

## Agenda

- Introduction to Design Patterns
- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- J2EE Design Patterns

K.Venkata Ramana



# GOF Design Patterns

## 1.0 Introduction to Design Patterns

### What is the design pattern?

- ▶ If a problem occurs over and over again, a solution to that problem has been used effectively. That solution is described as a pattern.
- ▶ The design patterns are language-independent strategies for solving common object-oriented design problems.
- ▶ Learning design patterns is good for people to communicate each other effectively.

### Do I have to use the design pattern?

- ▶ Learning design patterns speeds up your experience accumulation in OOA/OOD.
- ▶ You will learn the proven solutions for the problems designed by experienced developers.
- ▶ Helps in effective design of your application.



# GOF Design Patterns

## 1.0 Introduction to Design Patterns

### How many design patterns?

- Many. A site says at least 250 existing patterns are used in OO world, including Spaghetti which refers to poor coding habits.
- The 23 design patterns by GOF are well known, and more are to be discovered on the way.

### What is the relationship among these patterns?

Different designer may use different patterns to solve the same problem. Usually:

- ▶ Some patterns naturally fit together
- ▶ One pattern may lead to another
- ▶ Some patterns are similar and alternative
- ▶ Patterns are discoverable and documentable
- ▶ Patterns are not methods or framework
- ▶ Patterns give you hint to solve a problem effectively



The goal of Unified Modeling Language is –

- ▶ To represent complete systems using object oriented concepts.
- ▶ To establish an explicit coupling between concepts and executable code.
- ▶ To create a modeling language useable by both humans and machines.



# GOF Design Patterns

## 2.0 Understanding UML

### Multiplicity:

Specifies how many instances of one class may relate to a single instance of an associated class.

### Links and Associations:

Are the means of establishing relationships among objects and classes. E.g. Smith 'works for' Simplex company is a Link. A person works for a company is associations.

They are inherently bi-directional in relations ship.

### Aggregation

It is a part of relationship. It is tightly coupled form of association and transitive in nature i.e if A is part of B and B is part of C then A is part of C. Eg – Company and department.

### Composition

Is exactly like Aggregation except that the lifetime of the 'part' is controlled by the 'whole'.

### Generalization, Specialization and Inheritance:

Generalization - refers to relationship among classes.

Inheritance – mechanism of sharing of attributes and operations.

Specialization – refinement of sub-classes.



# GOF Design Patterns

## 2.0 Understanding UML

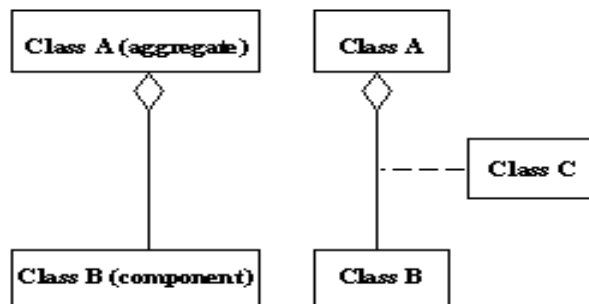
### CLASS NOTATION

Concrete Class

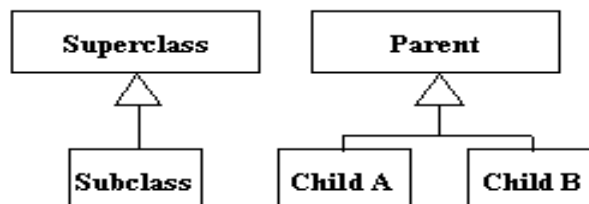
SEDRIS shades Abstract Classes

*Abstract Class*

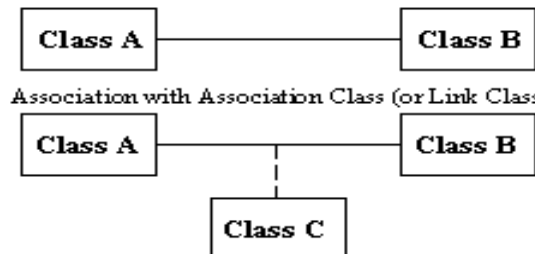
### AGGREGATION (has-a)



### INHERITANCE (is-a)



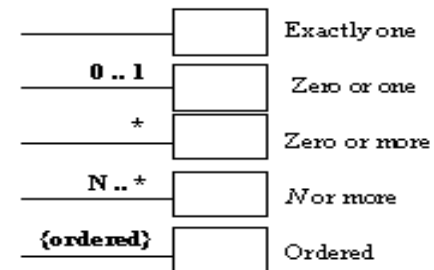
### ASSOCIATION



One-way (directed) Association



### MULTIPLICITY



## **Some Object-Oriented Design Principles**



# Principle #1

*Minimize The Accessibility  
of Classes and Members*





# Abstraction

- ◆ “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.”
- ◆ “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.”



# Encapsulation

- ◆ “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.”
- ◆ “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 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*



# 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 Of Inheritance

- ◆ New implementation is easy, since most of it is inherited
- ◆ Easy to modify or extend the implementation being reused



# Disadvantages Of Inheritance

- ◆ Breaks encapsulation, since it exposes a subclass to implementation details of its super class
- ◆ "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 runtime



# 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*.



# 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 of Composition

- ◆ 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 of Composition

- ◆ Resulting systems tend to have more objects
- ◆ Interfaces must be carefully defined in order to use many different objects as composition blocks



# Inheritance vs Composition Example

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

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

```
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

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?



# Inheritance vs Composition Example

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

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

