Some Object-Oriented Design Principles

Principle #1

Minimize The Accessibility of Classes and Members

The Meaning of Abstraction

- Tony Hoare: “Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.”
- Grady Booch: “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”
- Abstraction is one of the fundamental ways to deal with complexity
- An abstraction focuses on the outside view of an object and separates an object’s behavior from its implementation

Encapsulation

- Grady Booch: “Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”
- Craig Larman: “Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations.”
- Classes should be opaque
- Classes should not expose their internal implementation details

Information Hiding In Java

- Use private members and appropriate accessors and mutators wherever possible
- For example:
  - ⇨ Replace

```
public double speed;
```

  - ⇨ with

```
private double speed;
public double getSpeed() {
  return(speed);
}
public void setSpeed(double newSpeed) {
  speed = newSpeed;
}
```

Use Accessors and Mutators, Not Public Members

- You can put constraints on values

```
public void setSpeed(double newSpeed) {
  if (newSpeed < 0) {
    sendErrorMessage(...);
    newSpeed = Math.abs(newSpeed);
  }
  speed = newSpeed;
}
```

- If users of your class accessed the fields directly, then they would each be responsible for checking constraints

Use Accessors and Mutators, Not Public Members

- You can change your internal representation without changing the interface

```
// Now using metric units (kph, not mph)

public void setSpeedInMPH(double newSpeed) {
  speedInKPH = convert(newSpeed);
}

public void setSpeedInKPH(double newSpeed) {
  speedInKPH = newSpeed;
}
```

Use Accessors and Mutators, Not Public Members

- You can perform arbitrary side effects

```
public double setSpeed(double newSpeed) {
  speed = newSpeed;
  notifyObservers();
}
```

- If users of your class accessed the fields directly, then they would each be responsible for executing side effects

Principle #2

Favor Composition Over Inheritance

Composition

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms
- For example:
  - ⇨ Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - ⇨ Aggregation - when one object has a collection of objects that can exist on their own (UML)
  - ⇨ Containment - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object (Coad)

Composition

- Composition can be:
  - By reference
  - By value
- C++ allows composition by value or by reference
- But in Java all we have are object references!

Advantages/Disadvantages Of Composition

- Advantages:
  - ⇨ Contained objects are accessed by the containing class solely through their interfaces
  - ⇨ "Black-box" reuse, since internal details of contained objects are *not* visible
  - ⇨ Good encapsulation
  - ⇨ Fewer implementation dependencies
  - ⇨ Each class is focused on just one task
  - ⇨ The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type
- Disadvantages:
  - ⇨ Resulting systems tend to have more objects
  - ⇨ Interfaces must be carefully defined in order to use many different objects as composition blocks

Inheritance

- Method of reuse in which new functionality is obtained by extending the implementation of an existing object
- The generalization class (the superclass) explicitly captures the common attributes and methods
- The specialization class (the subclass) extends the implementation with additional attributes and methods

Advantages/Disadvantages Of Inheritance

- Advantages:
  - ⇨ New implementation is easy, since most of it is inherited
  - ⇨ Easy to modify or extend the implementation being reused
- Disadvantages:
  - ⇨ Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - ⇨ "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - ⇨ Subclasses may have to be changed if the implementation of the superclass changes
  - ⇨ Implementations inherited from superclasses can not be changed at run-time

Inheritance vs Composition Example

- This example comes from the book *Effective Java* by Joshua Bloch
- Suppose we want a variant of HashSet that keeps track of the number of attempted insertions. So we subclass HashSet as follows:

```
public class InstrumentedHashSet extends HashSet {

  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet(Collection c) {super(c);}
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }
```

Inheritance vs Composition Example (Continued)

```
  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public int getAddCount() {
    return addCount;
  }

}
```

Inheritance vs Composition Example (Continued)

- Looks good, right. Let's test it!

```
public static void main(String[] args) {
  InstrumentedHashSet s = new InstrumentedHashSet();
  s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
  System.out.println(s.getAddCount());
}
```

- We get a result of 6, not the expected 3. Why?
- It's because the internal implementation of addAll() in the HashSet superclass itself invokes the add() method. So first we add 3 to addCount in InstrumentedHashSet's addAll(). Then we invoke HashSet's addAll(). For each element, this addAll() invokes the add() method, which as overridden by InstrumentedHashSet adds one for each element. The result: each element is double counted.

Inheritance vs Composition Example (Continued)

- There are several ways to fix this, but note the fragility of our subclass. Implementation details of our superclass affected the operation of our subclass.
- The best way to fix this is to use composition. Let's write an InstrumentedSet class that is composed of a Set object. Our InstrumentedSet class will duplicate the Set interface, but all Set operations will actually be forwarded to the contained Set object.
- InstrumentedSet is known as a wrapper class, since it wraps an instance of a Set object
- This is an example of delegation through composition!

