I3305-GUI
Chapter 3
Design Patterns, Structural patterns

Abed Safadi,Ph.D

Structural patterns

Classes and relationships between statically defined classes

- data types handled
- specialization of these types of data
- relationships between all these types

Software also has a **dynamic side**

- objects created and manipulated
- relations between objects different from relations between classes
- can we speak of specialization of an object?

A large part of structural patterns

- concerns these dynamic aspects
- allows you to relate and specialize objects

Structural patterns

- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

- Structural patterns related to the problems of organization of objects in software. The essence of structural patterns is to organize the compositional structure of objects in a standard and efficient way.

Plan

- Flyweight
- Bridge
- Composite
- Proxy

Flyweight pattern

Intent

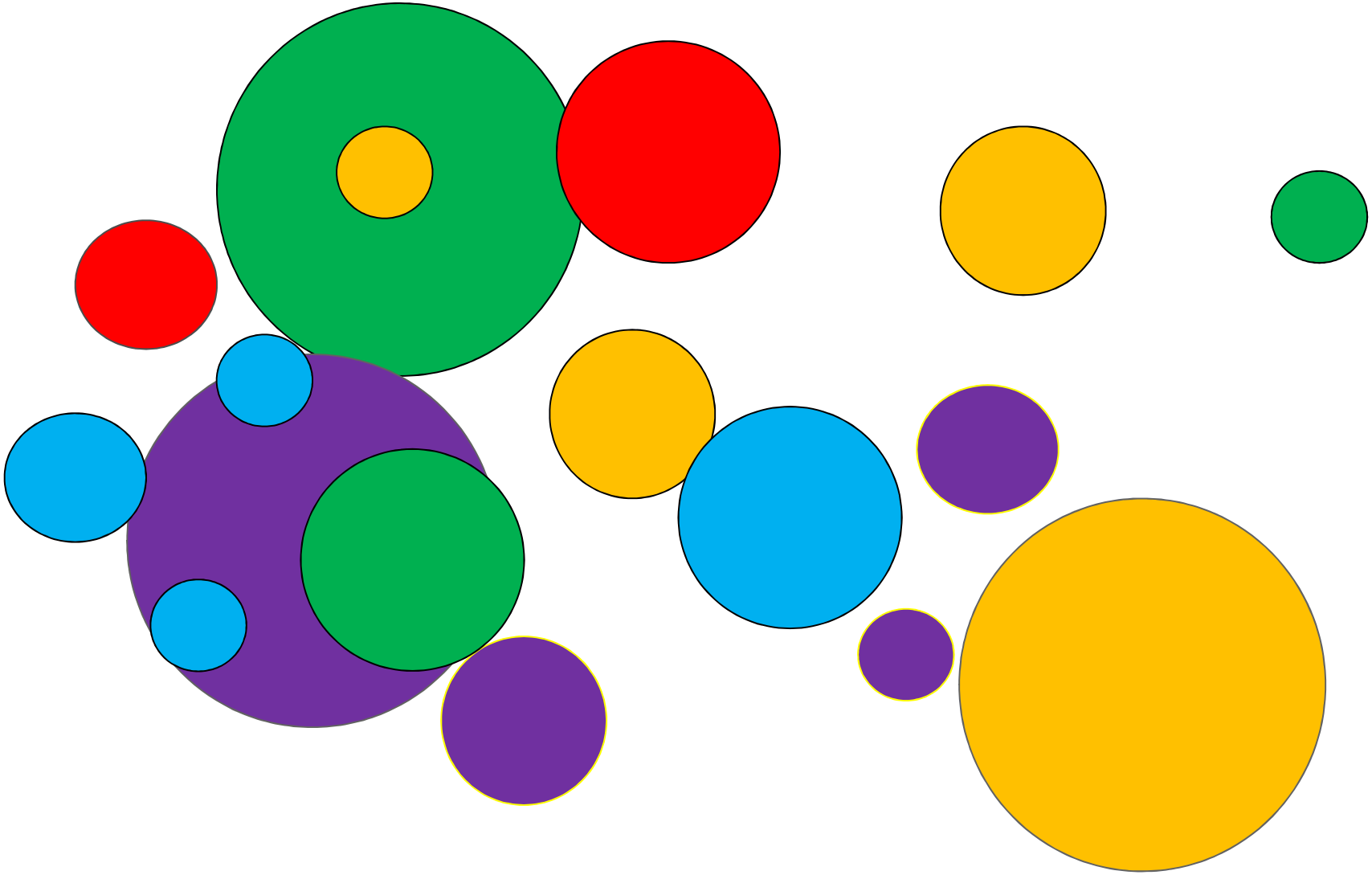
- **Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

- Used when you need to create a large number of similar objects

- To reduce memory usage you share Objects that are similar in some way rather than creating new ones

Example before Flyweight pattern

Creating 20 circles of different locations



Flyweight - Problem

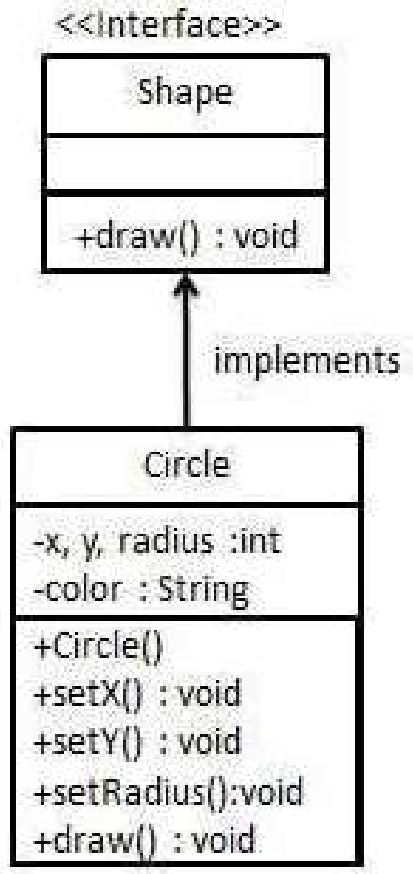
- Implementation : We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface.

Shape.java

```
public interface Shape {
  void draw();
}
```

Circle.java

```
public class Circle implements Shape {
  private String color;
  private int x;
  private int y;
  private int radius;

  public Circle(String color){
    this.color = color;
  }

  public void setX(int x) {
    this.x = x;
  }
  public void setY(int y) {
    this.y = y;
   }
  public void setRadius(int radius) {
    this.radius = radius;
   }
  @Override
  public void draw() {
    System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);
   }
}
```

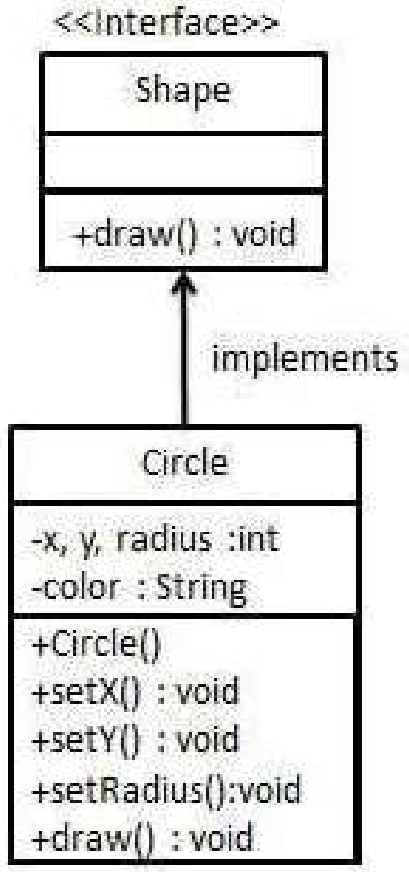


Flyweight - Problem

- Implementation : We are going to create a *Shape* interface and concrete class *Circle* implementing the *Shape* interface.

Demo.java

```
public class Demo {

   private static final String colors[] = { "Red", "Green", "Blue", “Black”,
     ”White”, };
   public static void main(String[] args) {
     for(int i=0; i < 20; ++i) {
        Circle circle = new Circle(color [(int)(Math.random()*colors.length)]);
           circle.setX(Math.random()*100);
           circle.setY(Math.random()*100);
           circle.setRadius(Math.random()*100);
           circle.draw();
       }
    }
}
```



Before Flyweight Design Patterns **we must create 20 objects** in order to draw 20 circles .

Flyweight - Solution

Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

A flyweight is an object through which we try to minimize memory usage by sharing data as much as possible.

