CSE201: Advanced Programming

Lecture 22: Adapter and Strategy Design Pattern

Vivek Kumar

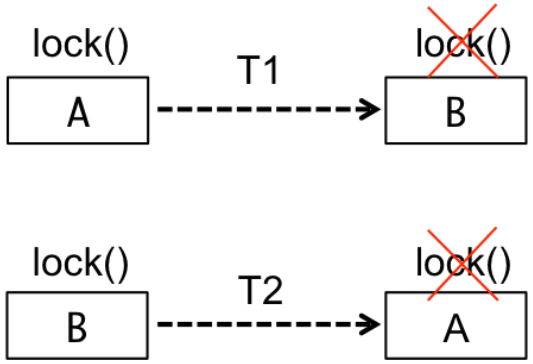
Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture

```
class Transfer {
    Account A, B;
    int amount;

    void run() {
     synchronized(A) {
        synchronized(B) {
           A.debit(amount);
           B.credit(amount);
        }
     }
    }
}
```



- Deadlocks
  - Deadlock occurs when multiple threads need the same locks but obtain them in different order
  - It could be avoided by using lock ordering
    - Ensure that all locks are taken in same order by any thread
- Design Patterns – it is a description or template for how to solve a repeatable problem in the software design
- Four examples
  - Iterator
    - Provides a solution to loop over all objects in any type of collection without changing client's code
  - Singleton
    - Provides a class that has at most one instance
  - Flyweight
    - Provides a class that has only one instance for each unique object

```
public class LengthComparator
                implements Comparator<String> {

    private static LengthComparator comp = null;

    public static LengthComparator getInstance()
    {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }

    private LengthComparator() {}

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

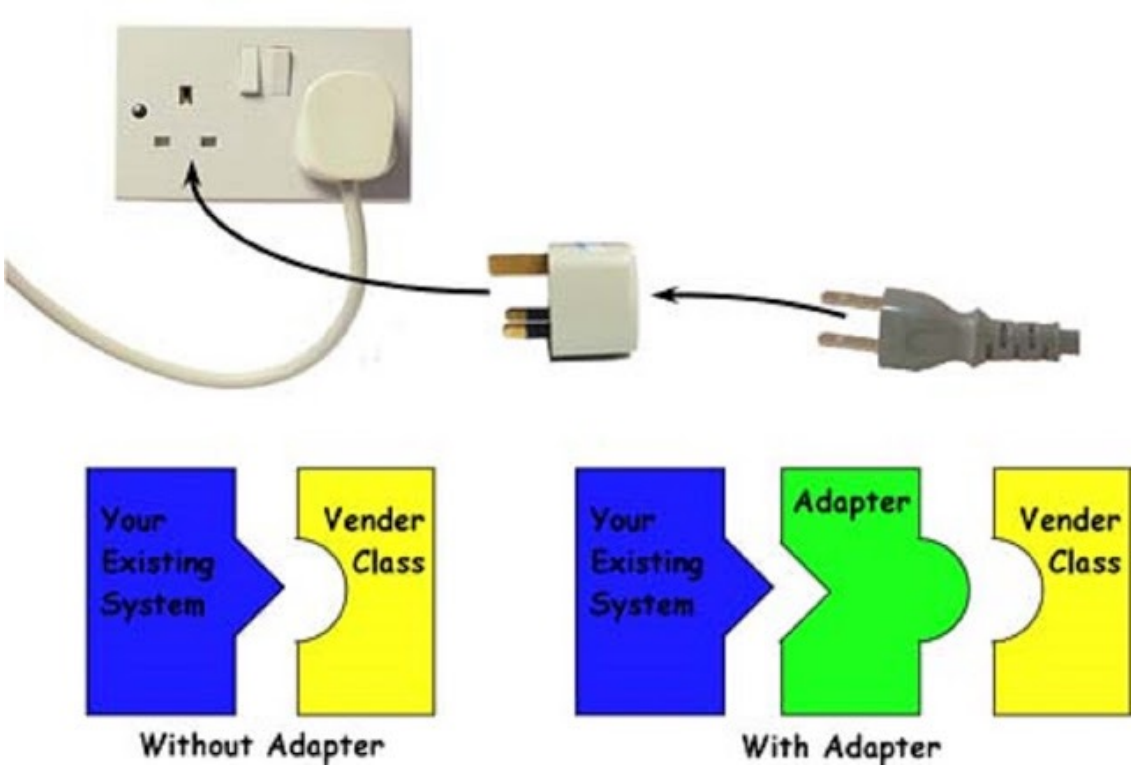
Today's Lecture

- Adapter design pattern (DP # 4)
- Strategy design pattern (DP # 5)

Pattern: Adapter

an object that fits another object into a given interface

Pattern: Adapter



- Recurring problem
  - We have an object that contains the functionality we need, but not in the way we want to use it
- Solution
  - Create an **adapter object** that bridges the provided and desired functionality

Fig source: http://javarevisited.blogspot.in/2016/08/adapter-design-pattern-in-java-example.html

Adapter Pattern Example (1/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());


    for(Movable obj: mylist) {
        obj.move();
    }
  }
}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

- The **adaptee** interface "Flyable" only implements fly() method, although it is similar to move() in Movable inteface
- Client class, Vehicle, doesn't understand Flyable and only use Movable
  - How to add Flyable type objects inside Movable type list in Vehicle?
  - We will code an adaptor that can serve this client by using this adaptee without any modifications

Adapter Pattern Example (2/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

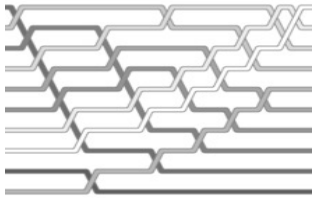
```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