```
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

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



# Inheritance vs Composition Example

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));
```



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

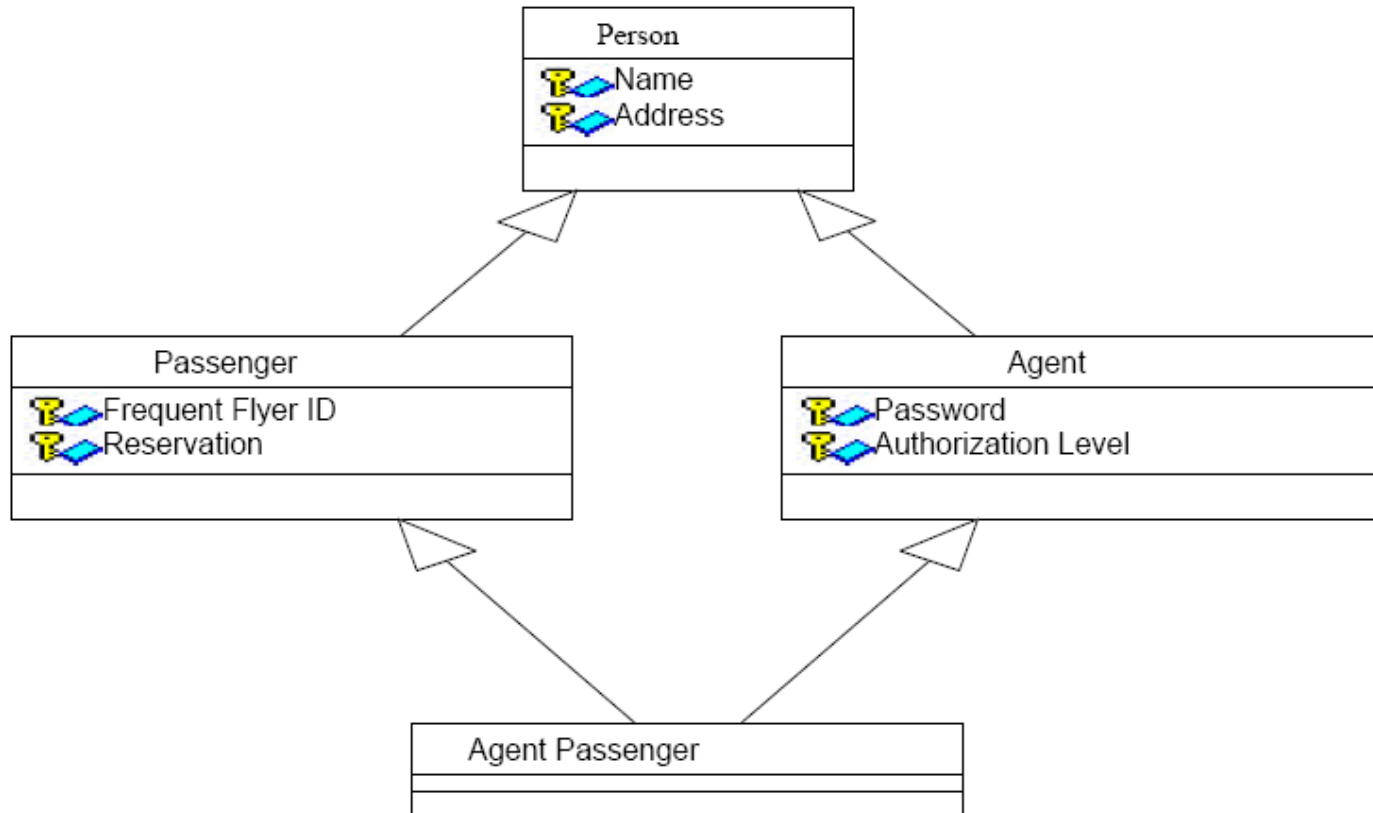
- ◆ A subclass expresses "is a special kind of" and not "is a role played by a"
- ◆ An instance of a subclass never needs to become an object of another class
- ◆ A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass
- ◆ A subclass does not extend the capabilities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





# GOF Design Patterns

## Inheritance/Composition Example 1



## Inheritance/Composition Example 1

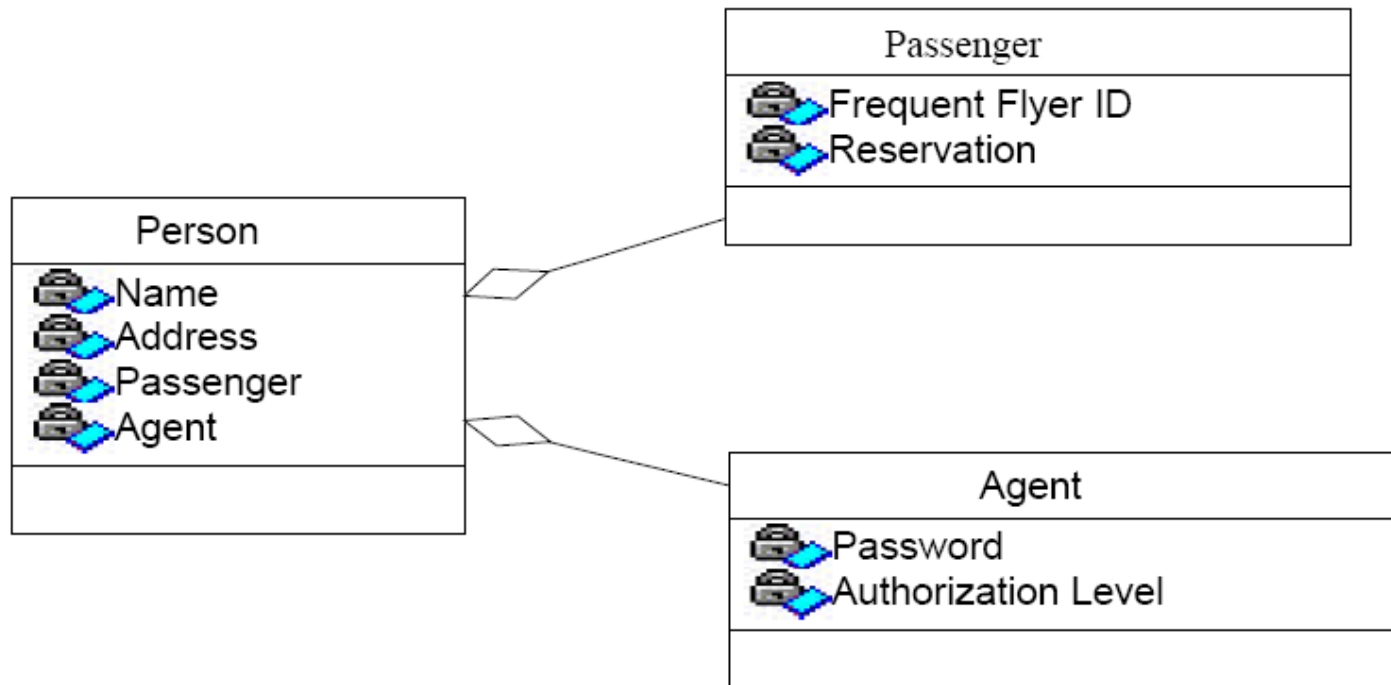
1. "Is a special kind of" not "is a role played by a"  
**Fail.** A passenger is a role a person plays. So is an agent.
2. Never needs to transmute  
**Fail.** A instance of a subclass of Person could change from Passenger to Agent to Agent Passenger over time
3. Extends rather than overrides or nullifies  
**Pass.**
4. Does not extend a utility class  
**Pass.**
5. 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!***

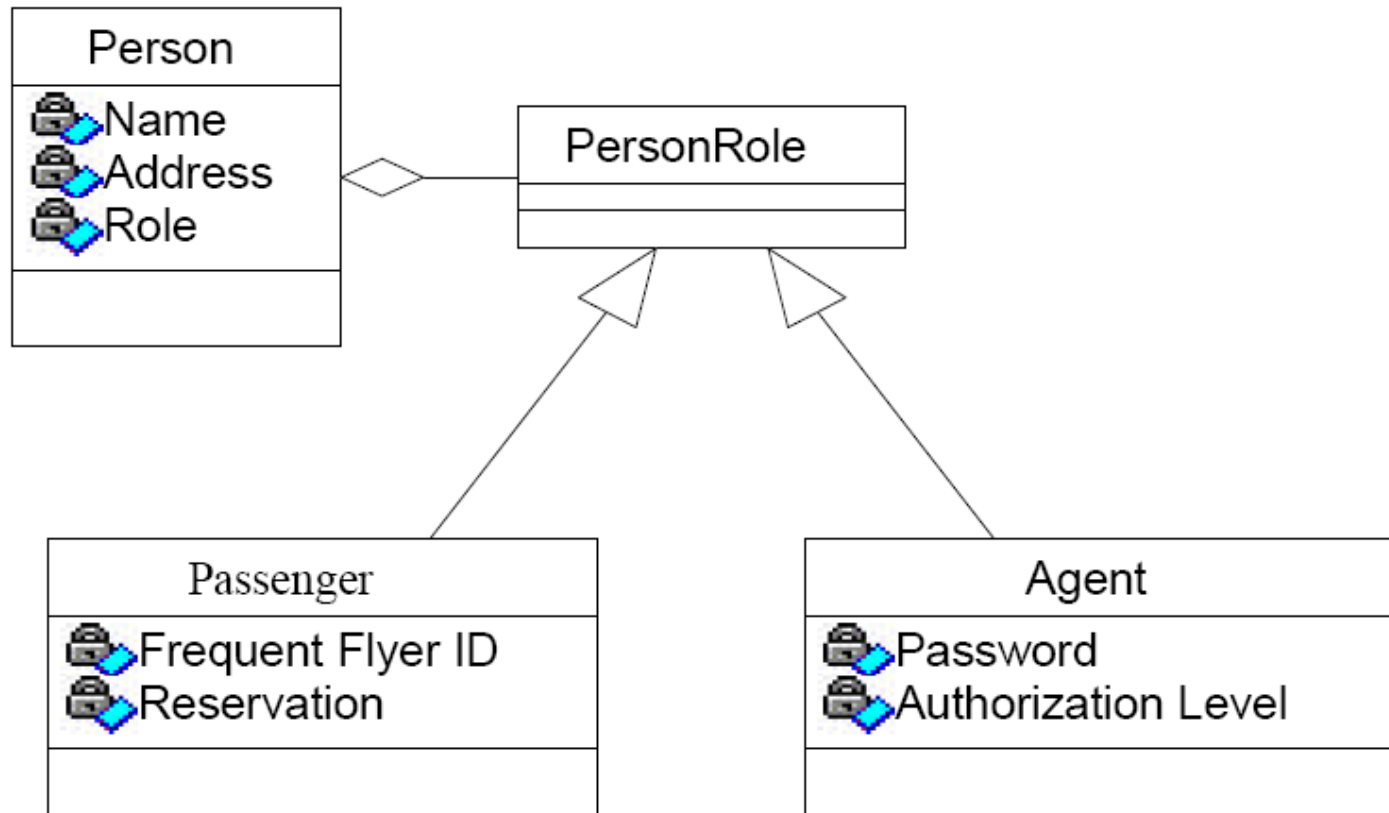


## Inheritance/Composition Example 1

*Composition to the rescue!*



## Inheritance/Composition Example 2



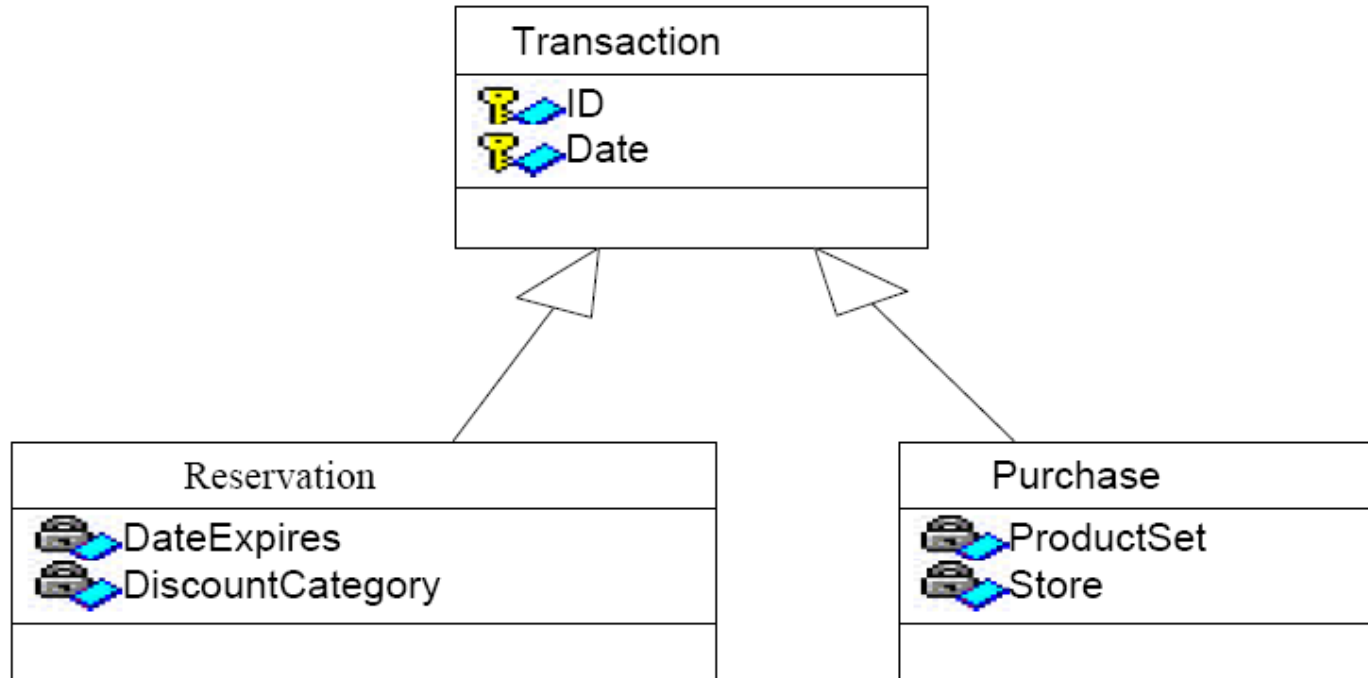
## Inheritance/Composition Example 2

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

*Inheritance is OK here*



## Inheritance/Composition Example 3



1. "Is a special kind of" not "is a role played by a"

**Pass.** Reservation and purchase are a special kind of transaction.

2. Never needs to transmute

**Pass.** A Reservation object stays a Reservation object; the same is true for a Purchase object.

3. Extends rather than overrides or nullifies

**Pass.**

4. Does not extend a utility class

**Pass.**

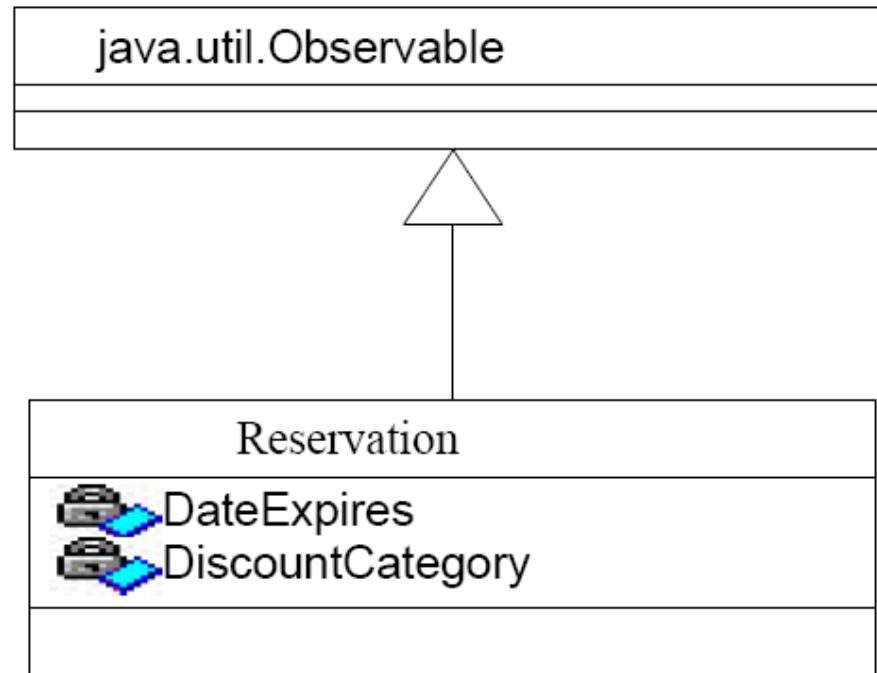
5. Within the Problem Domain, specializes a role, transaction or device

**Pass.** It's a transaction.

**Inheritance ok here!**



## Inheritance/Composition Example 4





1. "Is a special kind of" not "is a role played by a"

**Fail.** A reservation is not a special kind of observable.

2. Never needs to transmute

**Pass.** A Reservation object stays a Reservation object.

3. Extends rather than overrides or nullifies

**Pass.**

4. Does not extend a utility class

**Fail.** Observable is just a utility class.

5. 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!*



- 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*



## 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



# 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



## 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



# 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

- ♦ 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

- ◆ 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

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  
    }  
}
```