Inheritance vs Composition Example (Continued)

```
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {this.s = s;}

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {return addCount;}
```

Inheritance vs Composition Example (Continued)

```
// Forwarding methods (the rest of the Set interface methods)
public void clear()               { s.clear();            }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty()          { return s.isEmpty();   }
public int size()                 { return s.size();      }
public Iterator iterator()        { return s.iterator();  }
public boolean remove(Object o)   { return s.remove(o);   }
public boolean containsAll(Collection c)
                                  { return s.containsAll(c); }
public boolean removeAll(Collection c)
                                  { return s.removeAll(c);   }
public boolean retainAll(Collection c)
                                  { return s.retainAll(c);   }
public Object[] toArray()           { return s.toArray();  }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o)     { return s.equals(o);  }
public int hashCode()               { return s.hashCode(); }
public String toString()            { return s.toString(); }
}
```

Inheritance vs Composition Example (Continued)

- Note several things:
  - ⇨ This class is a Set
  - ⇨ It has one constructor whose argument is a Set
  - ⇨ The contained Set object can be an object of any class that implements the Set interface (and not just a HashSet)
  - ⇨ This class is very flexible and can wrap any preexisting Set object
- Example:

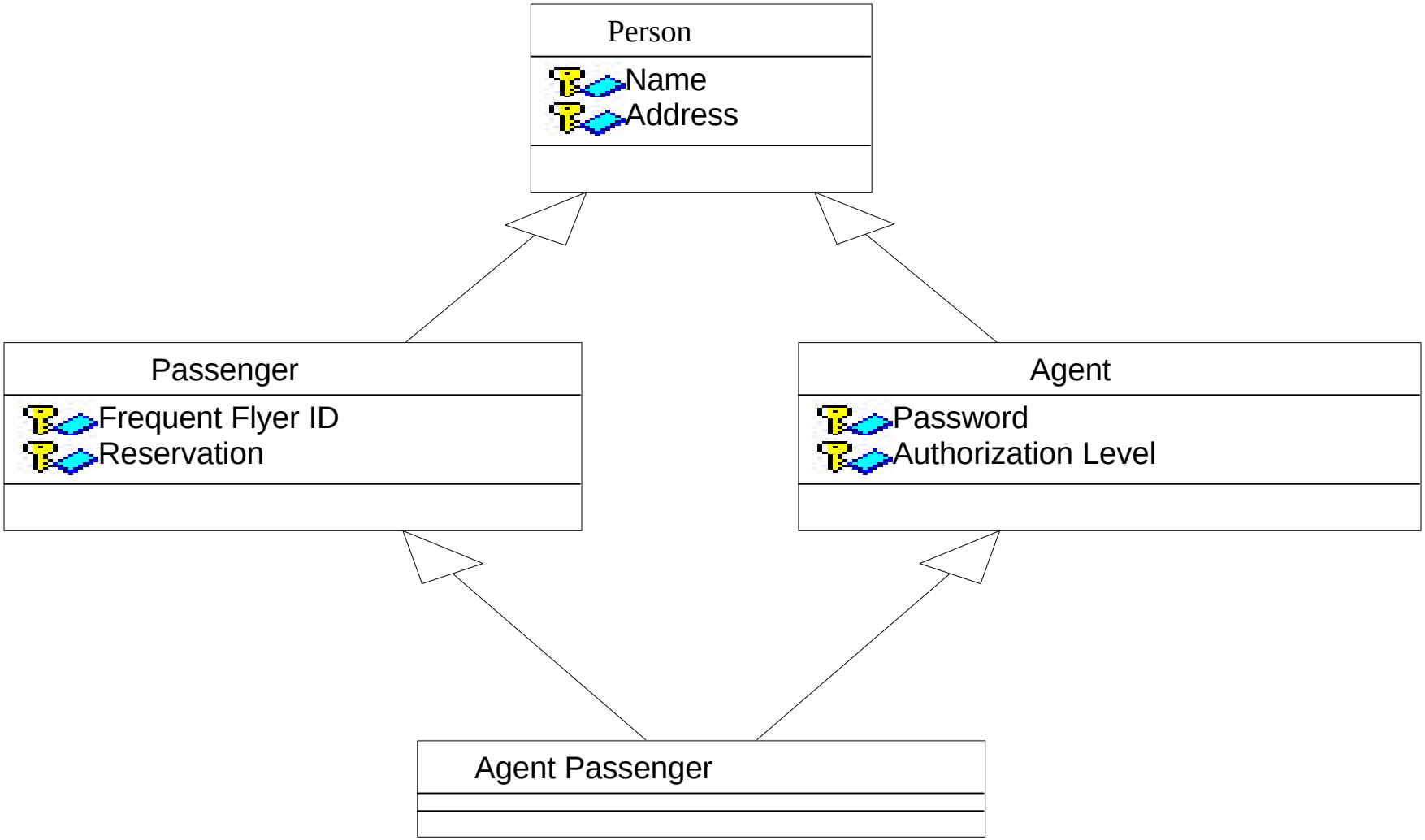
```
List list = new ArrayList();
Set s1 = new InstrumentedSet(new TreeSet(list));

int capacity = 7;
float loadFactor = .66f;
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

Coad's Rules

Use inheritance only when all of the following criteria are satisfied:

- A subclass expresses "is a special kind of" and not "is a role played by a"
- An instance of a subclass never needs to become an object of another class
- A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
- A subclass does not extend the capabilities of what is merely a utility class
- For a class in the actual Problem Domain, the subclass specializes a role, transaction or device

Inheritance/Composition Example 1



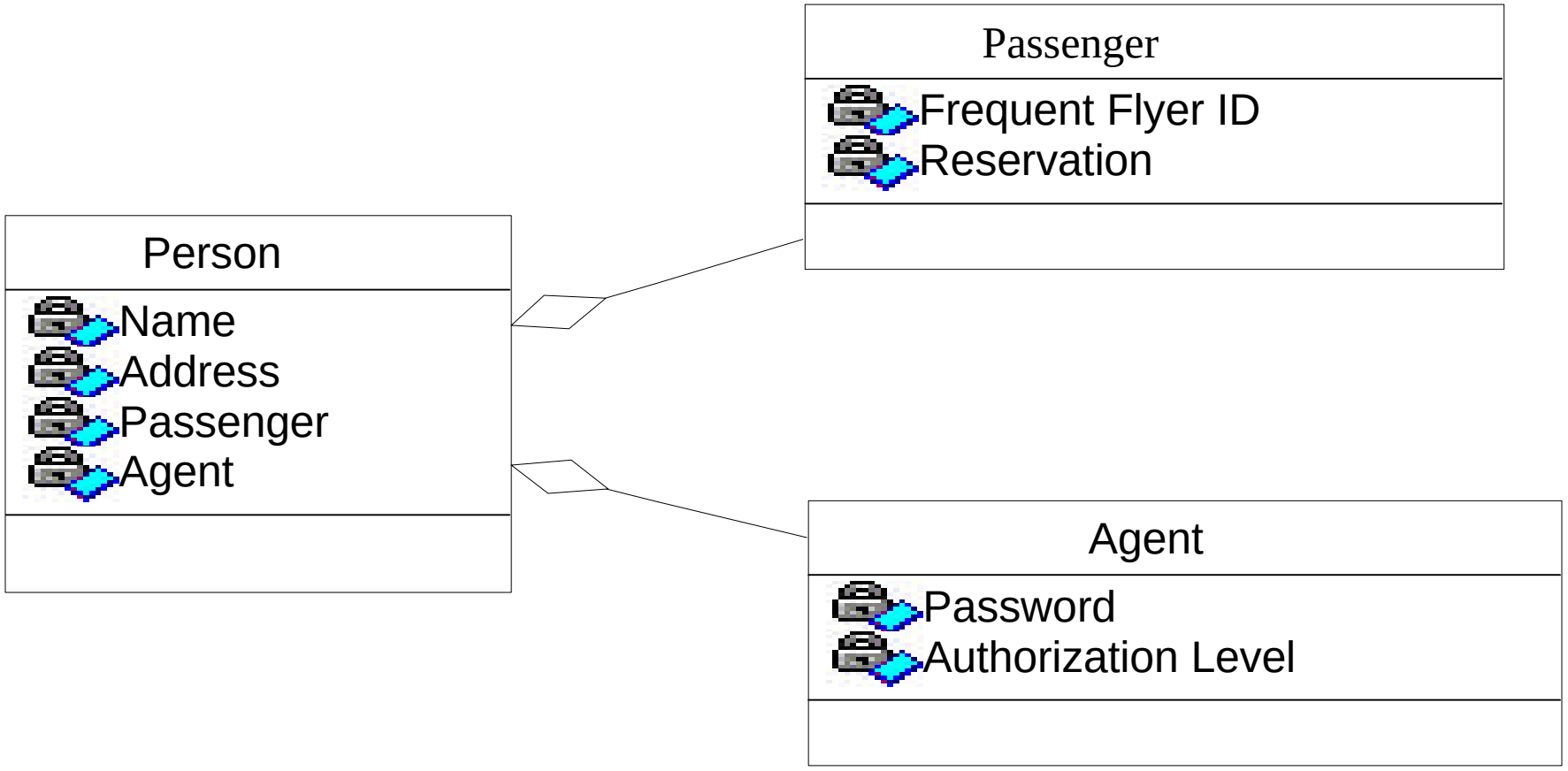
Inheritance/Composition Example 1 (Continued)

- "Is a special kind of" not "is a role played by a"
  - ⇨ **Fail**. A passenger is a role a person plays. So is an agent.
- Never needs to transmute
  - ⇨ **Fail**. A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time
- Extends rather than overrides or nullifies
  - ⇨ **Pass**.
- Does not extend a utility class
  - ⇨ **Pass**.
- Within the Problem Domain, specializes a role, transaction or device
  - ⇨ **Fail**. A Person is not a role, transaction or device.

