

# Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood\_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

# Strategy / Policy Pattern

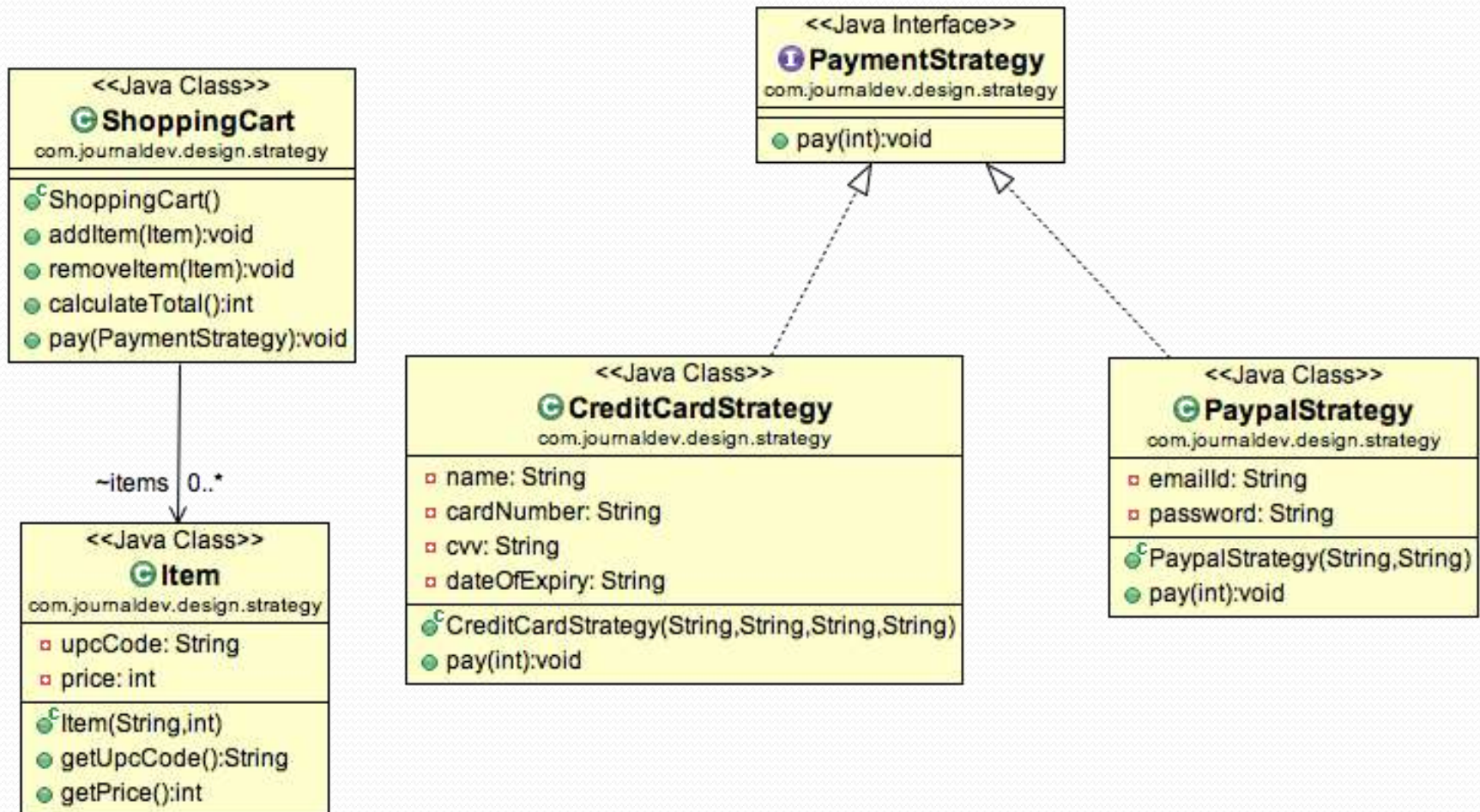
# Strategy Pattern

- Strategy design pattern is **behavioral design pattern**.
- Used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.
- Also known as **Policy Pattern**.
- We define multiple algorithms and let client application pass algorithm to be used as a parameter.
- Example : Collections.sort() method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

```
public void sort(ArrayList<String> list, Comparator<String> comp) {  
    Collections.sort(list);  
    Collections.sort(list, Comparator.reverseOrder());  
    Collections.sort(list, comp);  
}
```

# Ex. 1: Strategy Pattern

- For our example, we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.



