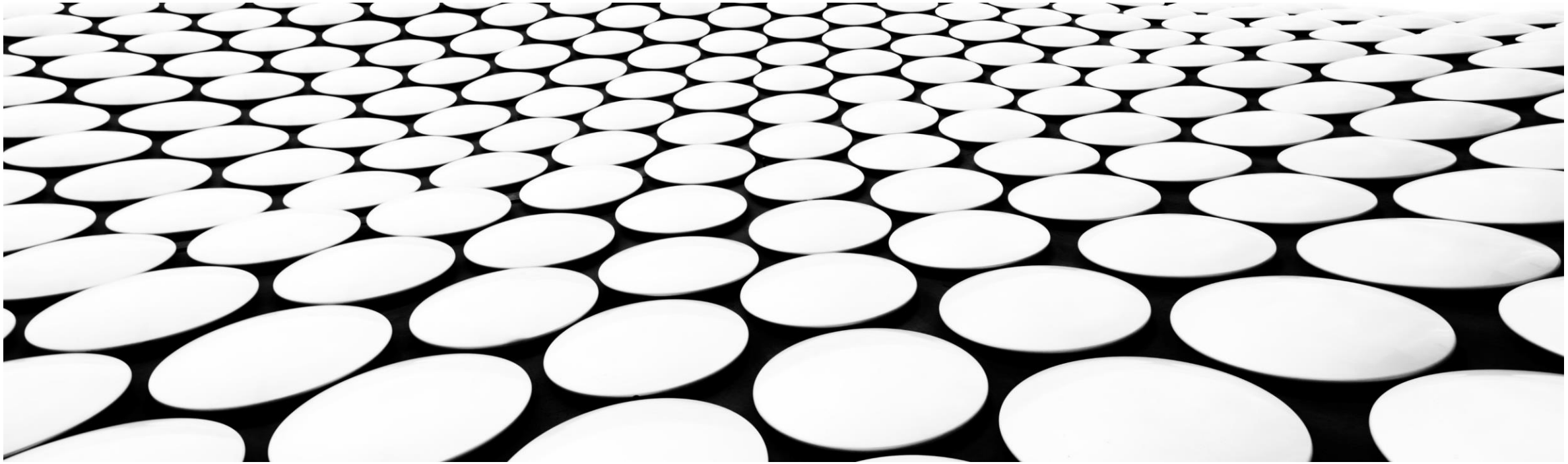
STRUCTURAL DESIGN PATTERNS

FACADE , FLYWEIGHT, PROXY PATTERN

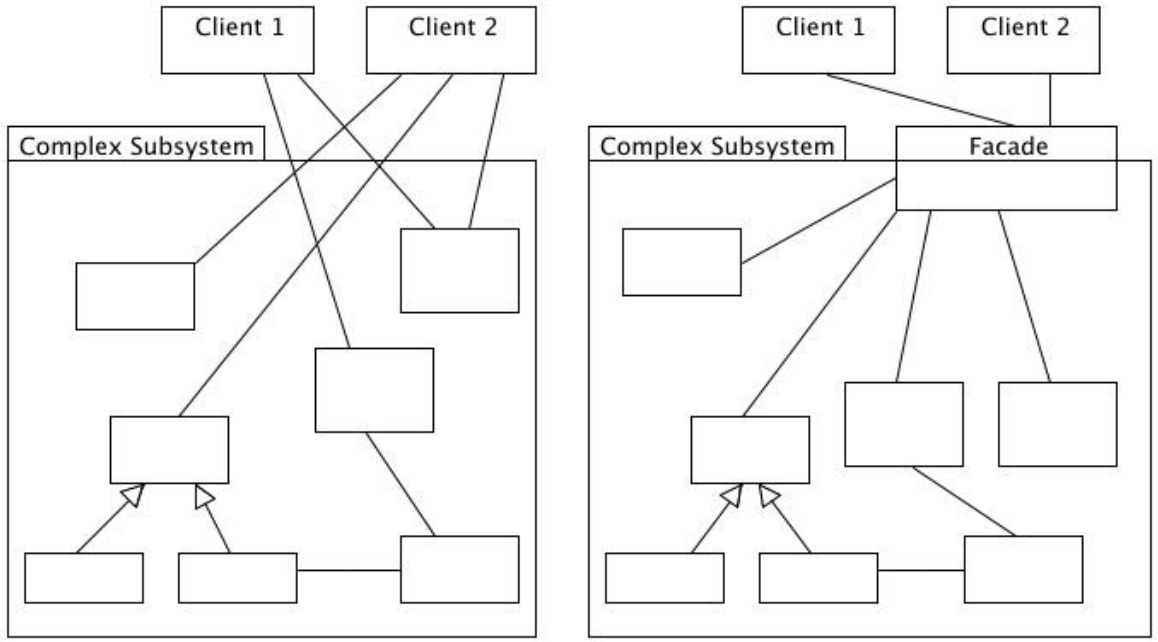


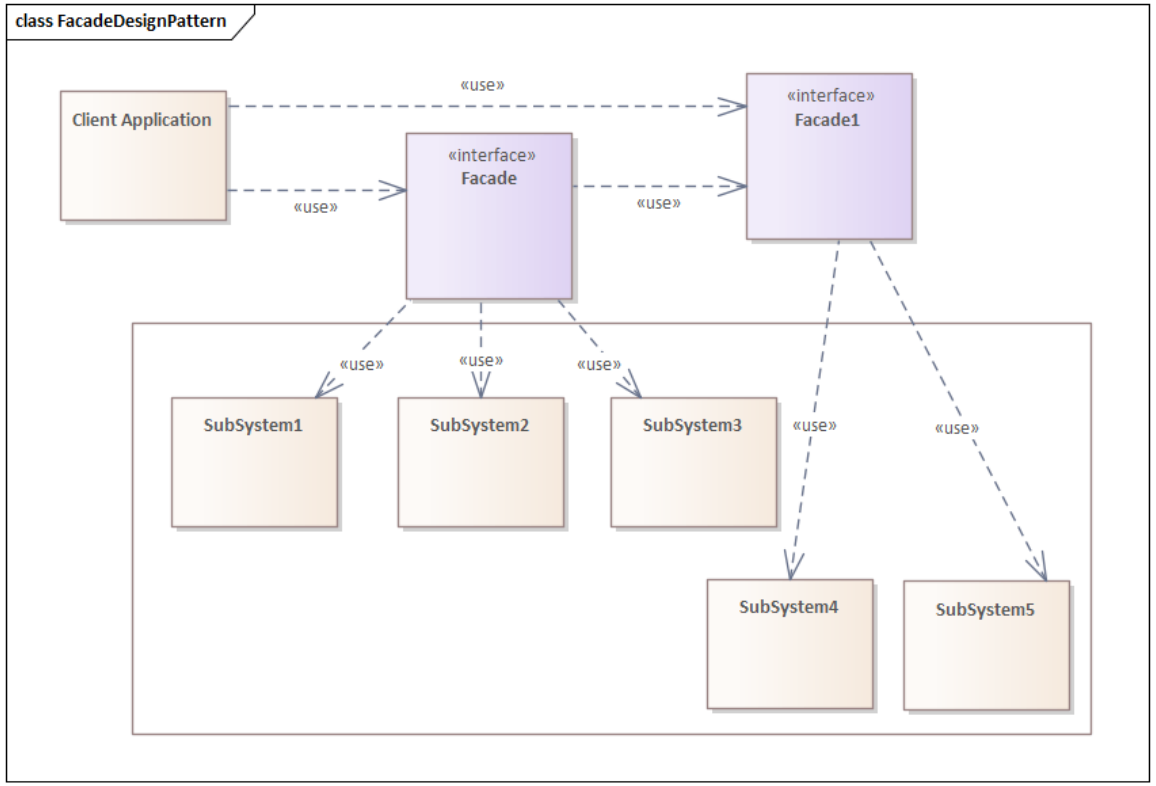
FACADE PATTERN

- Facade Pattern provides a unified interface to a set of interfaces in as subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- The Facade Pattern leaves the subsystem accessible to be used directly.
- Practically, **every Abstract Factory** is a type of **Facade.**
- Abstract Factory can serve as an alternative to Facade when you only want to hide the way the subsystem objects are created from the client code.
- So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.
- We can use Factory pattern with Facade to provide better interface to client systems

WHERE TO USE

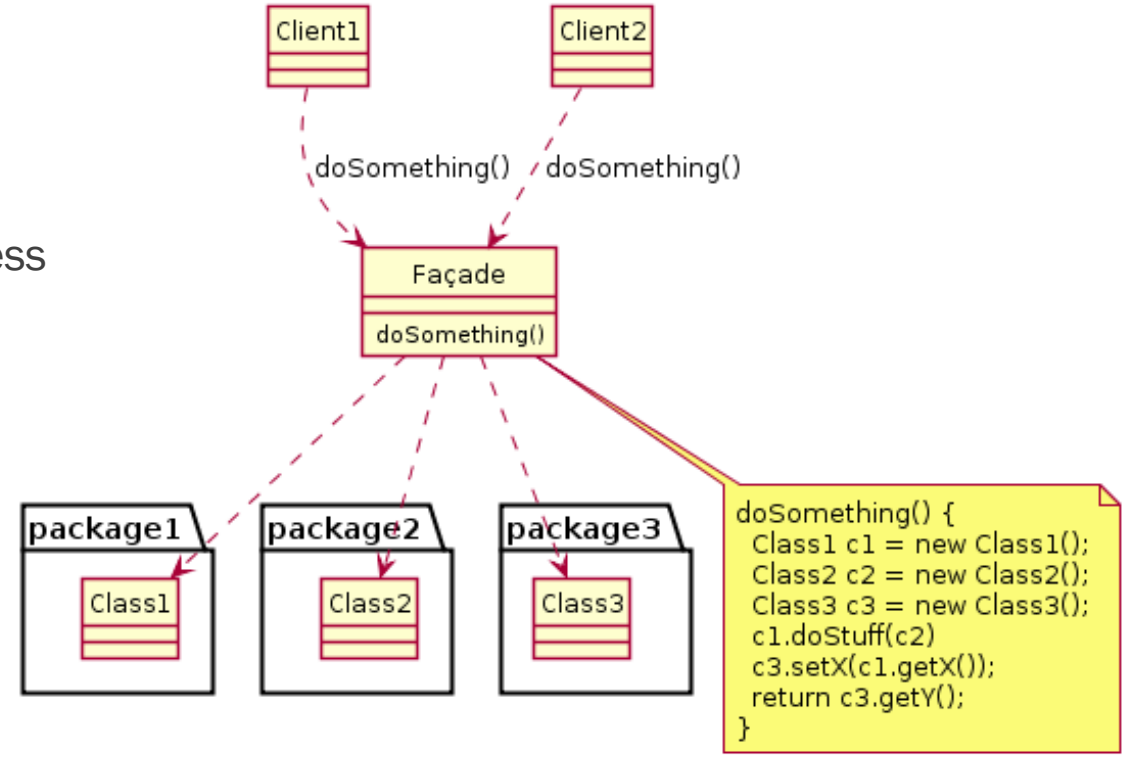
- using Facade is super-easy when we have to deal with a complex system/subsystem having lots of functionalities and different configurations..
- you want to reduce coupling between clients-subsystems or subsystems-subsystems.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level and make them communicate only through their facades, this can simplify the dependencies between them.





UML CLASS DIAGRAM

- Façade :The facade class abstracts Packages 1, 2, and 3 from the rest of the application.
- Clients : The objects are using the Facade Pattern to access resources from the Packages.



EXAMPLE:

- In this Example, our aim is to build robots. And from a user point of view he needs to supply only the color and material for his her robot through the RobotFacade .
- Our RobotFacade will in turn create objects for RobotBody, RobotColor, RobotMetal and will do the rest for the user.
- We need not worry about the creation of these separate classes and their calling sequence.
- All of the classes have their corresponding implementation.