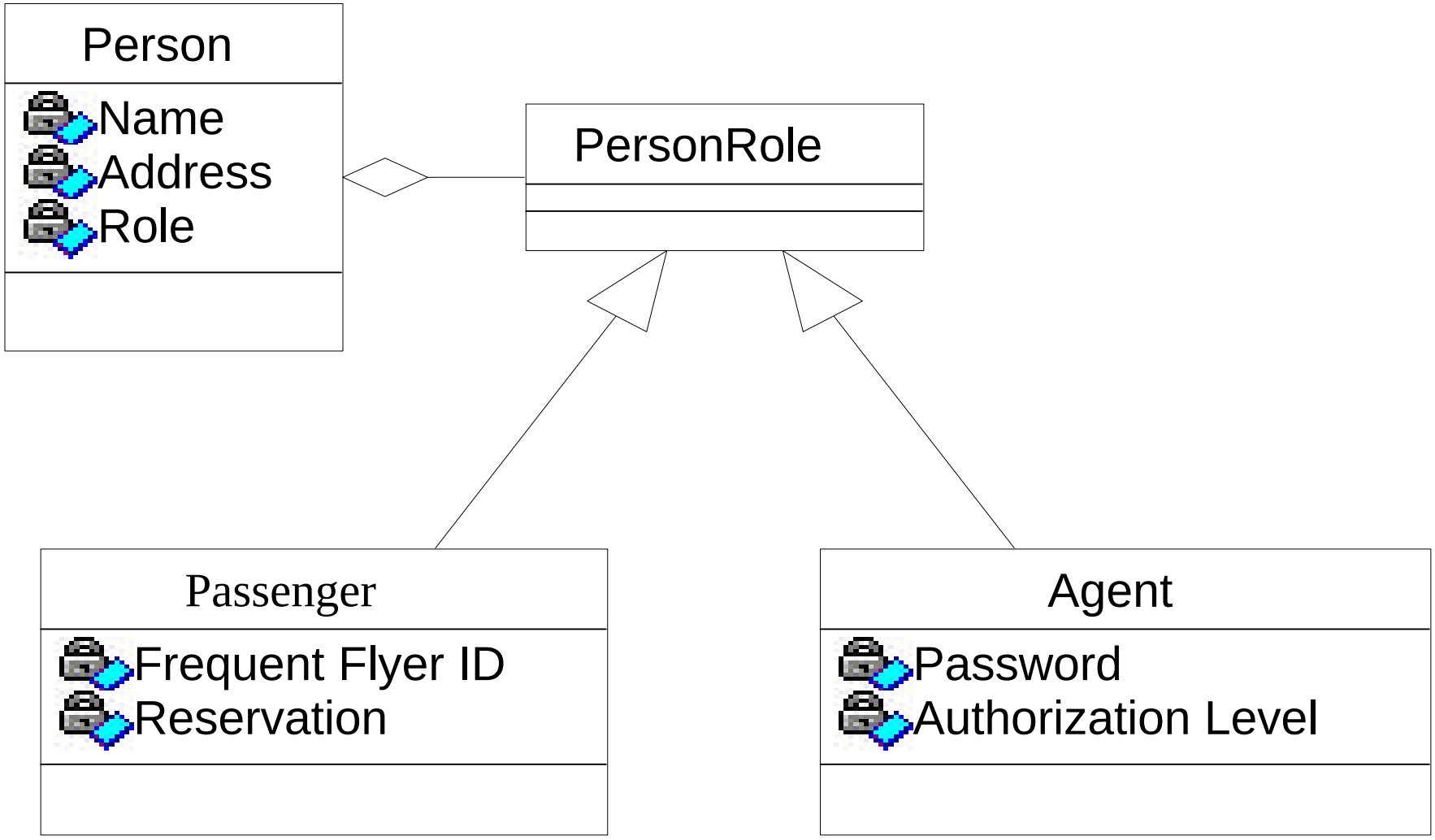
Inheritance does not fit here!

Inheritance/Composition Example 1 (Continued)

Composition to the rescue!



Inheritance/Composition Example 2

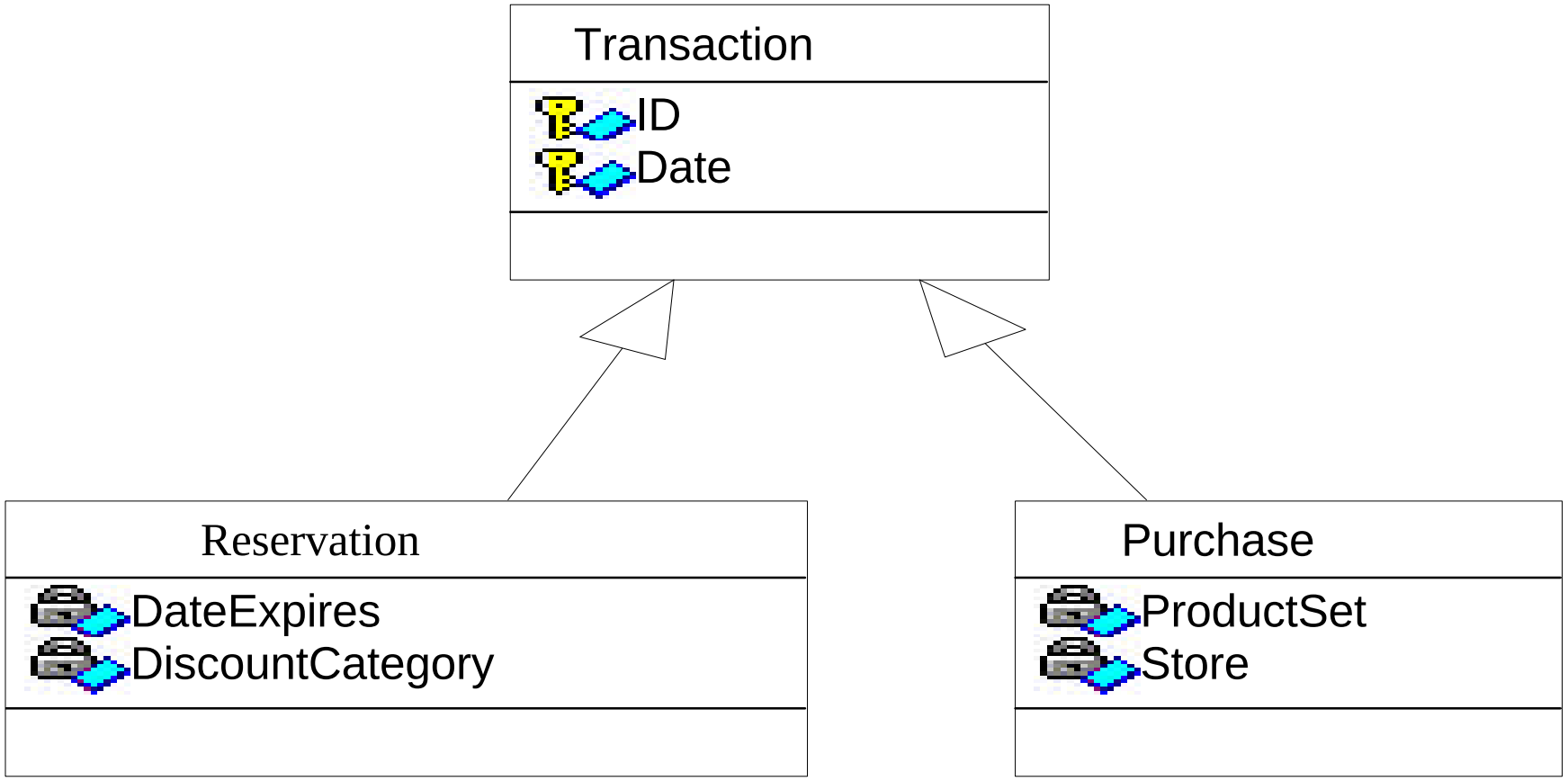


Inheritance/Composition Example 2 (Continued)

- "Is a special kind of" not "is a role played by a"
  - ⇨ **Pass**. Passenger and agent are special kinds of person roles.
- Never needs to transmute
  - ⇨ **Pass**. A Passenger object stays a Passenger object; the same is true for an Agent object.
- Extends rather than overrides or nullifies
  - ⇨ **Pass**.
- Does not extend a utility class
  - ⇨ **Pass**.
- Within the Problem Domain, specializes a role, transaction or device
  - ⇨ **Pass**. A PersonRole is a type of role.

Inheritance ok here!

Inheritance/Composition Example 3

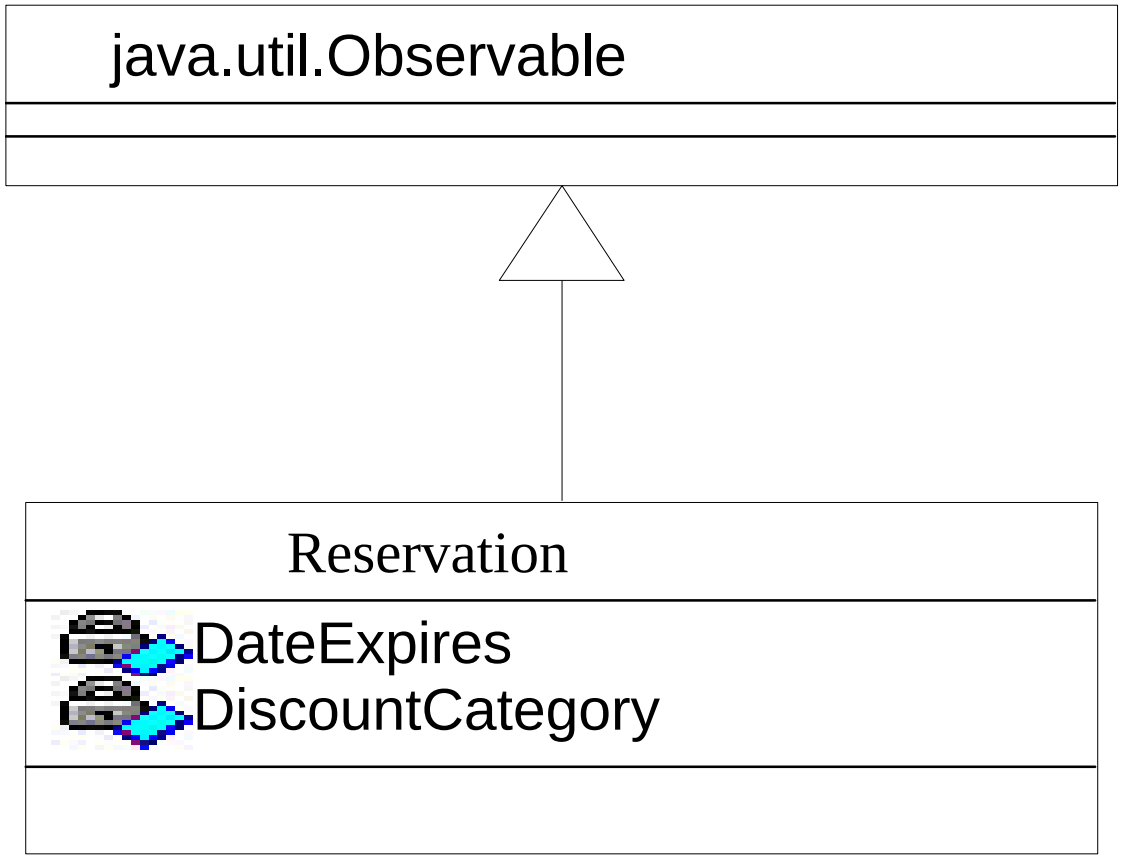


Inheritance/Composition Example 3 (Continued)

- "Is a special kind of" not "is a role played by a"
  - ⇨ **Pass**. Reservation and purchase are a special kind of transaction.
- Never needs to transmute
  - ⇨ **Pass**. A Reservation object stays a Reservation object; the same is true for a Purchase object.
- Extends rather than overrides or nullifies
  - ⇨ **Pass**.
- Does not extend a utility class
  - ⇨ **Pass**.
- Within the Problem Domain, specializes a role, transaction or device
  - ⇨ **Pass**. It's a transaction.

Inheritance ok here!