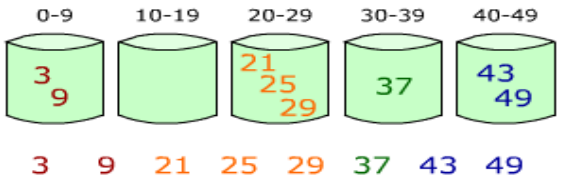
```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());

    mylist.add(new FlyableAdapter(new Airplane()));
    mylist.add(new FlyableAdapter(new Drone()));

    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

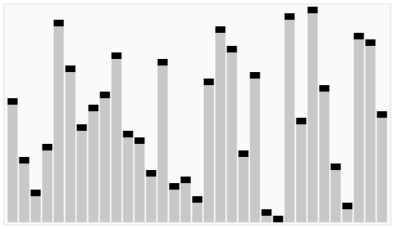
```
public class FlyableAdapter implements Movable {
    Flyable type;
    public FlyableAdapter(Flyable type) {
        this.type = type;
    }
    public void move() {
        type.fly();
    }
}
```

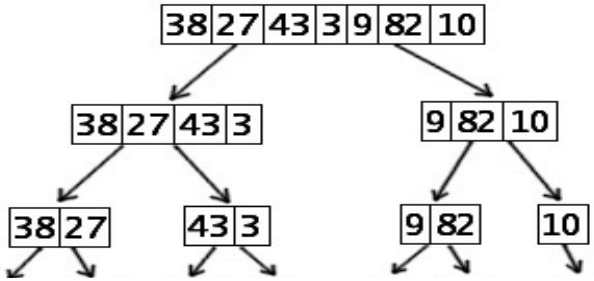
Pattern: Strategy

objects that hold different algorithms to solve a problem









The Ducks File Structure for Redux – S ...
medium.com

Wood Duck Identification, All Abou
allaboutbirds.org

Duck test - Wikipedia
en.wikipedia.org

Ever Wanted to Know About Ducks
thoughtco.com

Duck

Bufflehead

(*Bucephala albeola*)

Scientific classification

Kingdom:	Animalia
Phylum:	Chordata
Class:	Aves
Order:	Anseriformes
Superfamily:	Anatoidea
Family:	**Anatidae**

Subfamilies

see text

Amazon.com: Giant Duck F...
amazon.com

Different Kind of Ducks | D'Artagnan
dartagnan.com

Caring for Ducks in Winter | Modern ...
oysfarm.com

Different Types of Ducks With Examples
thespruce.com

Let’s Build a Duck Simulator!

- Concepts we will revisit
  - Inheritance
  - Interfaces
  - Polymorphism

What are their Characteristics?

- I'm Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I'm Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

How to Code a Duck Simulator?

- I'm Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I'm Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

Inheritance?

Lets See the Code

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```

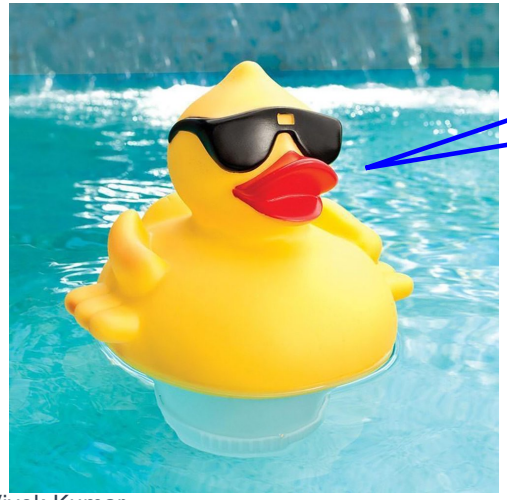
```
// Calling display on above two Duck type objects
I am Wood Duck
I can quack
I can swim
I can fly
My home is on trees
I am Dabbler Duck
I can quack
I can swim
I can fly
My home is on ground
```

Any Problems?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```



Please code me too ☹

- I'm Rubber duck
- **I can squeak**
- I can swim
- **I don't fly**
- Your home is my home

What are the Issues?

- Applying inheritance for code reuse sometimes backfires
- Poor solution for maintenance
  - Our assumption that all Ducks can Fly is incorrect
  - Our assumption that all Ducks make quack-quack sound is incorrect
- How to fix this issue?
  - Overriding both the methods fly() and speak() in subclass Rubber Duck

Let's Implement the Fix

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super("Rubber"); }

    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    @Override
    public void fly() {
        System.out.println("I don't Fly");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