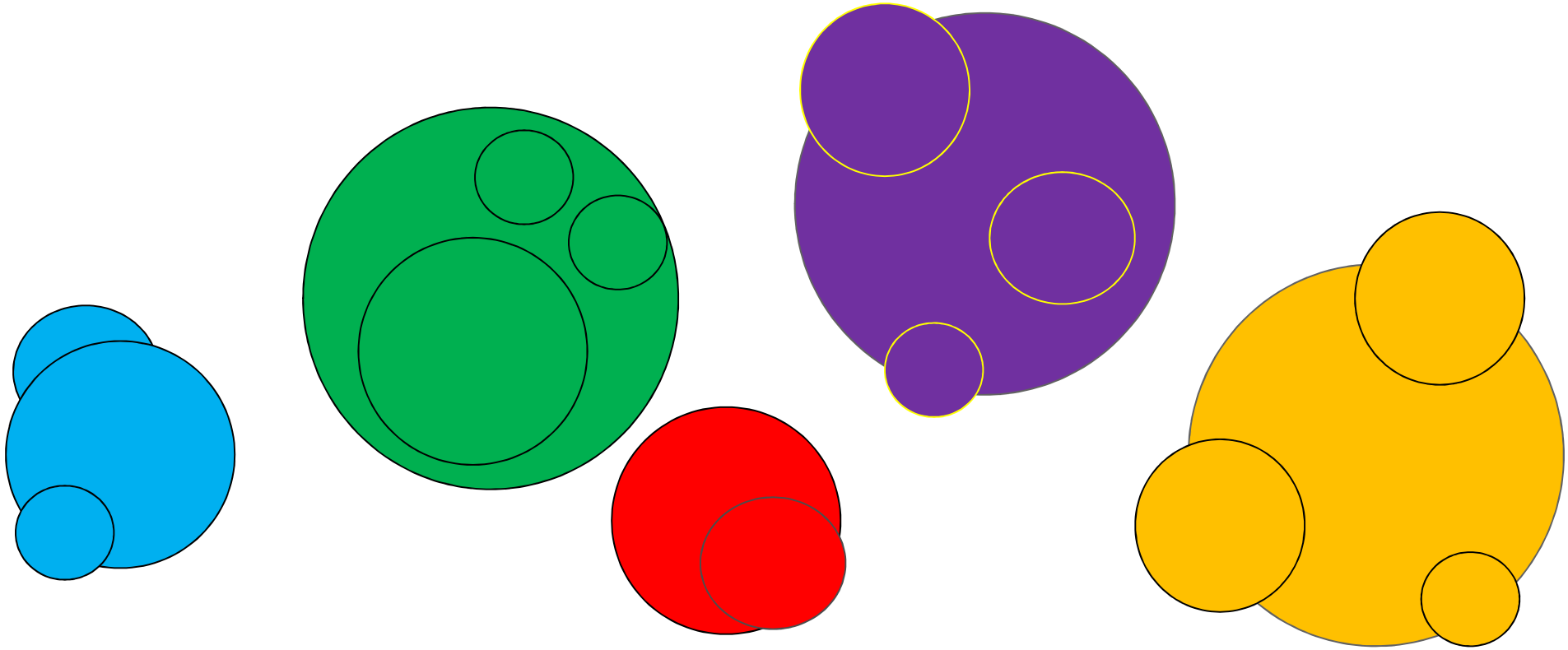
Two common terms are used here:
—*intrinsic* state : color
—*extrinsic* state: size.

The first category (intrinsic) can be stored in the flyweight and is shareable. The other one depends on the flyweight's context and is non-shareable.

➔We will demonstrate this pattern by drawing 20 circles of different locations but we will **create only 5 objects**

Flyweight - Solution

Only 5 colors are available so **color property (intrinsic state)** is used to **check** already existing *Circle* objects

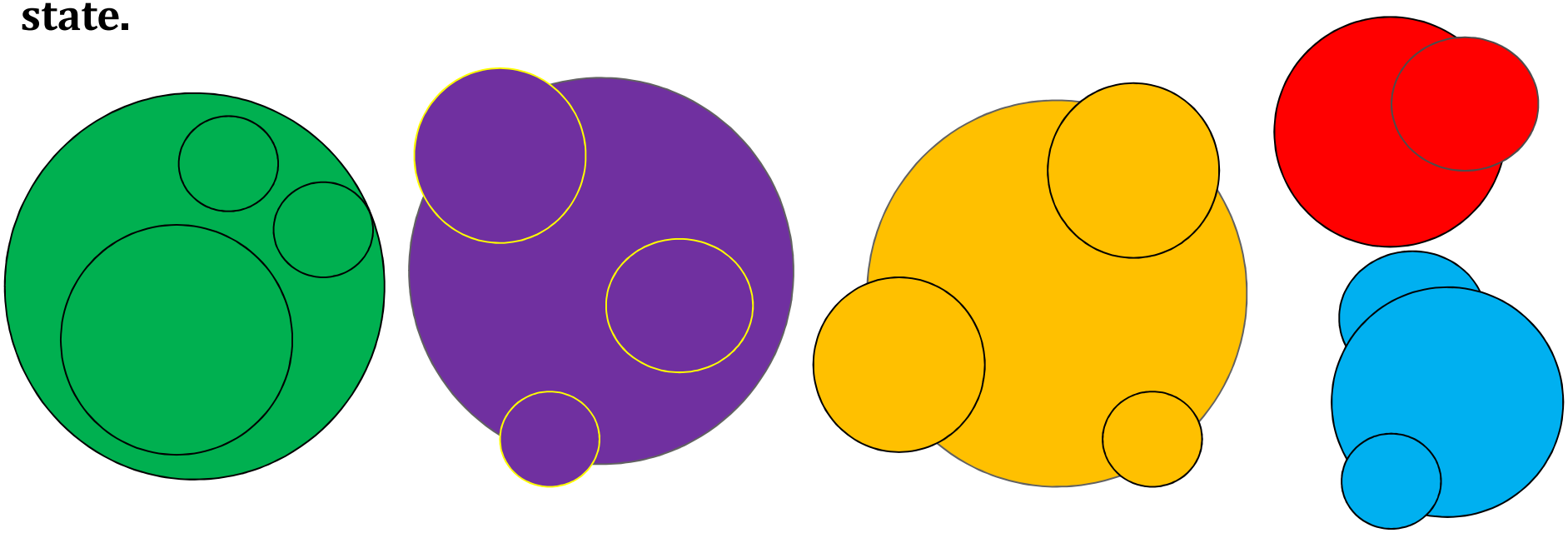


Flyweight - Solution

For example, in each group all circles have the same color. Other parts of a circle s state, such as radius is unique to each circle. After all, the values of this field change over time.

This data represents the always changing context in which the circle exists, while the color remain constant for each circle.

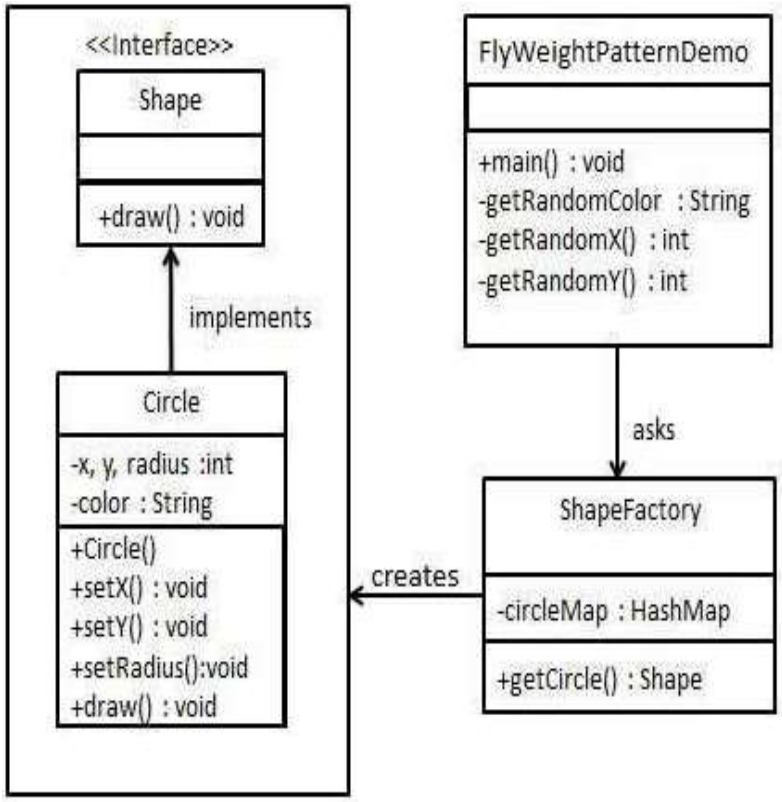
This constant data of an object is usually called the ***intrinsic state*****.** It lives within the object; other objects can only read it, not change it. The rest of the object's state, often altered "from the outside" by other objects, is called the **extrinsic state.**



Flyweight - Solution

- *ShapeFactory* has a *HashMap* of *Circle* having key as color (intrinsic state) of the *Circle* object. Whenever a request comes to create a circle of particular color to *ShapeFactory*, it checks the circle object in its *HashMap*, if object of *Circle* found, that object is returned otherwise a new object is created, stored in hashmap for future use, and returned to client.

- *FlyWeightPatternDemo*, our demo class, will use *ShapeFactory* to get a *Shape*object. It will pass information (*red / green / blue/ white/ black/*) to *ShapeFactory* to get the circle of desired color it needs.



Java translation

Create a factory to generate object of concrete class based on given information.

```
import java.util.HashMap;

public class ShapeFactory {

   private static final HashMap circleMap = new HashMap();

   public static Shape getCircle(String color) {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null) {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

ShapeFactory.java

Java translation

Use the factory to get object of concrete class by passing an information such as color.