Inheritance/Composition Example 4



Inheritance/Composition Example 4 (Continued)

- "Is a special kind of" not "is a role played by a"
  - ⇨ **Fail**. A reservation is not a special kind of observable.
- Never needs to transmute
  - ⇨ **Pass**. A Reservation object stays a Reservation object.
- Extends rather than overrides or nullifies
  - ⇨ **Pass**.
- Does not extend a utility class
  - ⇨ **Fail**. Observable is just a utility class.
- Within the Problem Domain, specializes a role, transaction or device
  - ⇨ **Not Applicable**. Observable is a utility class, not a Problem Domain class

Inheritance does not fit here!

Inheritance/Composition Summary

- Both composition and inheritance are important methods of reuse
- Inheritance was overused in the early days of OO development
- Over time we've learned that designs can be made more reusable and simpler by favoring composition
- Of course, the available set of composable classes can be enlarged using inheritance
- So composition and inheritance work together
- But our fundamental principle is:

Favor Composition Over Inheritance

Principle #3

Program To An Interface, Not An Implementation

Interfaces

- An *interface* is the set of methods one object knows it can invoke on another object
- An object can have many interfaces. (Essentially, an interface is a subset of all the methods that an object implements).
- A *type* is a specific interface of an object
- Different objects can have the same type and the same object can have many different types
- An object is known by other objects only through its interface
- In a sense, interfaces express "is a kind of" in a very limited way as "is a kind of that supports this interface"
- Interfaces are the key to pluggability!

Implementation Inheritance vs Interface Inheritance

- *Implementation Inheritance* (*Class Inheritance*) - an object's implementation is defined in terms of another's objects implementation
- *Interface Inheritance* (*Subtyping*) - describes when one object can be used in place of another object
- The C++ inheritance mechanism means both class and interface inheritance
- C++ can perform interface inheritance by inheriting from a pure abstract class
- Java has a separate language construct for interface inheritance - the Java interface
- Java's interface construct makes it easier to express and implement designs that focus on object interfaces

Benefits Of Interfaces

- Advantages:
  - Clients are unaware of the specific class of the object they are using
  - One object can be easily replaced by another
  - Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
  - Loosens coupling
  - Increases likelihood of reuse
  - Improves opportunities for composition since contained objects can be of any class that implements a specific interface
- Disadvantages:
  - Modest increase in design complexity

Interface Example

