Structural Design Patterns

Objectives

After completing this lesson, you should be able to do the following:

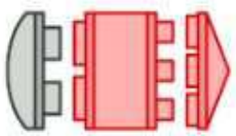
- Analysis of Structural DP
- Exploration of Adapter Pattern, Composite Pattern
- Proxy Pattern , Flyweight Pattern
- Facade Pattern, Bridge Pattern
- Decorator Pattern

Course Roadmap

Java Design Patterns

- Lesson 1: Introduction to Design Patterns
- Lesson02: Creational Design Patterns
- **Lesson03: Structural Design Patterns** — **You are here!**
- Lesson04: Behavioral Design Patterns
- Lesson 5: Most Useful Design Patterns

Structural Design Patterns



Adapter

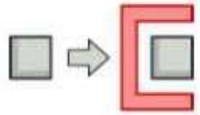
Allows objects with incompatible interfaces to collaborate.

Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

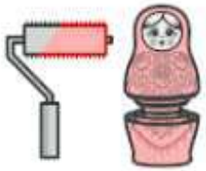
Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



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.

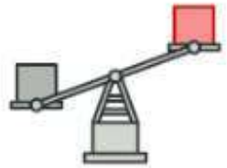


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Facade

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

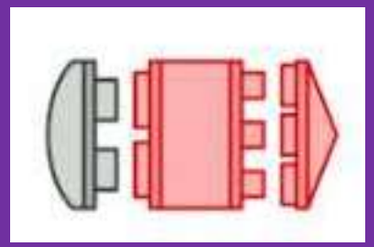


Flyweight

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.

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

- *For example, using Inheritance and Composition to create a large Object from small Objects*



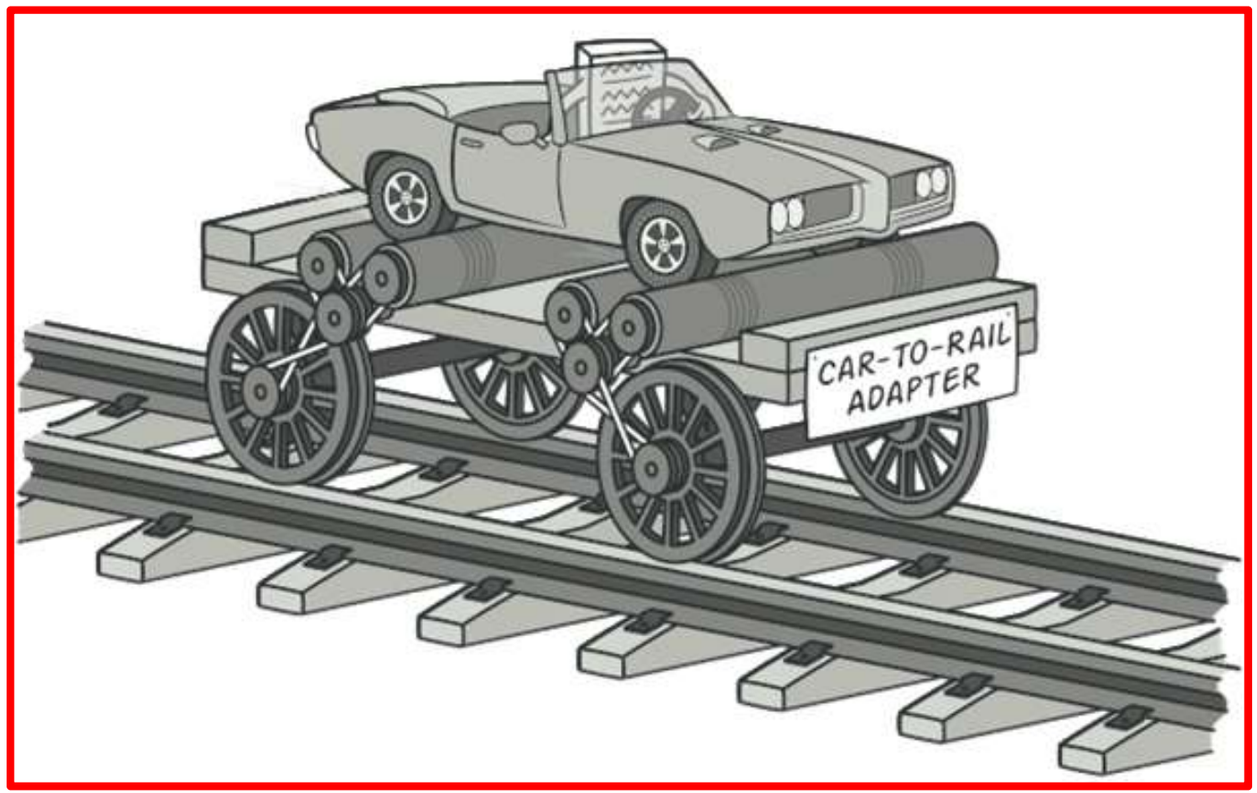
Adapter Pattern

Introduction

- ***Adapter*** *is a structural design pattern, which allows incompatible objects to collaborate.*
- *It is used so that two unrelated interfaces can work together.*
- *The object that joins these unrelated interfaces is called an adapter.*

Adapter Also known as: [Wrapper]

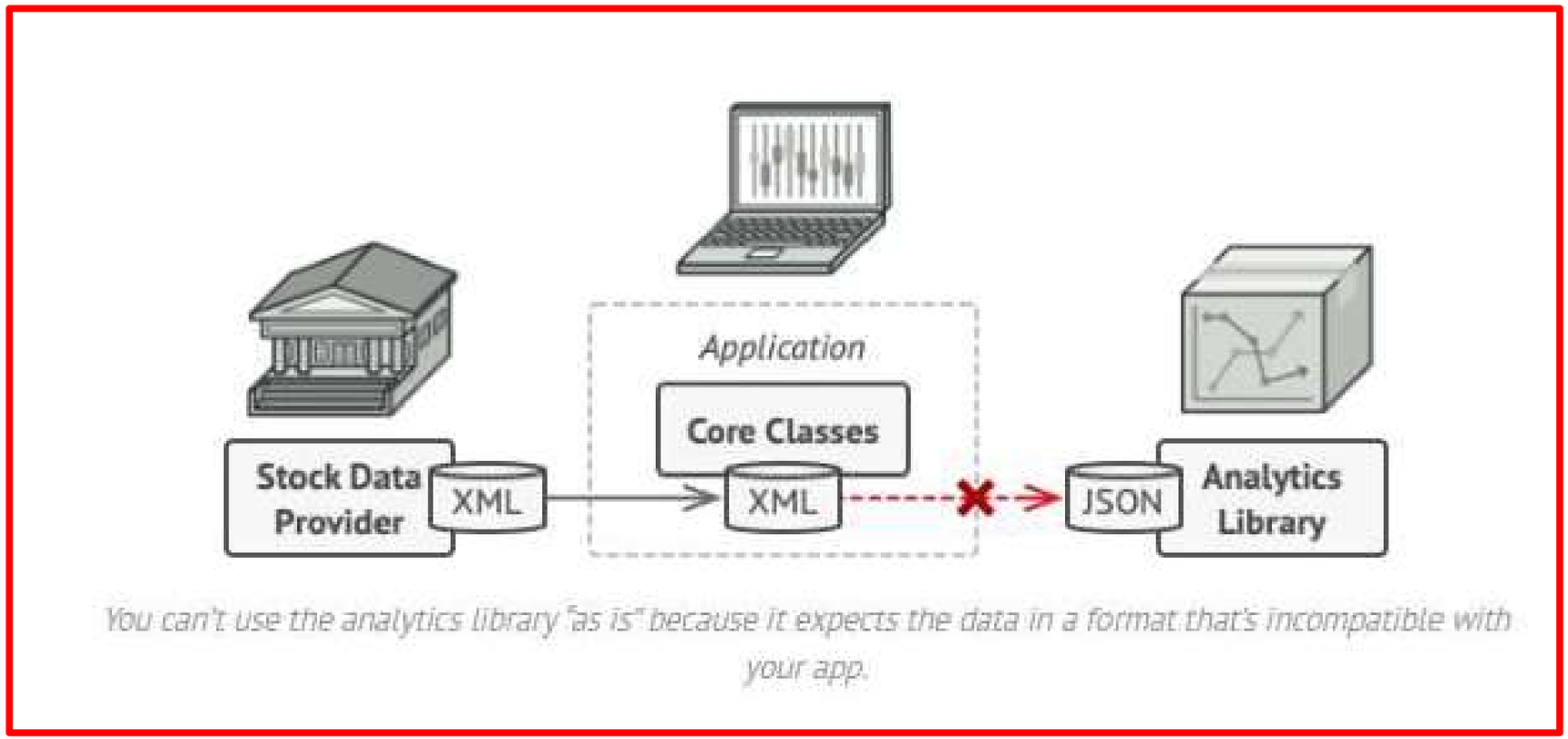
- **Intent**
  - **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate



☹ Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.



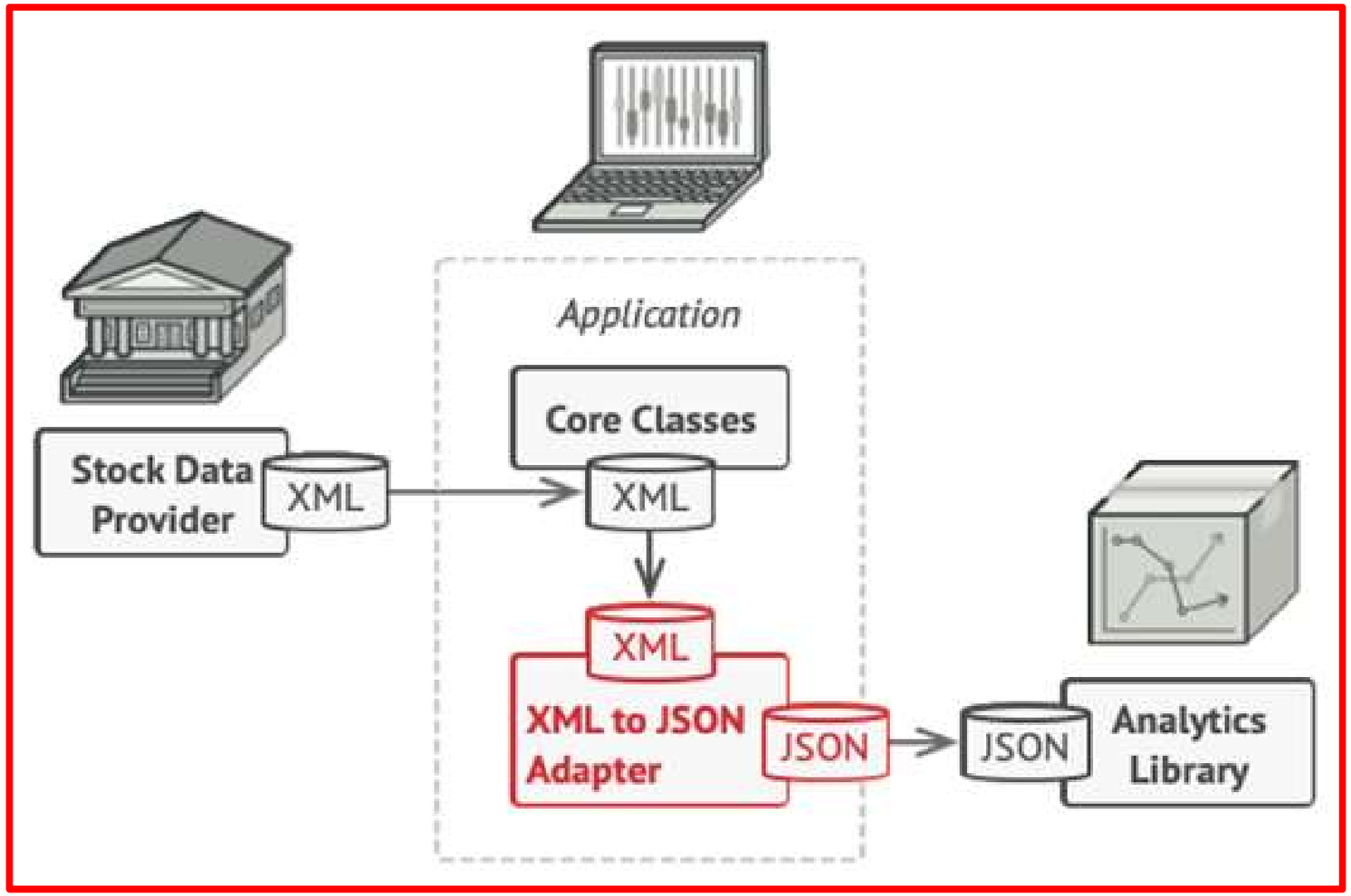
You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

- You can create an ***Adapter***. This is a special object that converts the interface of one object so that another object can understand it.

- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects


Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Complexity: ★☆☆

Popularity: ★★★

Usage examples: The Adapter pattern is pretty common in Java code. It's very often used in systems based on some legacy code. In such cases, Adapters make legacy code work with modern classes.

There are some standard Adapters in Java core libraries:

- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream)` (returns a `Reader` object)
- `java.io.OutputStreamWriter(OutputStream)` (returns a `Writer` object)
- `javax.xml.bind.annotation.adapters.XmlAdapter#marshal()` and `#unmarshal()`

Example

```
1.  public class Volt {
2.
3.      private int volts;
4.
5.      public Volt(int givenVolts) {
6.          this.volts = givenVolts;
7.      }

8.      public int getVolts() {
9.          return volts;
10.     }

11.     public void setVolts(int volts) {
12.         this.volts = volts;
13.     }
14.}
```

Example

```
1.  public class Socket {
2.
3.      public Volt getVolt() {
4.          return new Volt(120);
5.      }
6.
7.  }
```

Example

```
1.  public interface SocketAdapter {
2.
3.      public Volt get120Volt();
4.
5.      public Volt get12Volt();
6.
7.      public Volt get3Volt();
8.  }
```

Example

```
1. // Class Adapter
2. public class SocketClassAdapterImpl extends Socket implements
   SocketAdapter{


3.     @Override
4.     public Volt get120Volt() {
5.         return getVolt();
6.     }

7.     @Override
8.     public Volt get12Volt() {
9.         Volt v= getVolt();
10.        return convertVolt(v, 10);
11.    }
```

Example

```
12. @Override
13.     public Volt get3Volt() {
14.         Volt v= getVolt();
15.         return convertVolt(v, 40);
16.     }
17.
18.     private Volt convertVolt(Volt v, int i) {
19.         return new Volt(v.getVolts()/i);
20.     }
21.}
```

Example

```
1.  // Object Adapter [ Based on Composition ]
2.  public class SocketObjectAdapterImpl implements SocketAdapter{
3.
4.      // Composition
5.      private Socket socketRef = new Socket();

6.      @Override
7.      public Volt get120Volt() {
8.          return socketRef.getVolt();
9.      }

10.     @Override
11.     public Volt get12Volt() {
12.         Volt v = socketRef.getVolt();
13.         return convertVolt(v, 10);
14.     }
```

Example

```
15.  @Override
16.    public Volt get3Volt() {
17.        Volt v = socketRef.getVolt();
18.        return convertVolt(v, 40);
19.    }
20.
21.    private Volt convertVolt(Volt v, int i) {
22.        return new Volt(v.getVolts()/i);
23.    }
24.
25.}
```

Example

```
1.  public class AdapterPatternTester {
2.
3.      public static void main(String[] args) {
4.          testClassAdapter();
5.          testObjectAdapter();
6.      }
7.
8.      private static Volt getVolt(SocketAdapter sockAdapterRef, int i) {
9.          switch(i) {
10.             case 3   : return sockAdapterRef.get3Volt();
11.             case 12  : return sockAdapterRef.get12Volt();
12.             case 120 : return sockAdapterRef.get120Volt();
13.             default  : return sockAdapterRef.get120Volt();
14.         }
15.     }
```

Example

```
1.
2.   private static void testObjectAdapter() {
3.         SocketAdapter sockAdapterRef = new SocketObjectAdapterImpl();
4.         Volt v3 = getVolt(sockAdapterRef, 3);
5.         Volt v12 = getVolt(sockAdapterRef, 12);
6.         Volt v120 = getVolt(sockAdapterRef, 120);
7.
8.         System.out.println("V3 Volts Using Object Adapter = " + v3.getVolts());
9.         System.out.println("V12 Volts Using Object Adapter = " + v12.getVolts());
10.        System.out.println("V120 Volts Using Object Adapter = " + v120.getVolts());
11.   }
12.
13.
```

Example

```
1.  private static void testClassAdapter() {
2.          SocketAdapter sockAdapterRef = new SocketClassAdapterImpl();
3.          Volt v3 = getVolt(sockAdapterRef, 3);
4.          Volt v12 = getVolt(sockAdapterRef, 12);
5.          Volt v120 = getVolt(sockAdapterRef, 120);
6.
7.          System.out.println("V3 Volts Using Class Adapter = " + v3.getVolts());
8.          System.out.println("V12 Volts Using Class Adapter = " + v12.getVolts());
9.          System.out.println("V120 Volts Using Class Adapter = " + v120.getVolts());
10.     }
11.
12.}
```

Applicability

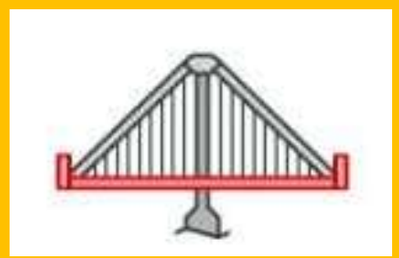
1. Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.

2. Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

Pros and Cons

- ✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

- ✗ *The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.*



Bridge Pattern

Introduction

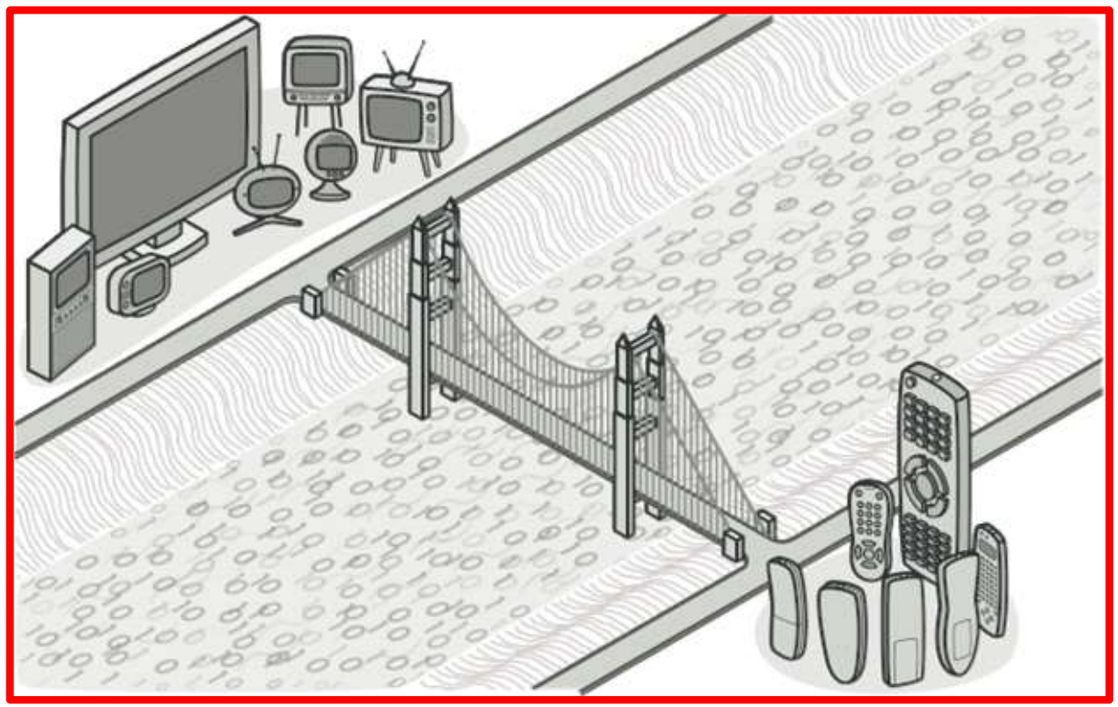
- ***Bridge*** *is a structural design pattern that divides business logic or huge class into separate class hierarchies that can be developed independently.*

- When we have interface hierarchies in both interfaces as well as implementations, then the bridge design pattern is used to decouple the interfaces from the implementation and to hide the implementation details from the client programs.

- The implementation of the bridge design pattern follows the notion of preferring ***Composition Over Inheritance.***

Intent

- **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

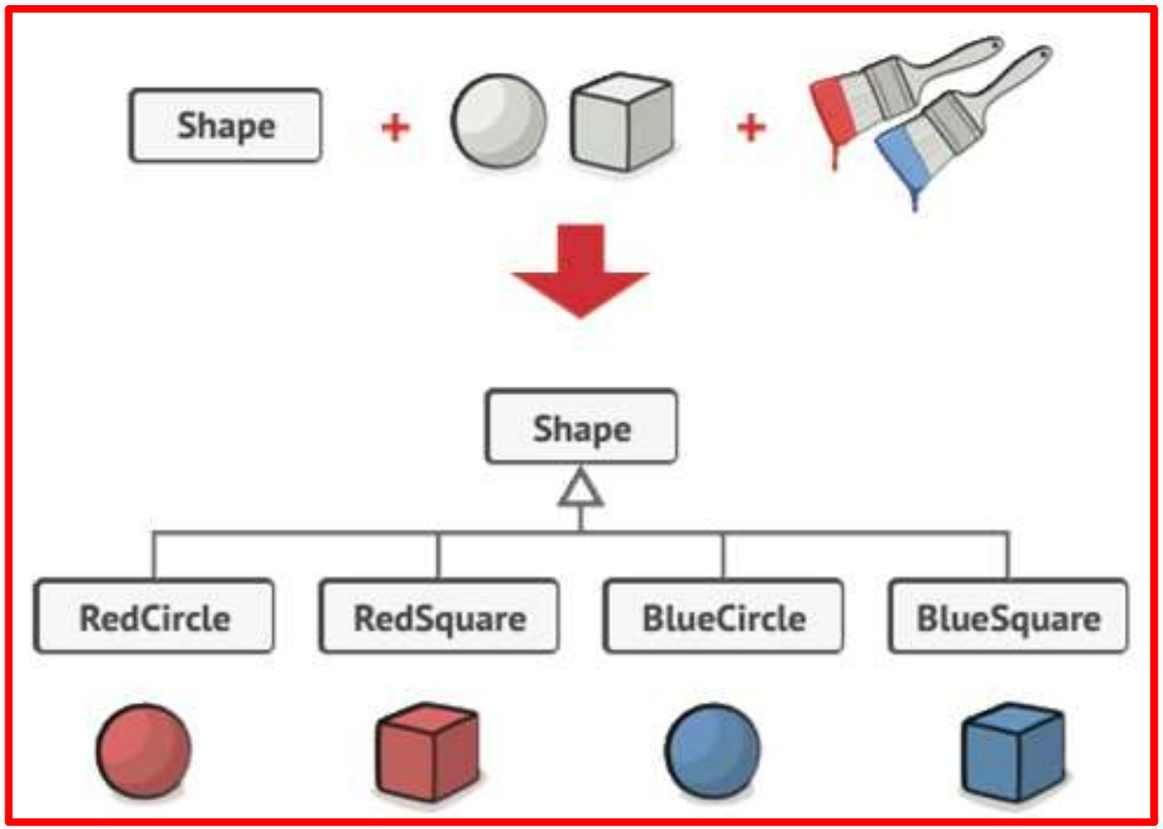


Problem

- *What is **Abstraction**?* What is ***Implementation**?*

- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.