```
public class FlyweightPatternDemo {
   private static final String colors[] = { "Red", "Green", "Blue", "White", "Black'};
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

FlyweightPatternDemo.java

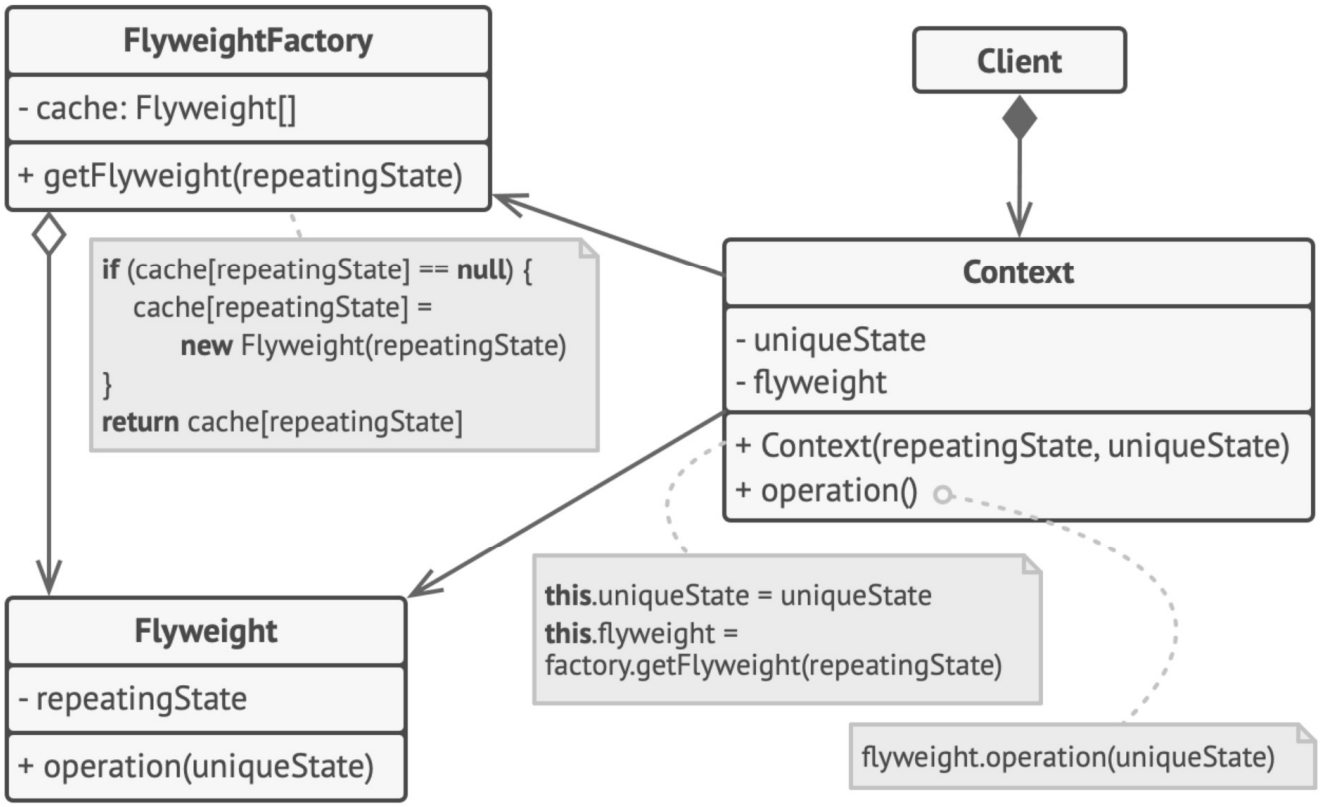
Java translation

- **Verify the output**

```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

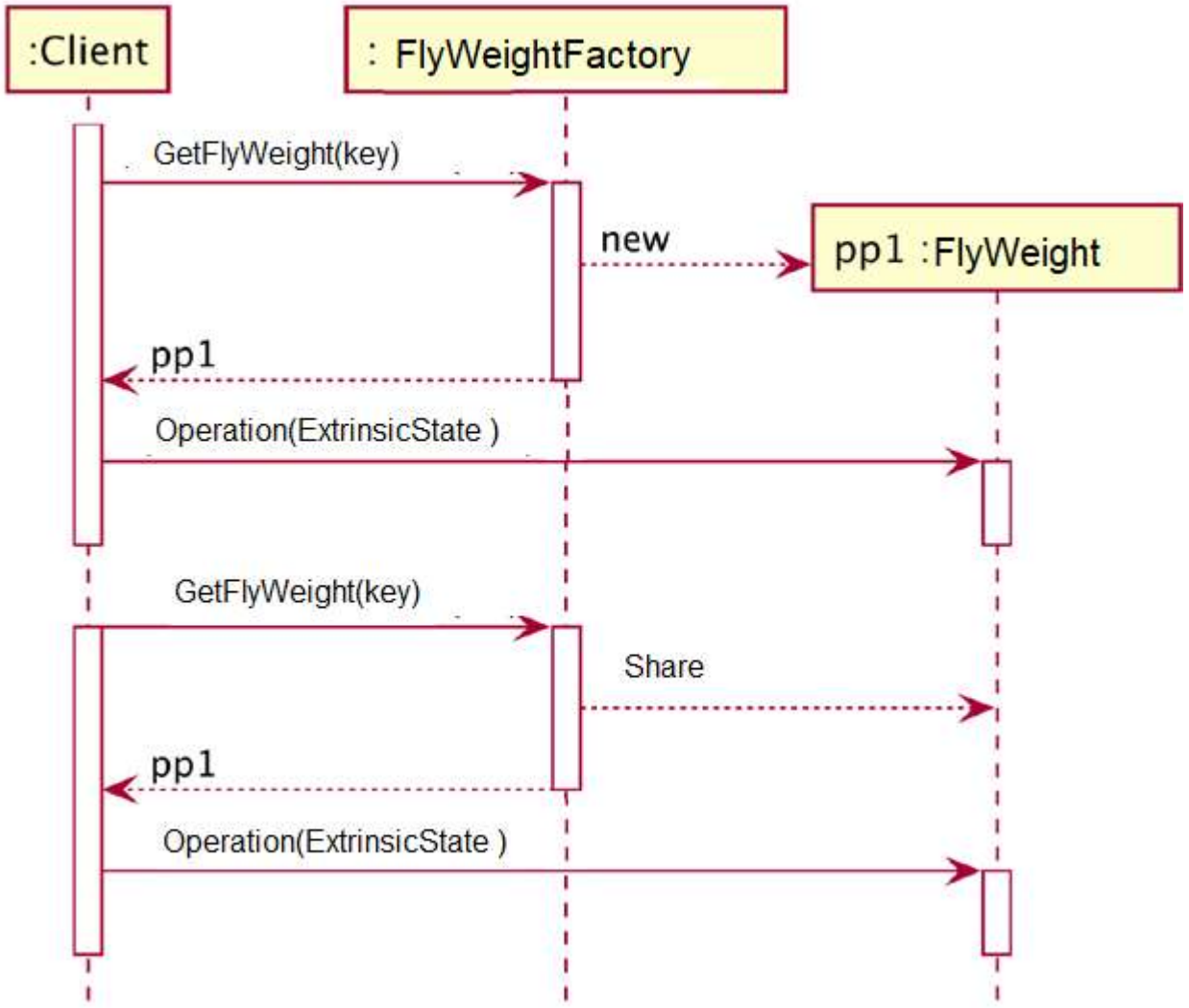
Flyweight Class diagram

The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.



The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called *intrinsic.* The state passed to the flyweight's methods is called *extrinsic.*

Flyweight sequence diagram



Plan

- Flyweight
- Bridge
- Composite
- Proxy

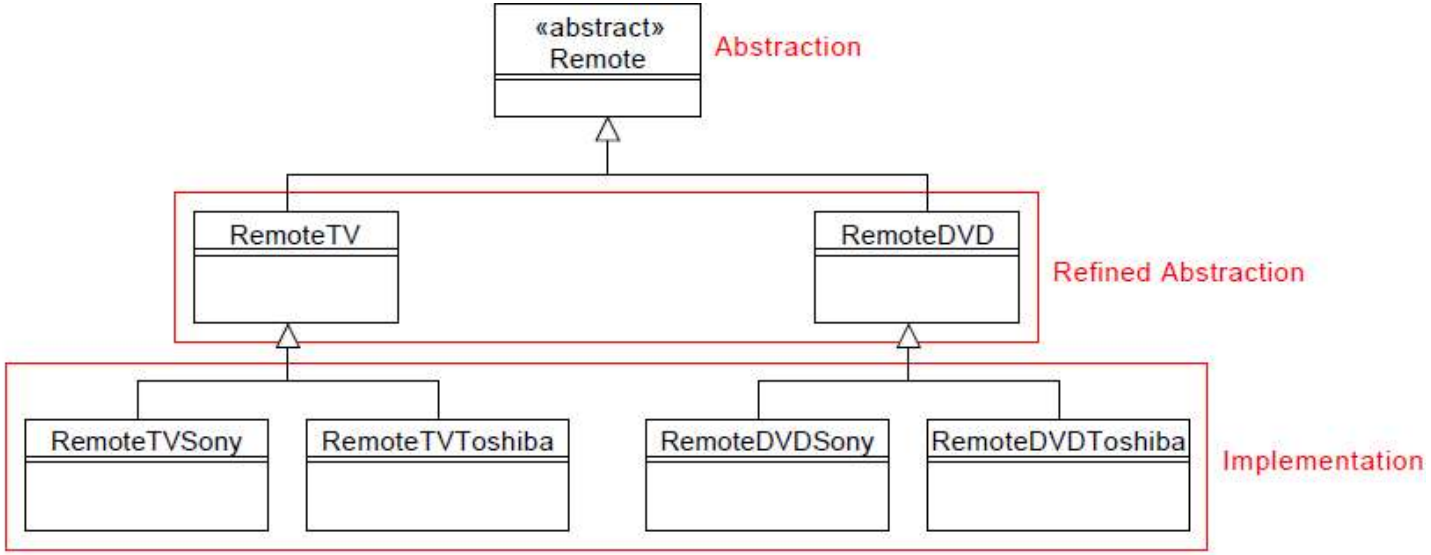
Bridge pattern

Intent

- Decouple an abstraction from its implementation so that the two can vary independently

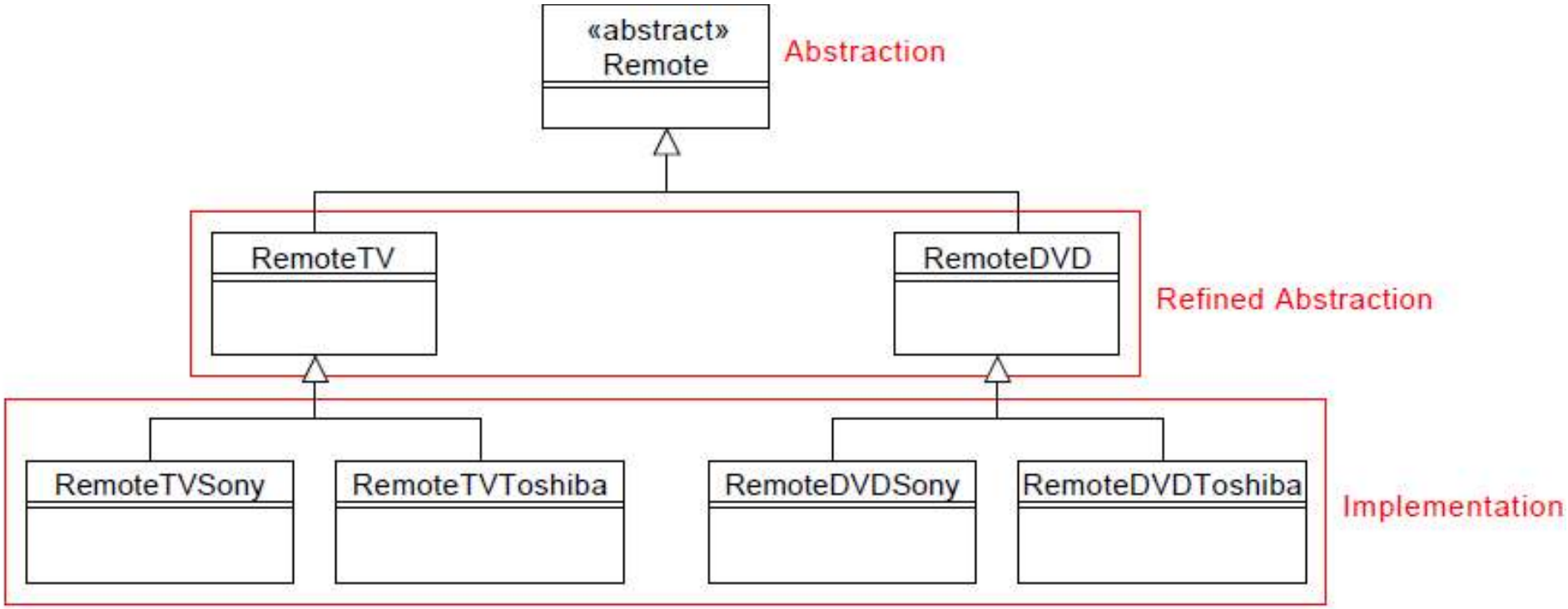
- In this pattern, the abstract class is separated from the implementation class and we provide a bridge interface between them. This interface helps us to make concrete class functionalities independent from the interface implementer class. We can modify these different kind of classes structurally without affecting each other.

Example : Before Bridge Design Pattern



- Remote is an abstract class. It contains:
  - an instance variable sound
  - two methods soundUp and soundDown
  - two abstract methods onPressMiddle and onPressNine
- If it is remote for TV: onPressNine ➔ go to the channel nine.
- If it is remote for DVD: onPressNine ➔pause.
- If it is Sony remote: onPressMiddle ➔open menu.
- If it is Toshiba remote: onPressMiddle ➔turn off.

Bridge pattern problem



- Problems
  - Create a remote for LCD creation of three new data types (RemoteLCD, RemoteLCDSony, RemoteLCDToshiba).
  - Code redundancy (duplication of methods)!

Java Translation