FACADE

```
public class RobotFacade {

    RobotColor rc;
    RobotMetal rm;
    RobotBody rb;

    public RobotFacade() {
        rc = new RobotColor();
        rm = new RobotMetal();
        rb = new RobotBody();

    }

    public void ConstructRobot(String color, String metal) {
        System.out.println("\nCreation of the Robot Start");
        rc.SetColor(color);
        rm.SetMetal(metal);
        rb.CreateBody();
        System.out.println(" \nRobot Creation End");
        System.out.println();
    }
}
```

SUBSYSTEM CLASSES

```java
public class RobotBody {

    public void CreateBody() {
        System.out.println("Body Creation done");
    }
}
```

```java
public class RobotColor {

    private String color;

    public void SetColor(String color) {
        this.color = color;
        System.out.println("Color is set to : " + this.color);
    }
}
```

```java
public class RobotMetal {

    private String metal;

    public void SetMetal(String metal) {
        this.metal = metal;
        System.out.println("Metal is set to : " + this.metal);
    }
}
```

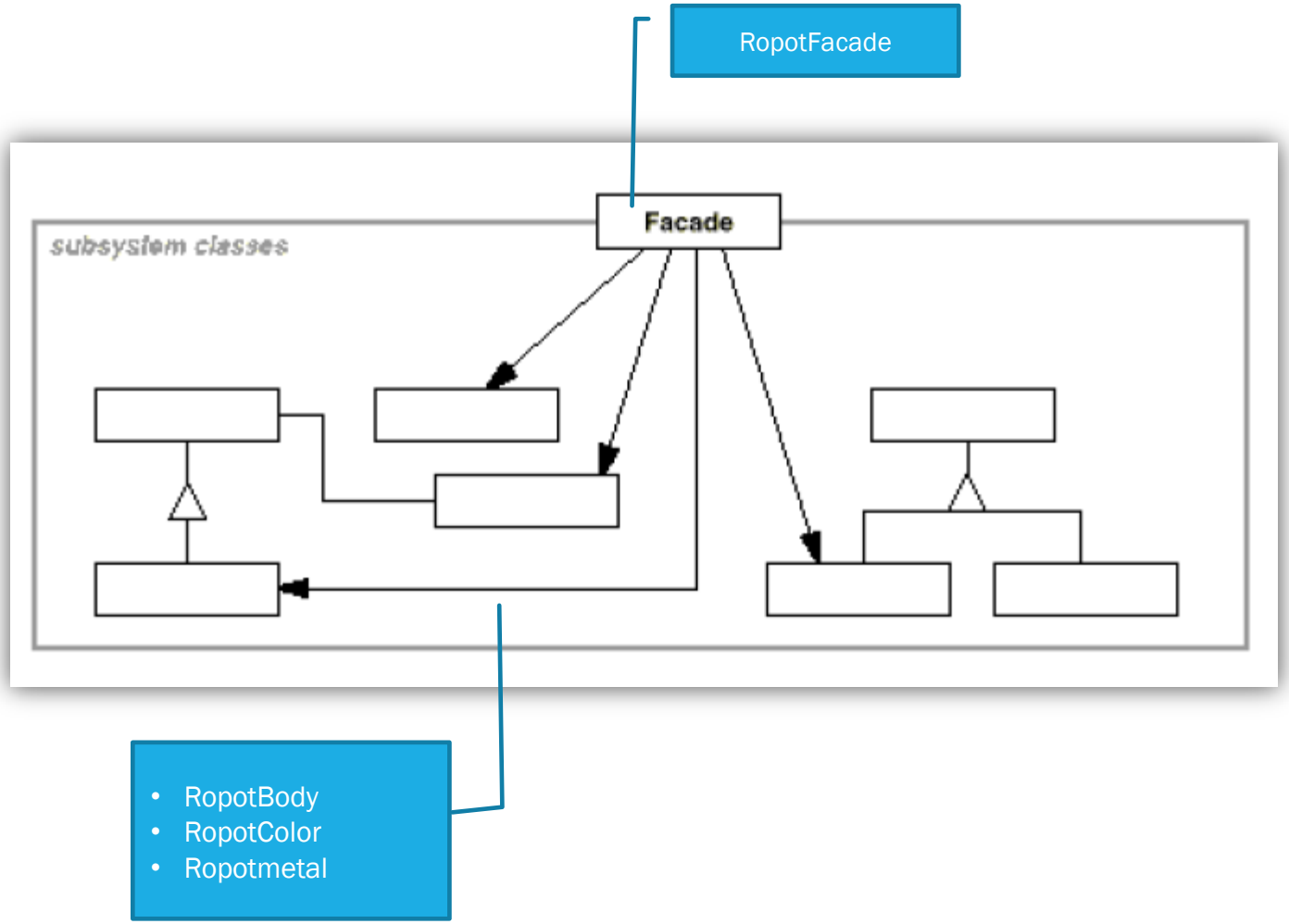
CLIENT

```
class FacadePatternEx {

    public static void main(String[] args) {
        System.out.println("***Facade Pattern Demo***");
        RobotFacade rf1 = new RobotFacade();
        rf1.ConstructRobot("Green", "Iron");
        RobotFacade rf2 = new RobotFacade();
        rf2.ConstructRobot("Blue", "Steel");

    }
}
```

OUTPUT:

```
run:
***Facade Pattern Demo***

Creation of the Robot Start
Color is set to : Green
Metal is set to : Iron
Body Creation done

Robot Creation End


Creation of the Robot Start
Color is set to : Blue
Metal is set to : Steel
Body Creation done

Robot Creation End
```

PARTICIPANTS:



PARTICIPANTS

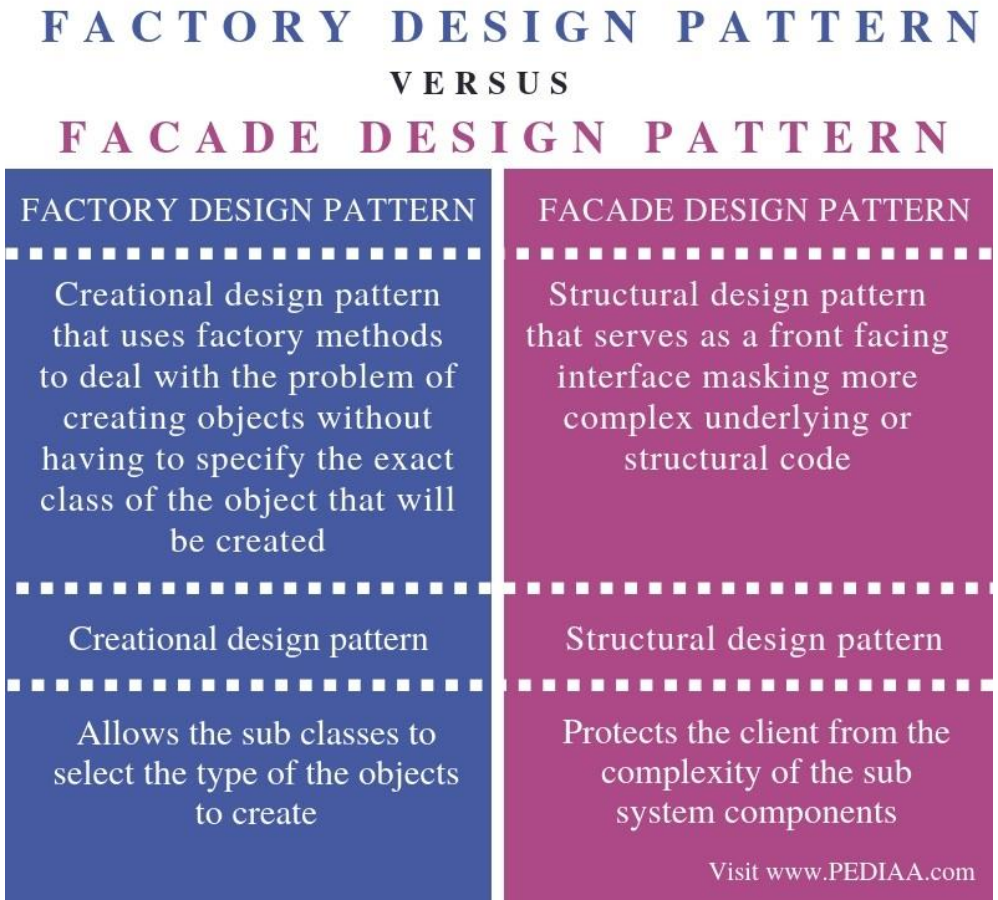
Participant	Class Or Interface
Facade	RopotFacade class
subsystem classes	• RopotBody • RopotColor • Ropotmetal
Client	FacadePatternEx class

