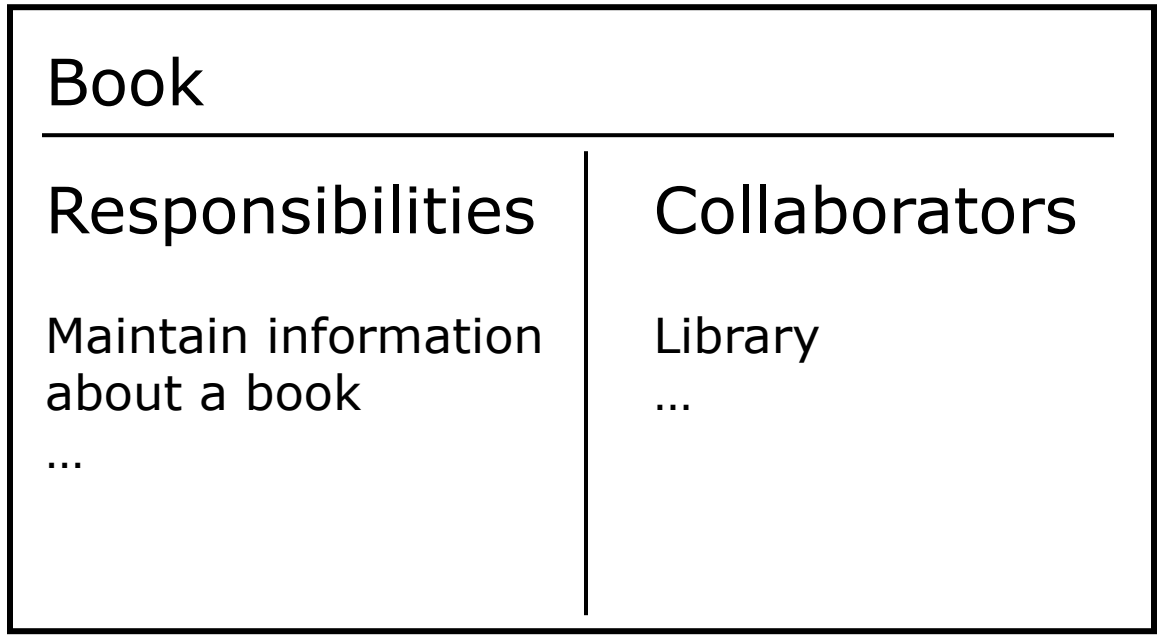
- Entity objects:
  - Things that model the problem domain
- Control objects:
  - Things that respond to events and coordinate services
- Boundary objects:
  - Things that interact with the system
    - E.g., other applications, devices, sensors, actors, roles, windows, forms

Use CRC Cards

- Class-Responsibility-Collaborator
  - Explore classes, their responsibilities, and their interactions
  - Organize index cards on a table



Use CRC Cards



Use CRC Cards

- Role playing:
  - Refine the cards by acting out a particular scenario with the candidate objects
  - "Become" the object
    - What do I do?
    - What do I need to remember?
    - With whom do I need to interact?
    - How do I respond?

Evaluate

- Principles:
  - During analysis, objects should initially be technology independent
  - If an object has only one attribute, perhaps it should not be a separate object at all
  - If an object has several highly related attributes, perhaps these attributes should form a separate object

Exercise

- Chess is a two-player, turn-based game where players move pieces on an 8x8 board. Each player begins a game with 16 initial pieces: 1 king, 1 queen, 2 rooks, 2 bishops, 2 knights, and 8 pawns. One player has the white pieces, and the other player has the black pieces. Pieces of different types move in specific ways or situations (e.g., navigating directionally, capturing an opponent's piece, castling, promoting, etc.).
- Consider a chess tutorial app, where a learner can tap on a piece and see its possible moves from which to choose. A learner can make a move upon the board. After making successive move(s), the learner can take back move(s).
- Using CRC cards, identify the key entity classes (and their generalizations as appropriate) in a well-designed, object-oriented model for this app. For each class, give it a good name, outline its main responsibilities, and list its collaborators.

Guidelines

- Get the big picture:
  - Understand the problem
    - Talk to the customer, end users, domain experts
  - Understand the target environment
    - Know the implementation constraints
  - Avoid reinventing the wheel
    - Reuse designs

Guidelines

- Modularity:
  - Increase cohesion
    - Class has a clear specific responsibility
  - Reduce coupling
    - Class is not connected to or knows too many others
  - Separate the layers
    - Identify entity, control, and boundary objects
    - Allow replacing layers

Guidelines

- Classes:
  - Use good names
    - Should be meaningful and explanatory
  - Avoid huge "blob" classes
    - A single class should not do everything
  - Use information hiding
    - Hide changeable details, reveal assumptions

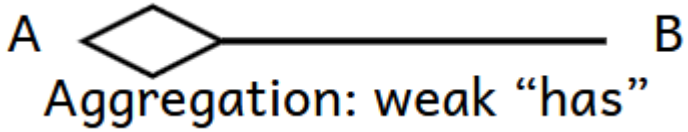
Guidelines

- Generalization:
  - Find superclasses
    - Look for and factor commonalities among classes
  - Apply Liskov principle for proper inheritance
    - Or use “is-a” test
  - “Is-a” test is not always enough
    - Class names can mislead, look at specific behaviour

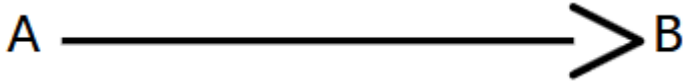
Guidelines

- Adaptation:
  - Hard to get it right the first time
    - Recognize problems and fix them
  - Your software won't go away
    - Make it easy to adapt to change
  - Simplicity (as simple as possible)
    - Does not always mean using the first thing that comes to mind
    - Elegant designs may need effort

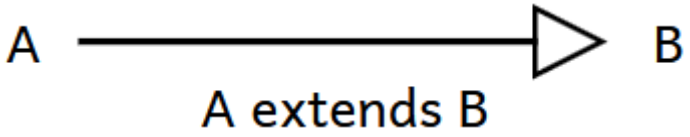
Association
Usually two-way navigability,
but sometimes just not specified

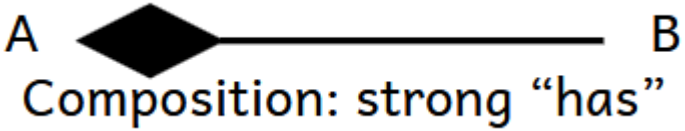


A has some Bs
The same Bs that A has can
be shared
Not exclusive!



One-way Navigability
We can only follow references from A to B





B is a part of A:
When instance of A is deleted,
all of its B are also deleted

More Information

- Books:
  - The Essence of Object-Oriented Programming with Java and UML
    - B. Wampler
    - Addison-Wesley, 2002
  - Java in a Nutshell
    - D. Flanagan
    - O'Reilly, 2005

More Information

- Books:
  - UML Distilled
    - M. Fowler
    - Addison-Wesley, 2003
  - The Elements of UML 2.0 Style
    - S. W. Ambler
    - Cambridge, 2005

More Information

- Link:
  - UML Quick Reference
    - http://www.holub.com/uml/
- Books:
  - Book: Developing applications with Java and UML
  - Book: Java Design: Objects, UML and Process
  - Book: Object-oriented design with UML and Java