```
public abstract class Remote {
    protected int sound = 0;

    public void soundUp() {
        if(sound < 100)
            sound++;
    }

    public void soundDown() {
        if(sound > 0)
            sound--;
    }
    public abstract void onPressNine();
    public abstract void onPressMiddle();
}
```

Java Translation

```
public abstract class RemoteDVD extends Remote {
    public void onPressNine() {
        System.out.println("Pause");
    }
}
```

```
public class RemoteDVDSony extends RemoteDVD {
    public void onPressMiddle() {
        System.out.println("Open Menu");
    }
}
```

```
public class RemoteDVDToshiba extends RemoteDVD {
    public void onPressMiddle() {
        System.out.println("Turn Off");
    }
}
```

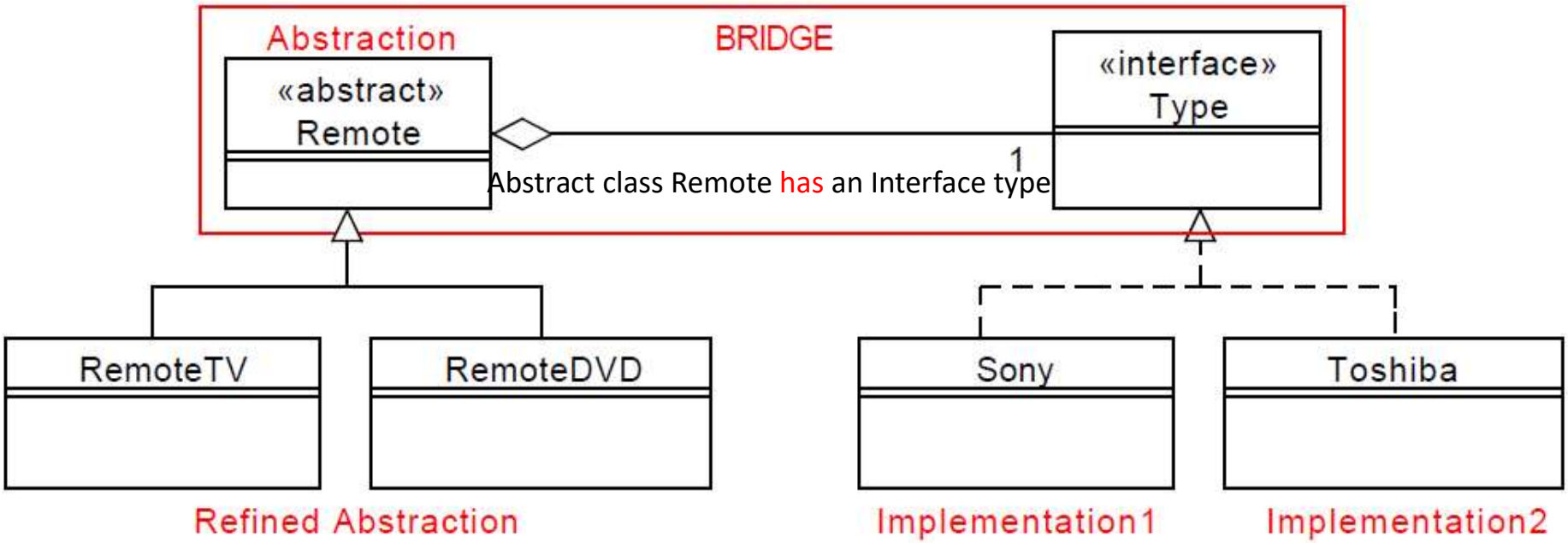
Java Translation

```
public abstract class RemoteTV extends Remote {
    public void onPressNine() {
        System.out.println("Go To Channel Nine");
    }
}
```

```
public class RemoteTVSony extends RemoteTV {
    public void onPressMiddle() {
        System.out.println("Open Menu");
    }
}
```

```
public class RemoteTVToshiba extends RemoteTV {
    public void onPressMiddle() {
        System.out.println("Turn Off");
    }
}
```

Bridge-Solution



The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.

Java Translation

```
public interface Type {
    public void onPressMiddle();
}
```

```
public class Toshiba implements Type {
    @Override
    public void onPressMiddle() {
        System.out.println("Turn Off");
    }
}
```

```
public class Sony implements Type {
    @Override
    public void onPressMiddle() {
        System.out.println("Open Menu");
    }
}
```

Java Translation