- However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.

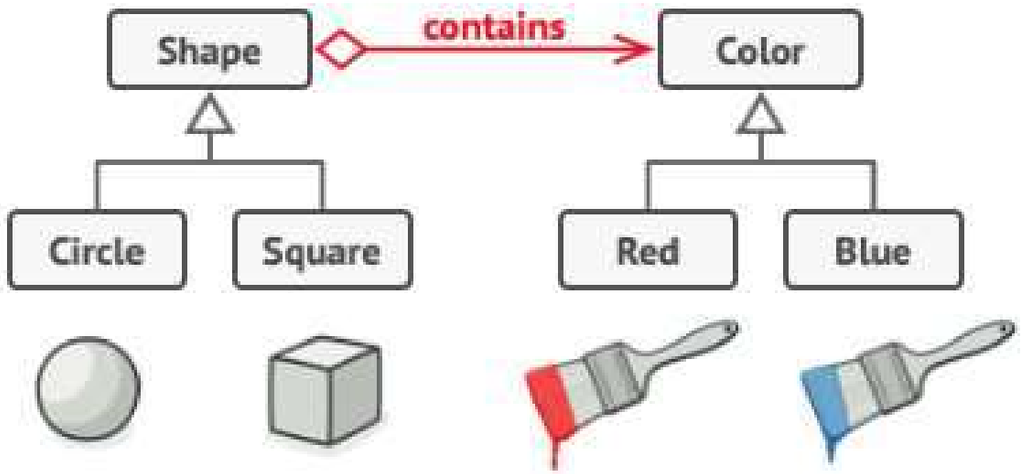


- Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape you'd need to introduce two subclasses, one for each color.

- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.

- 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.



You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue.

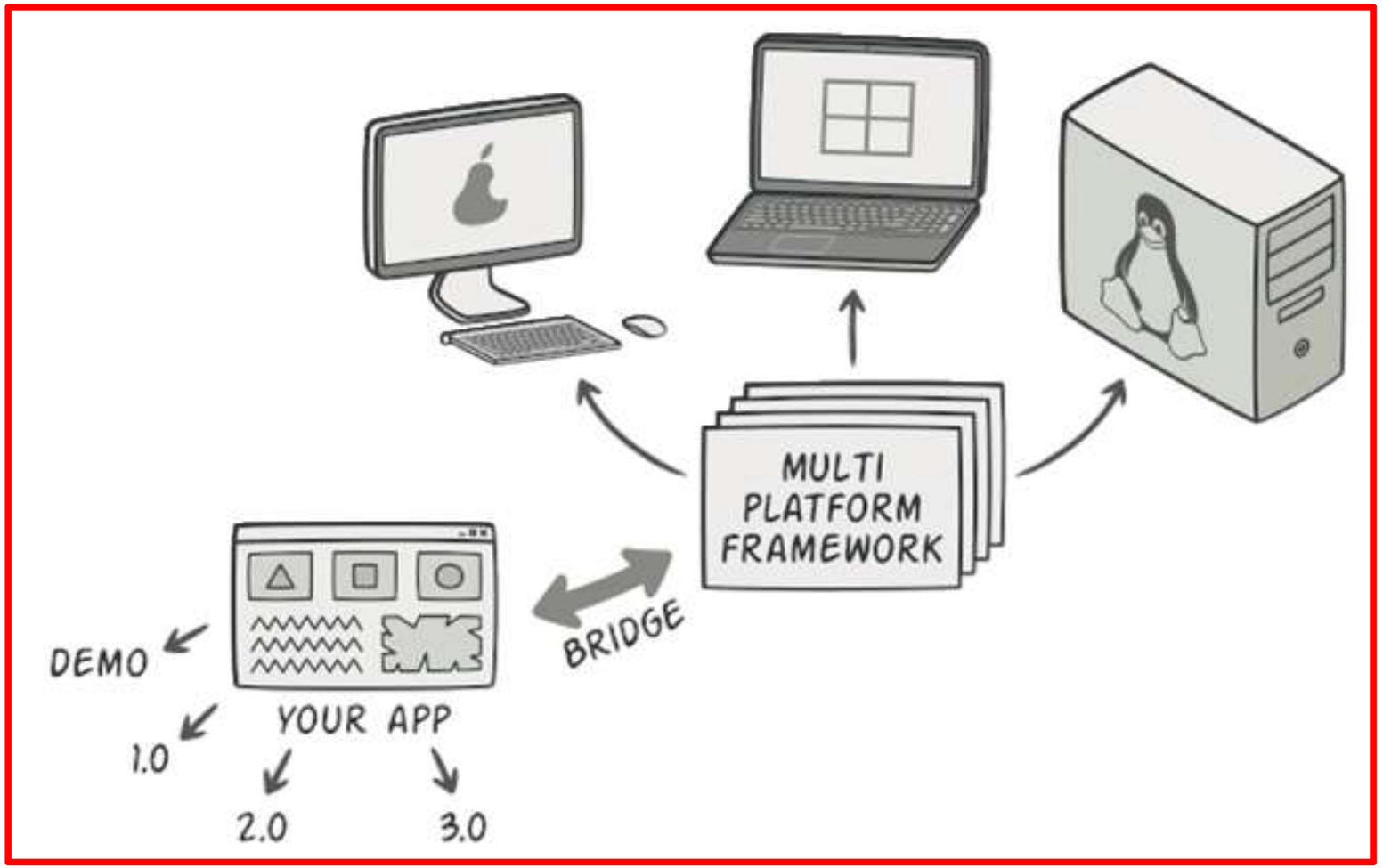
- The Shape class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object.

- That reference will act as a bridge between the Shape and Color classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.

- ***Abstraction*** (also called *interface*) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own.

- It should delegate the work to the ***implementation*** layer (also called *platform*).

Illustration



Complexity: ★★★

Popularity: ★☆☆

Usage examples: The Bridge pattern is especially useful when dealing with cross-platform apps, supporting multiple types of database servers or working with several API providers of a certain kind (for example, cloud platforms, social networks, etc.)

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).

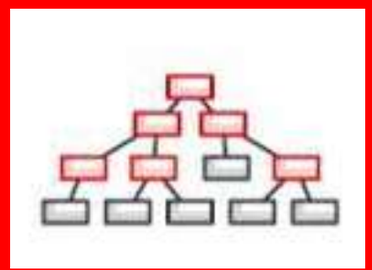
2. Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.

3. Use the Bridge if you need to be able to switch implementations at runtime.

Pros and Cons

- ✓ You can create platform-independent classes and apps.
- ✓ The client code works with high-level abstractions. It isn't exposed to the platform details.
- ✓ *Open/Closed Principle*. You can introduce new abstractions and implementations independently from each other.
- ✓ *Single Responsibility Principle*. You can focus on high-level logic in the abstraction and on platform details in the implementation.

- ✗ You might make the code more complicated by applying the pattern to a highly cohesive class.



Composite Pattern

Introduction

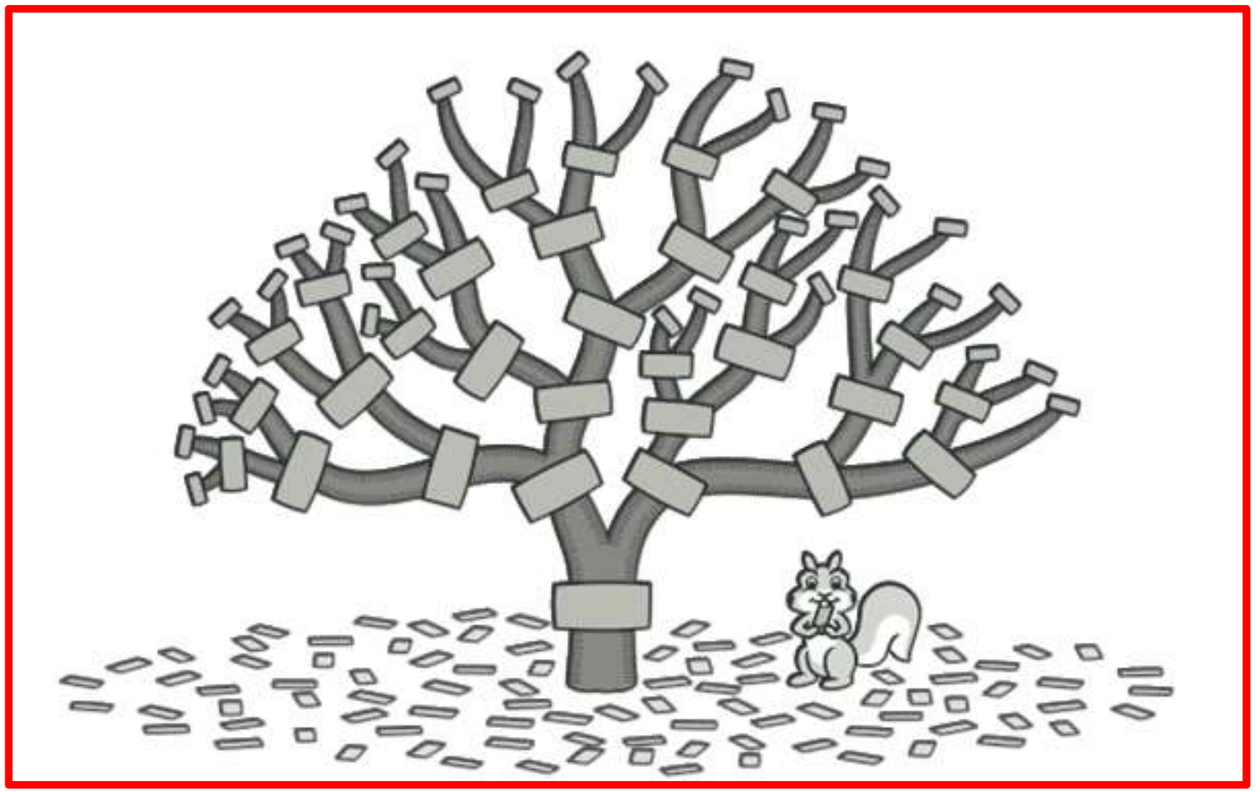
- ***Composite*** *is a structural design pattern that allows composing objects into a tree-like structure and work with the it as if it was a singular object.*

- The composite pattern is used when we have to represent a part-whole hierarchy. When we need to create a structure in a way that the objects in the structure have to be treated the same way.

- Composite became a pretty popular solution for the most problems that require building a tree structure. Composite's great feature is the ability to run methods recursively over the whole tree structure and sum up the results.

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.