FAÇADE VS. FACTORY

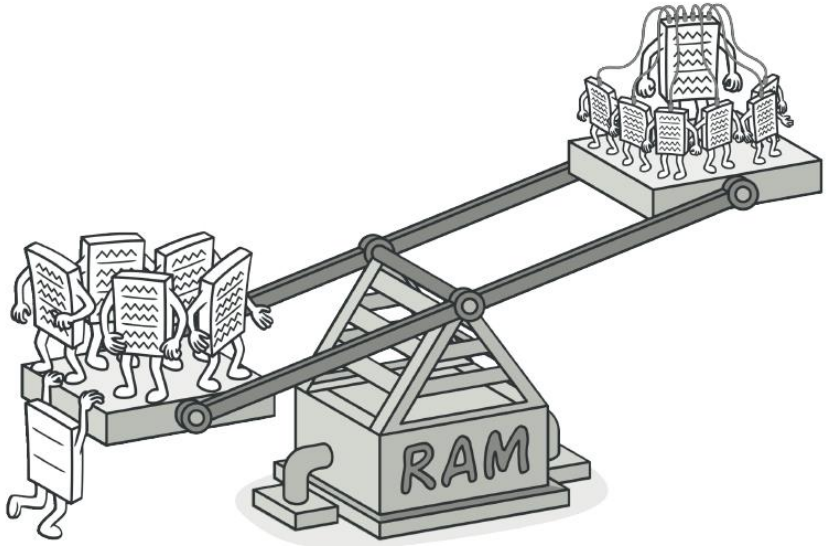
- **The facade pattern is used when you want to hide an implementation** or it is about changing interface of some class or set of classes. Builder hides the process of construction by decomposing it in smaller steps.
- **Abstract factory pattern is used when you want to hide the details on constructing instances.** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

The factory design pattern allows the subclasses to select the type of objects to create, whereas the facade design protects the client from the complexity of the subsystem components.

API's are Facades for example. The access to the inner workings are regulated through well defined points. You could restrict some elements that want to pass with a filter at that gateway, you could let messages inside and process it and deliver the right answer back or close off the door entirely.



FLYWEIGHT PATTERN



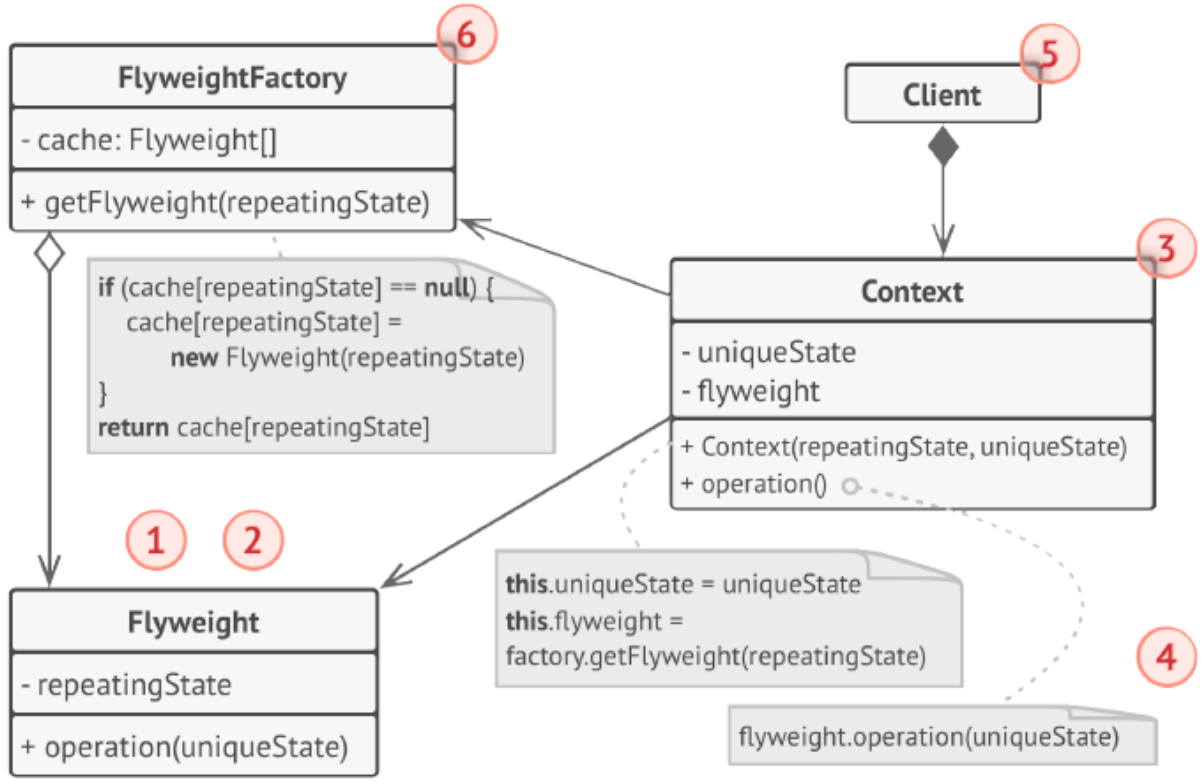
- You want to draw a forest of trees! How would you implement this?
- Millions of tree objects in the program? Memory constraint
- What is the optimal solution?
- Can we implement it so that we create only 1 tree object?
- Flyweight can be recognized by a creation method that returns cached objects instead of creating new.
- That's where flyweight comes in!
- If the objects you create have so much common properties (like trees, only positions are different).
- 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.