**But now we must modify each subclass of Part whenever the pricing policy changes!**



## Open-Closed Principle Example

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

/\*\*

**\* 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

- ◆ 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

*“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”*



# Liskov Substitution Principle (LSP)

- ◆ 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



# Liskov Substitution Principle (LSP)

- ◆ 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

- ◆ 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

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

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

```
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

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

- ◆ 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

- ◆ 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!



## Specification

A detailed description of design criteria for a piece of work

Java Specification Request (JSR) is the actual description of proposed and final specifications for the Java platform. JSRs are reviewed by the JCP and the public before a final release of a specification is made.

JSR 250 – Common Annotations for Java Platform

JSR 296 – Swing Application

JSR 127,252,314 – JSF

JSR 168,286,301 – Java Portlet Portability

JSR 160 – JMX

JSR 299 – Web Beans



## Framework

**Specific classifications that define a framework:**

- **Wrappers :**

- A wrapper simplifies an interface to a technology**

- reduces/eliminates repetitive tasks**

- increases application flexibility through abstraction**

- are often re-usable regardless of high level design considerations**

- **Architectures:**

- An architecture manages a collection of discrete objects**

- implements a set of specific design elements**

- **Methodologies**

- A methodology enforces the adherence to a consistent design approach**

- decouples object dependencies**

- are often re-usable regardless application requirements**



# GOF Design Patterns

A **wrapper** is way of repackaging a function or set of functions (related or not) to achieve one or more of the following goals (probably incomplete):

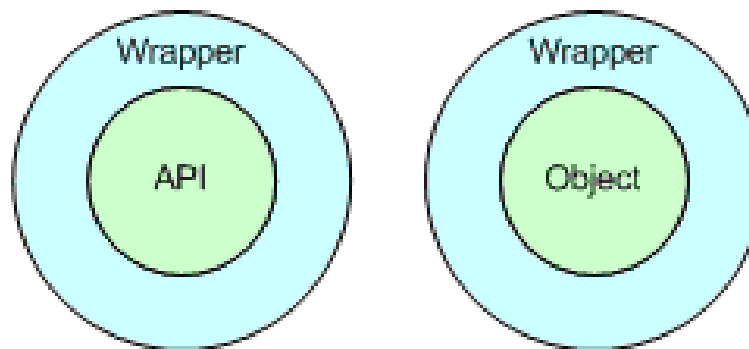
Simplification of use

Consistency in interface

Enhancement of core functionality

Collecting discrete processes into a logical association (an object)

Some Pictorial Examples



A Wrapper Provides Added Value

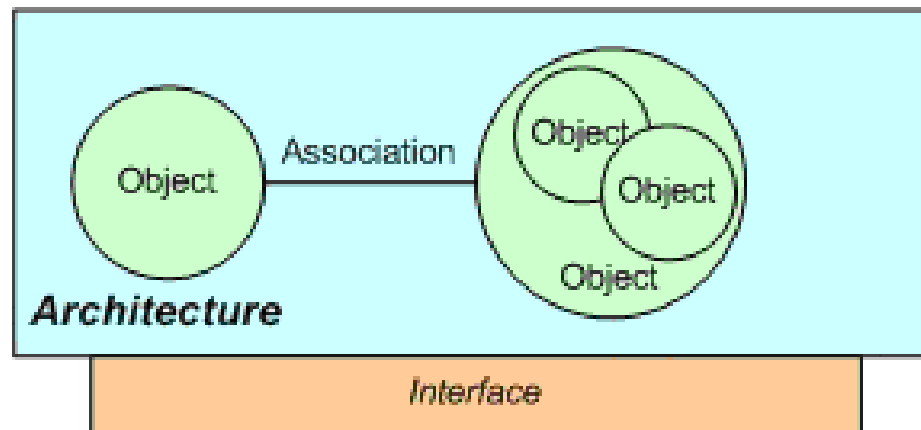


# GOF Design Patterns

An architecture is a style that incorporates specific design elements.

an architecture implements associations between objects--inheritance, container, proxy, collection, etc.

Architectures can and are useful because they create a re-usable structure (a collection of objects)



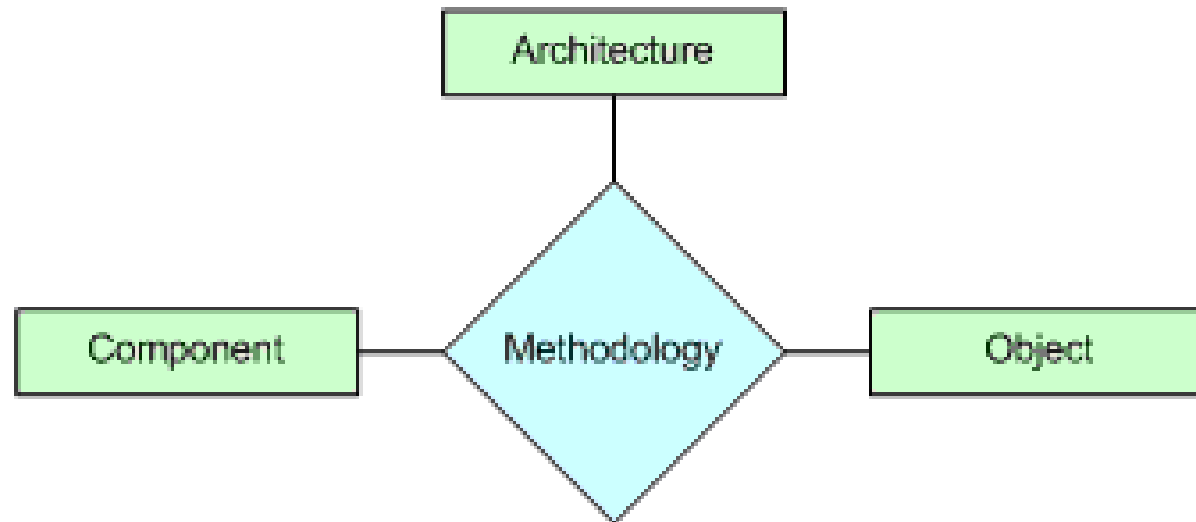
An Architecture Associates/Contains Objects  
(the interface is a wrapper of the architecture)





# GOF Design Patterns

**Methodology** is a body of practices, procedures, and rules used by those who work in a discipline.



A Methodology Defines The Interaction Between  
Architectures, Components, And Objects



## Design Patterns

Design patterns are both architectures and methodologies.

The structural patterns are more architecture, when the creational and behavioral patterns are more methodologies because their usage enforces a particular method of interaction.

“A Solution to a recurring problem for a particular context”



## Pattern Categories

**Patterns cover various ranges of scale and abstraction.**

### 1. Architectural Patterns

High-level patterns to help to specify the fundamental structure of an software system.

### 2. Design Patterns

Medium-scale patterns to organise subsystem functionality in application domain independent way.

### 3. Idioms

Low-level patterns to solve implementation-specific problems.



## Architectural Patterns

### Definition:

An architectural pattern expresses a fundamental structural organization Schema for software systems. It provides a set of predefined subsystems, Specifies their responsibilities, and includes rules and guidelines for Organizing the relationships between them.

This architecture needs patterns



## Architectural Patterns

Architectural patterns can be subdivided into four sub-categories

According to their properties:

- From Mud to Structure
  - > Layers, Pipes and filters, Blackboard
- Distributed Systems
  - > Broker, Pipes and Filters, Microkernel
- Interactive Systems
  - > MVC, IOC
- Adaptable Systems
  - > Reflection, Microkernel



## Layers

**The Layer patterns structures system into groups of subtasks  
Working on particular level of abstraction.**

Benefits :

Reuse of layers

Support for standardization

Dependencies are kept local

Exchangeability

Examples : OSI, Internet Protocol Suite

Liabilities:

Changing behaviour

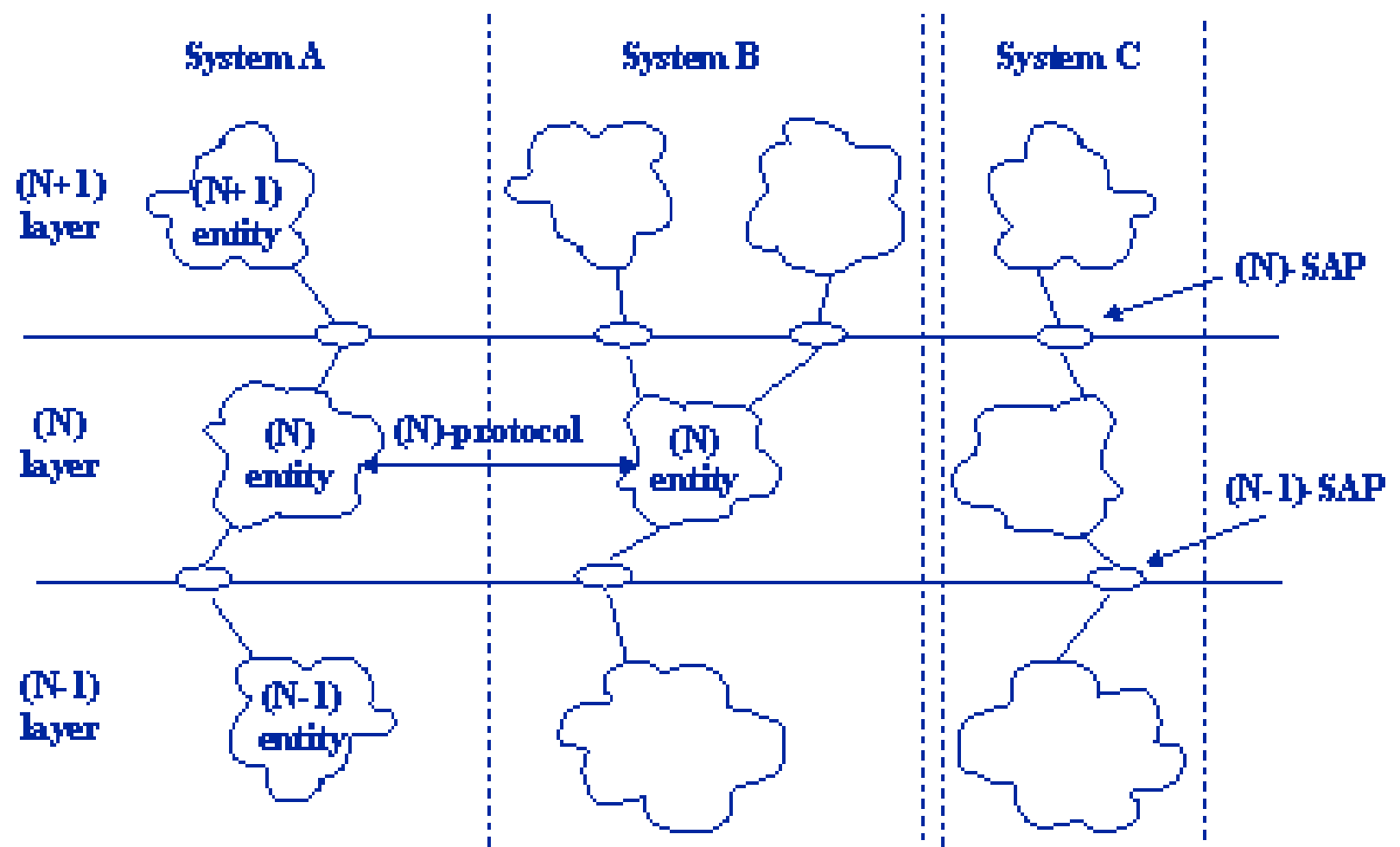
Lower efficiency

Unnecessary work

Difficulty of establishing  
correct granularity



# Layers



# GOF Design Patterns

---

## Broker

**A broker coordinates communication between distributed software Systems in order to enable remote use of services.**

### Benefits:

**Location transparency**

**System independency**

**Interoperability and portability**

**Changeability and reusability**

**Run-time configuration**

### Liabilities:

**Restricted efficiency**

**Lower fault tolerance**

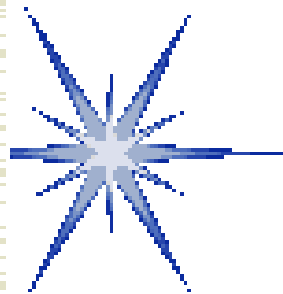
**Testing & debugging**

### Examples

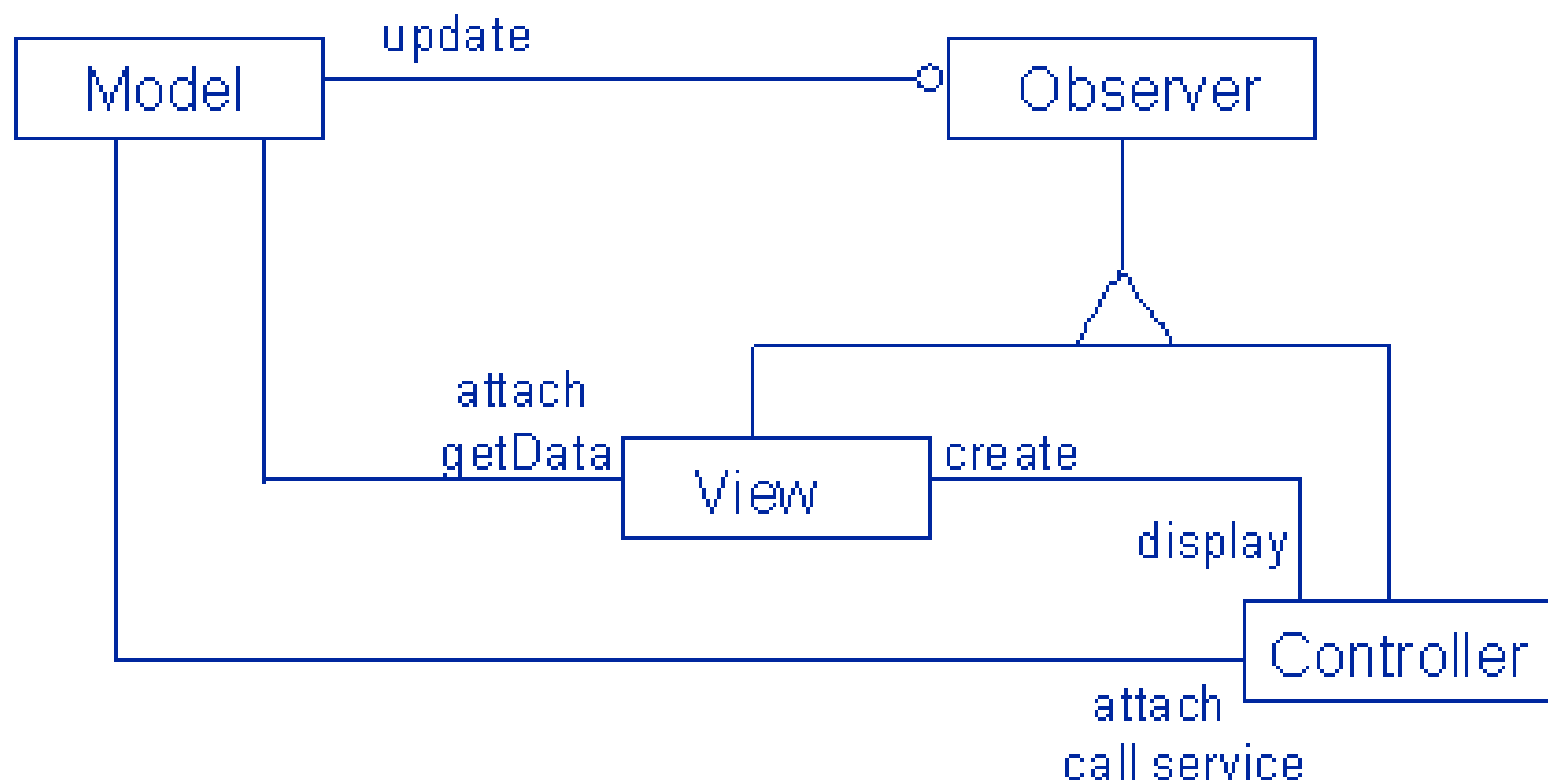
- CORBA (Common Object Request Broker Architecture)
- OLE/ActiveX , SOM/DSOM(IBM) and WWW







# Model-View-Controller



## Design Patterns

**A design pattern describes a commonly-recurring structure of  
Communicating components that solve a general design problem  
In a particular context**



## Design Patterns

**Design patterns can be grouped into categories of related patterns:**

- Structural Decomposition  
Whole-part, composite
- Organization of work  
Chain of Responsibility, Command, Mediator
- Access Control  
Proxy, Façade, Iterator
- Management  
Command processor, View Handler, Memento
- Communication  
Forward-Receiver, Client-Dispatcher-Server, Publisher-Subscriber



## **Publisher-Subscriber**

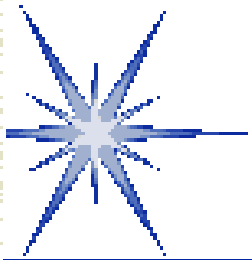
**Publisher-Subscriber pattern (Observer) helps to keep the state  
Of cooperating components synchronized by enabling one-way  
Propagation of changes**

**Examples :**

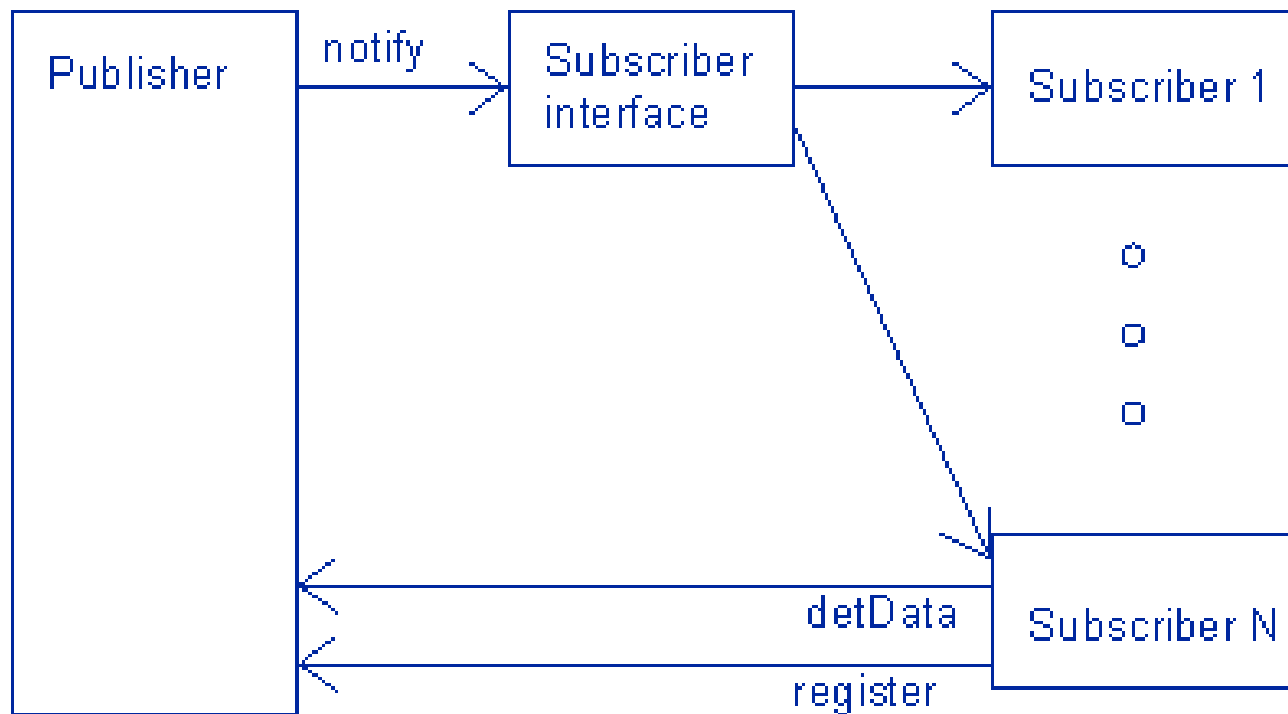
