

## Flyweight

In particular, it is to ensure that the state of some of the objects is shared and the other part is customizable. Thus, even if multiple instances are created for objects with customizable states, shared states can be maintained as a single copy in memory. In this way, memory usage is optimized.

### Problem

Let's say you're going to make a survival game and the main character will fight hundreds of zombies coming around it. The first thing to consider is to create a zombie class and create objects from it.

Zombie
health : int
power : int
sprite : image

The problem here, which is not visible at first glance but is a big one, is the image size of the sprite field of this class. Let's say an image is 20Kb, if we produce thousands of these zombies, there will be space to take up in RAM ( $20 * \text{number of zombies}$ ) and this can reach very large megabytes and cause problems.

So what could be the solution to this problem?

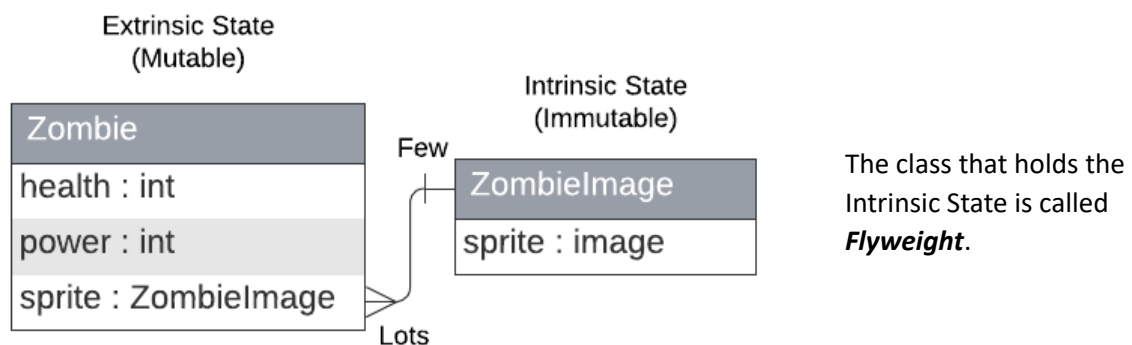
### Solution

Instead of keeping 1 image for each zombie, can we keep only 1 image for all zombies? If we do this, we can eliminate the problem because the area occupied by the image in the game will be reduced from ( $20 * \text{number of zombies}$ ) to only 20.

We can achieve this if we divide our Class into two: values that are common to all zombies and values that are not common, and if all objects read the common values from the same place.

Values that are common and do not change are called **intrinsic state**.

Values that are not common and differ in each object (for example, life and power) are called **extrinsic state**.



**Flyweight** specifically aims to reduce memory usage. This pattern is useful in object-oriented programming when creating a large number of similar objects. The basic idea of the flyweight pattern is to reduce memory usage by using shared objects instead of repetitive objects.

The benefit of applying the pattern largely depends on how and where it is used. It is most useful in these situations:

- If an application needs to create a large number of similar objects.
- If it consumes all available RAM on the device.
- If objects contain repeated states that can be shared among multiple objects.

When you think about it, the ***Flyweight*** pattern can be compared to the ***Singleton*** pattern. But there are two main differences between these patterns:

- A Singleton must have only 1 instance, whereas the Flyweight class can have multiple instances with different internal states.
- Singleton object can be mutable. Flyweight objects are immutable.

**The Flyweight pattern can increase the complexity of the code, so applying the pattern may not be suitable for every situation, it is especially effective in cases where RAM usage is critical and should be used when the program being written must support a large number of objects that barely fit in RAM.**

## Adapter

The Adapter design pattern is used to provide a connection between incompatible interfaces or classes. This pattern solves communication problems by adding a middleware that allows one class or interface to be compatible with another expected class or interface.

### Problem

Imagine being one of two kingdoms that have been fighting for generations and never defeated each other. As a result of your many years of experiments, plans and wars, you have realized that you cannot conquer the other kingdom without going inside the castle. You cannot enter by swimming because the water around the kingdom is full of crocodiles, all your attempts to climb the walls of the kingdom have failed so far, you can actually enter by flying but the dragons have not been tamed yet. So, how can you get into the opposing kingdom and conquer it?