# Ex. 1: Strategy Pattern

- Step 1: create the interface for strategy pattern

```
public interface PaymentStrategy {  
    public void pay(int amount);  
}
```

- Step 2: create concrete implementation of algorithms for payment using credit/debit card or through paypal.

```
public class CreditCardStrategy implements PaymentStrategy {  
    private String name;  
    private String cardNumber;  
    private String cvv;  
    private String dateOfExpiry;  
    public CreditCardStrategy(String nm,String ccNum,String cvv,String expDate){  
        this.name=nm;  
        this.cardNumber=ccNum;  
        this.cvv=cvv;  
        this.dateOfExpiry=expDate;  
    }  
    @Override  
    public void pay(int amount) {  
        System.out.println(amount + " paid with credit/debit card");  
    }  
}
```

# Ex. 1: Strategy Pattern

- Step 3: create concrete implementation of algorithms for payment using paypal.

```
public class PaypalStrategy implements PaymentStrategy {  
    private String emailId;  
    private String password;  
    public PaypalStrategy(String email, String pwd){  
        this.emailId=email;  
        this.password=pwd;  
    }  
    @Override  
    public void pay(int amount) {  
        System.out.println(amount + " paid using Paypal.");  
    }  
}
```

# Ex. 1: Strategy Pattern

- Step 4: create item class

```
public class Item {  
    private String upcCode;  
    private int price;  
    public Item(String upc, int cost){  
        this.upcCode=upc;  
        this.price=cost;  
    }  
    public String getUpcCode() {  
        return upcCode;  
    }  
    public int getPrice() {  
        return price;  
    }  
}
```



# Ex. 1: Strategy Pattern

Step 5: create ShoppingCart class

```
import java.util.ArrayList;
import java.util.List;
public class ShoppingCart {
    List<Item> items;
    public ShoppingCart(){
        this.items=new ArrayList<Item>();
    }
    public void addItem(Item item){
        this.items.add(item);
    }
    public void removeItem(Item item){
        this.items.remove(item);
    }
    public int calculateTotal(){
        int sum = 0;
        for(Item item : items){
            sum += item.getPrice();
        }
        return sum;
    }
    public void pay(PaymentStrategy paymentMethod){
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}
```



# Ex. 1: Strategy Pattern

- Step 6:

```
public class ShoppingCartTest {  
    public static void main(String[] args) {  
        ShoppingCart cart = new ShoppingCart();  
        Item item1 = new Item("Timato Catchup",100);  
        Item item2 = new Item("7up",40);  
        cart.addItem(item1);  
        cart.addItem(item2);  
        //pay by paypal  
        cart.pay(new PayPalStrategy("myemail@example.com", "mypwd"));  
        //pay by credit card  
        cart.pay(new CreditCardStrategy("Salman Lakhani",  
            "1234567890123456", "786", "12/15"));  
    }  
}
```

- Output:

140 paid using Paypal.