**Even model**

**Push-technology in web (Webcasting)**





## Publisher-Subscriber



# GOF Design Patterns

## Idioms (Coding Patterns)

**An Idiom is a low-level pattern specific to a programming language.**

**An Idiom describes how to implement particular aspects of Components or the relationships between them using the Features of the given language.**

**In practice ;**

**Naming conventions**

**Source code formats**

**Memory management**



# GOF Design Patterns

---

## Idioms (Coding Patterns)

### Notes :

No clear line between design patterns and idioms

Less portable implementations of design patterns

Experienced programmers apply idioms

Documented idioms can help problem solving, team work and

Training of new programmers.

### Examples ;

Counted Pointer(C++)

Implementation of Singleton design patterns



### Class vs. interface Inheritance

- ◆ A class as a polymorphic type (like an array of type Animal or a method that takes a Canine argument) the objects can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type.
- ◆ For example, an argument of type Canine can accept a Wolf and Dog, but not a Cat or Hippo.



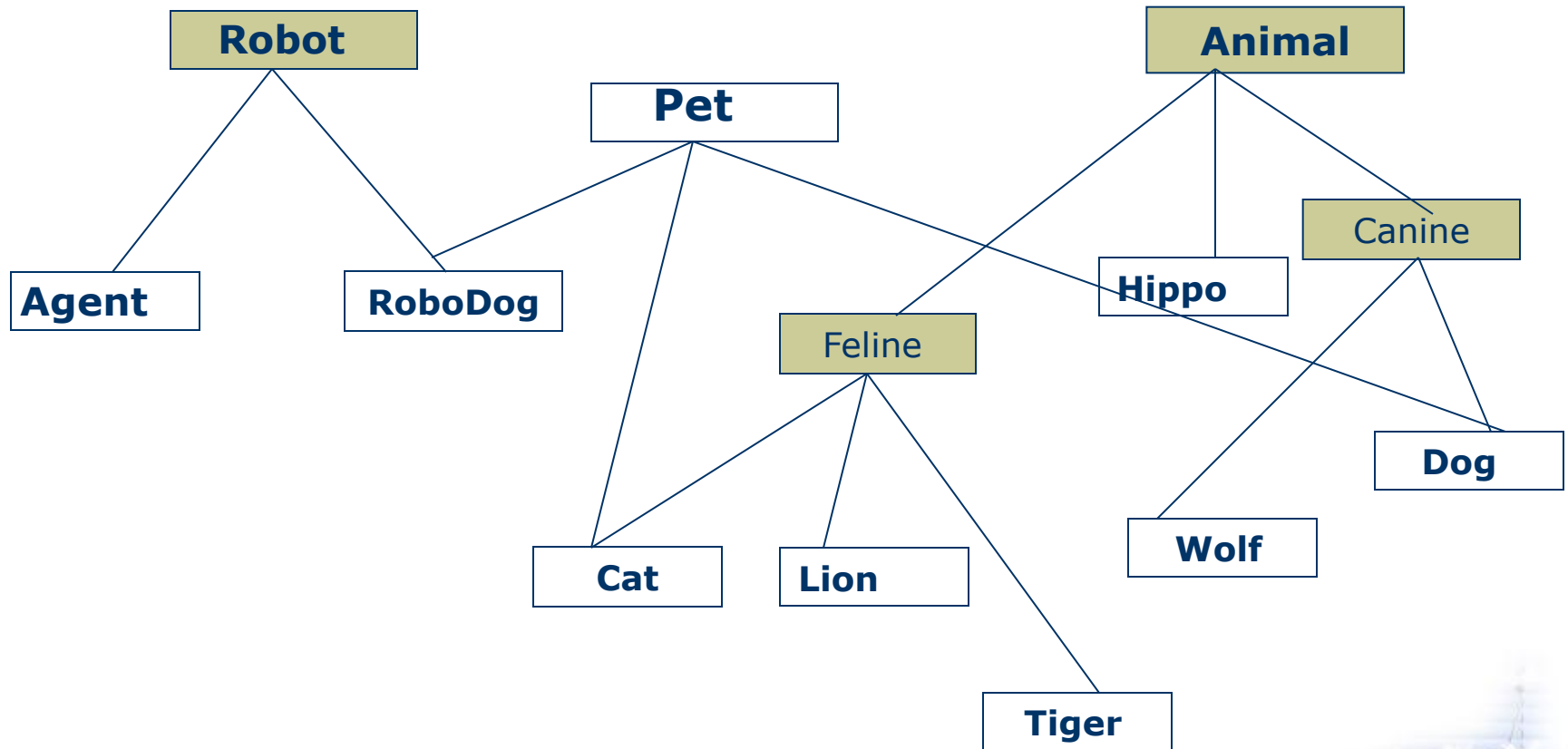


# Class vs. interface Inheritance

- ◆ An interface as a polymorphic type (like an array of Pets), the objects can be from anywhere in the inheritance tree.
- ◆ The only requirement is that the objects are from a class that implements the interface.
- ◆ Allowing classes in different inheritance trees to implement a common interface is crucial in the java API.
- ◆ For example we want an object to save its state to a file? Implement the Serializable interface.
- ◆ We want objects to run their methods in a separate thread of execution? Implement Runnable.



Classes from different inheritance trees can implement the same interface.



# GOF patterns

- ◆ The GOF broke the patterns into two scopes:
- ◆ Class patterns require implementation inheritance(extends)
- ◆ Object patterns should be implemented using nothing but interface inheritance(implements)



# GOF Team

- ◆ Erich Gamma
- ◆ Richard Helm
- ◆ John Vlissides
- ◆ Ralph Johnson



# GOF Design Patterns

		Purpose		
		Creational	Structural	Behavioral
S C O P E	Class	Factory Method	Class Adapter	Interpreter Template Method
	O B J E C T	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



# GOF Design Patterns

## 3 Creational Patterns in a Nutshell

Creational Patterns involve object instantiation and all provide a way to decouple a client from the objects it need to instatiate.

### Abstract Factory

- ▶ Provides one level of interface higher than the factory pattern. It is used to return one of several factories.
- ▶ Provides a class library of products, exposing interface not implementation.
- ▶ Suppose you need to write a program to show data from two different places - local or a remote database. You need to make a connection to a database before working on the data. You may use abstract factory design pattern to design the interface

### Builder Pattern

- ▶ Construct a complex object from simple objects step by step.
- ▶ Want to decouple the process of building a complex object from the parts that make up the object.
- ▶ Give you finer control over the construction process.
- ▶ E.g. To build a house, we will take several steps: build foundation, build frame, build exterior, build interior.



# GOF Design Patterns

## 3 Creational Patterns

### Factory

- Provides an abstraction or an interface and lets subclass or implementing classes decide which class or method should be instantiated or called, based on the conditions or parameters given.
- To paint a picture, you may need several steps. A shape is an interface. Several implementing classes may be designed like circle, square etc.

### Prototype

- Cloning an object by reducing the cost of creation.
- Dynamic loading is a typical object-oriented feature and prototype example. For example, overriding method is a kind of prototype pattern.

### Singleton

- One instance of a class or one value accessible globally in an application.
- Make a method or a variable public or/and static.
- Access to the instance by the way you provided.
- E.g. Printer

### Object Pool

- Reuse and share objects that are expensive to create.



# GOF Design Patterns

## 4 Structural Patterns

Structural patterns let you compose classes or objects into largest structures.

### Adapter Pattern

► Convert the existing interfaces to a new interface to achieve compatibility and reusability of the unrelated classes in one application. E.g. Resource Adapters

### Bridge

► The Bridge pattern is used to separate the interface of class from its implementation, so that either can be varied separately.

► You can vary or replace the implementation without changing the client code.  
E.g. Corba

### Composite

► Composite to build part-whole hierarchies or to construct data representations of trees.

► In summary, a composite is a collection of objects, any one of which may be either a composite, or just a primitive object.

### Decorator

► The Decorator pattern provides us with a way to modify the behavior of individual objects without having to create a new derived class.





# GOF Design Patterns

## 4 Structural Patterns

E.g. Design a class that prints a number. Create a decorator class that adds a text to the Number object to indicate the number eg. a random number. We can subclass the Number class to achieve the same goal. But the decorator pattern provides us an alternative way.

### Façade

Make a complex system simpler by providing a unified or general interface, which is a higher layer to these subsystems. E.g. JDBC interfaces

### Flyweight

Make instances of classes on the fly to improve performance efficiently, like individual characters or icons on the screen. E.g File System

### Proxy

Use a simple object to represent a complex one or provide a placeholder for another object to control access to it. E.g. Security Proxy.



## 5 Introduction to Behavioral Pattern

Behavioral pattern is concerned with how classes and objects interact and distribute responsibility.

(OR)

Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

These patterns deals with –

- Assignment of responsibilities between objects
- Encapsulating behavior in an object and
- Delegating requests to objects



# GOF Design Patterns

## 5.1 Classification of Creational Patterns

Creational Patterns	
<b>Abstract Factory</b>	<b>Creates an instance of several families of classes</b>
<b>Factory method</b>	<b>Creates an instance of several derived classes</b>
<b>Builder</b>	<b>Separates object construction from its representation</b>
<b>Singleton</b>	<b>A class in which only a single instance can exist</b>