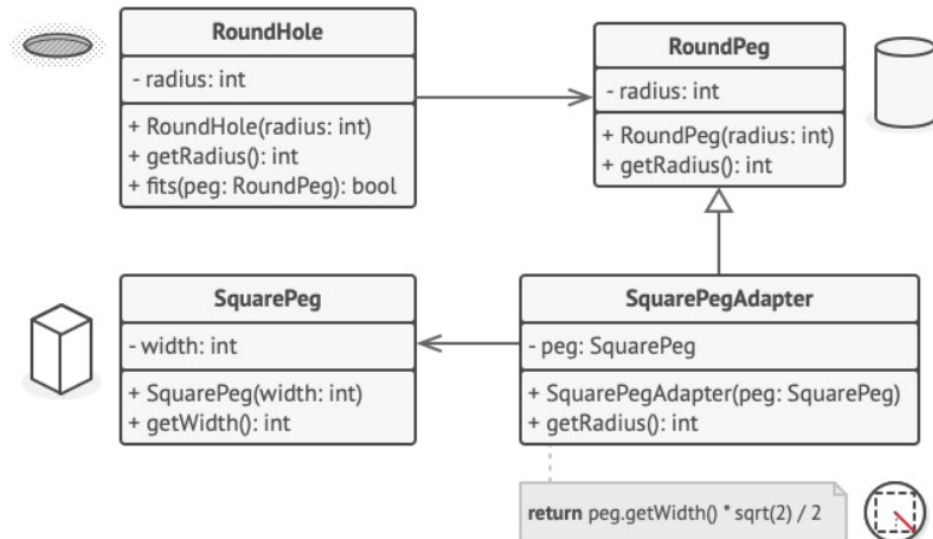
### Solution

It looks like the only way in is through the door. So how do we get in with our arms around? By looking like something that would be admitted inside, of course.

This is exactly the opportunity **Adapter** provides us. We make **two incompatible interfaces** compatible by making them similar to each other. In other words, we create a **Wrapper Class** for the incompatible class, similar to the compatible class. Thus, we can use the incompatible class as a compatible class, in addition, we can use the advantages of polymorphism and pass the incompatible class as a parameter.



While creating the **adapter**, the adapter inherits from the compatible class, then the object of your incompatible class is sent to the constructor of this adapter. Then, the object you send is assigned to the field in the adapter. Thus, we inherit from the compatible class and create a class that is similar to the compatible class but contains an object of the incompatible class. Then, by overriding the functions of the compatible class, you create a new structure from your incompatible class that can act like a compatible class.



In this photo, we are performing a similar process as in the trojan horse analogy. Since SquarePeg cannot enter through the RoundHole, we can make it look like a cylinder (create a cylinder adapter) and insert it through the RoundHole.

java code of SquarePegAdapter:

```

java
Copy code

class RoundPeg {
    private double radius;

    // Constructor, getters, other methods...
}

class SquareHole {
    private double width;

    // Constructor, getters, other methods...

    public boolean fits(RoundPeg peg) {
        // Check if the peg fits into the hole
    }
}

class SquarePegAdapter extends RoundPeg {
    private SquarePeg peg;

    SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        // Calculate an appropriate radius for the square peg
        // so that it can fit into the round hole
    }
}
  
```

It may make sense to apply the Adapter pattern in the following situations:

- When you want to use some classes but your code is not compatible with the interfaces of these classes, create an adapter class and make your code compatible.
  - Adapter class becomes an intermediate layer between your code and non-compliant code.
- You can use it when you want to reuse existing subclasses that lack common functionality that can be added to a superclass. The first thing that comes to mind is adding the desired function to the subclasses one by one, but we can achieve a better solution with an adapter.
  - Let's have classes that can play different types of media files in a sound system application. Let's say there are classes MP3Player, WAVPlayer, and FLACPlayer, and each can only play a certain type of file. However, none of these classes have a functionality to read the metadata of the music file (artist, album, duration, etc.), and this functionality cannot be added to the common super class of each class.