```
/**
 * Interface IManeuverable provides the specification
 *   for a maneuverable vehicle.
 */
public interface IManeuverable {
  public void left();
  public void right();
  public void forward();
  public void reverse();
  public void climb();
  public void dive();
  public void setSpeed(double speed);
  public double getSpeed();
}
```

Interface Example (Continued)

```
public class Car
  implements IManeuverable {  // Code here. }


public class Boat
  implements IManeuverable {  // Code here. }


public class Submarine
  implements IManeuverable {  // Code here. }
```

Interface Example (Continued)

- This method in some other class can maneuver the vehicle without being concerned about what the actual class is (car, boat, submarine) or what inheritance hierarchy it is in

```
public void travel(IManeuverable vehicle) {
  vehicle.setSpeed(35.0);
  vehicle.forward();
  vehicle.left();
  vehicle.climb();
}
```

Principle #4

The Open-Closed Principle:

Software Entities Should Be Open For Extension, Yet Closed For Modification

The Open-Closed Principle

- The Open-Closed Principle (OCP) says that we should attempt to design modules that never need to be changed
- To extend the behavior of the system, we add new code. We do not modify old code.
- Modules that conform to the OCP meet two criteria:
  - ⇨ Open For Extension - The behavior of the module can be extended to meet new requirements
  - ⇨ Closed For Modification - the source code of the module is not allowed to change
- How can we do this?
  - ⇨ Abstraction
  - ⇨ Polymorphism
  - ⇨ Inheritance
  - ⇨ Interfaces

The Open-Closed Principle

- It is not possible to have all the modules of a software system satisfy the OCP, but we should attempt to minimize the number of modules that do not satisfy it
- The Open-Closed Principle is really the heart of OO design
- Conformance to this principle yields the greatest level of reusability and maintainability

Open-Closed Principle Example

- Consider the following method of some class:

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    total += parts[i].getPrice();
  }
  return total;
}
```

- The job of the above function is to total the price of each part in the specified array of parts
- If Part is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of parts *without* having to be modified!
- It conforms to the OCP

Open-Closed Principle Example (Continued)

- But what if the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price.
- How about the following code?

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    if (parts[i] instanceof Motherboard)
      total += (1.45 * parts[i].getPrice());
    else if (parts[i] instanceof Memory)
      total += (1.27 * parts[i].getPrice());
    else
      total += parts[i].getPrice();
  }
  return total;
}
```

Open-Closed Principle Example (Continued)

- Does this conform to the OCP? No way!
- Every time the Accounting Department comes out with a new pricing policy, we have to modify the totalPrice() method! It is *not* Closed For Modification. Obviously, policy changes such as that mean that we have to modify code somewhere, so what could we do?
- To use our first version of totalPrice(), we could incorporate pricing policy in the getPrice() method of a Part

Open-Closed Principle Example (Continued)

- Here are example Part and ConcretePart classes:

```
// Class Part is the superclass for all parts.
public class Part {
  private double price;
  public Part(double price) (this.price = price;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
  public double getPrice() {
    // return (1.45 * price);   //Premium
    return (0.90 * price);      //Labor Day Sale
  }
}
```

Open-Closed Principle Example (Continued)

- But now we must modify each subclass of Part whenever the pricing policy changes!
- A better idea is to have a PricePolicy class which can be used to provide different pricing policies:

```
// The Part class now has a contained PricePolicy object.
public class Part {
  private double price;
  private PricePolicy pricePolicy;

  public void setPricePolicy(PricePolicy pricePolicy) {
    this.pricePolicy = pricePolicy;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return pricePolicy.getPrice(price);}
}
```

Open-Closed Principle Example (Continued)