```
public abstract class Remote {

    public Type type;

    public Remote(Type t) {
        type = t;
    }

    protected int sound = 0;

    public void soundUp() {
        if(sound < 100)
            sound++;
    }

    public void soundDown() {
        if(sound > 0)
            sound--;
    }

    public void onPressMiddle() {
        type.onPressMiddle();
    }
    public abstract void onPressNine();
}
```

Java Translation

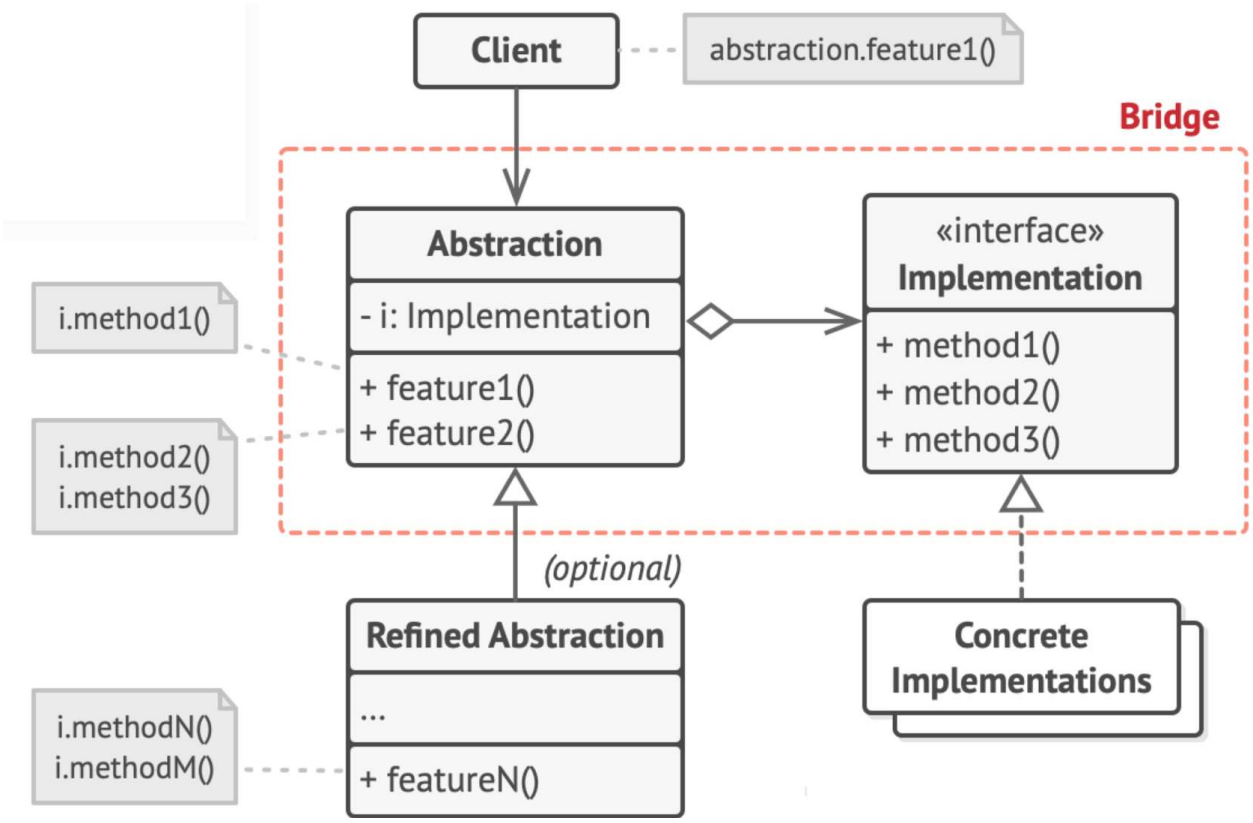
```
public class RemoteDVD extends Remote {
    public RemoteDVD(Type t) {
        super(t);
    }
    public void onPressNine() {
        System.out.println("Pause");
    }
}
```

```
public class RemoteTV extends Remote {
    public RemoteTV(Type t) {
        super(t);
    }
    public void onPressNine() {
        System.out.println("Go To Channel Nine");
    }
}
```

Java Translation

```
public class Test {
    public static void main(String[] args) {
        Remote r1 = new RemoteDVD(new Toshiba());
        Remote r2 = new RemoteTV(new Sony());

        r1.onPressMiddle();
        r2.onPressNine();
    }
}
```

Bridge Pattern UML diagram



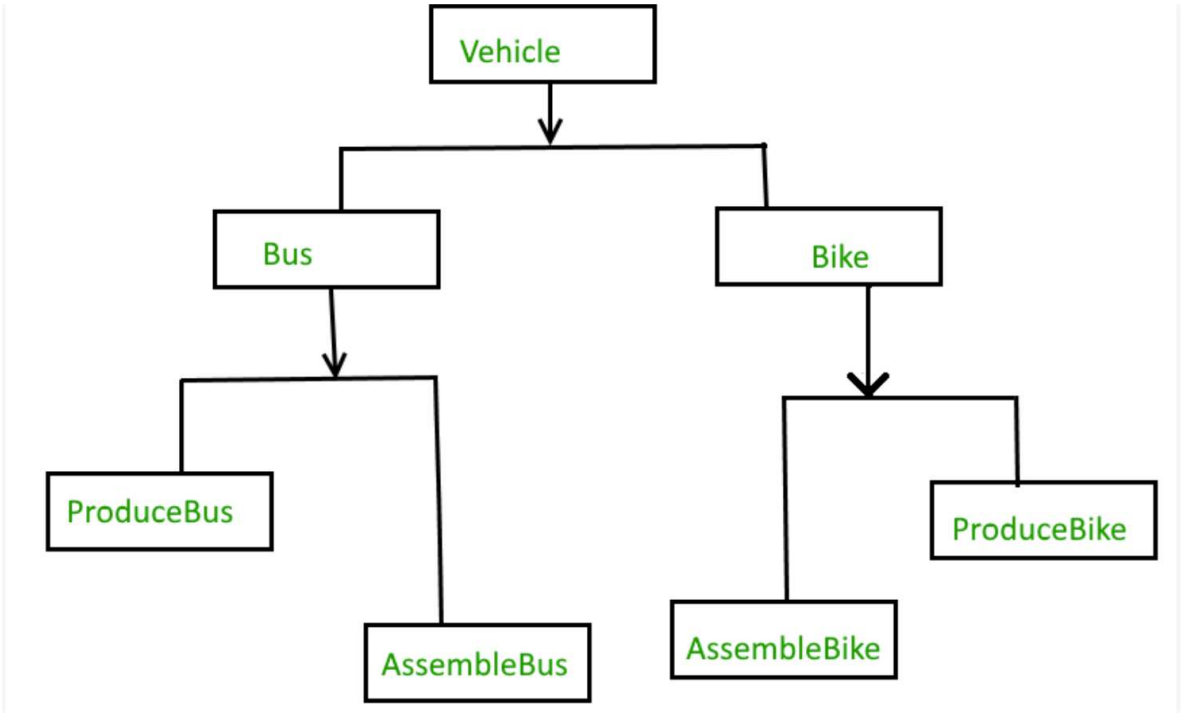
Bridge Pattern-Example 2

Suppose there is a company that manufactures various types of vehicles, like bikes, cars, and buses. There are frequent changes in the vehicle, as new models of bikes and cars can be introduced and have different processes to manufacture them. There can be other new vehicles, like a scooter or truck in future as well.

All the vehicle has to do is use the WorkShop to produce, assemble, or quality check (test) its work using the work() method for production.

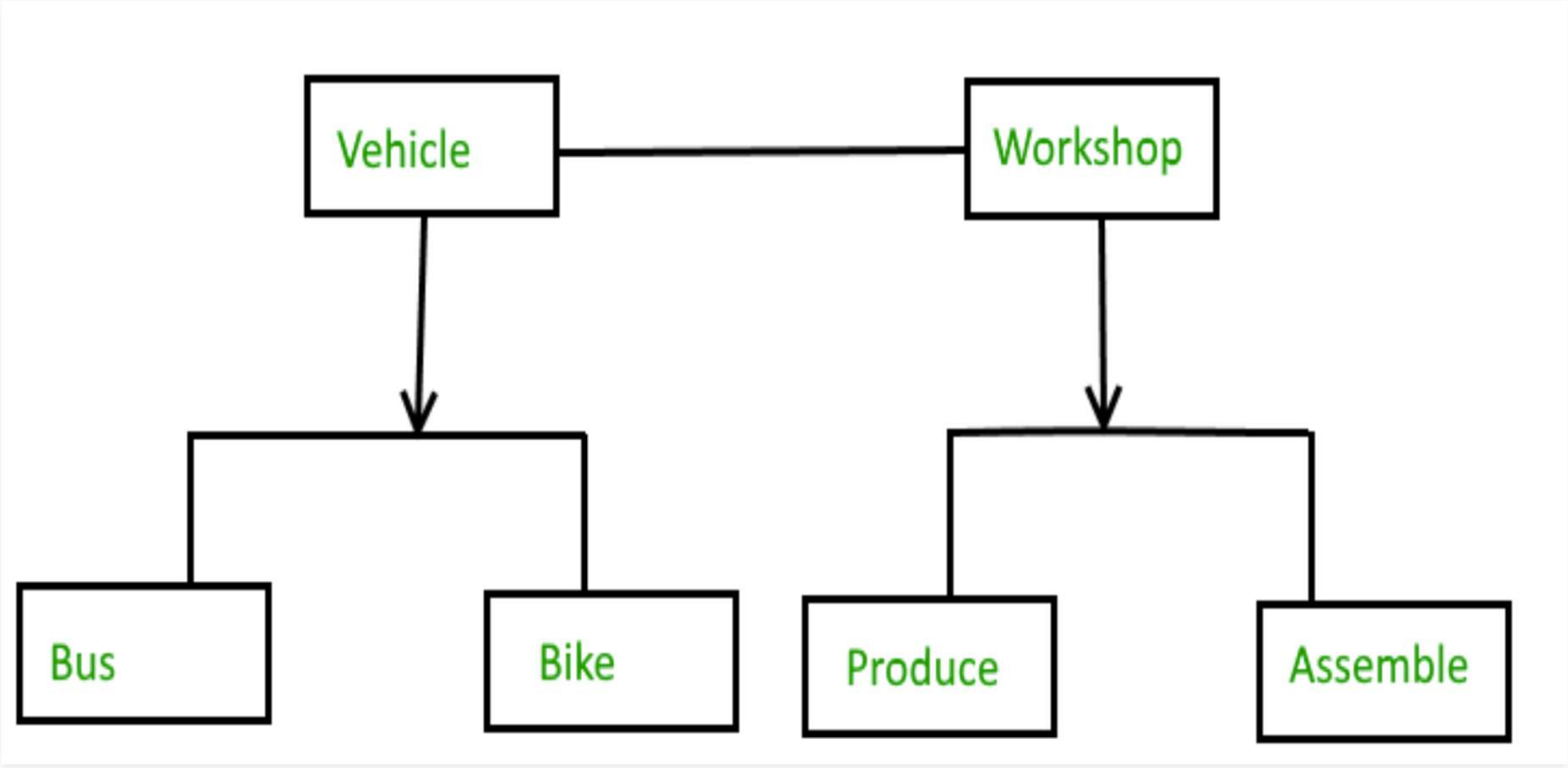
Bridge Pattern-Example 2

Without Bridge design pattern



Bridge Pattern-Example 2

Using Bridge design pattern



Bridge pattern example-Java Translation