In this case, we can create an adapter class called MusicPlayerAdapter and define the functionality to read metadata in this adapter:

```
java Copy code  
  
interface MediaPlayer {  
    void play();  
    // Diğer ortak metotlar...  
}  
  
class MP3Player implements MediaPlayer {  
    // MP3 dosyalarını oynatma işlevselliği...  
}  
  
class MusicPlayerAdapter implements MediaPlayer {  
    private MediaPlayer player;  
  
    public MusicPlayerAdapter(MusicPlayer player) {  
        this.player = player;  
    }  
  
    @Override  
    public void play() {  
        player.play();  
    }  
  
    public String getMetadata() {  
        // Mevcut müzik dosyasının metadatasını döndür  
    }  
}
```

## Composite

This pattern allows treating individual objects, such as an employee, and combinations of objects, such as a team of employees, in the same way. Thus, clients can interact with these objects regardless of the distinction between individual objects and combinations of objects.

## Problem

You are working in the software department of a holding company consisting of many companies, and you have been asked to write a program that allows each employee to send messages to the employees under him. In your system, an employee can directly access the employees under him, but the employees under him do not have access to the employees under him. In other words, while a regional manager can access branch managers, he cannot directly access the employees in the branch, and in the same way, a branch manager can access the employees in the branch.

## Solution

When the problem is considered a little, it becomes obvious that it is actually a kind of "n-ary tree" problem. If we navigate this tree correctly, we can convey the message we want. What we need here is to navigate through the hierarchy in a recursive way. We can also do this with a composite pattern.

The logic of the **composite** pattern is to treat two or more structures as if they were the same thing. It would be more meaningful if we explained this pattern directly through a tree. In the composite pattern, objects that do not have anything under them are called **leaf** (same as **leaf** in trees), while objects that have other things under them are called **composite** (such as the branch manager and the employees working under him).

An interface is created. Leaf and composite classes implement this interface, and since these classes will contain functions with the same signatures, the two classes can be treated as if they were the same. In this way, operations can be carried out by easily navigating the hierarchy.

```
java Copy code

public interface Employee {
    void sendMessage(String message);
}
```

*Composite Interface*

```
java Copy code

public class StaffMember implements Employee {
    private String name;

    public StaffMember(String name) {
        this.name = name;
    }

    @Override
    public void sendMessage(String message) {
        System.out.println(name + " received message: " + message);
    }
}
```

*Leaf Class*

```
java Copy code

import java.util.ArrayList;
import java.util.List;

public class Manager implements Employee {
    private String name;
    private List<Employee> subordinates = new ArrayList<>();

    public Manager(String name) {
        this.name = name;
    }

    public void addSubordinate(Employee employee) {
        subordinates.add(employee);
    }

    @Override
    public void sendMessage(String message) {
        for (Employee subordinate : subordinates) {
            subordinate.sendMessage(message);
        }
    }
}
```

*Composite Class*

### In Which Situations Should *Composite* Be Used?

- Composite pattern is used when you need to implement a tree-like object structure.
  - This pattern provides two basic element types with a common interface: leaf and composite container. A container can contain both leaves and other containers. This allows you to create a nested, repeating object structure that resembles a tree.
- This pattern is used when you want the client code to treat both simple and complex elements the same.
  - All elements defined by the composite pattern have a common interface. Thanks to this interface, the client does not have to worry about the concrete classes of the objects it is working with.

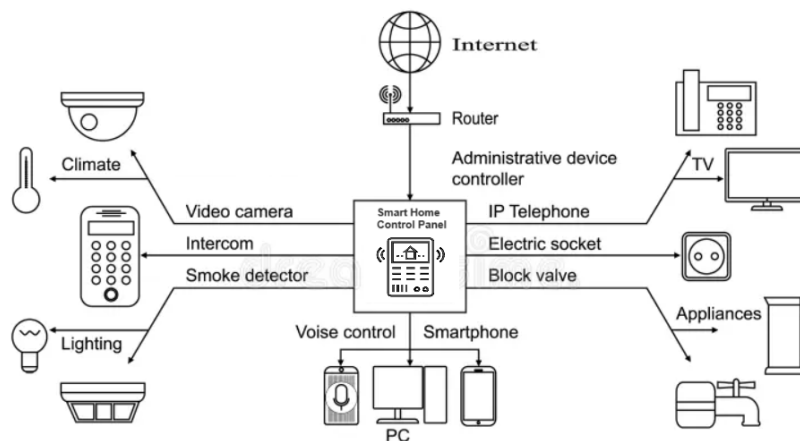


## Facade

It is a design pattern that gathers complex subsystems (library, framework, etc.) under a single interface to facilitate use.