```
/**
 * Class PricePolicy implements a given price policy.
 */
public class PricePolicy {
  private double factor;

  public PricePolicy (double factor) {
    this.factor = factor;
  }

  public double getPrice(double price) {return price * factor;}

}
```

Open-Closed Principle Example (Continued)

- With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to
- Of course, in an actual application, both the price of a Part and its associated PricePolicy could be contained in a database

The Single Choice Principle

A corollary to the OCP is the Single Choice Principle

The Single Choice Principle:

Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives

Principle #5

The Liskov Substitution Principle:

Functions That Use References To Base (Super) Classes Must Be Able To Use Objects Of Derived (Sub) Classes Without Knowing It

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) seems obvious given all we know about polymorphism
- For example:

```
public void drawShape(Shape s) {
        // Code here.
}
```

- The drawShape method should work with any subclass of the Shape superclass (or, if Shape is a Java interface, it should work with any class that implements the Shape interface)
- But we must be careful when we implement subclasses to insure that we do not unintentionally violate the LSP

The Liskov Substitution Principle

- If a function does not satisfy the LSP, then it probably makes explicit reference to some or all of the subclasses of its superclass. Such a function also violates the Open-Closed Principle, since it may have to be modified whenever a new subclass is created.

LSP Example

- Consider the following Rectangle class:

```
// A very nice Rectangle class.
public class Rectangle {
  private double width;
  private double height;

  public Rectangle(double w, double h) {
    width = w;
    height = h;
  }
  public double getWidth() {return width;}
  public double getHeight() {return height;}
  public void setWidth(double w) {width = w;}
  public void setHeight(double h) {height = h;}
  public double area() {return (width * height);
}
```

LSP Example (Continued)

- Now, had about a Square class? Clearly, a square is a rectangle, so the Square class should be derived from the Rectangle class, right? Let's see!
- Observations:
  - ⇨ A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So, each Square object wastes a little memory, but this is not a major concern.
  - ⇨ The inherited setWidth() and setHeight() methods are not really appropriate for a Square, since the width and height of a square are identical. So we'll need to override setWidth() and setHeight(). Having to override these simple methods is a clue that this might not be an appropriate use of inheritance!

LSP Example (Continued)

- Here's the Square class:

```
// A Square class.
public class Square extends Rectangle {

  public Square(double s) {super(s, s);}

  public void setWidth(double w) {
    super.setWidth(w);
    super.setHeight(w);
  }

  public void setHeight(double h) {
    super.setHeight(h);
    super.setWidth(h);
  }
}
```

LSP Example (Continued)

- Everything looks good. But check this out!

```
public class TestRectangle {

  // Define a method that takes a Rectangle reference.
  public static void testLSP(Rectangle r) {
    r.setWidth(4.0);
    r.setHeight(5.0);
    System.out.println("Width is 4.0 and Height is 5.0" +
                       ", so Area is " + r.area());
    if (r.area() == 20.0)
      System.out.println("Looking good!\n");
    else
      System.out.println("Huh?? What kind of rectangle is
                                this??\n");
  }
```

LSP Example (Continued)

```
  public static void main(String args[]) {

    //Create a Rectangle and a Square
    Rectangle r = new Rectangle(1.0, 1.0);
    Square s = new Square(1.0);

    // Now call the method above.  According to the
    // LSP, it should work for either Rectangles or
    // Squares.  Does it??
    testLSP(r);
    testLSP(s);
  }

}
```

LSP Example (Continued)

- Test program output:

```
Width is 4.0 and Height is 5.0, so Area is 20.0
Looking good!

Width is 4.0 and Height is 5.0, so Area is 25.0
Huh?? What kind of rectangle is this??
```

- Looks like we violated the LSP!

LSP Example (Continued)

- What's the problem here? The programmer of the testLSP() method made the reasonable assumption that changing the width of a Rectangle leaves its height unchanged.
- Passing a Square object to such a method results in problems, exposing a violation of the LSP
- The Square and Rectangle classes look self consistent and valid. Yet a programmer, making reasonable assumptions about the base class, can write a method that causes the design model to break down
- Solutions can not be viewed in isolation, they must also be viewed in terms of reasonable assumptions that might be made by users of the design

LSP Example (Continued)

- A mathematical square might be a rectangle, but a Square object is not a Rectangle object, because the behavior of a Square object is not consistent with the behavior of a Rectangle object!
- Behaviorally, a Square is *not* a Rectangle! A Square object is not polymorphic with a Rectangle object.

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) makes it clear that the ISA relationship is all about behavior
- In order for the LSP to hold (and with it the Open-Closed Principle) all subclasses must conform to the behavior that clients expect of the base classes they use
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable
- If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- The guarantee of the LSP is that a subclass can always be used wherever its base class is used!