WHERE TO USE

- When there is a very large number of objects that may not fit in memory.
- When most of an objects state can be stored on disk or calculated at runtime.
- When there are groups of objects that share state.

CONCEPT

- 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 and extrinsic state.
  - coordinates, movement vector and speed, are unique to each particle. After all, the values of these fields change over time. This data represents the always changing context in which the particle exists, while the color and sprite remain constant for each particle.
  - 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*.
  - The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts.
  - an object that only stores the intrinsic state is called a flyweight.
  - a thousand small contextual objects can reuse a single heavy flyweight object instead of storing a thousand copies of its data.



- The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.

- **Flyweight factory** you can create a factory method that manages a pool of existing flyweight objects. The method accepts the intrinsic state of the desired flyweight from a client, looks for an existing flyweight object matching this state, and returns it if it was found. If not, it creates a new flyweight and adds it to the pool.
- 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."
- The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.

RENDERING A FOREST

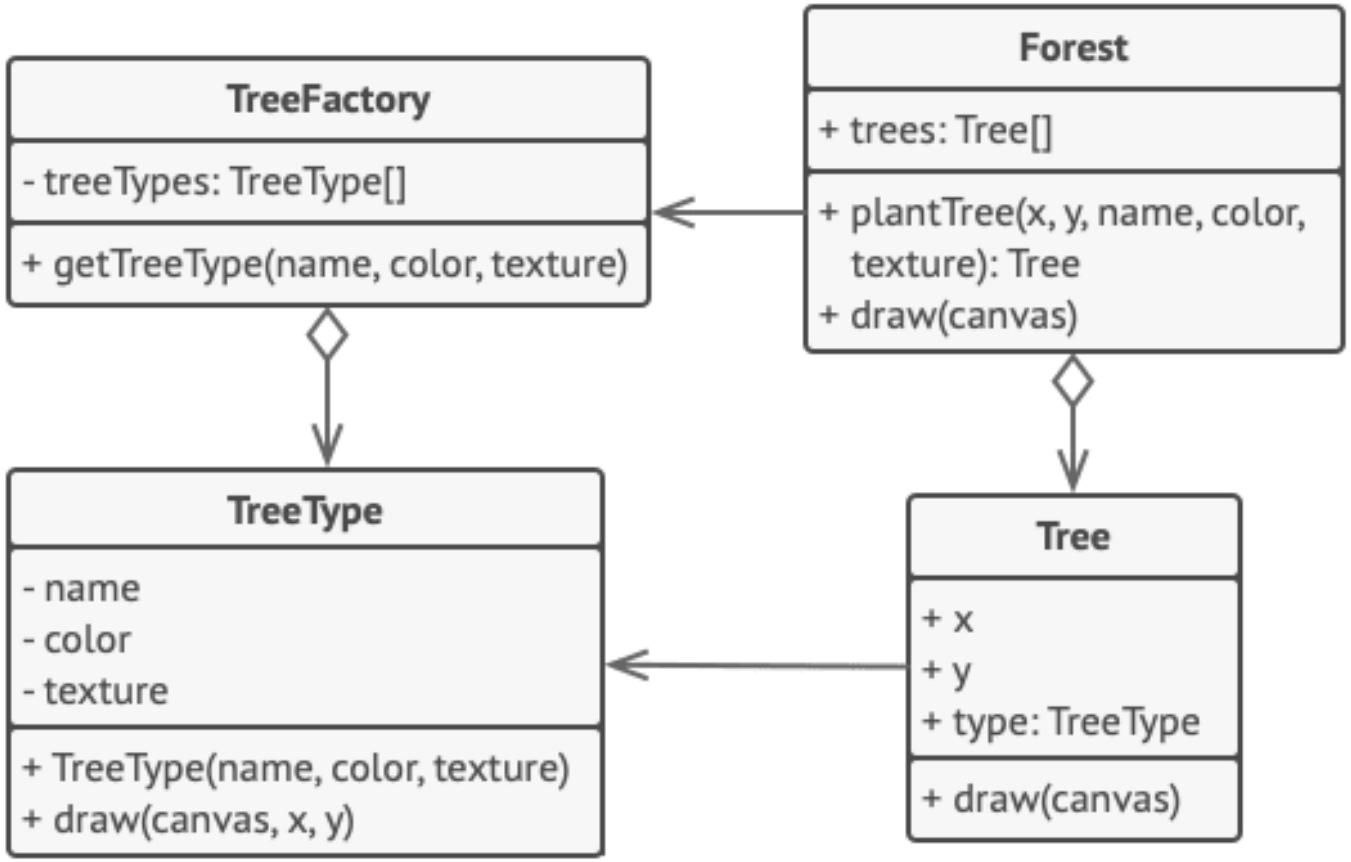
- In this example, we're going to render a forest (1.000.000 trees)! Each tree will be represented by its own object that has some state (coordinates, texture and so on). Although the program does its primary job, naturally, it consumes a lot of RAM.

- The reason is simple: too many tree objects contain duplicate data (name, texture, color). That's why we can apply the Flyweight pattern and store these values inside separate flyweight objects (the TreeType class). Now, instead of storing the same data in thousands of Tree objects, we're going to reference one of the flyweight objects with a particular set of values.

- The client code isn't going to notice anything since the complexity of reusing flyweight objects is buried inside a flyweight factory.

UML OF FOREST EXAMPLE

- The pattern extracts the repeating intrinsic state from a main Tree class and moves it into the flyweight class TreeType.