## Problem

Let's say you are slowly turning your home into a smart home. First, you made the lights smart and you can control them from your phone, then you made your cameras smart and you can view your door from your phone when your doorbell is pressed, and you have the convenience of controlling other cameras from your phone whenever you want, then you made the heating system of the house smart. You are gradually making your entire home smart, but over time, you started to feel uncomfortable with the space these smart home applications took up on your phone, and it became increasingly difficult to control them because there were many different applications on your phone. It is obvious that what you need here is a control panel. If you could control all your systems from one place, it would not bother you so much that your smart home system is getting bigger and bigger, because they would all be controlled from the same place anyway.



## Solution

If you have sufficient technical knowledge, you can write this control panel yourself, if not, you can use some of the ready-made applications and gather your entire system under one roof.

What the **Facade** pattern does is the same process of gathering many complex systems in one simple place in this analogy. It allows you to create a more useful and simple interface by isolating many of the work that the system does in the background from the user.

java

Copy code

```
// Facade
public class SmartHomeControlPanel {
    private HeatingSystem heatingSystem = new HeatingSystem();
    private LightingSystem lightingSystem = new LightingSystem();
    private SecuritySystem securitySystem = new SecuritySystem();

    public void activateMorningRoutine() {
        heatingSystem.increaseTemperature();
        lightingSystem.turnOnLights();
        securitySystem.deactivateAlarm();
    }
}

// Subsystems
class HeatingSystem {
    public void increaseTemperature() {
        System.out.println("Heating system is increasing temperature.")
    }
}

class LightingSystem {
    public void turnOnLights() {
        System.out.println("Lighting system is turning on lights.");
    }
}

class SecuritySystem {
    public void deactivateAlarm() {
        System.out.println("Security system is deactivating the alarm.")
    }
}
```

### A simpler example..

A simpler example is a computer. In fact, there is no single part called a computer; we call the whole formed by the case into which we put the parts a computer. **The computer is a kind of facade** because it isolates us from the complex components inside it and communicates with all the components after pressing a single button, and then raises the system and displays an image on the screen.



**If it is not used carefully and a face covering the entire system is made, this face can turn into a god object. To prevent this, many facades can be used and even facades can be made in layers.**

## Proxy

Proxy pattern is used to control access to an object. It allows clients to access the object (indirectly) through a proxy, instead of directly accessing the relevant object. The proxy object communicates with the principal object only when necessary and the relevant operations are performed. The aim is to reduce costs and ensure security.

## Problem

Let's say you have a company of 30 people. Apart from your own work, you also listen to your employees' requests/complaints/suggestions. But after a while, you became unable to spare time for your own work because your employees came to you with all kinds of questions. What solution method do you follow?



## Solution

You can solve this problem by assigning a person to handle the questions. This person you appoint can evaluate and filter the incoming questions and convey what needs to be conveyed to you, so that you will not be concerned with unnecessary questions and you can spare time for your other work.

This is what is done with the **proxy** pattern. You access an object through a proxy, and this proxy can only communicate with the object when necessary.

## A real example

Let's imagine that you wrote an application for children and that you will access the internet through this application. The first problem here is to create a structure that will control the sites that children cannot access. We can achieve this with the Proxy pattern.


```
java Copy code

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

java Copy code

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

java

 Copy code


```
import java.util.ArrayList;
import java.util.List;

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

    static {
        bannedSites = new ArrayList<>();
        bannedSites.add("banned-site.com");
        bannedSites.add("example-bad-site.com");
        bannedSites.add("restricted-site.com");
    }

    @Override
    public void connectTo(String serverHost) throws Exception {
        if(bannedSites.contains(serverHost.toLowerCase())) {
            throw new Exception("Access Denied to " + serverHost);
        }
        internet.connectTo(serverHost);
    }
}
```

java

 Copy code

```
public class ProxyDemo {
    public static void main(String[] args) {
        Internet internet = new ProxyInternet();
        try {
            internet.connectTo("example.com");
            internet.connectTo("banned-site.com");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