<b>Prototype</b>	<b>A fully initialized instance to be copied or cloned</b>
<b>Object Pool</b>	<b>Reuse and share objects that are expensive to create.</b>



# GOF Design Patterns

## 5.1 Classification of Structural Patterns

Structural Patterns	
<b>Adapter</b>	<b>Match interfaces of different classes</b>
<b>Bridge</b>	<b>Separates an object's abstraction from its implementation</b>
<b>Composite</b>	<b>A tree structure of simple and composite objects</b>
<b>Decorator</b>	<b>Add responsibilities to objects dynamically</b>
<b>Façade</b>	<b>A single class that represents an entire subsystem</b>
<b><i>Flyweight</i></b>	<b>A fine-grained instance used for efficient sharing</b>
<b><i>Proxy</i></b>	<b>An object representing another object A placeholder for a real object</b>



# GOF Design Patterns

## 5.1 Classification of Behavioral Patterns

Behavioral Patterns	
<b>Chain of Responsibilities</b>	A way of passing a request between a chain of objects.
<b>Command</b>	Encapsulate a command request as an object.
<b>Iterator</b>	Sequentially access the elements of a collection.
<b>Interpreter</b>	A way to include language elements in a program.
<b>State</b>	Alter an object's behavior when its state changes.
<b><i>Mediator</i></b>	Defines simplified communication between classes And simplifies maintenance of the system by Centralizing control logic
<b><i>Memento</i></b>	Capture and restore an object's internal state(like serialization).
<b><i>Observer</i></b>	A way of notifying change to a number of classes.
<b><i>Strategy</i></b>	Encapsulates an algorithm inside a class.
<b><i>Template Method</i></b>	Defer the exact steps of an algorithm to a subclass.
<b><i>Visitor</i></b>	Defines a new operation to a class without change

## Class Inheritance

Class Inheritance, or subclassing, allows a subclass Implementation to be defined in terms of the parent class implementation.

This type of reuse is often called **white-box reuse**.

This term refers to the fact that with inheritance, the parent class Implementation is often visible to the subclasses.



## Object composition

Object composition is a different method of reusing functionality.

Objects are composed to achieve more complex functionality.

This approach requires that the objects have well-defined Interface since the internals of the objects are unknown.

Because objects are treated only as “black-boxes”.

This type of reuse is often called black-box reuse.



## Class Inheritance Advantages

Class inheritance is done statically at compile-time and is Easy to use.

## Class Inheritance Disadvantages

The subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable.

“Inheritance breaks encapsulation” – one way around this problem is to only inherit from abstract classes.

Another problem with class inheritance is that the implementation inherited from a parent class can't be changed at run-time.





# GOF Design Patterns

In object composition, functionality is acquired dynamically at Run-time by objects collecting references to other objects.

## **Object composition Advantages:**

- Implementations can be replaced at run-time.
- Objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type.
- Since each object is defined in terms of object interfaces, there are less implementation dependencies.

## **Object composition Disadvantages:**

- The behavior of the system may be harder to understand just by looking at the source code.
- A system using object composition may be very dynamic in nature so it may require running the system to get a deeper understanding of how the different objects cooperate.



**Object composition** should be favored over inheritance.

Object composition promotes smaller, more focused classes and Smaller Inheritance hierarchies.

A design based on object composition will have less classes, but more objects. The behavior of such a system depends more on the interrelationships between objects rather than being defined in a particular class.

Due to the flexibility and power of object composition, most Design patterns emphasize object composition over Inheritance whenever it is possible.



**Overuse of inheritance**, resulting in large inheritance hierarchies that can become hard to deal with.

However, inheritance is still necessary. We can't always get all the necessary functionality by assembling existing components. This is because "the set of components is never quite rich enough in practice"

Inheritance can be used to create new components that can composed with old components. As a result, the two methods work together.



## Inheritance vs. Composition

- ◆ Both creates a coupling
- ◆ Which of the two couplings are more maintainable?
- ◆ Aspects than can change in any class are:
  - Changing method signature
  - Adding a method
  - Deleting a method
  - Changing the implementation of a method.



Changing the method signature:

Changing the method signature effects both the Cases  
(Inheritance and Composition)



Adding a method:

Adding a method will not have a direct effect, unless we discover that the new method does something an old method did in a better way. In this case we may want to refactor our code to use the new method.

**Implication is same in both Cases.**



Deleting a method:

If a method is deleted in superclass or aggregate, then we will have to refactor if we were using that method.

Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of subclass will no longer be overriding superclass methods.

These methods will become independent methods in their own right.



Changing the implementation of a method

Changing the implementation of a method should not have any effect either because the implementation is supposed to be black box.

**Inheritance creates a tighter coupling.**

Ofcourse here we are assuming that we do not need to use polymorphism and we are using Inheritance purely for code reuse.





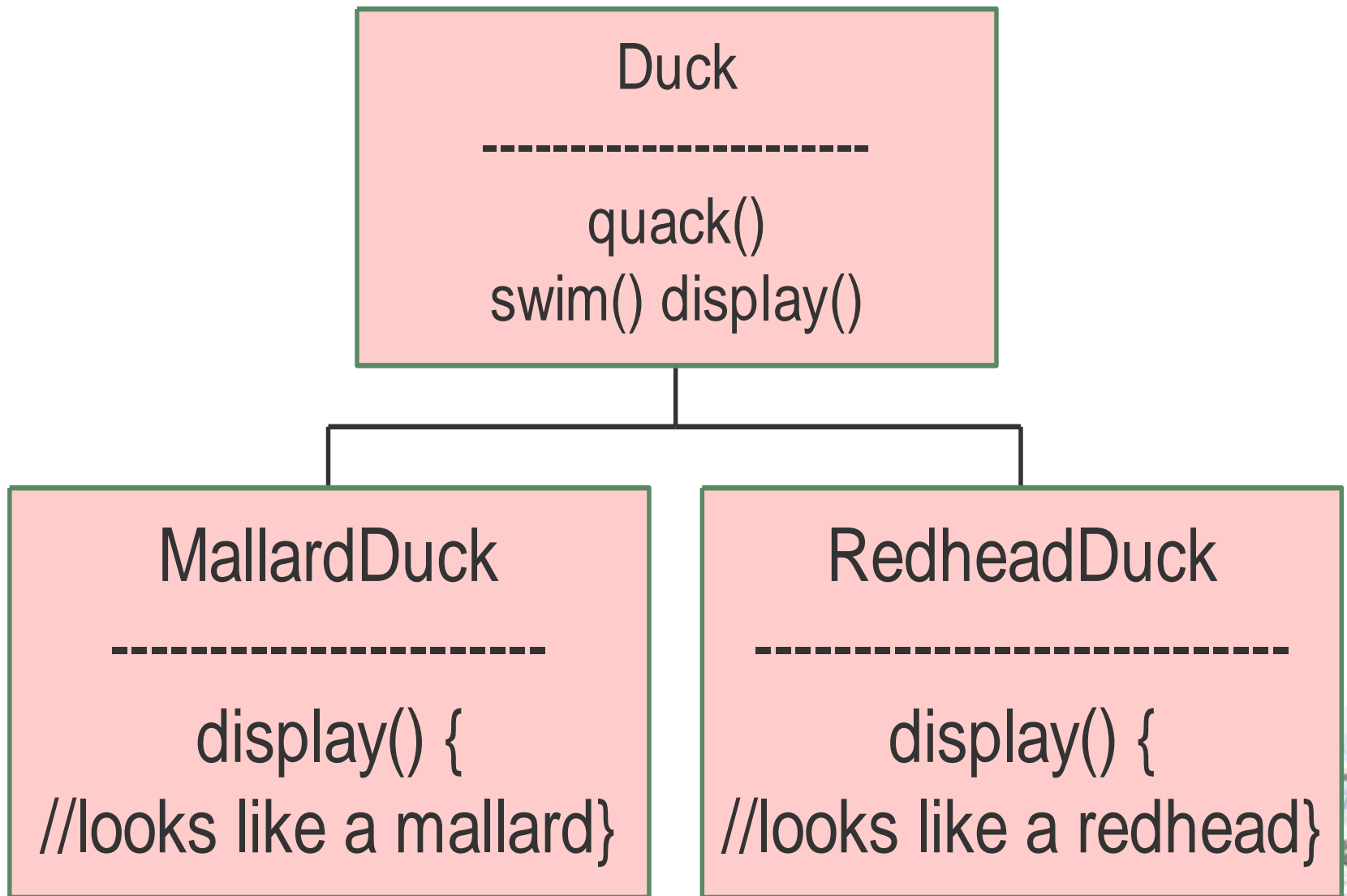
## Object Composition Vs Inheritance

Object composition and inheritance are two techniques for reusing functionality in object-oriented systems.



# Duck Pond Simulation Game



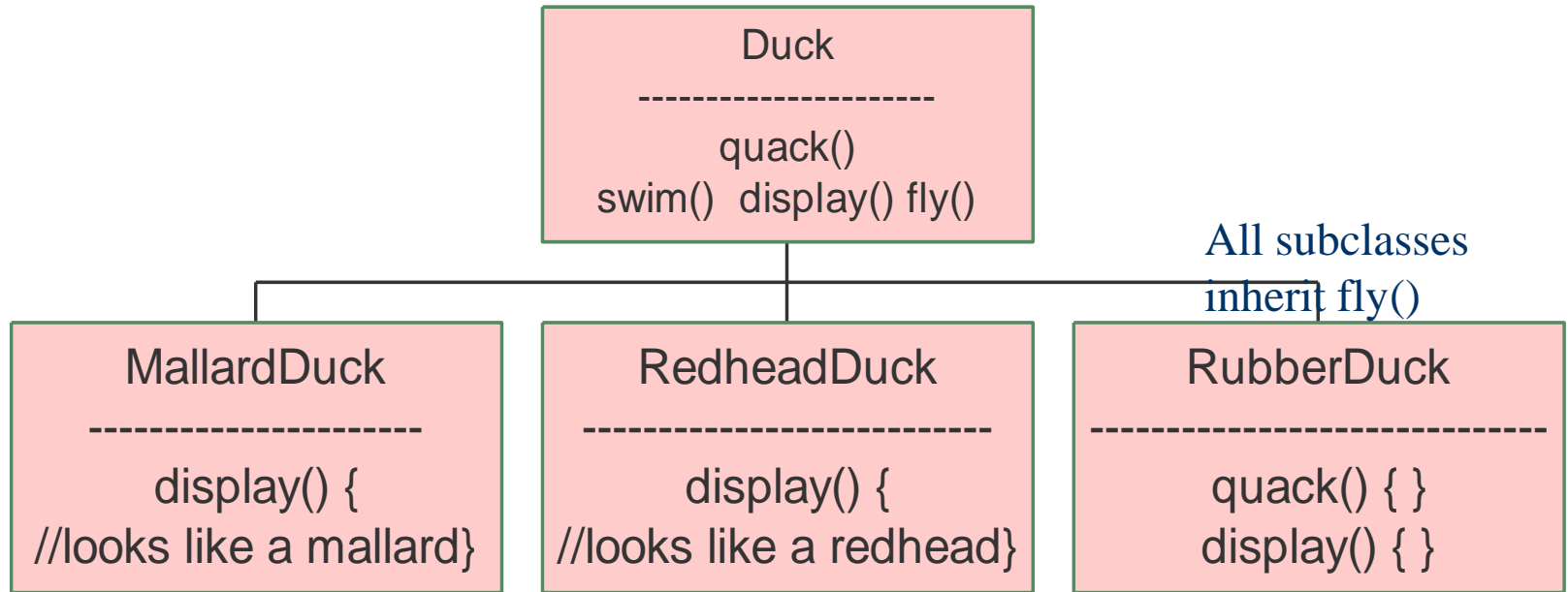


In the last year, the company has been under increasing pressure from competitors.

Time for a big invocation!!!!



# GOF Design Patterns



But something went horribly wrong....

Rubber duckies flying around the screen!

Oh!.. By putting fly() in the superclass, all ducks got flying ability, including those that shouldn't...

The great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance....



I could always just  
override the fly() method  
in rubber duck...

## RubberDuck