```
// Java code to demonstrate
// bridge design pattern

// abstraction in bridge pattern
abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;

    protected Vehicle(Workshop workShop1, Workshop workShop2)
    {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();
}
```

Bridge pattern example-Java Translation

```
// Refine abstraction 2 in bridge pattern
class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture()
    {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }
}
```

Bridge pattern example-Java Translation

```
// Refine abstraction 1 in bridge pattern
class Bus extends Vehicle {
    public Bus(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture()
    {
        System.out.print(" Bus ");
        workShop1.work();
        workShop2.work();
    }
}
```

Bridge pattern example-Java Translation

```
interface Workshop
{
    abstract public void work();
}

// Concrete implementation 1 for bridge pattern
class Produce implements Workshop {
    @Override
    public void work()
    {
        System.out.print("Produced");
    }
}

// Concrete implementation 2 for bridge pattern
class Assemble implements Workshop {
    @Override
    public void work()
    {
        System.out.print(" And");
        System.out.println(" Assembled.");
    }
}
```

Bridge pattern example-Java Translation

```
// Demonstration of bridge design pattern
class BridgePattern {
    public static void main(String[] args)
    {
        Vehicle vehicle1 = new Bus(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();
    }
}
```

Output :

```
Bus Produced And Assembled.
Bike Produced And Assembled.
```

Plan

- Flyweight
- Bridge
- Composite
- Proxy

Composite design pattern

Intent

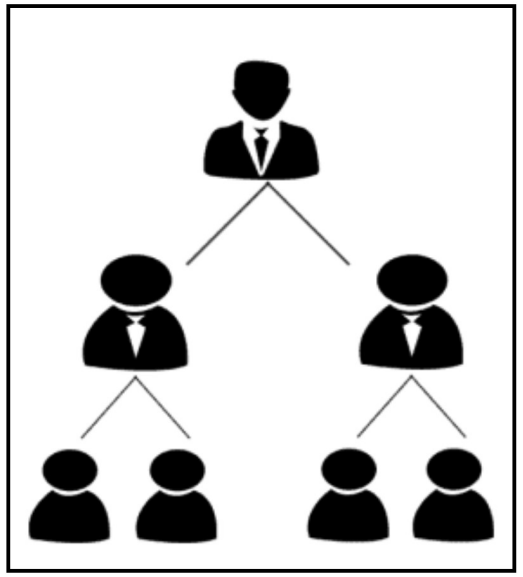
- **Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

- In other words, when we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern

- In object-oriented programming, we make a composite object when we have many objects with common functionalities. This relationship is also termed a “has-a” relationship among objects.

Composite-Example

We can think of any organization that has many departments, and in turn each department has many employees to serve.



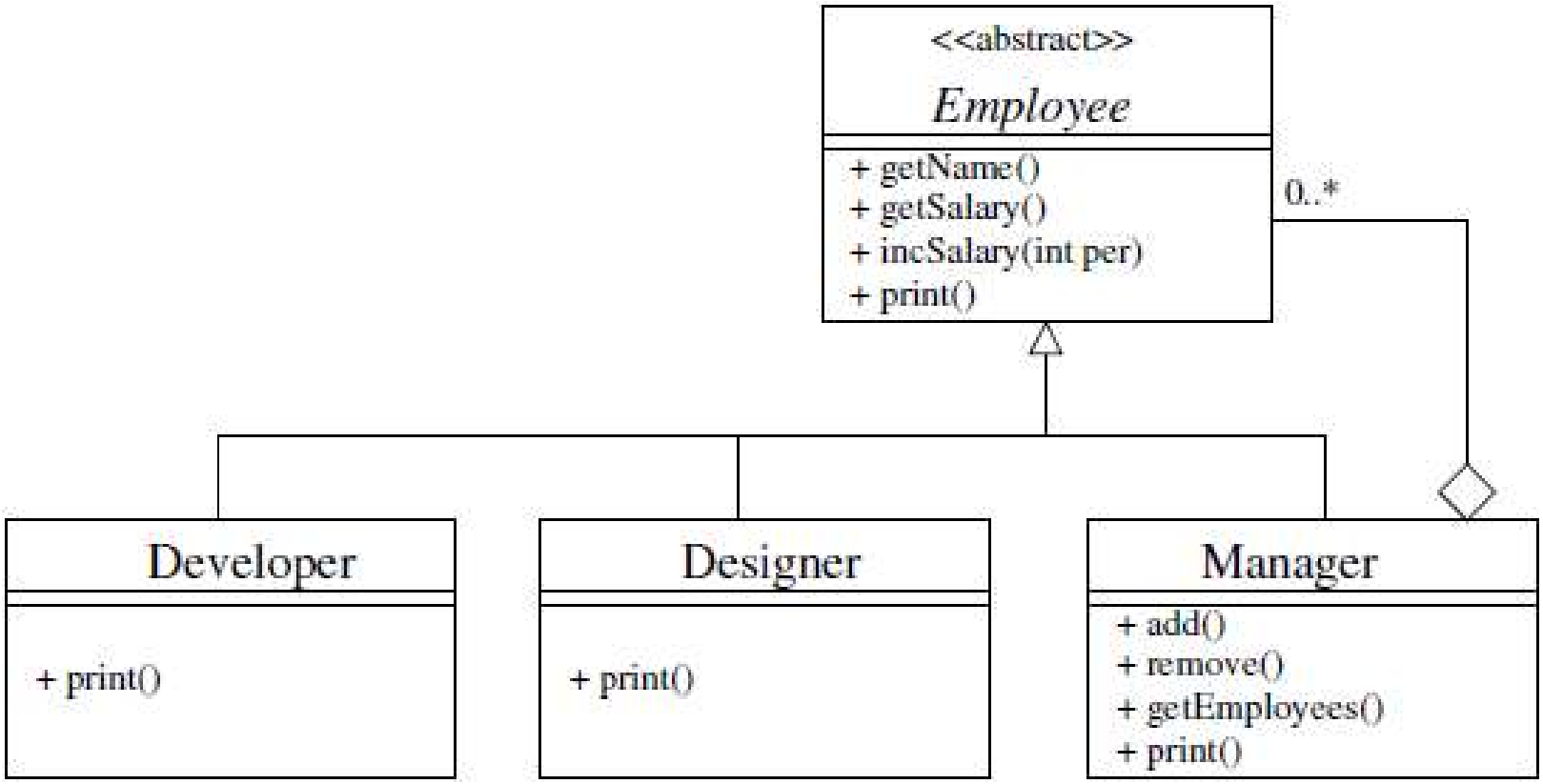
Please note that actually all are employees of the organization.

Groupings of employees create a department, and those departments ultimately can be grouped together to build the whole organization.

Example (Employees)

- You have a set of employees
- An employee could be a manager (of a set of employees), a developer, or a designer

UML Diagram


<<abstract>>
Employee
+ getName()
+ getSalary()
+ incSalary(int per)
+ print()
0..*
Developer
+ print()
Designer
+ print()
Manager
+ add()
+ remove()
+ getEmployees()
+ print()

Java Translation