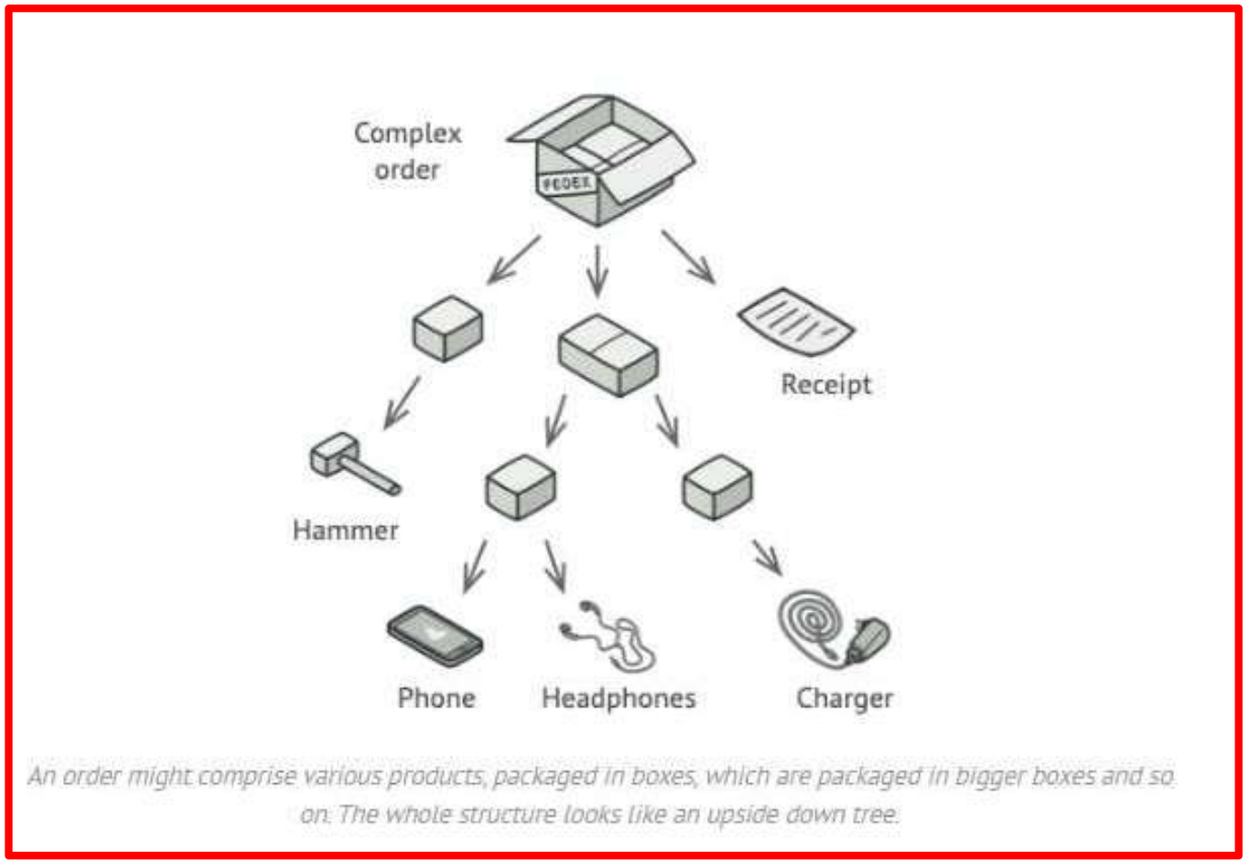


Problem

- Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

- For example, imagine that you have two types of objects: Products and Boxes. 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.

- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?

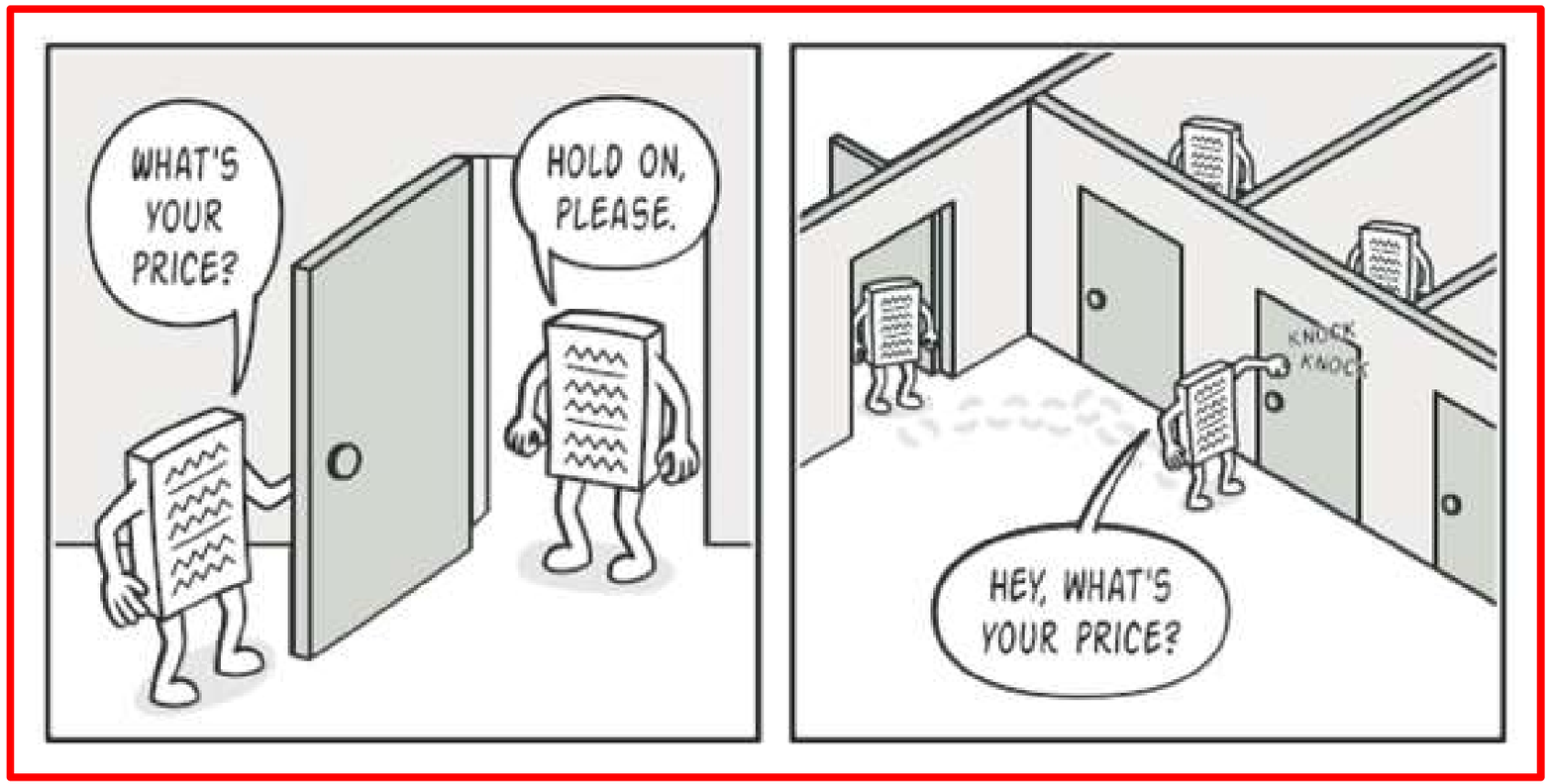


An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.

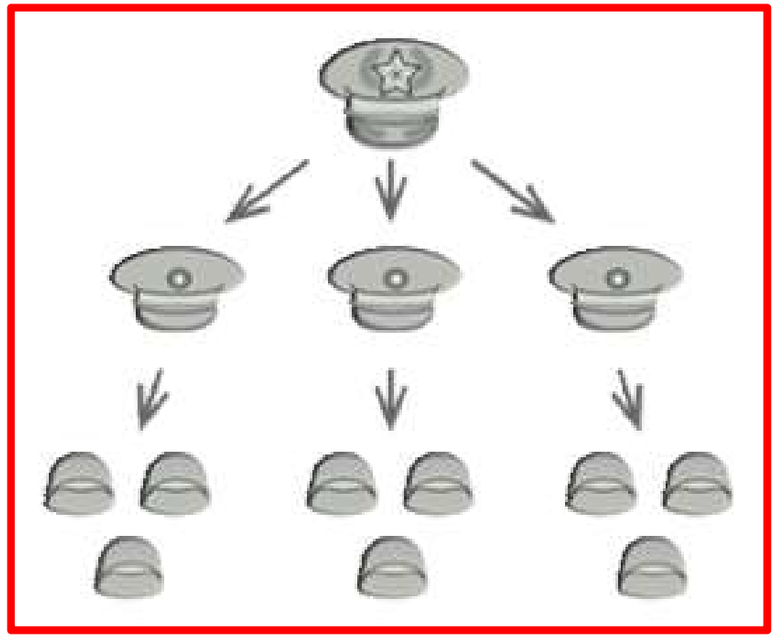
- You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop

- The ***Composite pattern*** suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

- How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.


WHAT'S YOUR PRICE?
HOLD ON, PLEASE.
KNOCK KNOCK
HEY, WHAT'S YOUR PRICE?

Real-World Analogy



- Armies of most countries are structured as hierarchies. An army consists of several divisions; a division is a set of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done.

Complexity: ★★☆

Popularity: ★★☆

Usage examples: The Composite pattern is pretty common in Java code. It's often used to represent hierarchies of user interface components or the code that works with graphs.

Here are some composite examples from standard Java libraries:

- `java.awt.Container#add(Component)` (practically all over Swing components)
- `javax.faces.component.UIComponent#getChildren()` (practically all over JSF UI components)

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Use the Composite pattern when you have to implement a tree-like object structure.
2. Use the pattern when you want the client code to treat both simple and complex elements

Pros and Cons

- ✓ You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- ✓ *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

- ✗ It might be difficult to provide a common interface for classes whose functionality differs too much.

Decorator Pattern

Introduction

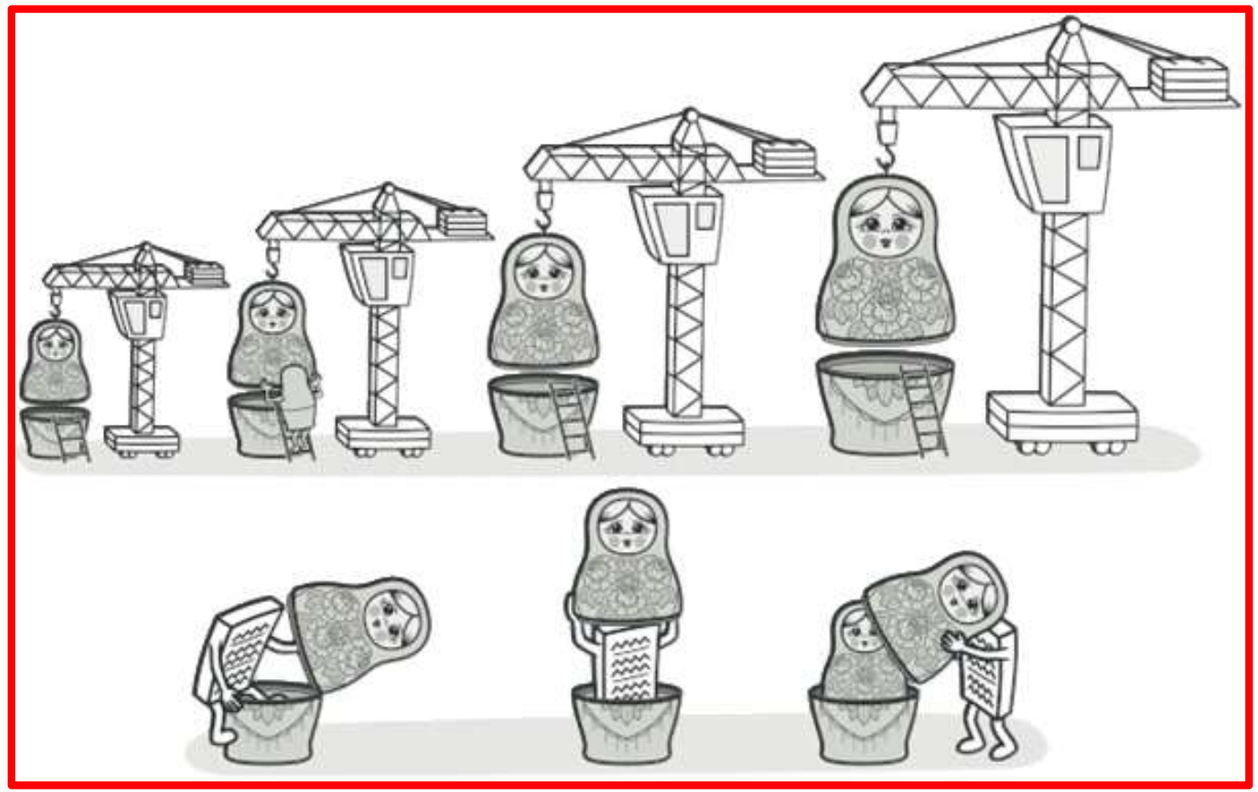
- ***Decorator*** *is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects, called decorators.*

- The decorator design pattern is used to modify the functionality of an object at runtime. At the same time, other instances of the same class will not be affected by this, so the individual object gets the modified behavior.

