Abram Hindle
Department of Computing Science
University of Alberta

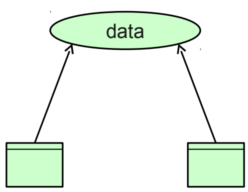
Objects, UML, and Java

Slides originally by Ken Wong

Modeling Principles

Language Evolution

COBOL, Fortran:
- subprograms (subroutines) access global data
- break up system into subroutines



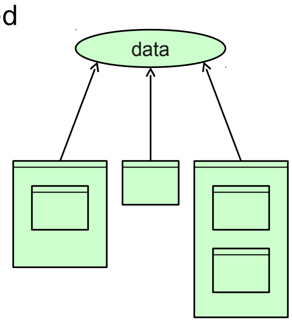
subprograms

Language Evolution

Algol, Pascal:

(nested) procedures with block structured scope

break up system into nested procedures



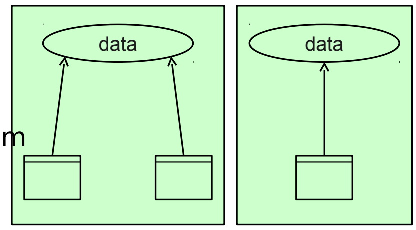
nested procedures

Language Evolution

Modula-2, C:

modules (files) of related data and functions

break up system into modules (e.g., abstract data types)



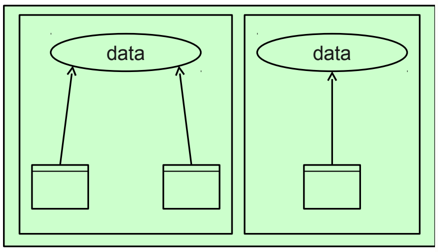
modules

Language Evolution

Smalltalk, C++, Java:

classes with data and methods

classes as “factories” for objects



classes

break up system into classes

Discussion

Question:

What software engineering design principles drove this evolution?

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
  - life time 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 ...

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 inventors:
- Grady Booch, Ivar Jacobson, James Rumbaugh

Purpose:
- express object-oriented designs visually
- programming language independent
- communicate, evaluate, and reuse designs
- make design intent more explicit

- can think about design, before coding

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 ) { … }
}
```


UML Class

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

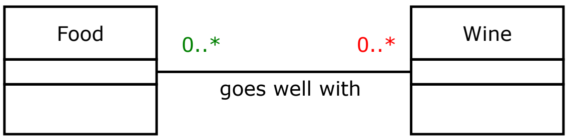
– private
\+ public

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
0 or more Wine *objects*

a Wine *object* is associated with
0 or more Food *objects*

Decomposition

Aggregation relationship:

weak “has-a” relationship

whole “has-a” part

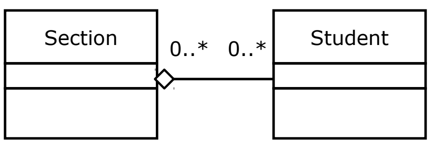
a part may belong to (be shared with)
other wholes

e.g., a Section and a Student

Java and UML Aggregation

Dynamic number of aggregated objects:

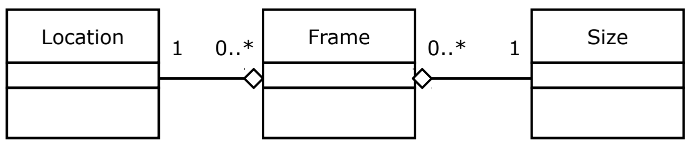
```
public class Section {
    private ArrayList<Student> roster;
    …