- Tree.java: Contains state unique for each tree

```
import java.awt.*;

public class Tree {
    private int x;
    private int y;
    private TreeType type;
    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;    }
    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

- TreeType.java: Contains state shared by several trees

```
import java.awt.*;

public class TreeType {
    private String name;
    private Color color;
    private String otherTreeData;

    public TreeType(String name, Color color, String
otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;
    }
    public void draw(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);
    }
}
```

- TreeFactory.java: Encapsulates complexity of flyweight creation

```
import java.awt.*;
import java.util.HashMap;
import java.util.Map;

public class TreeFactory {
    static Map<String, TreeType> treeTypes = new
HashMap<>();

    public static TreeType getTreeType(String name, Color color,
String otherTreeData) {
        TreeType result = treeTypes.get(name);
        if (result == null) {
            result = new TreeType(name, color, otherTreeData);
            treeTypes.put(name, result);
        }
        return result;
    }
}
```

- Forest.java: Forest, which we draw

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name, Color color,
String otherTreeData) {
        TreeType type = TreeFactory.getTreeType(name, color,
otherTreeData);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);    }

    @Override
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }    }   }
```

```
public class Demo {
    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 1000000;
    static int TREE_TYPES = 2;

    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Summer Oak", Color.GREEN, "Oak texture stub");
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");        }
        forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
        forest.setVisible(true);

        System.out.println(TREES_TO_DRAW + " trees drawn");
        System.out.println("-------------------");
        System.out.println("Memory usage:");
        System.out.println("Tree size (8 bytes) * " + TREES_TO_DRAW);
        System.out.println("+ TreeTypes size (~30 bytes) * " + TREE_TYPES + "");
        System.out.println("-------------------");
        System.out.println("Total: " + ((TREES_TO_DRAW * 8 + TREE_TYPES * 30) / 1024 / 1024) +
                "MB (instead of " + ((TREES_TO_DRAW * 38) / 1024 / 1024) + "MB");     }

    private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));      }      }
```

OUTPUT

RAM usage stats

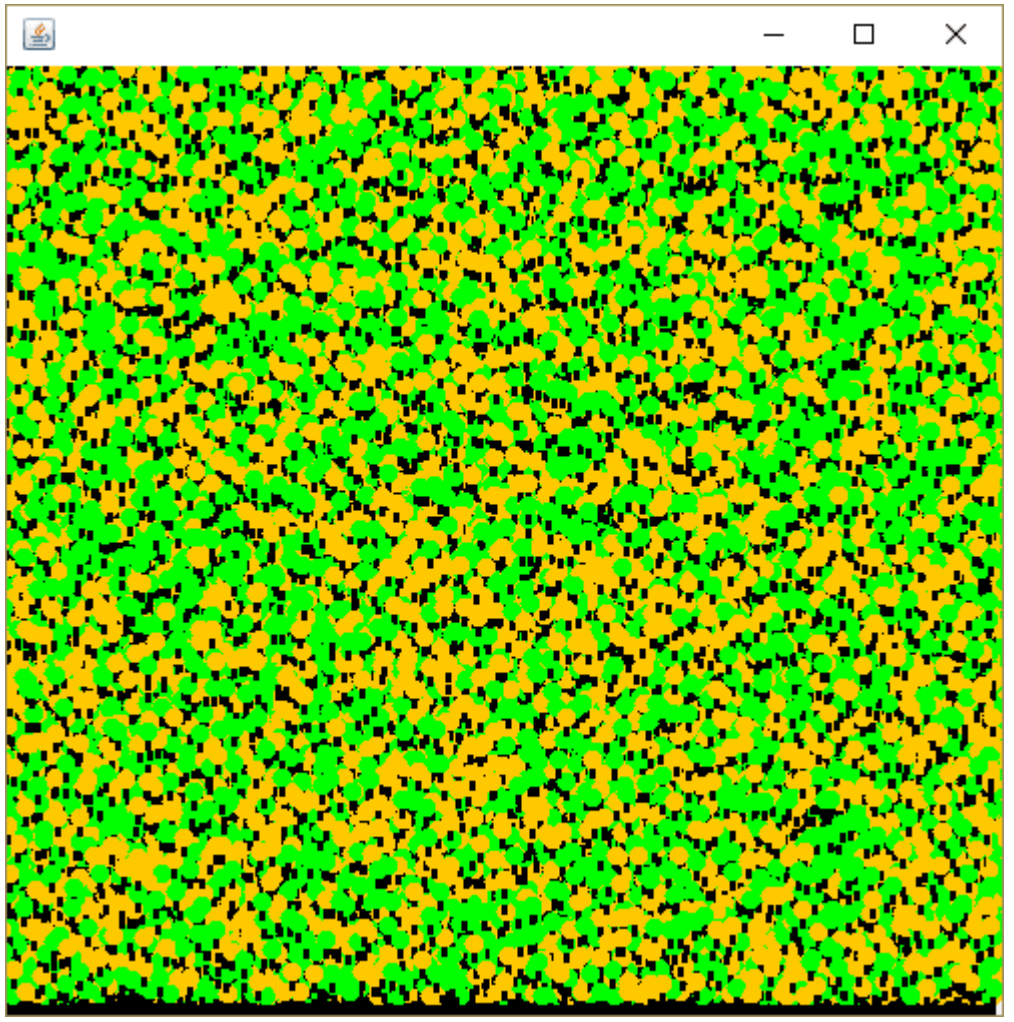
1000000 trees drawn

Memory usage:

Tree size (8 bytes) * 1000000

+ TreeTypes size (~30 bytes) * 2

Total: 7MB (instead of 36MB)



PROS & CONS

- Pros
  - You can save lots of RAM, assuming your program has tons of similar objects.

- Cons
  - One of the main disadvantages of this pattern is that all class objects are connected, so that only one class object can not be independent of other instances.
  - Flyweight design pattern can cause disruptions that can take place better during the earlier load if a vast number of objects are required at once. (You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.)