140 paid with credit/debit card

# Strategy

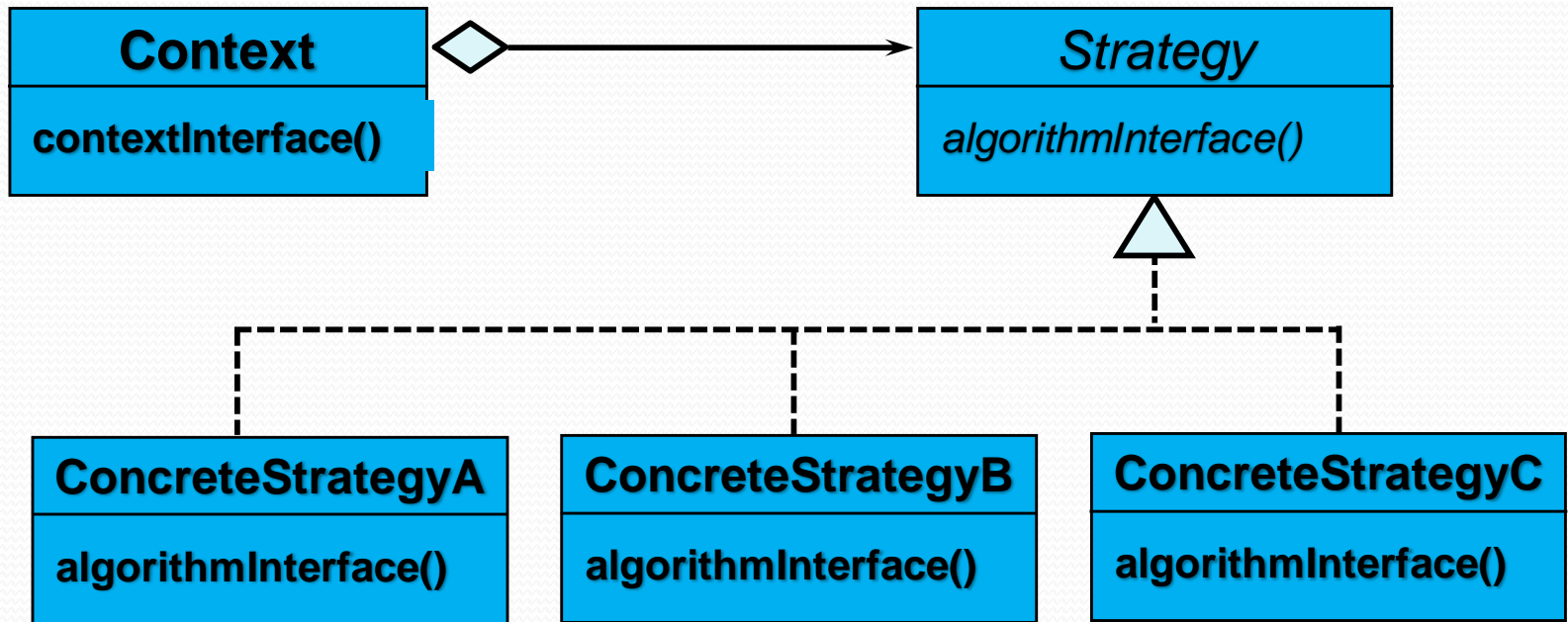
In a Strategy design pattern, you will:

- Define a family of algorithms
- Encapsulate each one
- Make them interchangeable

# You should use Strategy when:

- You have code with a lot of algorithms
- You want to use these algorithms at different times
- You have algorithm(s) that use data the client should not know about