- We use inheritance or composition to extend the behavior of an object, but this is done at compile-time, and it's applicable to all the instances of the class. We can't add any new functionality to remove any existing behavior at runtime this is when the decorator pattern is useful.

Intent

- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

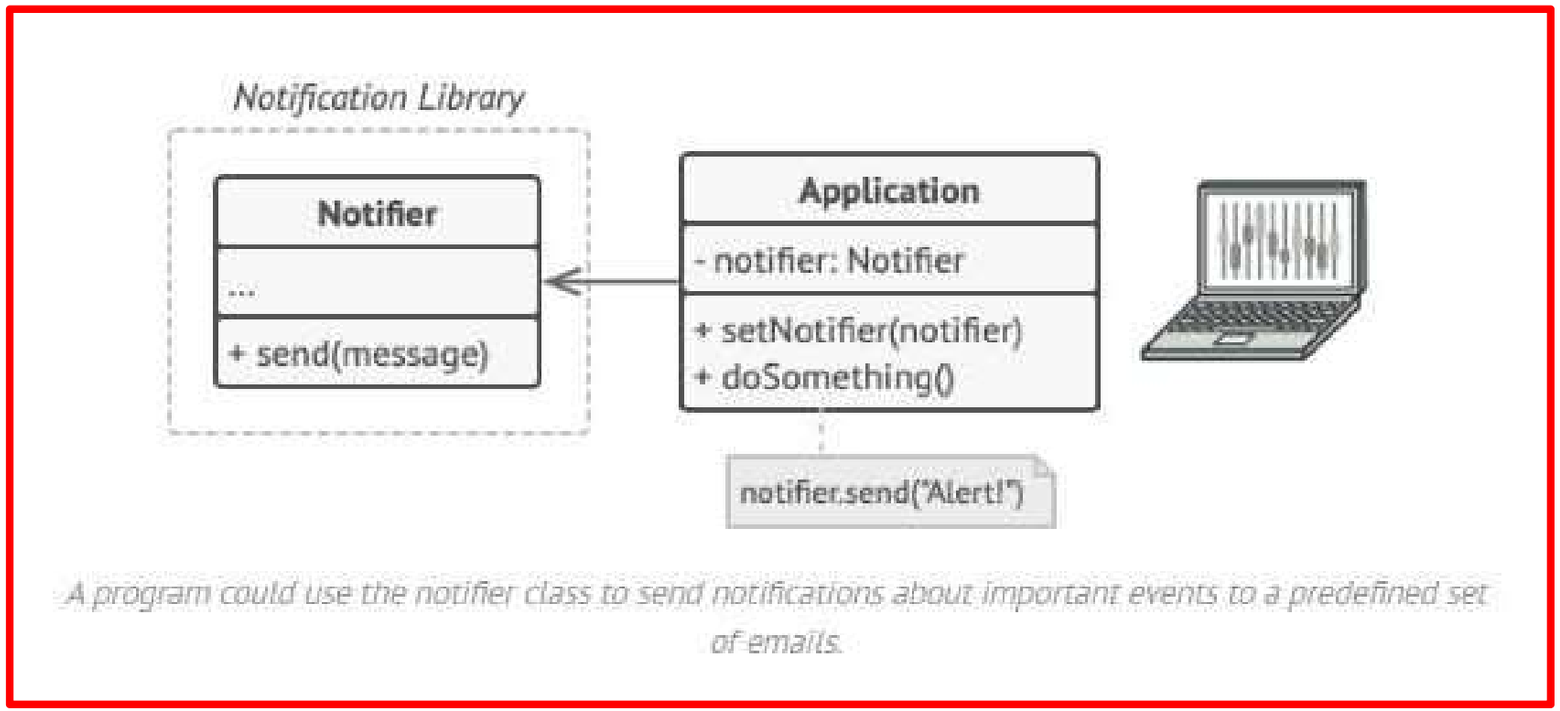


Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.

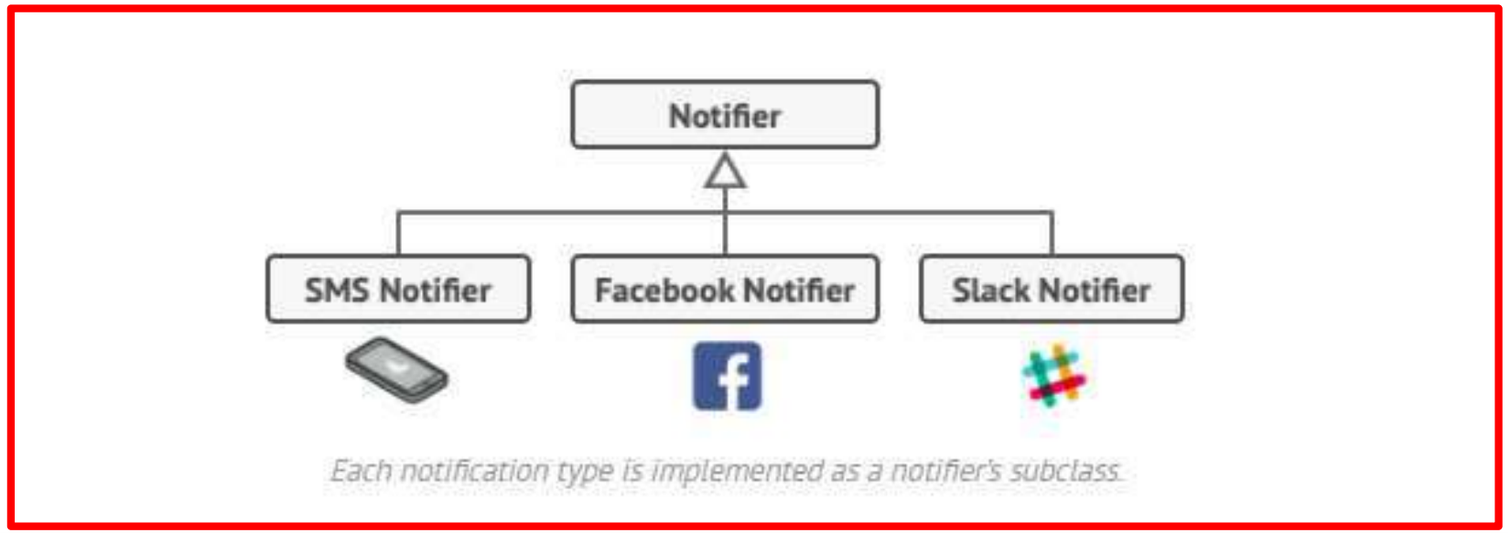
- The initial version of the library was based on the Notifier class that had only a few fields, a constructor and a single send method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor.

- A third-party app which acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.

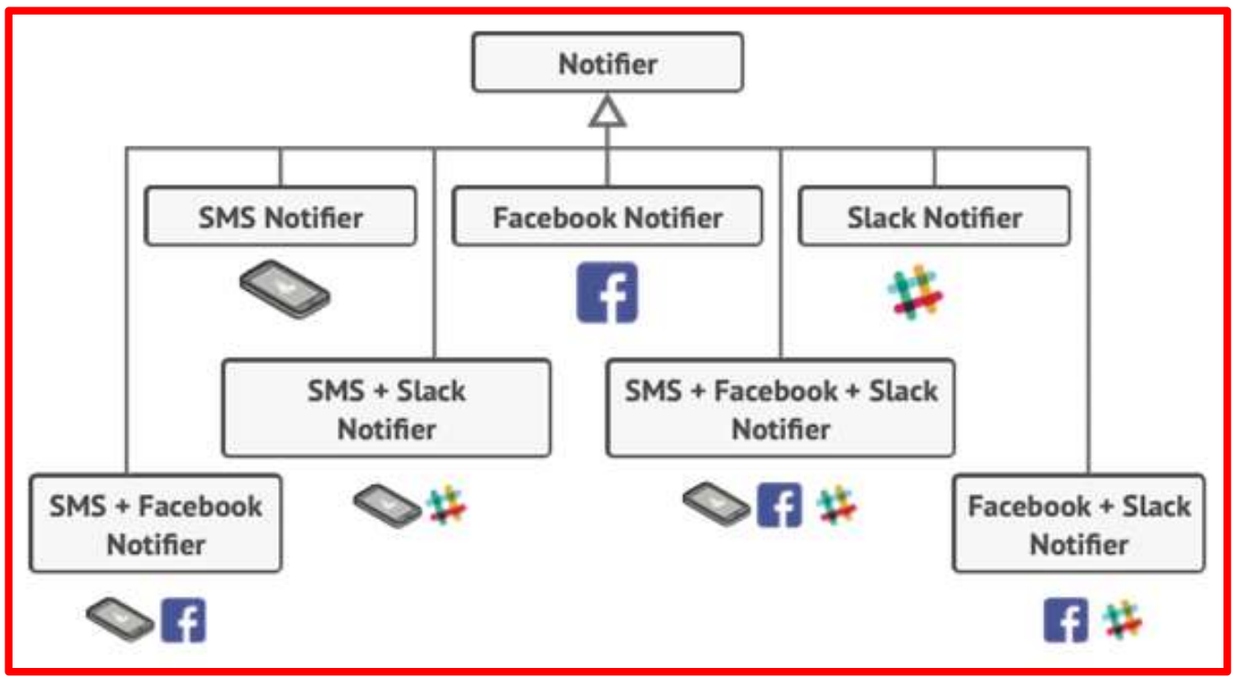


- At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.

- How hard can that be? You extended the Notifier class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.
- But then someone reasonably asked you, “Why can’t you use several notification types at once? If your house is on fire, you’d probably want to be informed through every channel.”



- We tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.



- Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.