```
public abstract class Employee {
   protected String name;
   protected int salary;

   public Employee(String name, int salary) {
      this.name = name;
      this.salary = salary;
   }

   public String getName() {
      return name;
   }

   public int getSalary() {
      return salary;
   }

   public void increaseSalary(int percentage) {
      salary *= 1 + percentage/100.0;
   }

   public abstract void print();
}
```

Java Translation

```
public class Manager extends Employee {
   protected List<Employee> employees = new LinkedList<Employee>();

   public Manager(String name, int salary) {
      super(name, salary);
   }

   public void add(Employee employee) {
      employees.add(employee);
   }

   public Iterator<Employee> getEmployees() {
      return employees.iterator();
   }

   public void remove(Employee employee) {
      employees.remove(employee);
   }

   public void print() {
      System.out.println("Manager " + name);
   }
}
```

Java Translation

```
public class Developer extends Employee {
    public Developer(String name, int salary) {
        super(name, salary);
    }

    public void print() {
        System.out.println("Developer " + name);
    }
}
```

```
public class Designer extends Employee {
    public Designer(String name, int salary) {
        super(name, salary);
    }

    public void print() {
        System.out.println("Designer " + name);
    }
}
```

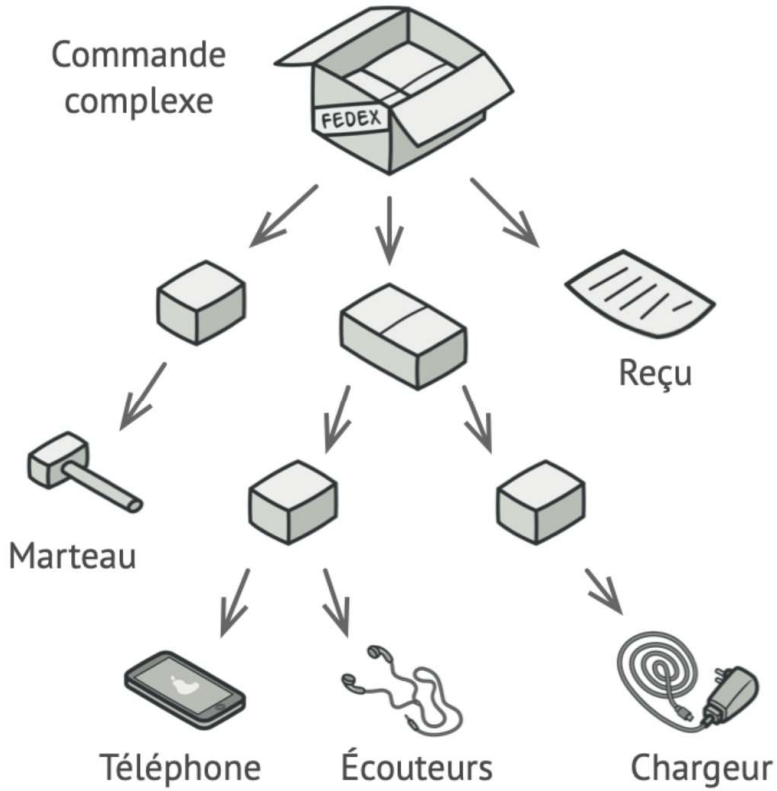
Java Translation

```
public class Company
{
    public static void main (String[] args)
    {
        Developer dev1 = new Developer("Lokesh Sharma", 100 )
        Developer dev2 = new Developer("Vinay Sharma",101 )
        Manager engDirectory = new Manager("Vikram Sharma ", 201)
        engDirectory.add(dev1);
        engDirectory.add(dev2);

        Manager accDirectory = new Manager("Kushagra Garg",200)

        Designer d1 = new Designer  ("Lok rma", 100 )
        Designer d2 = new Designer  ("Vin Sha",101 )
        accDirectory.add(d1);
        accDirectory.add(d2);

       Manager directory = new Manager("ram Vikrma ", 221)
       directory.add(engDirectory);
       directory.add(accDirectory);