### In what situations should Proxy be used?

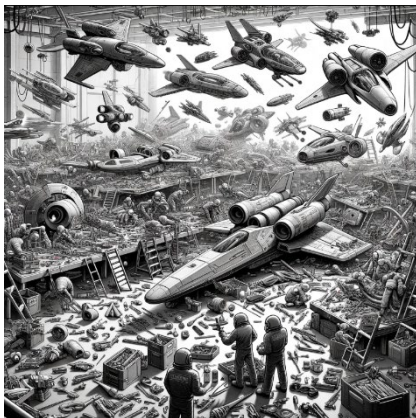
- For whatever reason, whenever it needs to control access to an object, subsystem or system, or hide the subsystem or system, it can use a **proxy**.
- JPA can be used in ORM frameworks such as Hibernate or EntityFramework, especially to implement lazy loading structure.
  - In all collection structures where postloading is applied, proxy collections and proxy entity objects are created for the entities whose fields will be loaded later.
- When there is a heavy service object that is needed only occasionally but is always running, wasting system resources, a **proxy** can be used to perform lazy initialization.
- A **proxy** can be used as a control structure when you want only certain clients to be able to use the service object. For example, blocking websites that can be accessed from the application can be given.

## Decorator

Decorator pattern allows you to dynamically add additional behaviors to an object or change the object's properties. To add different features to an object, it is the process of adding a new feature to that object at runtime, instead of inheriting its class and creating a new class with the new feature.

## Problem

Humanity is no longer in a situation where it cannot fit into the world, fortunately, technology has developed a lot and interplanetary travel is now possible. For this reason, you are a member of a research institution that travels to planets that you think have life. Your mission is to produce, maintain and develop traveling spaceships. You design ships according to requests for each trip. A ship that can travel a long distance without needing fuel, a ship with very strong shields, a ship with weapons, a compact ship, a fast ship, ....



After a while, building a new spaceship according to each request becomes very complicated. You don't have enough space to put the spaceships, it becomes very difficult to keep track of the spaceships because there are too many of them, and the costs increase because you produce a ship that we may only use on one planet and put it on the shelf.

## Solution

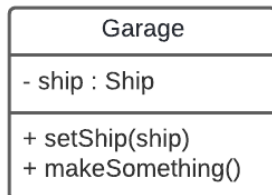
If spaceships were modular and desired features could be added or removed, there would be no need to constantly produce new ships. So, if you had a basic spaceship and this spaceship was modified according to the planet to be visited, you could obtain multiple ships from a single ship.

## Implementation

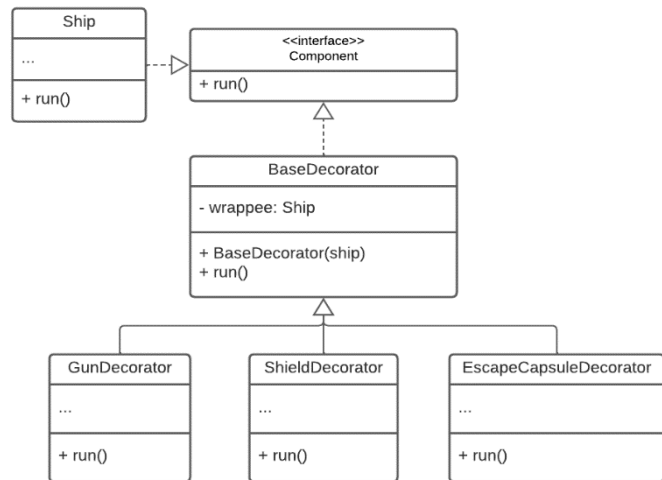
We achieve the "modularity" function here by wrapping the object with the feature we want. Our class structure generally looks like the diagram below. First, we create a Ship object, then we put it into the decorator we want, then we repeat the same process if we want different features to be added.

```
stack = new Ship();
if (gunEnabled)
    stack = new GunDecorator(stack)
if (shiledEnabled)
    stack = new ShieldDecorator(stack)

garage.setShip(stack)
```



*makeSomething() gemiyi çalıştırma gibi bir işlemi gerçekleştiriyor olabilir.*



## In what situations should decorator be used?

- It can be used when you need to be able to assign extra behaviors to objects in the runtime without breaking the code that uses these objects.
- Can be used when extending the behavior of an object using inheritance is difficult or not possible.
  - For a *final* class, the only way to reuse existing behavior is to wrap the class with your own wrapper using the Decorator pattern.