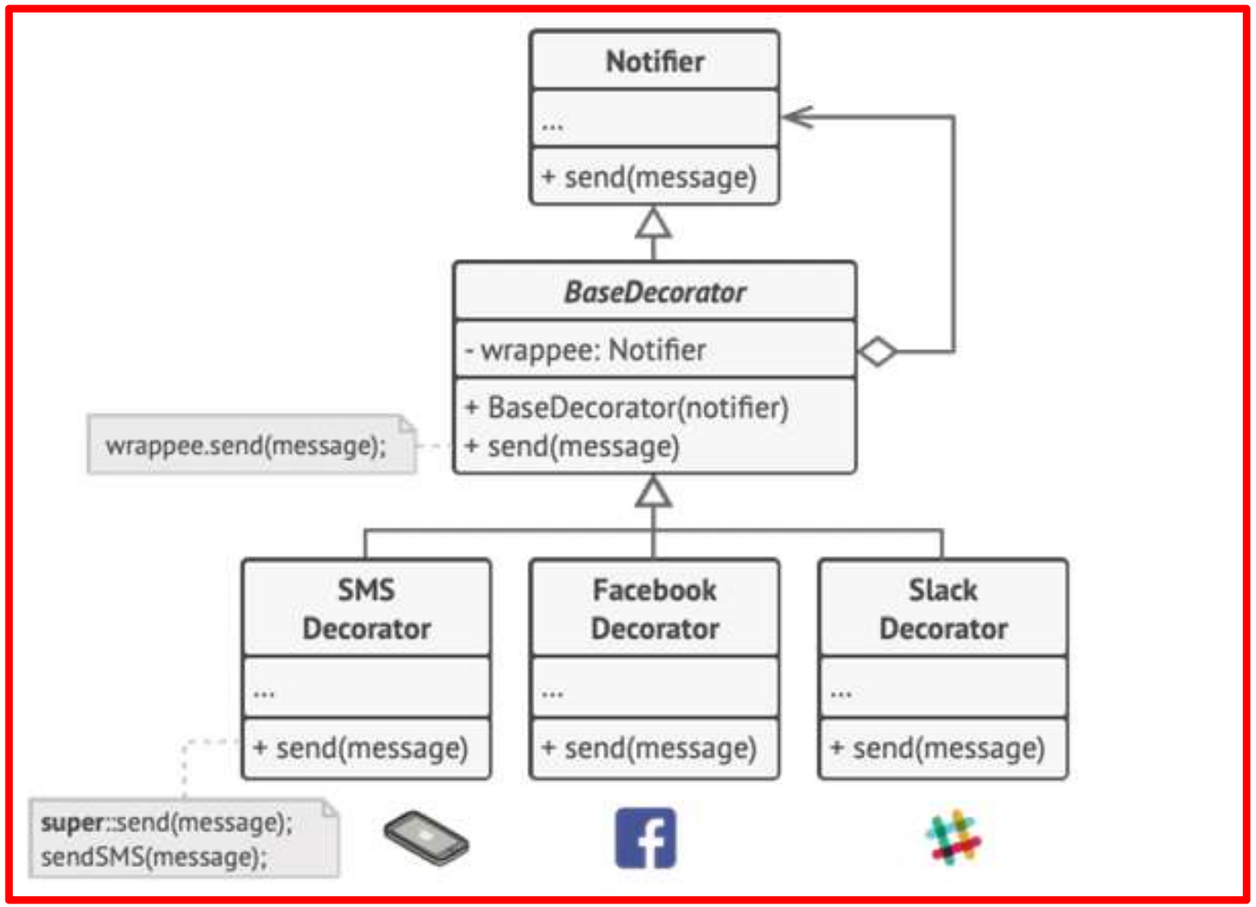
- ***Inheritance is static.*** You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.

- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

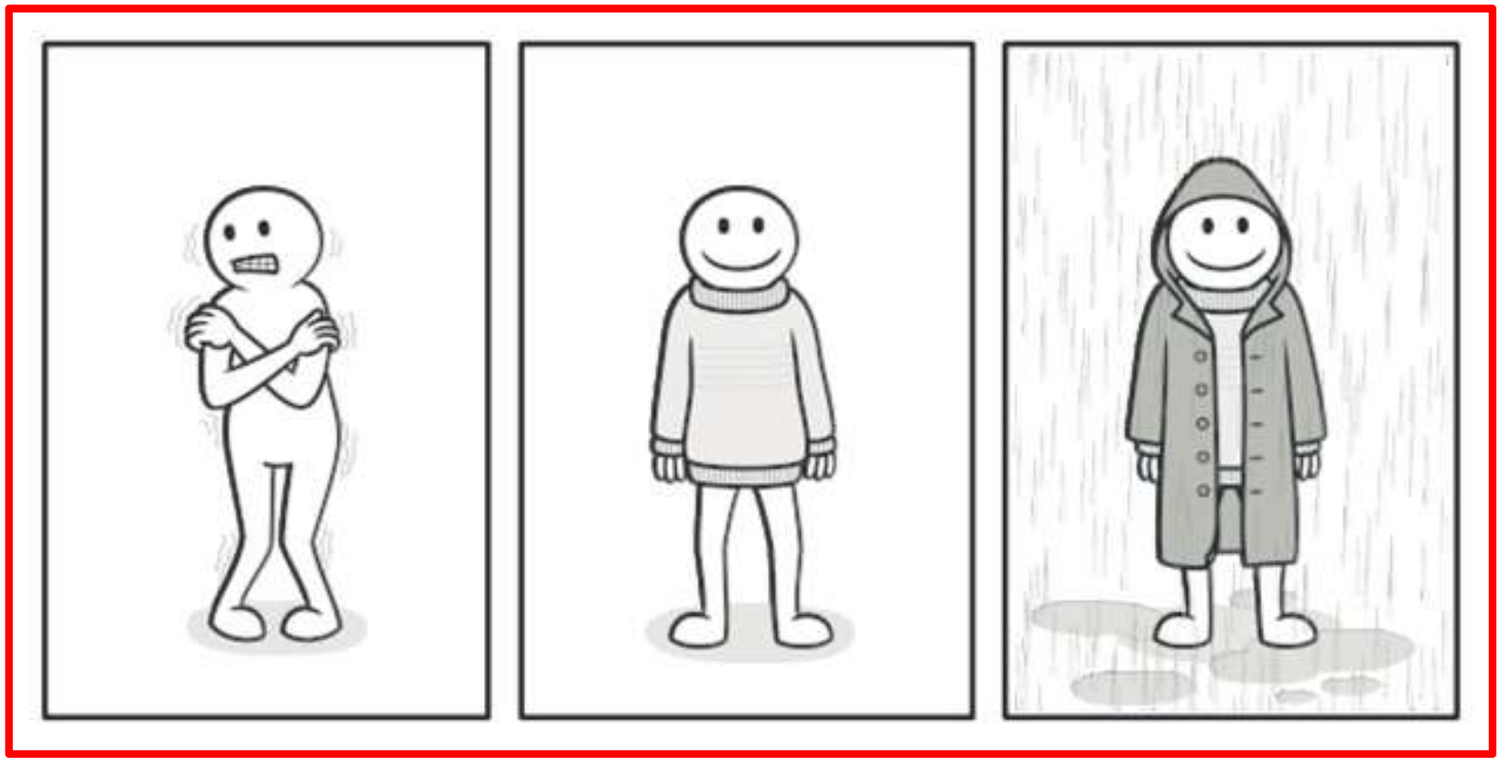
- One of the ways to overcome these caveats is by using ***Aggregation* or *Composition*** instead of ***Inheritance***. Both of the alternatives work almost the same way: one object *has a* reference to another and delegates it some work, whereas with inheritance, the object itself *is* able to do that work, inheriting the behavior from its superclass.

- With this new approach you can easily substitute the linked “helper” object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work.

- ***"Wrapper"*** is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern. A *wrapper* is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.


Notifier
...
+ send(message)
BaseDecorator
- wrappee: Notifier
+ BaseDecorator(notifier)
+ send(message)
wrappee.send(message);
SMS Decorator
...
+ send(message)
Facebook Decorator
...
+ send(message)
Slack Decorator
...
+ send(message)
super::send(message);
sendSMS(message);

Real-World Analogy



Complexity: ★★☆

Popularity: ★★☆

Usage examples: The Decorator is pretty standard in Java code, especially in code related to streams.

Here are some examples of Decorator in core Java libraries:

- All subclasses of `java.io.InputStream`, `OutputStream`, `Reader` and `Writer` have constructors that accept objects of their own type.
- `java.util.Collections`, methods `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()`.
- `javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

2. Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

Pros and Cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

- ✗ It's hard to remove a specific wrapper from the wrappers stack.
- ✗ The initial configuration code of layers might look pretty ugly.

Facade Pattern

Introduction

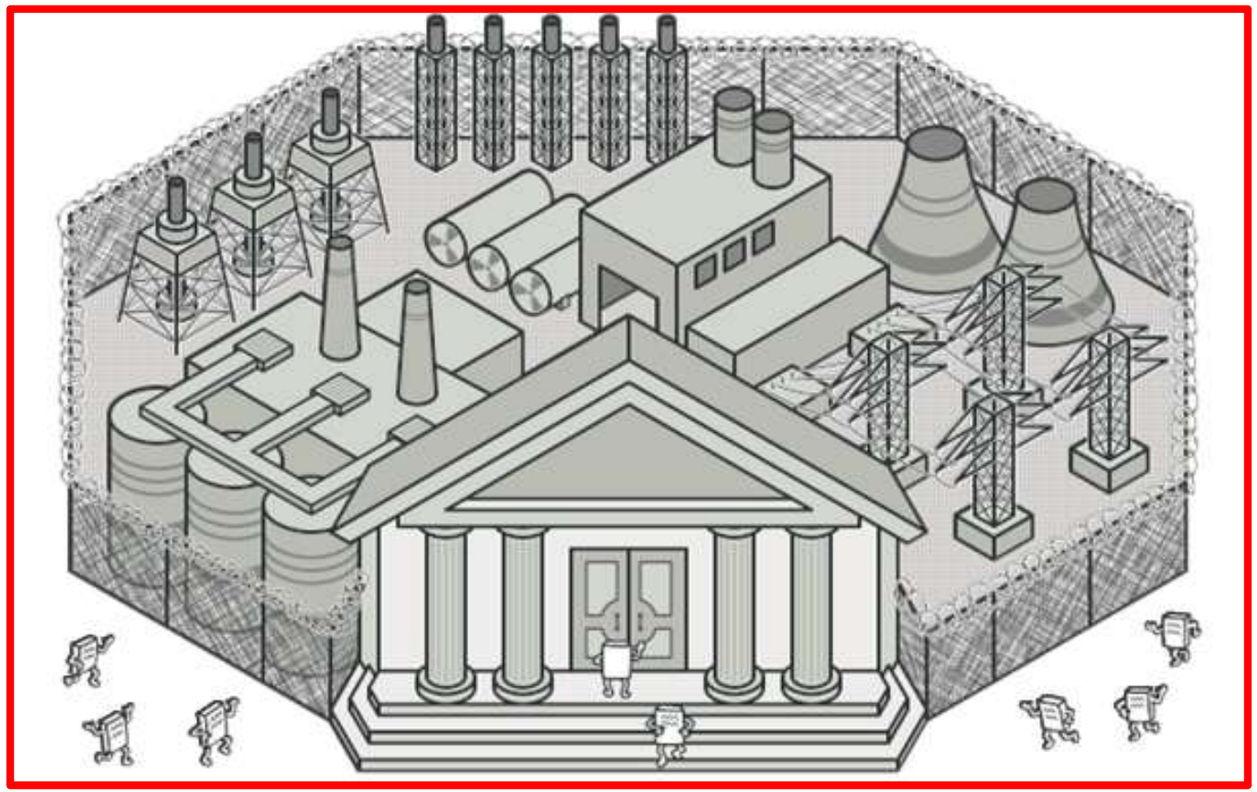
- ***Facade*** *is a structural design pattern that provides a simplified (but limited) interface to a complex system of classes, library or framework.*

- The facade pattern is used to help client applications easily interact with the system.