```
Quack()  
display()  
fly() {  
    //override to do nothing  
}
```



# GOF Design Patterns

But then what happens  
when we add wooden decoy  
ducks to the program? They aren't  
supposed to fly or quack...

## DecoyDuck

```
Quack() {  
    //override to do nothing }  
display()  
fly() {  
    //override to do nothing  
}
```



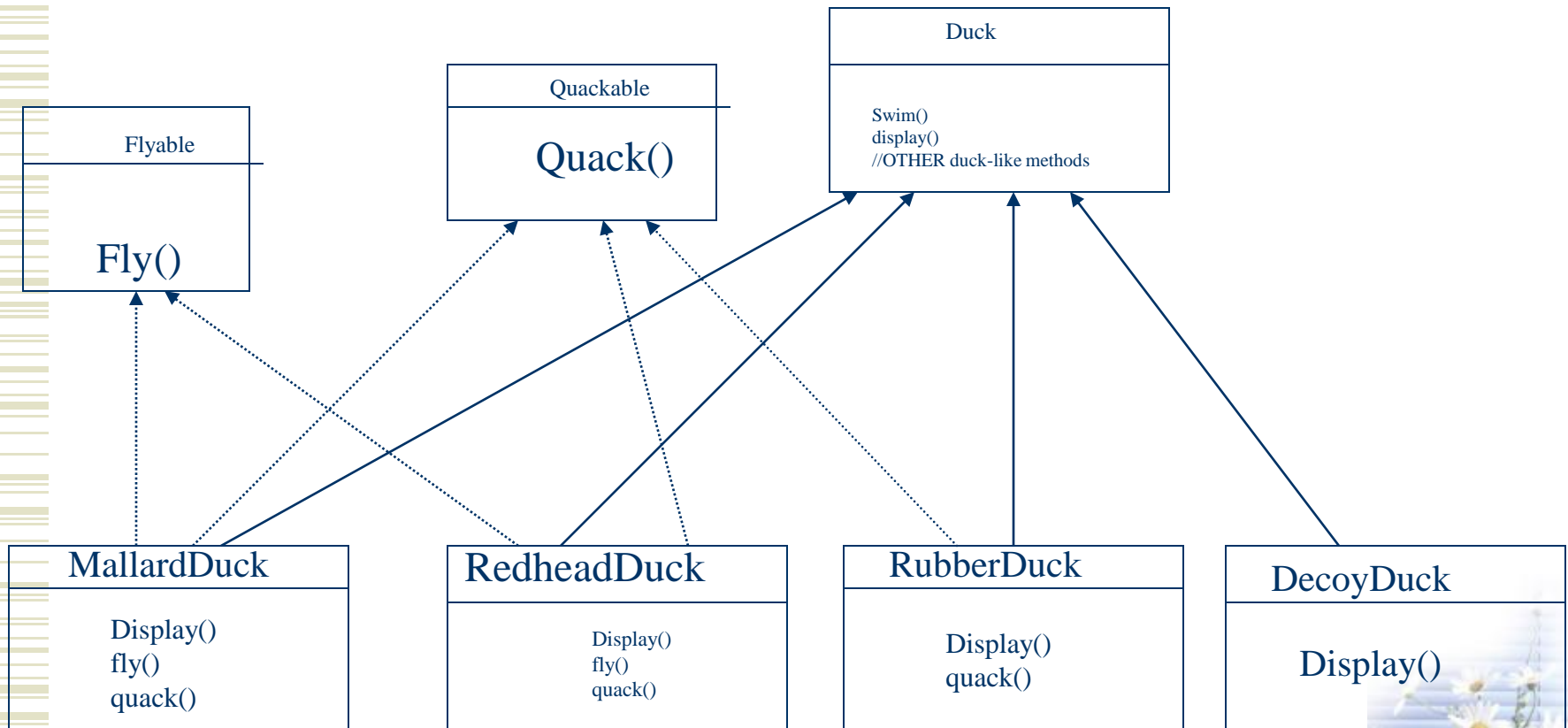


How about an interface?



# GOF Design Patterns

What do YOU think about this design?



That is, like, the dumbest idea we have come up with. We can say, “duplicate code”? If you thought having a override a few methods was bad, how are you gonna feel when you need to make a little change to the flying behavior.. In all 48 of the flying Duck subclasses? And of course there might be more than one kind of flying behavior even among the ducks that do fly....



Yes we understand.... Now we know only  