```
// Calling display on Rubber Duck type object
I am Rubber Duck
I can Squeak
I can swim
I don't Fly
Your home is my home
```

Wait.. What if we get other non-flyable Duck?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super("Rubber"); }

    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    @Override
    public void fly() {
        System.out.println("I don't Fly");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

- If we have to code a **Domestic Duck** then they too don't fly
  - This means we need to Override the fly() method even inside Domestic Duck class

What are the Issues?

- Another Duck type could speak in a language other than “Quack” and “Squeak”
  - Examples:
    - **Decoy** Duck can’t speak
    - **Whistling** Duck make whistles
  - As there are **several possible** ways to speak, we don’t have any choice other than **Overriding** the speak() method
- However, the flying capability could be either true or false only. As the options for flying capability is limited, can we write a better code?
  - How about using an interface called Flyable that has fly() method?
    - Again there will be lot of duplicate code as each Duck type will have to implement this interface to show their flying capability

Recap: Design Principals

- Program to a supertype and not for an implementation
  - We used Duck as superclass in past
- Identify the aspects of the implementation that differs and separate them out from what stays the same
  - We took out similar functionality inside the superclass Duck and left the specialized implementation inside subclass

Using Strategy Pattern for Final Fix

1. We will still use **Flyable** interface BUT will limit its implementation in only **two** classes
2. Create a field of **Flyable** type in supertype (Duck)
3. Each subclass will simply instantiate this field inside their constructor with correct flying ability. The flying capability are defined inside the two classes mentioned in Step-1
4. display() method in Duck will use polymorphism to show the correct flying capability

Applying Strategy Pattern: The Final Fix!

```
public interface Flyable {
    public void fly();
}
```

```
public abstract class Duck {
    private String name;
    private Flyable flyStatus;
    public Duck(String n, Flyable f) {
        this.name = n;
        this.flyStatus = f;
    }
    ........
    ........
    public void tryFlying() {
        flyStatus.fly();
    }
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.tryFlying();
        this.home();
    }
}
```

```
public class CannotFly implements Flyable {
    public void fly() {
        System.out.println("I don't Fly");
    }
}
```

```
public class CanFly implements Flyable {
    public void fly() {
        System.out.println("I can Fly");
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() {
        super("Dabbler", new CanFly());
    }
    ........
}
```

```
public class Rubber extends Duck {
    public Rubber() {
        super("Rubber", new CannotFly());
    }
    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

Summary: Strategy Pattern

- In Strategy pattern, a class behavior (or its algorithm) can be changed at run time
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- The strategy object changes the executing algorithm of the context object
- This type of design pattern comes under behavior pattern

Next Lecture

- More design patterns
- **Quiz-5**