- While Facade decreases the overall complexity of the application, it also helps to move unwanted dependencies to one place.

Intent

- **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



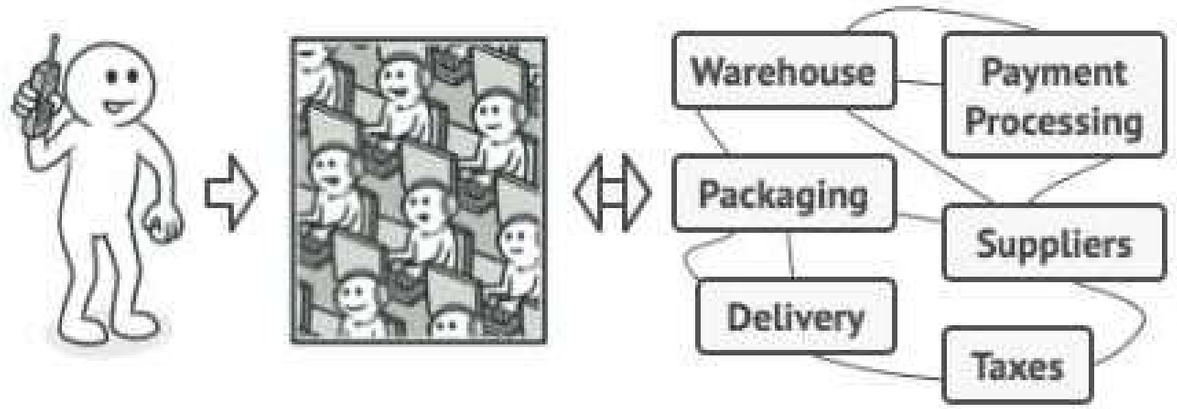
☹ Problem

- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.
- Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.
- For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you'll have your first facade.

Real-World Analogy



Placing orders by phone.

Complexity: ★☆☆

Popularity: ★★☆

Usage examples: The Facade pattern is commonly used in apps written in Java. It's especially handy when working with complex libraries and APIs.

Here are some Facade examples in core Java libs:

- `javax.faces.context.FacesContext` uses `LifeCycle`, `ViewHandler`, `NavigationHandler` classes under the hood, but most clients aren't aware of that.
- `javax.faces.context.ExternalContext` uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse` and others inside.

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

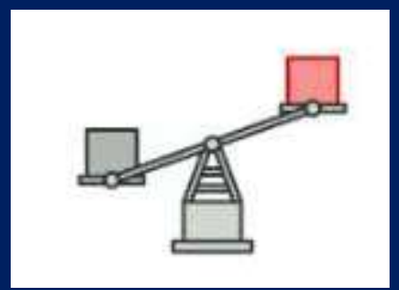
1. Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.

2. Use the Facade when you want to structure a subsystem into layers.

Pros and Cons

✓ You can isolate your code from the complexity of a subsystem.

✘ A facade can become ***A God Object*** coupled to all classes of an app.



Flyweight Pattern

Introduction

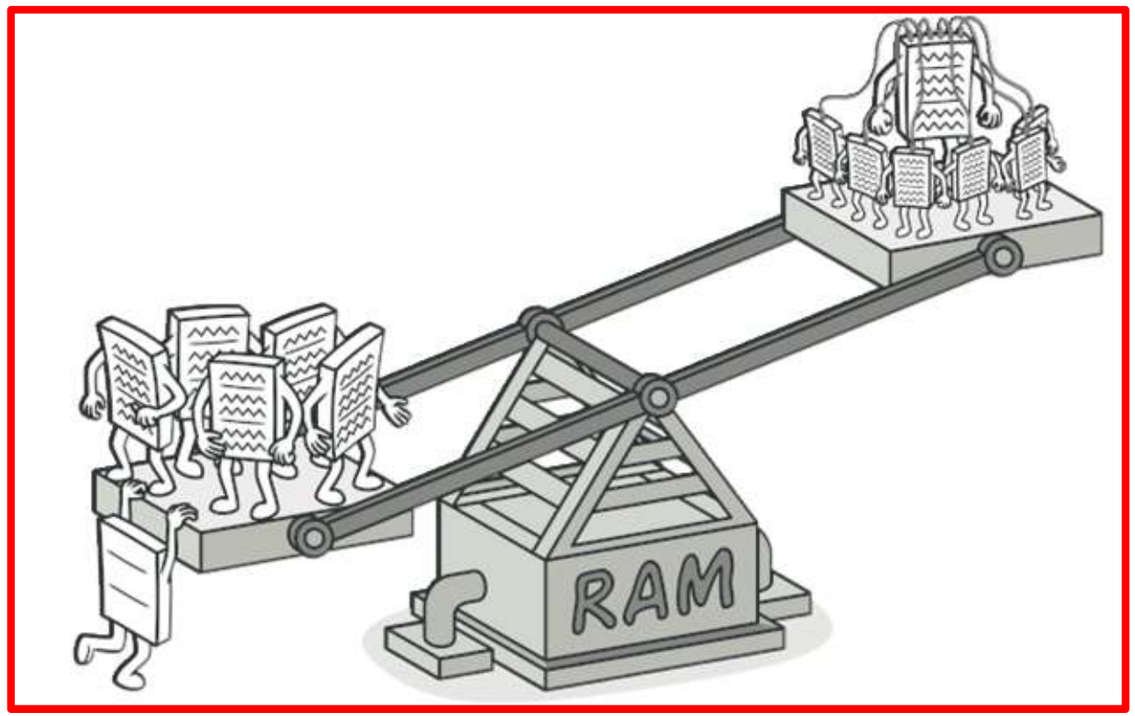
- ***Flyweight*** *is a structural design pattern that allows programs to support vast quantities of objects by keeping their memory consumption low.*

- The pattern achieves it by sharing parts of object state between multiple objects. In other words, the Flyweight saves RAM by caching the same data used by different objects.

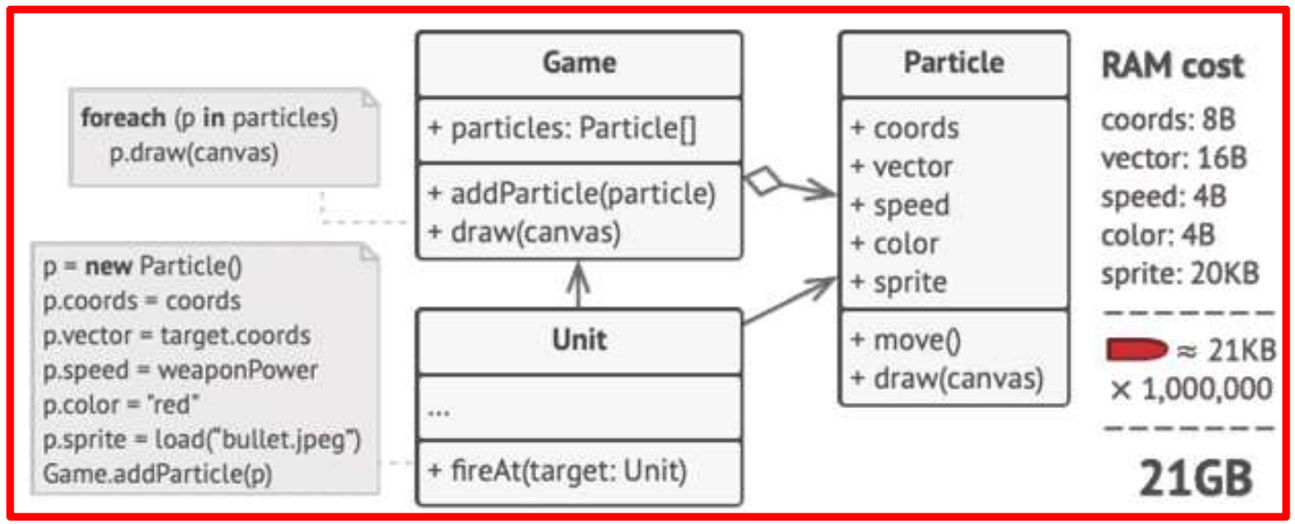
- The flyweight design pattern is used when we need to create a lot of Objects of a Class. Since every Object consumes memory space that can be crucial for low-memory devices (such as mobile devices or embedded systems), the flyweight design pattern can be applied to reduce the load on memory by sharing Objects.
  - String pool implementation in Java is one of the best examples of flyweight pattern implementation.

Intent

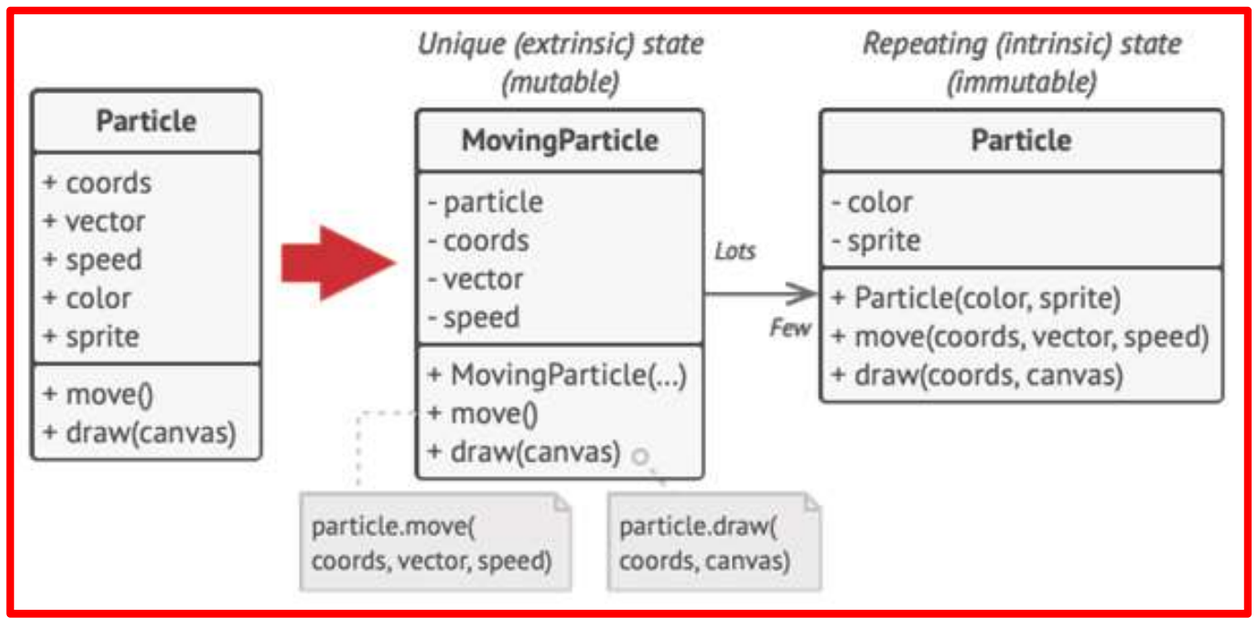
- **Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