God can help us...  
Oh..Great Design Pattern has come to  
solve this problem?



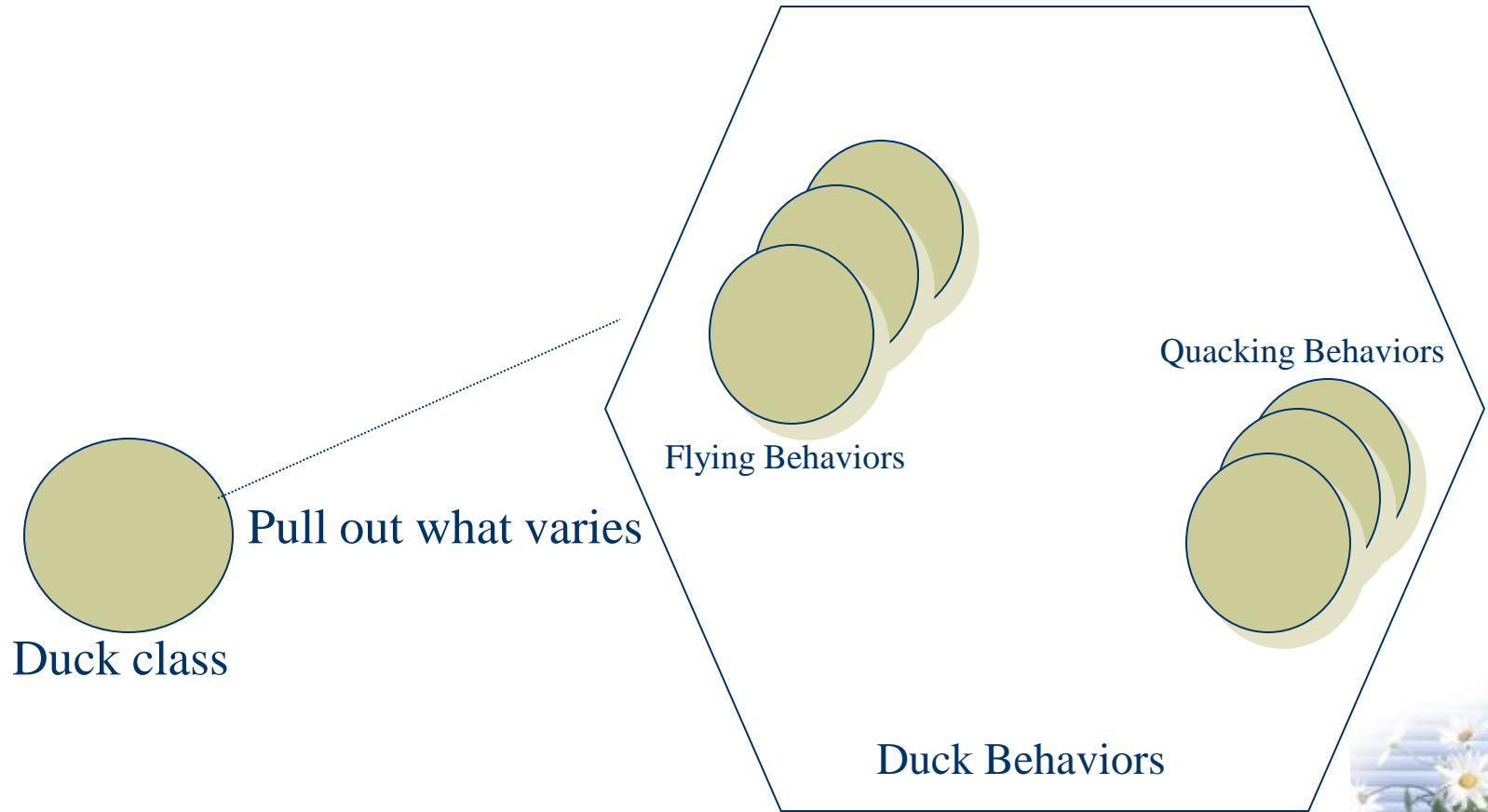
## Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

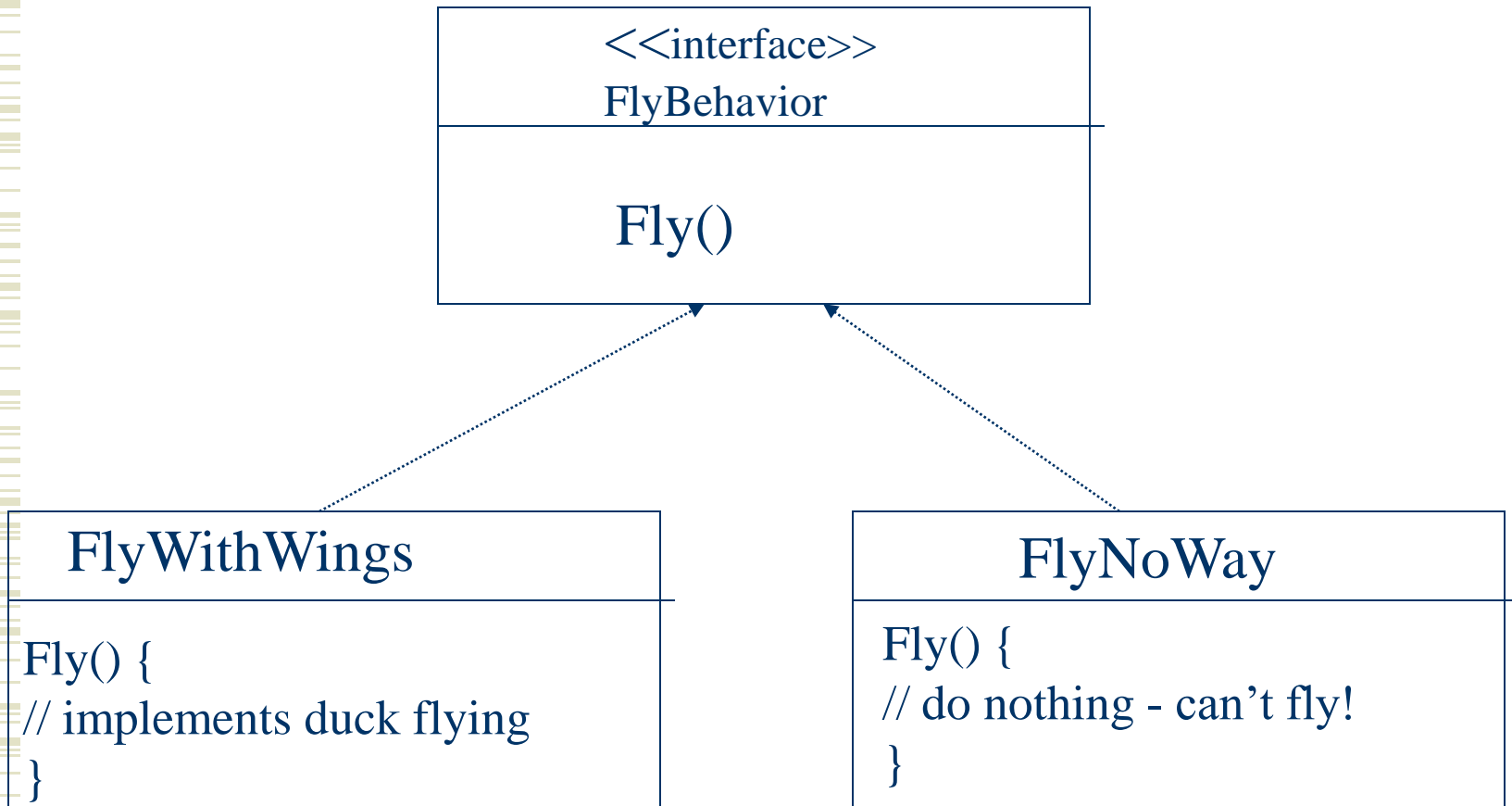


# GOF Design Patterns

Separating what changes from what stays the same



# GOF Design Patterns

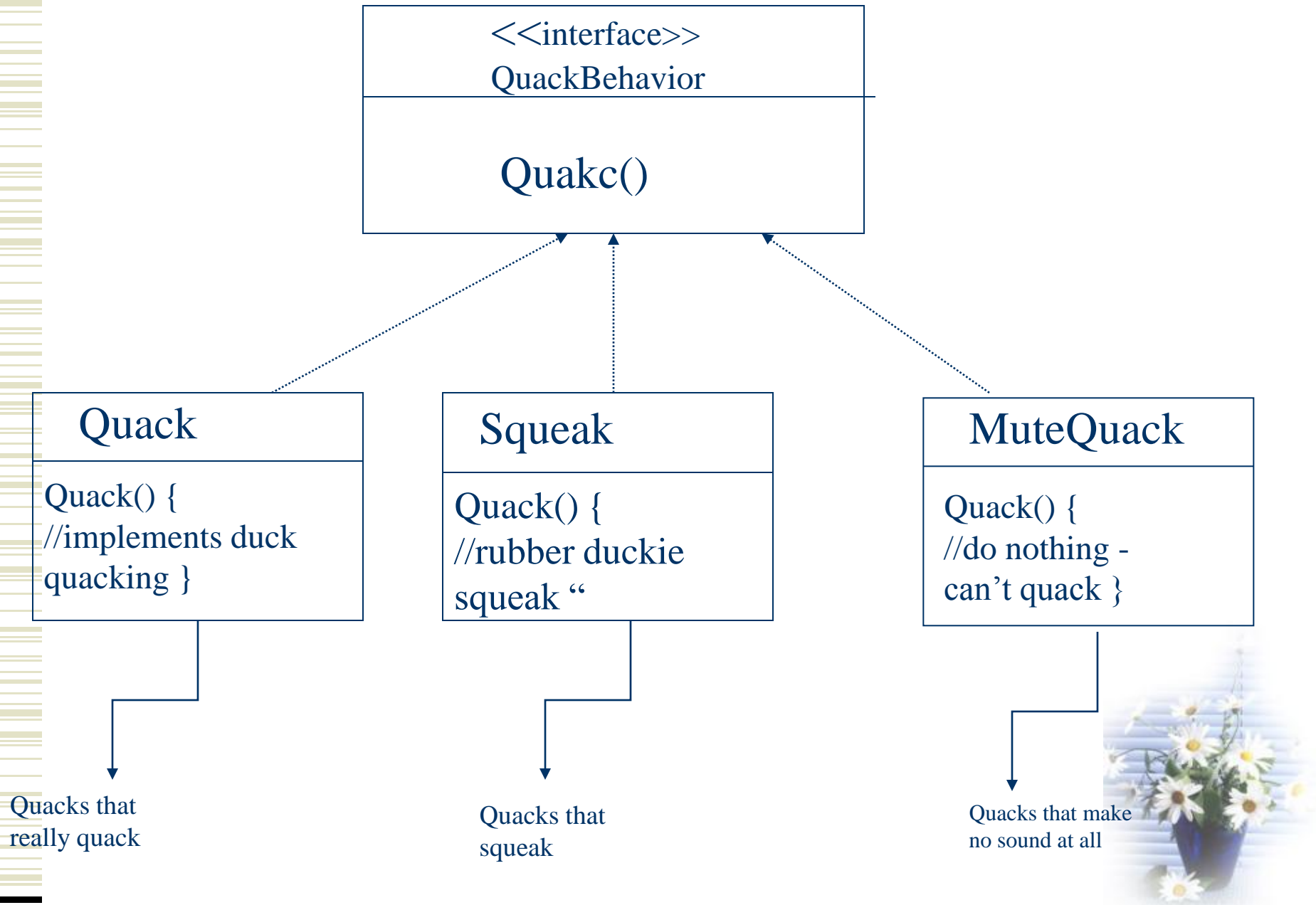


The implementation  
of flying for all ducks  
that have wings

Implementation of all  
ducks that can't fly



# GOF Design Patterns





With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with Inheritance.

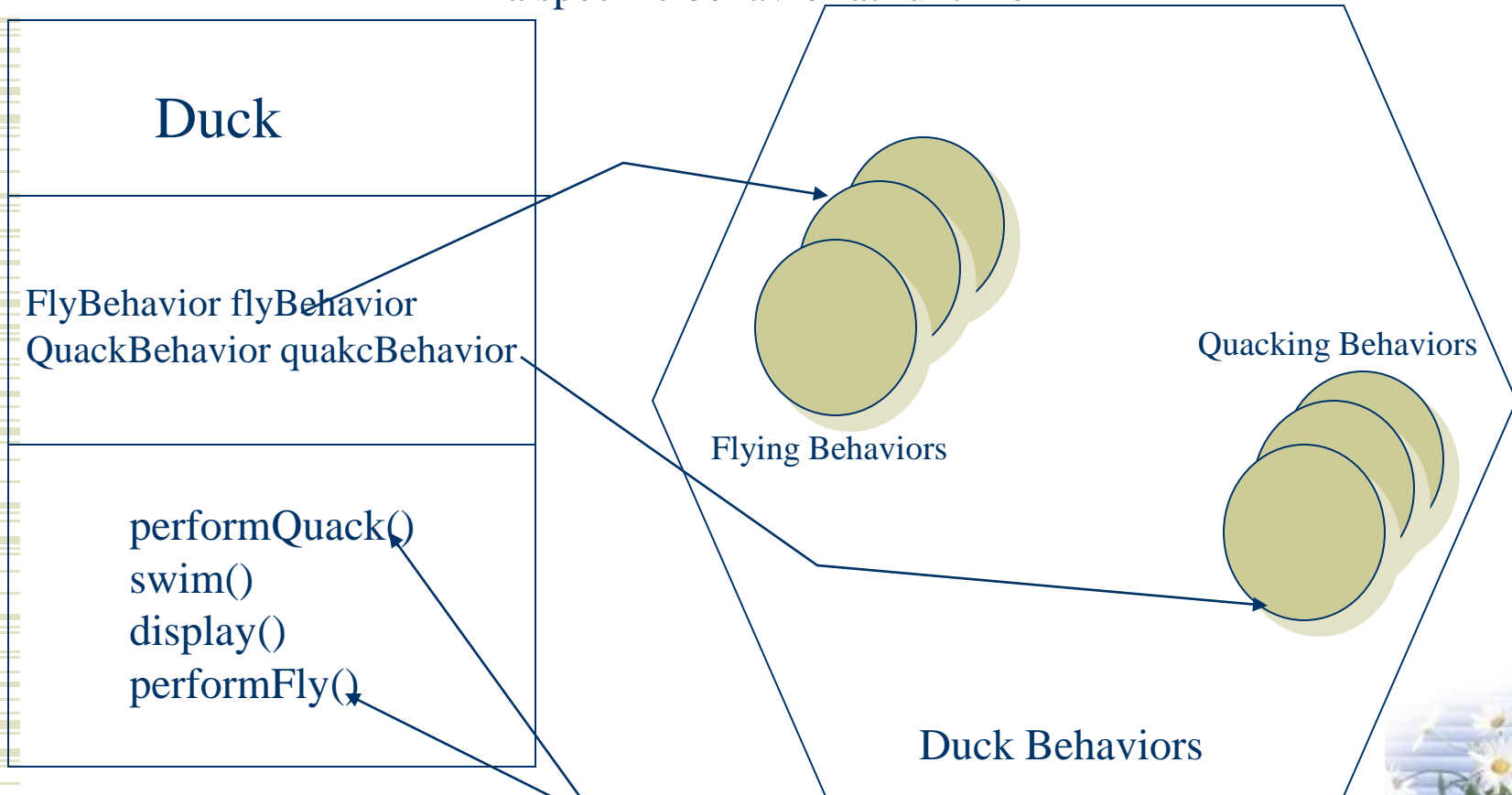


# GOF Design Patterns

## Integrating the Duck Behavior

1. First we'll add two instance variables :

Instance variables hold a reference to a specific behavior at runtime



These methods replace  
`fly()` and `quack()`



2. Now we implement performQuack() :