FLYWEIGHT VS PROTOTYPE

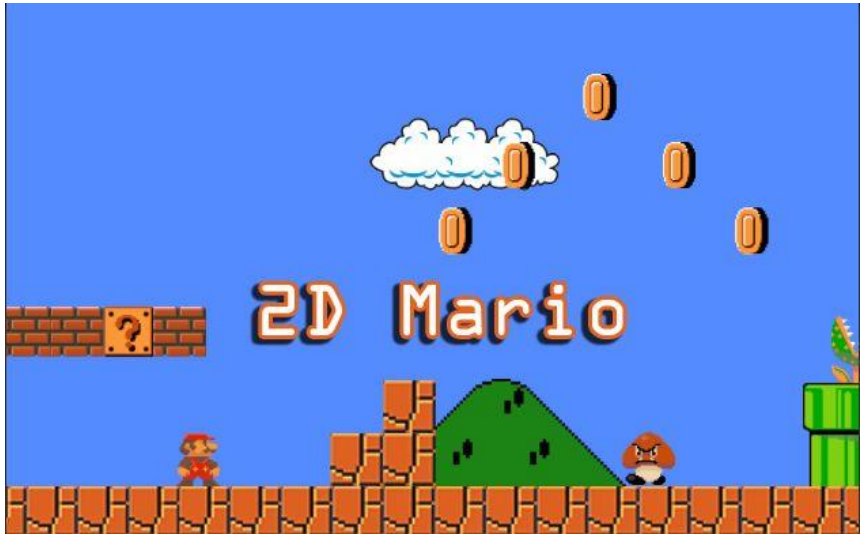
- Flyweight is a **structural** pattern and the prototype is a **creational** design pattern.
- In the prototype pattern, we **simply clone repeatedly one object**, and in the Flyweight pattern we **use the created objects again by sharing them**, new objects are created only when there is no such or similar object in the application.
- The prototype is for creating new **instances or cloning them**, Flyweight is for **creating and sharing them**.
- Flyweight is used when creating multiple type of single object.
  Prototype is used when creating single type of single object.
- In Flyweight, object is immutable. In Prototype, object is mutable.

FLYWEIGHT VS SINGLETON

- Flyweight is when you have *many different kinds* of a single thing. Singleton is when you have a single thing.
- For example, you might use the Flyweight Pattern to represent keyboard characters. One object for `a`, one for `b`, etc. They are both characters, but different *kinds* of characters. In contrast, you might use a singleton to represent the keyboard. There can only be one.

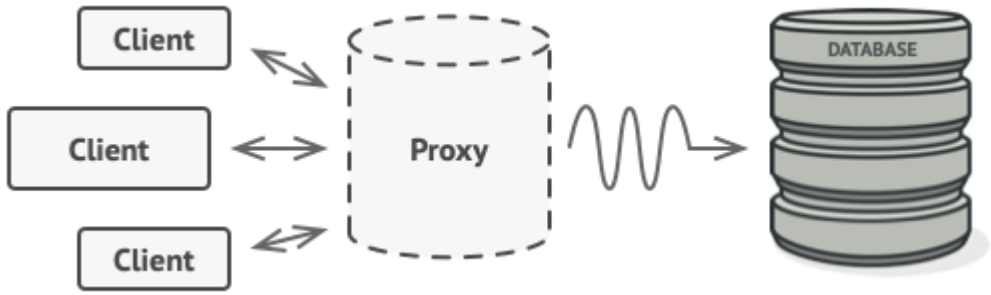
SCENARIO TO DISTINGUISH BETWEEN PROTOTYPE AND FLYWEIGHT

- You ‘ve asked to programmed 2D platformer game like Mario (in Java). There are a lot of enemies which are the same the only difference is their position [x,y]. There are also walls which are built from a huge number of rectangles and again the only difference is their position [x,y]
- Should you use prototype to clone objects via cloneable and then set [x,y]?
- Is it better to use flyweight - when you need new object just return them from hashmap and then set [x,y]?
- In both scenarios to avoid using new operator but which one to use.



- Prototype is used to create new instances, Flyweight is used to allow sharing of instances.
- Prototype would mean that you have an EnemyPrototype (or several) and you create a new enemy from that. In a naive implementation this would duplicate all the data, including the graphics. So for 100 enemies you would have the same image 100 times in memory (not a good thing).
- As for Flyweight, you would share the graphics.

PROXY PATTERN



- Provide a surrogate or placeholder for another object to control access to it.
  - Provides extra level of indirection
  - Protects real component from undue complexity
- Also Known as : **SURROGATE**
- **Proxy** is a structural design pattern that provides an object that acts as a substitute for a real service object used by a client. A proxy receives client requests, does some work (access control, caching, etc.) and then passes the request to a service object.
- The proxy object has the same interface as a service, which makes it interchangeable with a real object when passed to a client.

WHERE EXACTLY IS THE PROXY PATTERN USED ?

- When the creation of one object is relatively expensive it can be a good idea to replace it with a proxy that can make sure that instantiation of the expensive object is kept to a minimum.
- Proxy pattern implementation allows for login and authority checking before one reaches the actual object that's requested.
- Can provide a local representation for an object in a remote location.