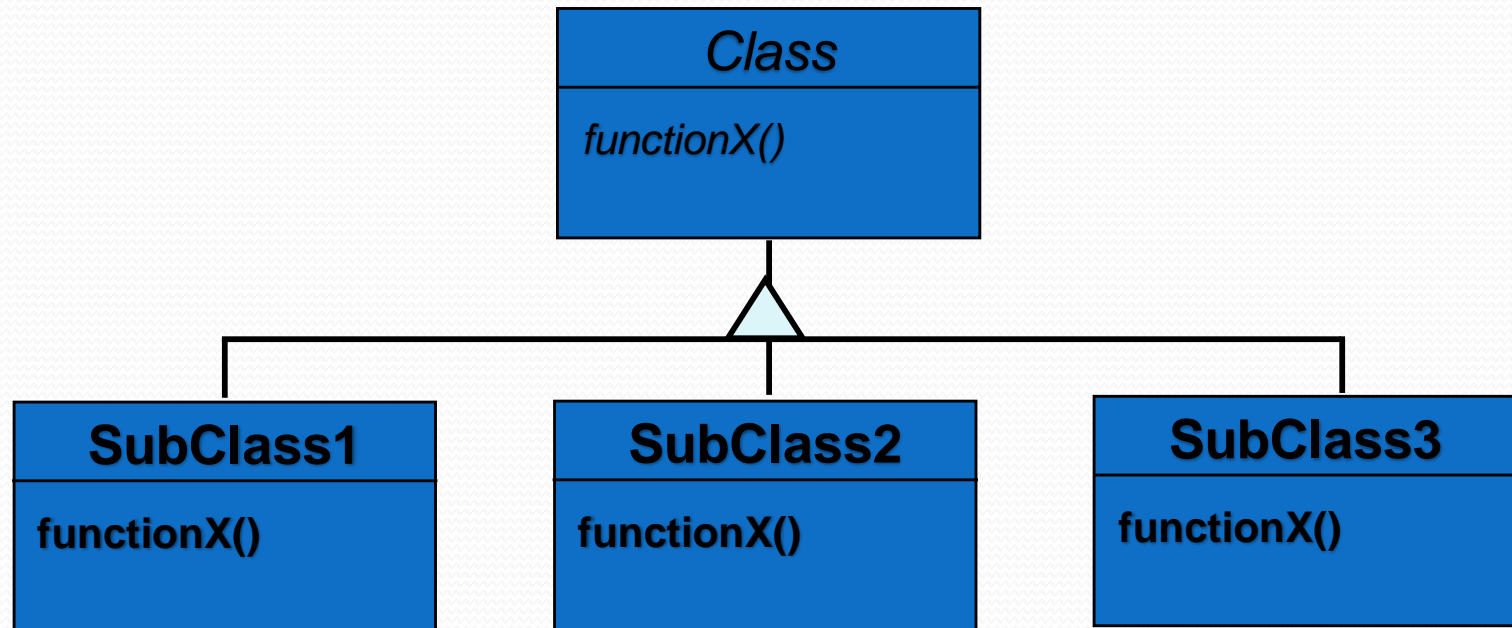
# Strategy Class Diagram



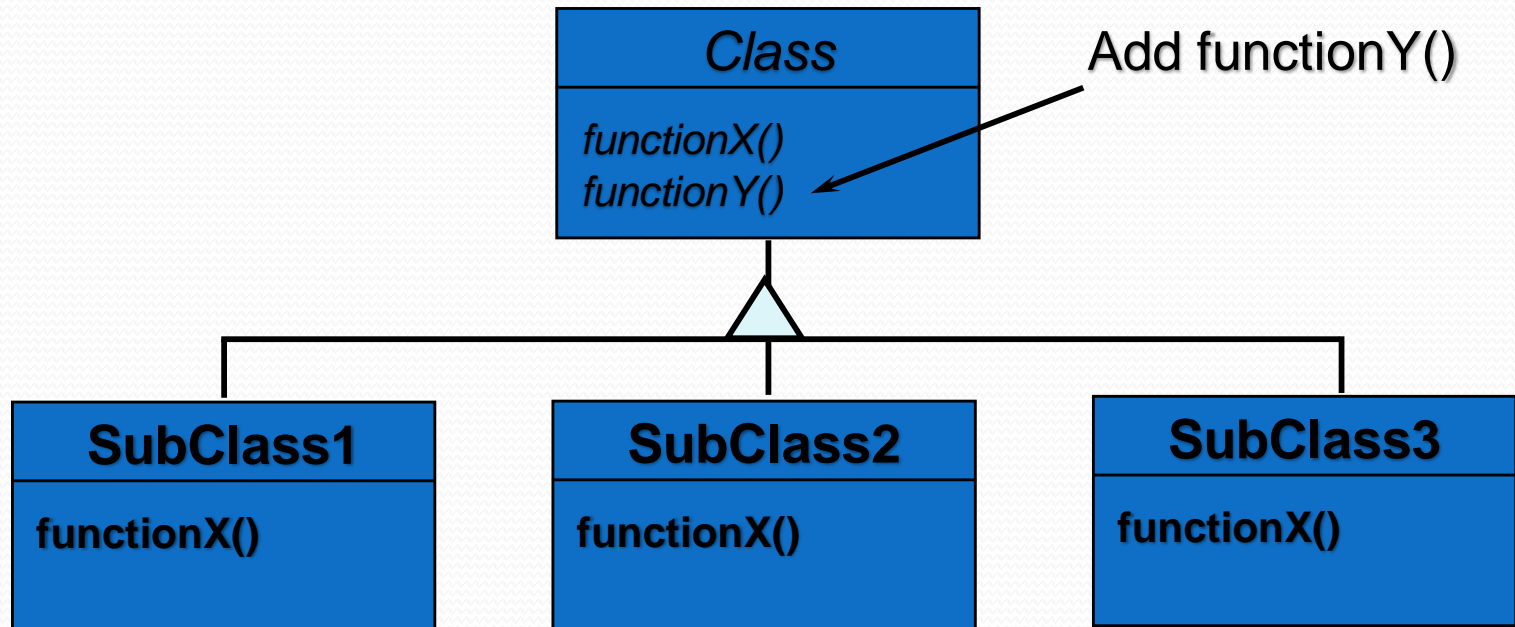
# Strategy vs. Subclassing

- Strategy can be used in place of subclassing
- Strategy is more dynamic
- Multiple strategies can be mixed in any combination where subclassing would be difficult