```
public class Duck {  
    QuackBehavior quackBehavior;  
    //more
```

```
public void performQuack() {  
    quackBehavior.quack(); } }
```

Pretty simple.. To perform the quack, a Duck just allows the object that is referenced by quackBehavior to quack for it. In this part of the code we don't care what kind of object it is, all we care about is that it knows how to quack()!



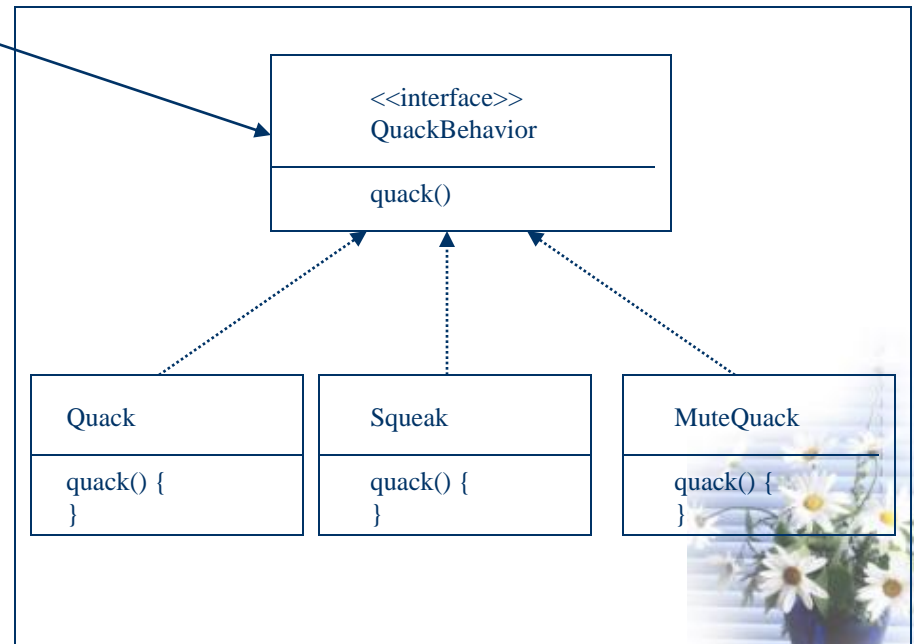
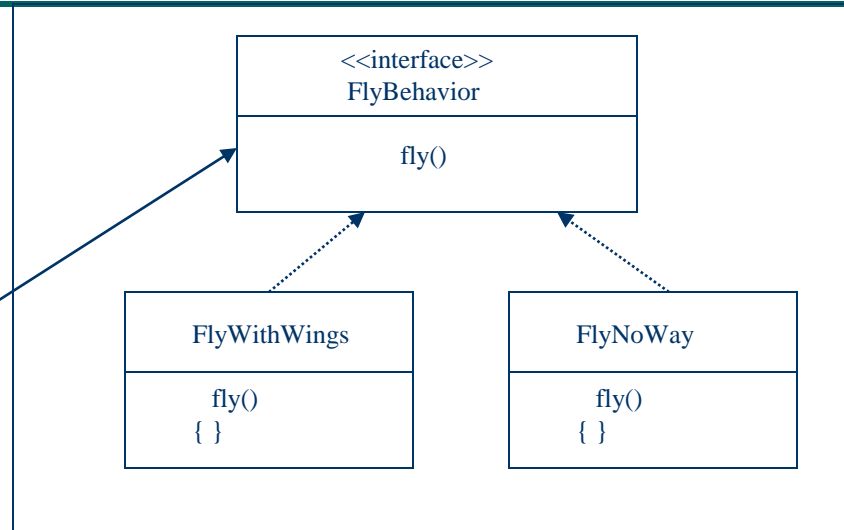
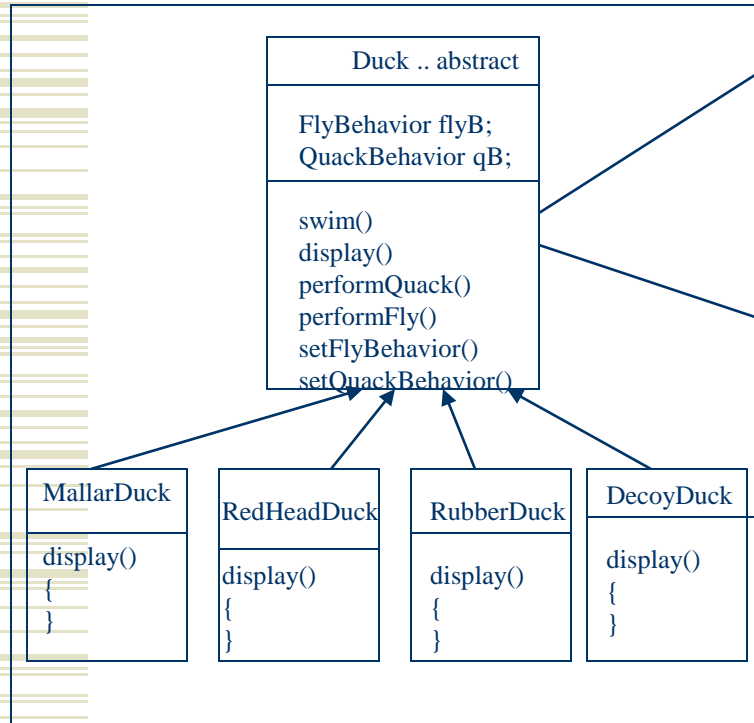
# GOF Design Patterns

3. Okay, time to worry about how the flyBehavior and quackBehavior instance variables are set. Let's take a look at the MallardDuck class:

```
Public class MallardDuck extends Duck {  
    public MallardDuck() {  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() {  
        System.out.println("I'm a real Mallard duck"); } } }
```



# GOF Design Patterns



# HAS-A can be better than IS-A

Design Principle

Favor composition over inheritance\*

\*Change the behavior at runtime



Great.... We made wonders!!!!!!

Unknowingly we desinged a pattern????

**The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it



## OO Principles

Encapsulate what varies

Favor composition over Inheritance

Program to interfaces, not implementation





## DESIGN PUZZLE



# GOF Design Patterns

Character

WeaponBehavior weapon;

flight();

Queen

fight() {....}

KnifeBehavior

useWeapon() { }

BowAndArrowBehavior

useWeapon() { }

Troll

fight() {....}

WeaponBehavior

useWeapon();

King

fight() {....}

AxeBehavior

useWeapon() { }

Knight

fight() {....}

setWeapon(WeaponBehavior w)

```
{  
  this.weapon = w;  
}
```




SwordBehavior

useWeapon() {....}



Your task :

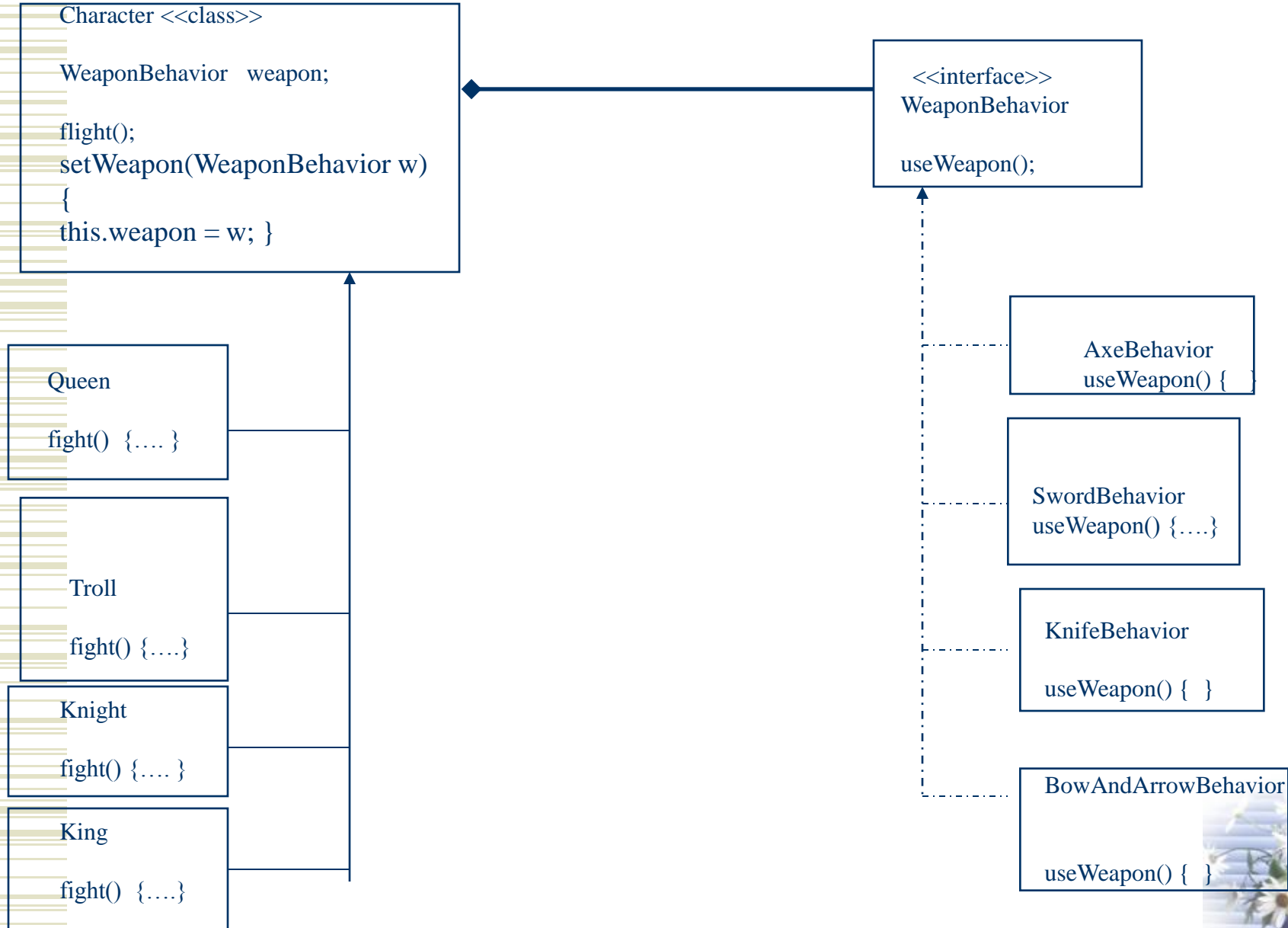
1. Arrange the classes.
2. Identify one abstract class, one interface and eight classes
3. Draw arrows between classes

- Draw  For inheritance (“extends”)
- Draw  For interface (“implements”)
- Draw  For HAS-A

4. Put the method setWeapon() into the right class.



# GOF Design Patterns



## OO Principles

Encapsulate what varies.

Favor composition over  
Inheritance.

Program to interfaces, not  
implementations.

## OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

## OO Patterns

Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Factory Method – Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory method lets a class defer instantiation to the subclasses.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

## OO Patterns

Singleton – Ensure a class only has one instance  
and provide a global point of access to it.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

## OO Patterns

Prototype – Hides the complexities of making  
new instances from the client.





# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

## OO Patterns

Builder – Use the Builder pattern to encapsulate the construction of a product and allow it to be constructed in steps.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

## OO Patterns

Adapter – Converts the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interface.

Facade – Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over  
Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

Don't call us, we'll call you

## OO Patterns

Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

Don't call us, we'll call you

Strive for loosely coupled  
designs between objects that  
interact.

## OO Patterns

Observer – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over  
Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

Don't call us, we'll call you

Strive for loosely coupled  
designs between objects that  
interact.

## OO Patterns

Command – Encapsulate a request as an object, thereby  
letting you parameterize clients with different requests,  
queue or log requests, and support undoable operations.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over

Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

Don't call us, we'll call you

Strive for loosely coupled  
designs between objects that  
interact.

Classes should be open for  
extension but closed for  
modification.

## OO Patterns

Decorator – Attach additional responsibilities to an object  
dynamically. Decorators provide a flexible alternative to  
sub classing for extending functionality.



# GOF Design Patterns

## OO Principles

Encapsulate what varies.

Favor composition over  
Inheritance.

Program to interfaces, not  
implementations.

Depend on abstractions. Do not  
depend on concrete classes.

Only talk to your friend

Don't call us, we'll call you

Strive for loosely coupled  
designs between objects that  
interact.

Classes should be open for  
extension but closed for  
modification.

A Class should have only one  
reason to change.

## OO Patterns

Iterator – Provide a way to access the elements of an aggregate  
object sequentially without exposing its underlying representation.

Composite – Compose objects into tree structure to represent  
part-whole hierarchies. Composite lets clients treat individual  
objects and compositions of objects uniformly.