    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 myLocation; // shared object
    private Size mySize; // shared object
    …
}
```



Decomposition

Composition relationship:

strong “has-a” relationship

exclusive containment of parts

related object life times

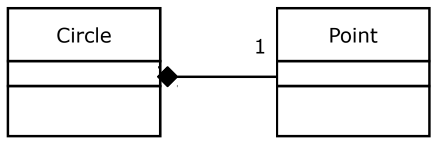
- the whole cannot exist without having the parts; if the whole is destroyed, the parts should also be destroyed

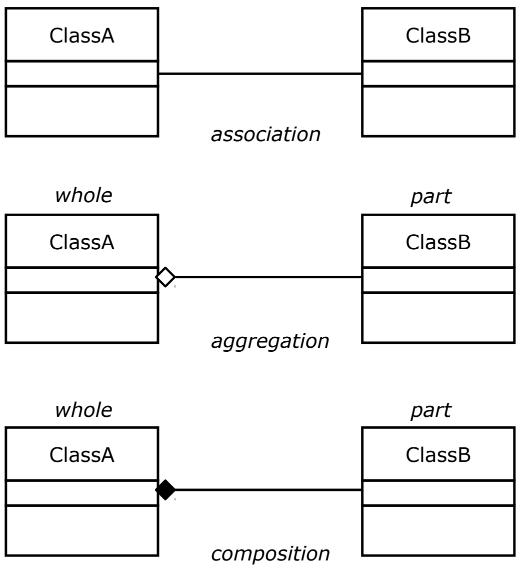
often access the parts through the whole

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)





Exercise

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

Generalization

Generalization

Look for commonalities:
- common attributes
  - e.g., all vehicles have ?
- common methods (behavior)
  - 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

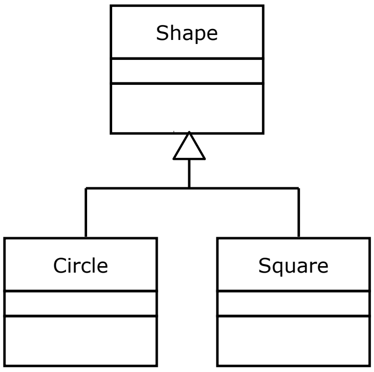
Specific part:
- a derived class (or "subclass") is endowed with the attributes and methods of its base class
- a subclass may "extend" a superclass by adding attributes and methods, or overriding an existing method

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 {

    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
}
class Circle extends Shape {
    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();
}
class Circle implements Bordered {
    public double area() { … }
    public double perimeter() { … }
}
```


Interface Inheritance

Java interface:

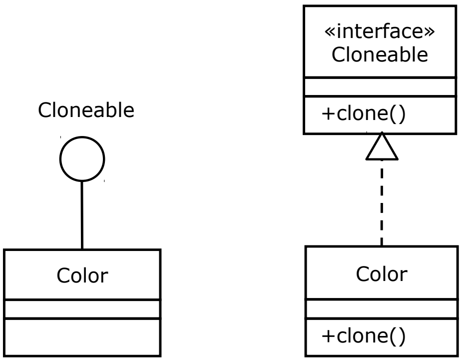
- a “contract”, specifying a *capability* that an implementing classes must provide
- gives method signatures, but no implementation
- cannot be instantiated
- may extend other (sub)interfaces

```
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 );
Color redClone = red.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

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

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

```
public class Base {
    // default implementation
    public void op() { … }
}
public class Derived1 extends Base {
    // does not override op()
}
public class Derived2 extends Base {
    // override ...
    public void op() { … }
}
```

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

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


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 6 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 6 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 6 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

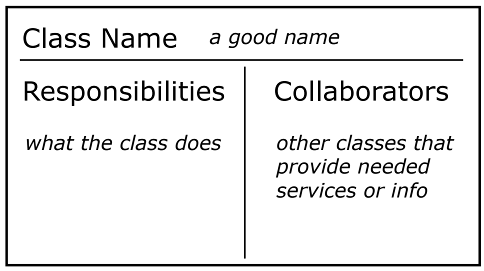
Boundary objects:
  things that interact with the system

- e.g., other applications, devices, sensors, actors, roles, windows, forms

Use CRC Cards

Class-Responsibility-Collaborator
  explore classes, their responsibilities, their interactions

  organize index cards on a table



use the back for more details

Use CRC Cards

Book	
Responsibilities	**Collaborators**
maintain information about a book ...	Library ...

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

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 can’t 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 behavior

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

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/goodies/uml/