# Subclassing

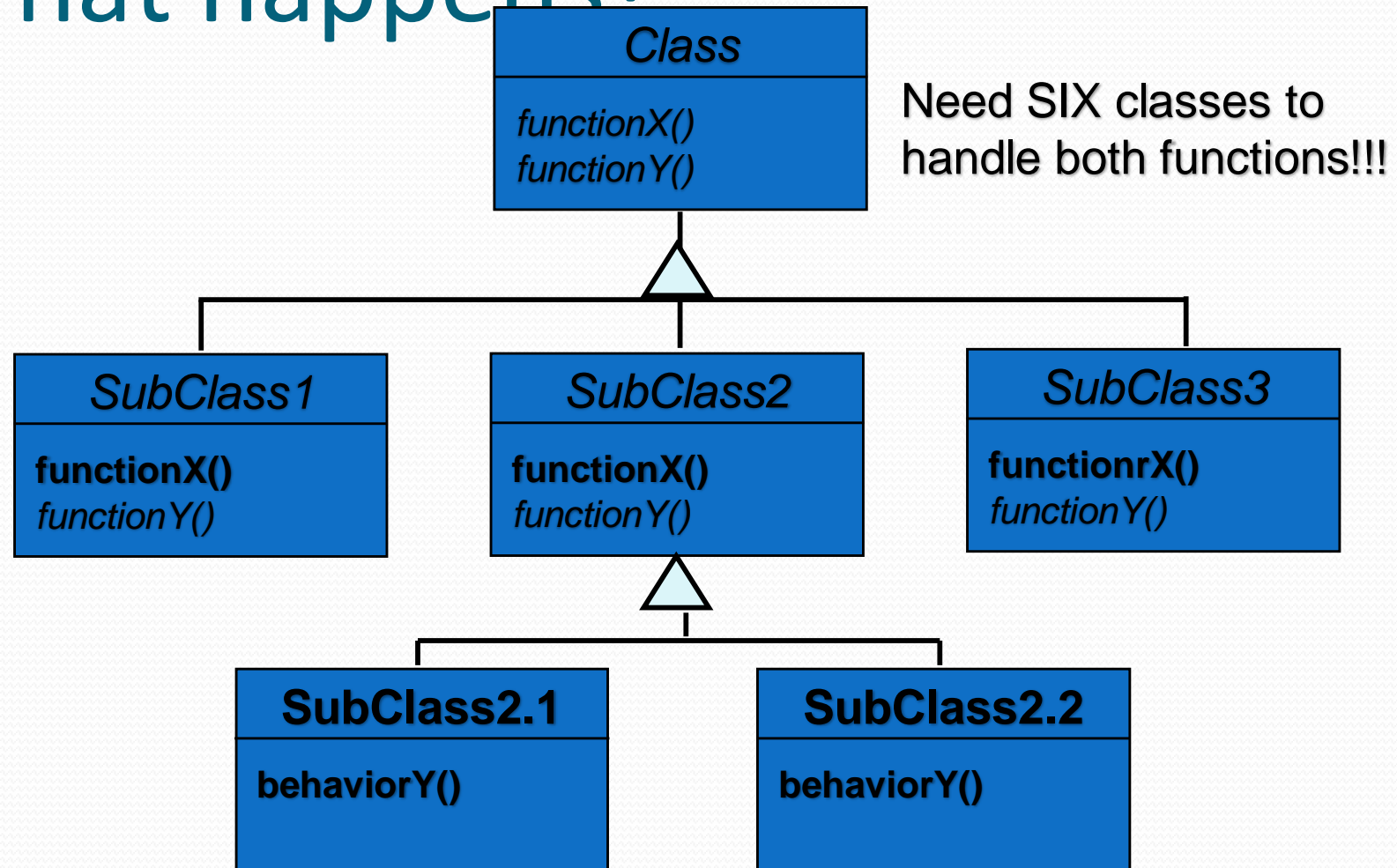


# Add a function

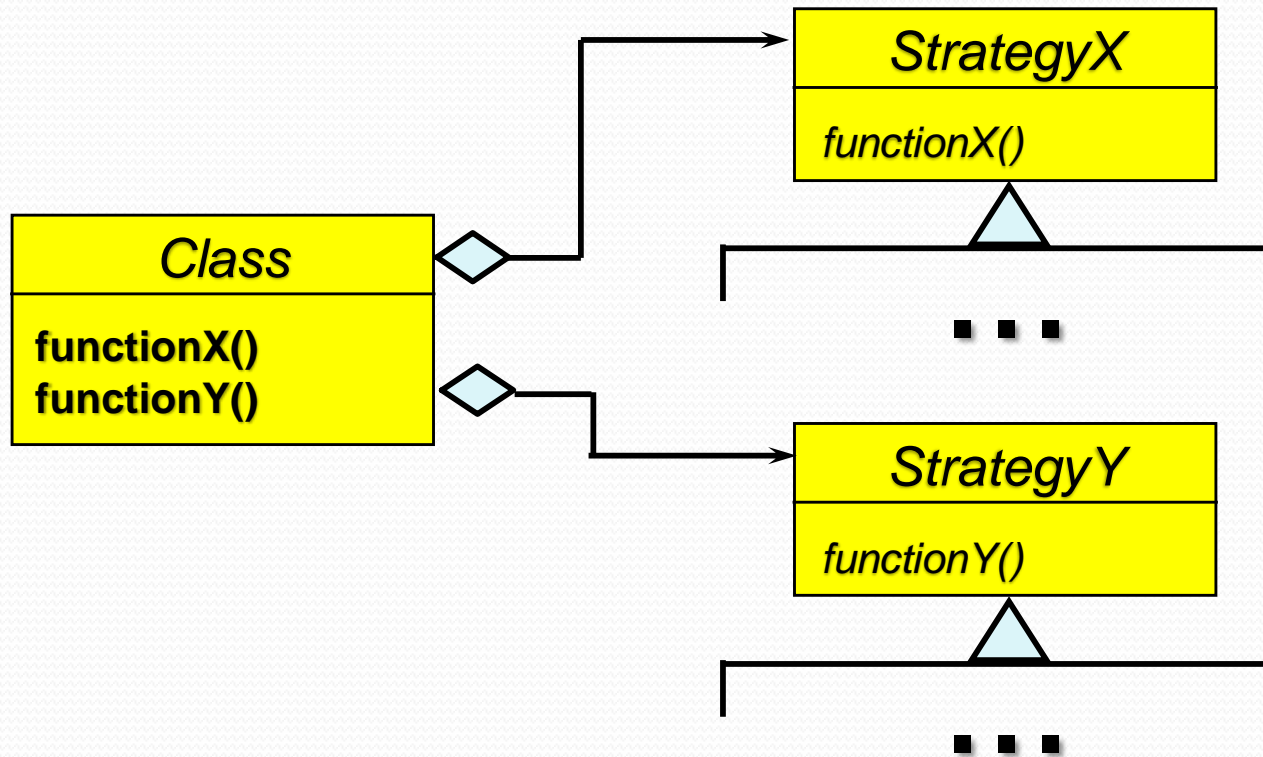




# What happens?



# Strategy makes this easy!

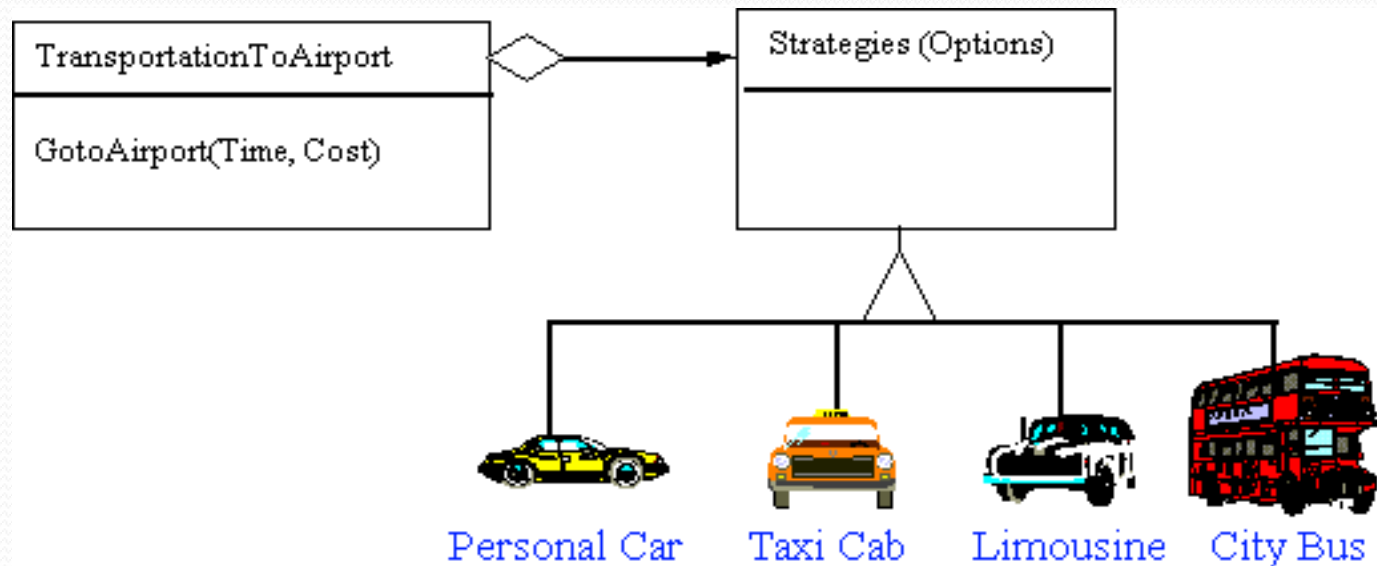


# Applicability

- Many related classes differ only in their behavior.
- You need different variants of an algorithm. Strategy can be used as a class hierarchy of algorithms.
- An algorithm use data structures that clients shouldn't know about.
- A class defines many behaviors, and these appear as multiple conditionals in its operation.