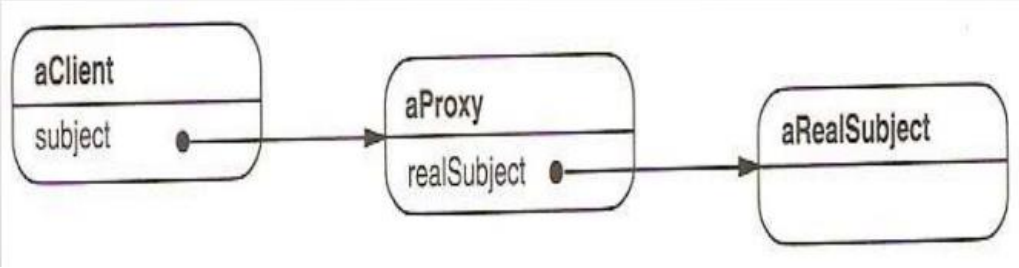
TYPES OF PROXY DESIGN PATTERN

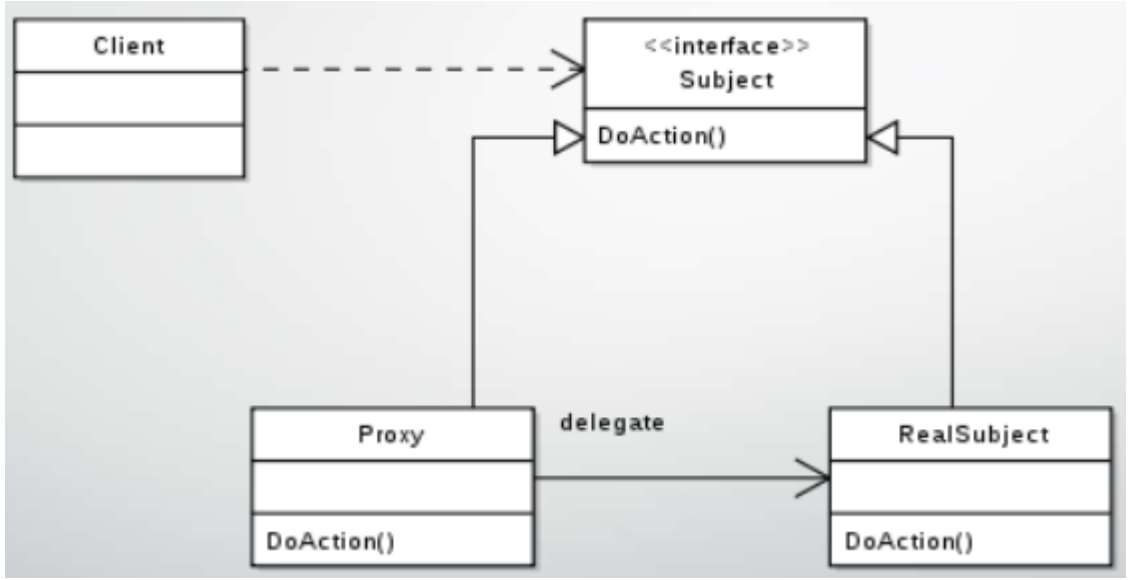
- **Virtual Proxy:** When the subject is pretty resource-intensive to instantiate, we might choose to use this pattern. The proxy class we create here is known as a virtual proxy. Some example use cases would include loading very high-resolution images on a web page. The idea is to delay the creation of an expensive resource until the time it is needed
- **Protection Proxy:** We can also use a proxy class to control access to our real subject class. For instance, allowing users to access a website based on their specific user roles .The protective proxy acts as an authorization layer to verify if the actual user has access to appropriate content. An example can be thought about the proxy server which provides restrictive internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
- **Remote Proxy:** A real-world example of this implementation would be that of a Google Docs. The web browser holds the proxy objects locally which are then synced with the objects at the remote server
- **Smart Proxy**. Smart proxies add extra functionality to the calls to the real object's members foe ex confirm that it locked the actual object before accessing it to make sure that no other object may alter it. You could also use a smart proxy to log calls to the underlying object's members.

WHAT IS LIKE ,THE STRUCTURE OF THIS PATTERN ?

- **Proxy:**
  - Maintains a reference that lets the proxy access the real subject.
  - Provides an interface identical to subject's so that a proxy can by substituted for the real subject.
  - Controls access to the real subject and may be responsible for creating and deleting it.
- **Subject**: Defines the common interface for Real Subject and Proxy so that a Proxy can be used anywhere a Real Subject is expected.
- **Real Subject**: Defines the real object that the proxy represents.

STRUCTURE – OBJECT DIAGRAM





- Proxy forwards requests to Real Subject when appropriate, depending on the kind of proxy

PROTECTION PROXY EXAMPLE

- Most of the organizations provide restricted access to the internet within their premises. So, how is it implemented?
- The idea is to create a protection proxy.
- Let's start by defining a *WebServer* interface:

```
public interface WebServer {
    void makeRequest(String url);
}
```

- Here, the makeRequest() method is responsible for making a call to the webserver with a specific endpoint.
- Let's now implement the RealWebServer class which does the actual job of hitting a URL via network API calls:

```
public class RealWebServer implements WebServer {

    @Override
    public void makeRequest(String url) {
    //code to hit a particular url
    }
}
```

- Finally, we'll create a proxy server and expose it to our clients:

```
public class ProxyWebServer implements WebServer {

    private RealWebServer realServer;
    private List<String> blockedSites = new ArrayList<>();

    public ProxyWebServer() { this.realServer = new
RealWebServer(); }

    public void blockWebsite(String url) {
        this.blockedSites.add(url);        }

    @Override
    public void makeRequest(String url) {
        if(!blockedSites.contains(url)) {
            this.realServer.makeRequest(url);
        }
        else {
            System.out.println("This website is blocked. Contact
your administrator");            }                }        }
```

With this, all the blocked websites will remain unavailable within the premises:

```
//code in main method
WebServer server = new ProxyWebServer();
server.blockWebsite("www.facebook.com");
...
server.makeRequest("www.facebook.com");
  // Prints 'This website is blocked. Contact
your administrator'
```

VIRTUAL PROXY EXAMPLE

```
interface Image {
public void displayImage();
}
class RealImage implements Image {
private String filename;
public RealImage(String filename) {
this.filename = filename;
System.out.println("Loading "+filename);
}
public void displayImage() { System.out.println("Displaying
"+filename); }
}
```

```
class ProxyImage implements Image {
private String filename;
private RealImage image;
public ProxyImage(String filename) { this.filename =
filename; }
public void displayImage() {
if (image == null) {
image = new RealImage(filename); // load only on demand
}
image.displayImage();
}
```

```
public class VProxyExample {



    public static void main(String[] args) {
        ArrayList<Image> images = new ArrayList<Image>();
images.add( new ProxyImage("HiRes_10MB_Photo1") );
images.add( new ProxyImage("HiRes_10MB_Photo2") );
images.add( new ProxyImage("HiRes_10MB_Photo3") );
images.get(0).displayImage(); // loading necessary
images.get(1).displayImage(); // loading necessary
images.get(0).displayImage(); // no loading necessary; already done
// the third image will never be loaded - time saved!
    }

}
```

PROS & CONS

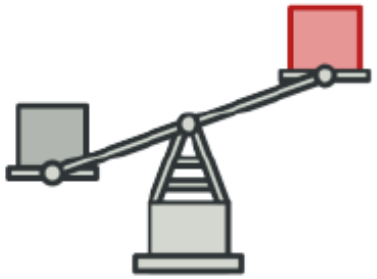
- Pros
  - The Proxy pattern introduces a level of indirection when accessing an object. This additional indirection has many uses.
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Copying a large and complicated object can be an expensive operation. If the copy is never modified, then there's no need to incur this cost. By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it's modified.
- Cons
  - The code may become more complicated since you need to introduce a lot of new classes.
  - The response from the service might get delayed.

COMPARISON BETWEEN PATTERNS

- Bridge and decorator
  - The Decorator should match the interface of the object you're decorating. i.e. it has the same methods.
  - In the Bridge it separates the Abstraction from implementation - both can vary without impact in client
- proxy and façade
- Proxy object provides access control to the single target object while Facade object provides simplified higher level interface to a subsystem of objects/components.
- Proxy object has the same interface as that of the target object and holds references to target objects

- Adapter makes things work after they're designed; Bridge makes them work before they are.

CONCLUSION

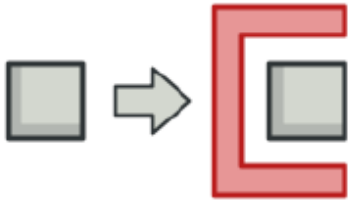


Flyweight

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.

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Proxy

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.