   }
}
```

Composite pattern-Problem

A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.

How would you determine the total price of such an order?



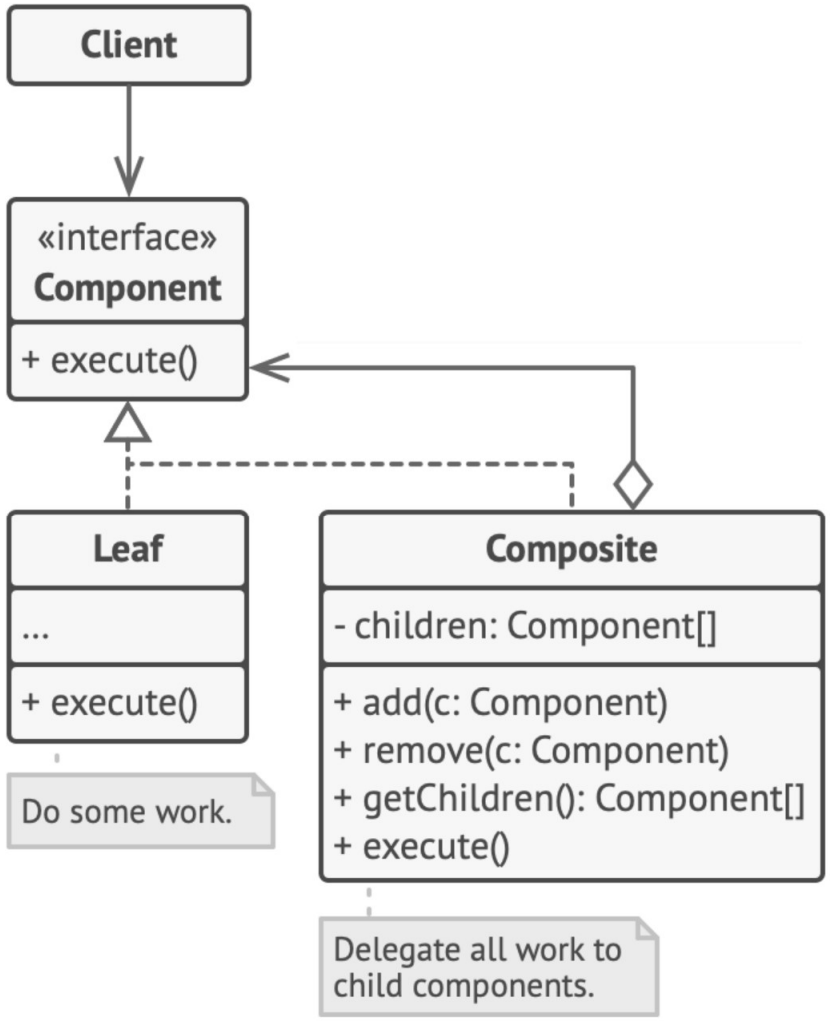
You could try the **direct approach**: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop. You have to know the classes of Products and Boxes you're going through, the nesting level of the boxes and other nasty details before hand. All of this makes the direct approach either too awkward or even impossible.

Composite pattern-Solution

The **Component** interface describes operations that are common to both simple and complex elements of the tree.

The **Leaf** is a basic element of a tree that doesn't have sub-elements.

The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.
Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.



Plan

- Flyweight
- Bridge
- Composite
- Proxy

Proxy design pattern

Intent

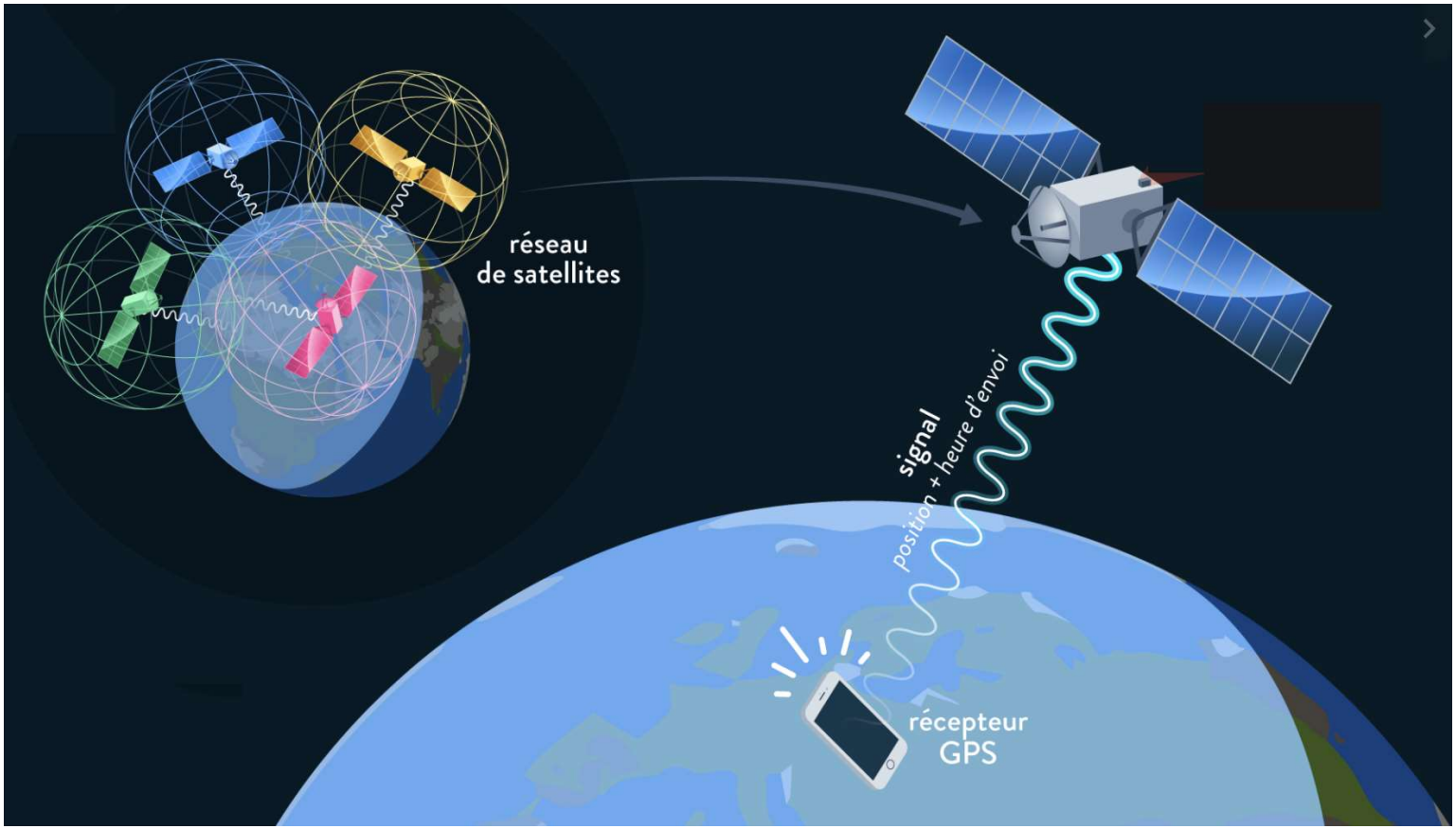
- **Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Types of proxies:

- **Virtual proxy**: create an expensive object if needed only
- **Protection proxy**: controls access to the original object
- .....

Poxy design pattern-Problem

A GPS navigation system loads satellite images of the planet over the network, which is very expensive. However, we want to see all of these images as a single image taking advantage of the fact that downloading a section is only necessary when that section is actually displayed.



Poxy design pattern-Problem/Solution

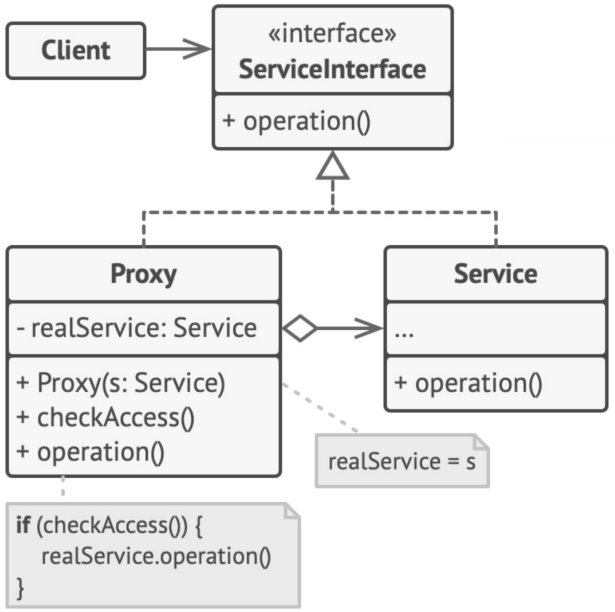
Problem : How to implement the interface of an image without having all the data of this image loaded in memory.

Solution : The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

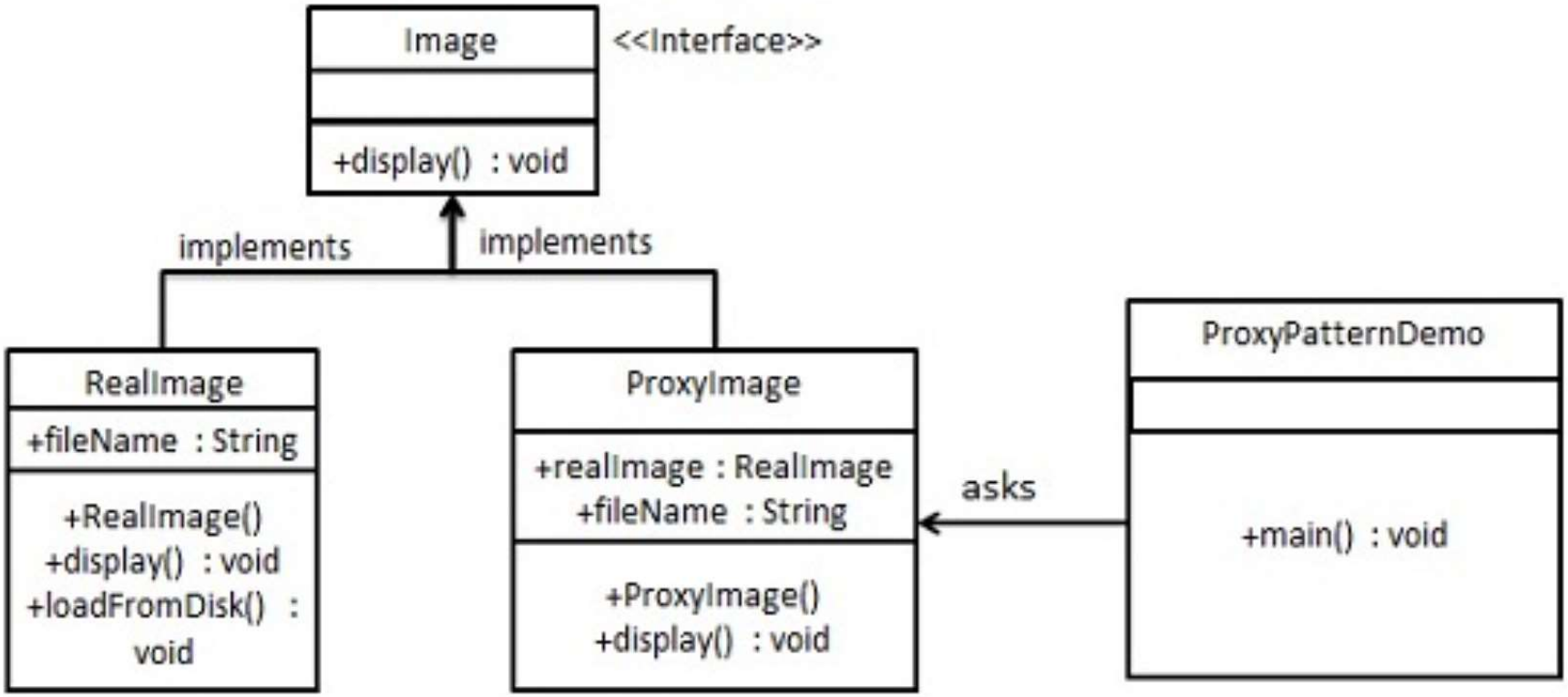
The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

The **Service** is a class that provides some useful business logic.



Poxy design pattern-Problem/Solution



Java Translation

Example, virtual proxy

```
public interface image {
    public void affichage ();
}

public class RealImage implements image {
     private String fileName ;
   public RealImage  (String fileName) {
      this.fileName = fileName;
      loadFromDisk (fileName);
   }
   @Override
   public void affichage () {
      System.out.println ( "Résultats " + fileName);
   }
   private void loadFromDisk (String fileName) {
      System.out.println ( "Chargement " + fileName);
   }
}
```

Java Translation

```
public class ProxyImage implements image{
    private RealImage realImage;
    private String fileName ;

    public ProxyImage  (String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void affichage() {
        if (realImage == null) {
            realImage = new RealImage (fileName);
        }
        realImage.affichage();
    }
}
```

Java Translation

```
public class ProxyPatternDemo {
        public static void main(String[] args) {
        image img = new ProxyImage ( "de test_10mb.jpg");
        img.affichage();
    }
}
```

Result:

```
Chargement de test_10mb.jpg
Résultats de test_10mb.jpg
```

Example – Protection Proxy

A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Java Translation

```
public interface Internet
{
    public void connectTo(String serverhost) throws Exception;
}
```

```
public class RealInternet implements Internet
{
    @Override
    public void connectTo(String serverhost)
    {
        System.out.println("Connecting to "+ serverhost);
    }
}
```

Java Translation

```
public class ProxyInternet implements Internet
{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;

    static
    {
        bannedSites = new ArrayList<String>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
        bannedSites.add("lnm.com");
    }

    @Override
    public void connectTo(String serverhost) throws Exception
    {
        if(bannedSites.contains(serverhost.toLowerCase()))
        {
            throw new Exception("Access Denied");
        }

        internet.connectTo(serverhost);
    }

}
```

Java Translation

```
public class Client
{
    public static void main (String[] args)
    {
        Internet internet = new ProxyInternet();
        try
        {
            internet.connectTo("geeksforgeeks.org");
            internet.connectTo("abc.com");
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Result

```
Connecting to geeksforgeeks.org
Access Denied
```

Questions?