# Example

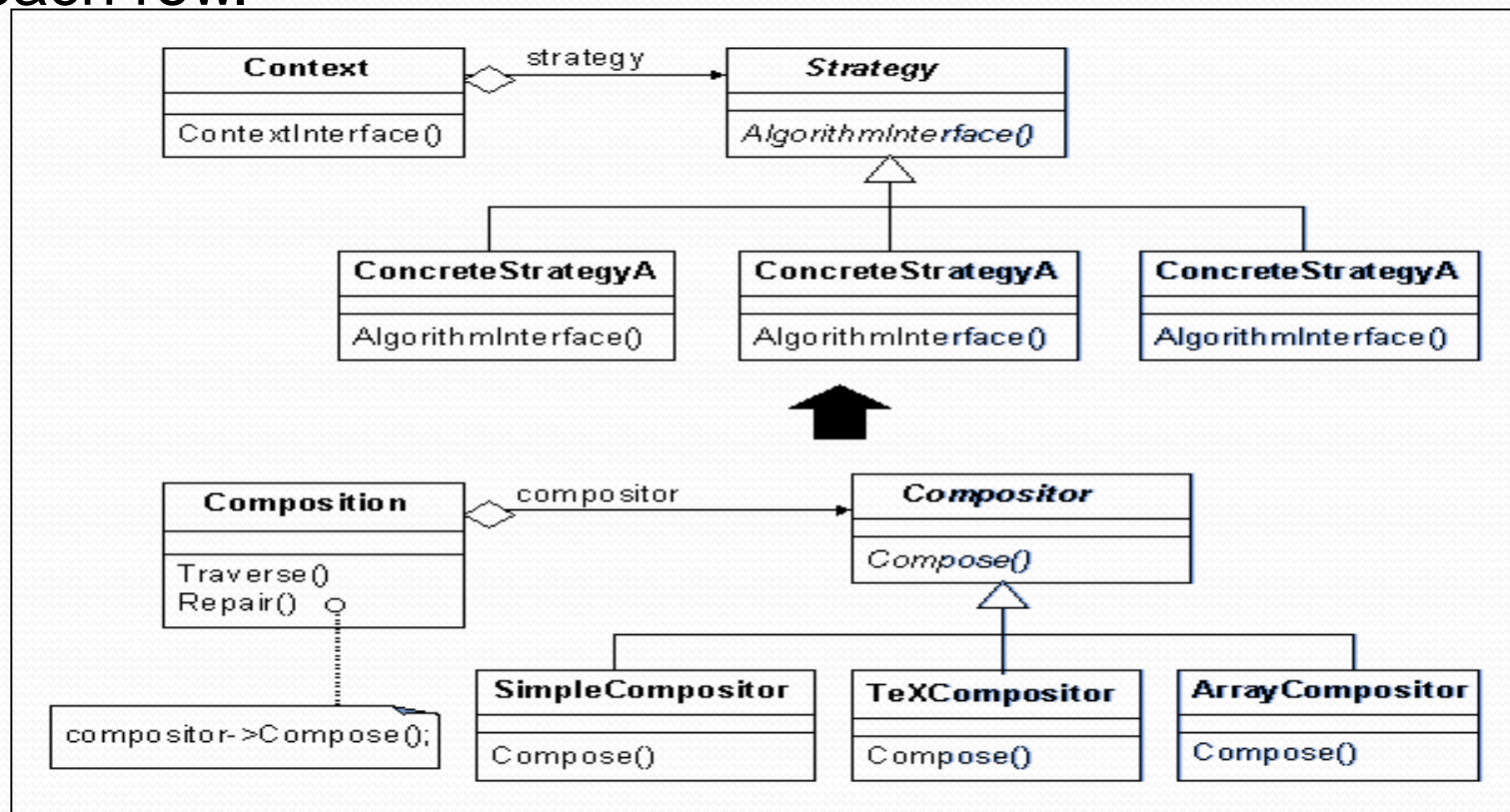
Modes of transportation to an airport is an example of a Strategy. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on tradeoffs between cost, convenience, and time.



# Example

Many algorithms exist for breaking a string into lines.

- Simple Compositor is a simple linebreaking method.
- TeX Compositor uses the TeX linebreaking strategy that tries to optimize linebreaking by breaking one paragraph at a time.
- Array Compositor breaks a fixed number of items into each row.



# Participants

- *Strategy*  
declares an interface common to all supported algorithms. Context uses its interface to call the algorithm defined by a ConcreteStrategy.
- *ConcreteStrategy*  
implements a specific algorithm using the Strategy interface.
- *Context*
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface for Strategy to use to access its

# Consequences

- ***Families of related algorithms***
  - Hierarchies of Strategy factor out common functionality of a family of algorithms for contexts to reuse.
- ***An alternative to subclassing***
  - Subclassing a Context class directly hard-wires the behavior into Context, making Context harder to understand, maintain, and extend.
  - Encapsulating the behavior in separate Strategy classes lets you vary the behavior independently from its context, making it easier to understand, replace, and extend.
- ***Strategies eliminate conditional statements.***
  - Encapsulating the behavior into separate Strategy classes eliminates conditional statements for selecting desired behavior.
- ***A choice of implementations***
  - Strategies can provide different implementations of the same behavior with different time and space trade-offs.



# Consequences (cont..)

- ***Clients must be aware of different strategies.***
  - A client must understand how Strategies differ before it can select the appropriate one.
  - You should use the Strategy pattern only when the variation in behavior is relevant to clients.
- ***Communication overhead between Strategy and Context.***
  - The Strategy interface is shared by all ConcreteStrategy classes.
  - It's likely that some ConcreteStrategies will not use all the information passed to them through this common interface.
  - To avoid passing data that get never used, you'll need tighter coupling between Strategy and Context.