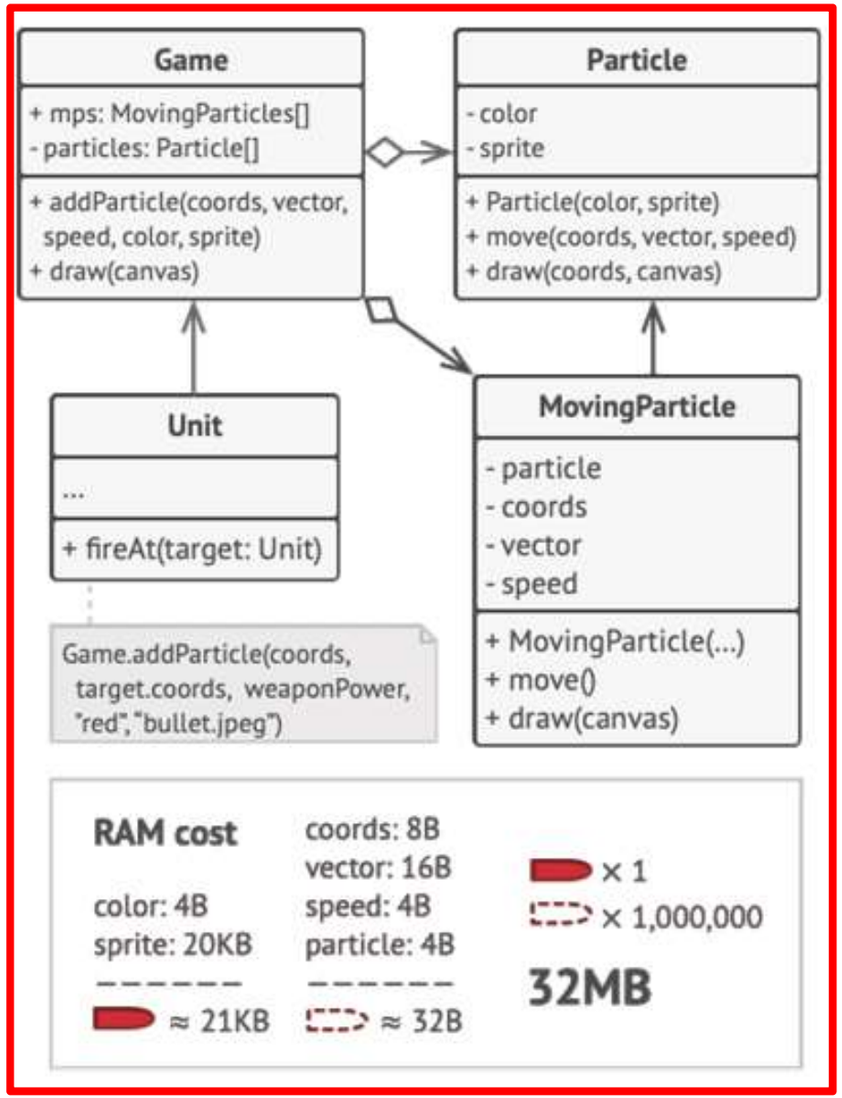
- Lets take an Example of Video Game
  - Game one Machine-1 Goes Super Fine
  - Same Game on Machine-2 Crashes Repeatedly



- On closer inspection of the Particle class, you may notice that the color and sprite fields consume a lot more memory than other fields. What's worse is that these two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.



- 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. As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.


Game
+ mps: MovingParticles[]
- particles: Particle[]
+ addParticle(coords, vector, speed, color, sprite)
+ draw(canvas)
Particle
- color
- sprite
+ Particle(color, sprite)
+ move(coords, vector, speed)
+ draw(coords, canvas)
Unit
...
+ fireAt(target: Unit)
Game.addParticle(coords, target.coords, weaponPower, "red", "bullet.jpeg")
MovingParticle
- particle
- coords
- vector
- speed
+ MovingParticle(...)
+ move()
+ draw(canvas)
RAM cost
color: 4B
sprite: 20KB
≈ 21KB
coords: 8B
vector: 16B
speed: 4B
particle: 4B
≈ 32B
× 1
× 1,000,000
32MB

Complexity: ★★★

Popularity: ★☆☆

Usage examples: The Flyweight pattern has a single purpose: minimizing memory intake. If your program doesn't struggle with a shortage of RAM, then you might just ignore this pattern for a while.

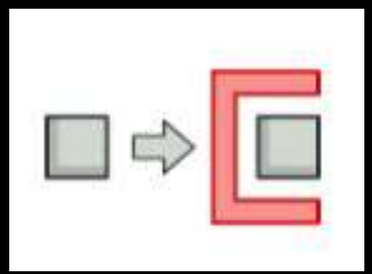
Examples of Flyweight in core Java libraries:

`java.lang.Integer#valueOf(int)` (also `Boolean`, `Byte`, `Character`, `Short`, `Long` and `BigDecimal`)

1. Use the ***Flyweight pattern*** only when your program must support a huge number of objects which barely fit into available RAM.

Pros and Cons

- ✓ You can save lots of RAM, assuming your program has tons of similar objects.
- ✘ 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.
- ✘ The code becomes much more complicated.



Proxy Pattern

Introduction

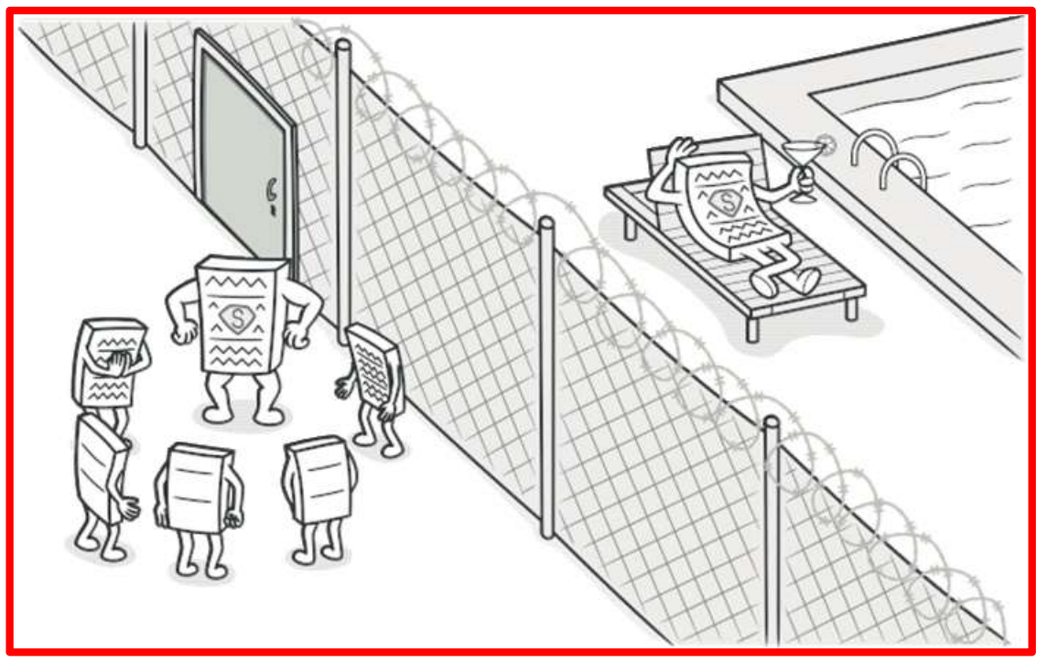
- ***Proxy*** *is a structural design pattern that provides an object that acts as a substitute for a real service object used by a client. A proxy receives client requests, does some work (access control, caching, etc.) and then passes the request to a service object.*

- The proxy object has the same interface as a service, which makes it interchangeable with a real object when passed to a client.

- The proxy pattern provides a placeholder for another Object to control access to it. This pattern is used when we want to provide controlled access to functionality.

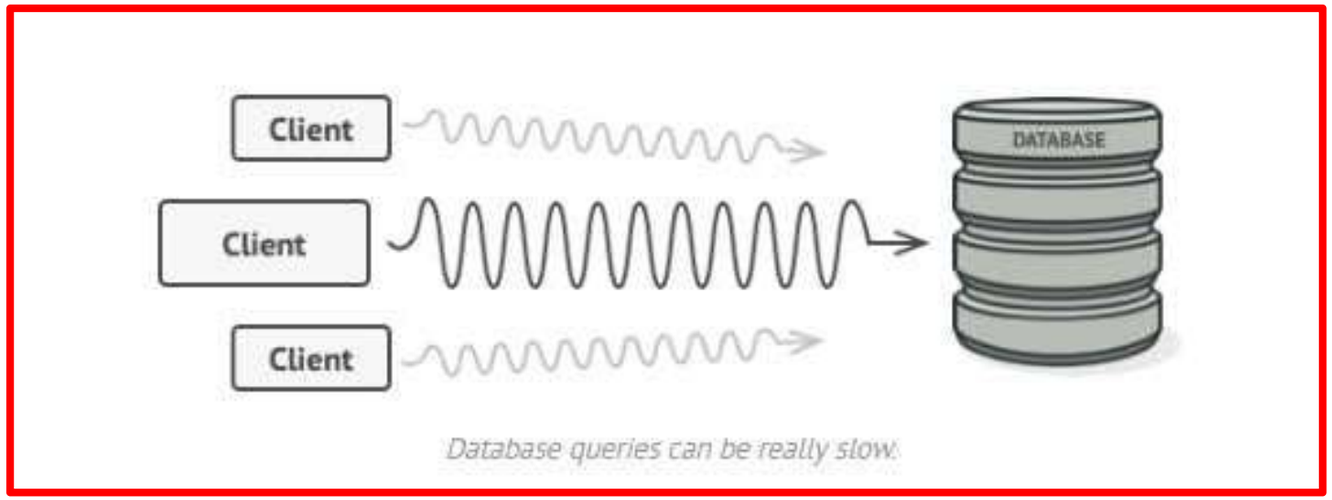
Intent

- **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.



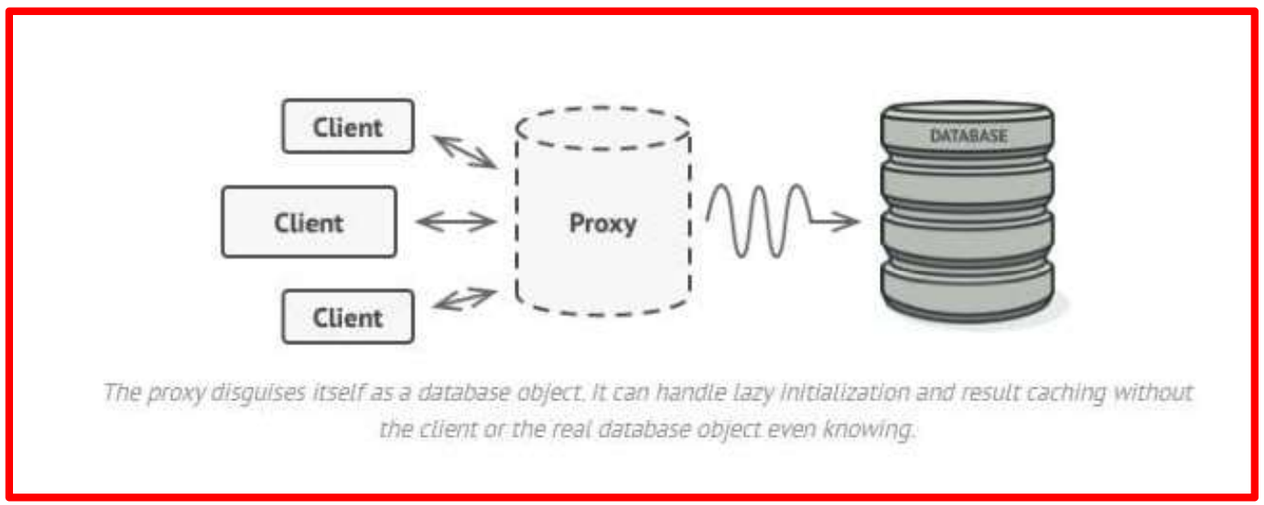
Problem

- Why would you want to control access to an object? Here is an example: you have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.



- You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

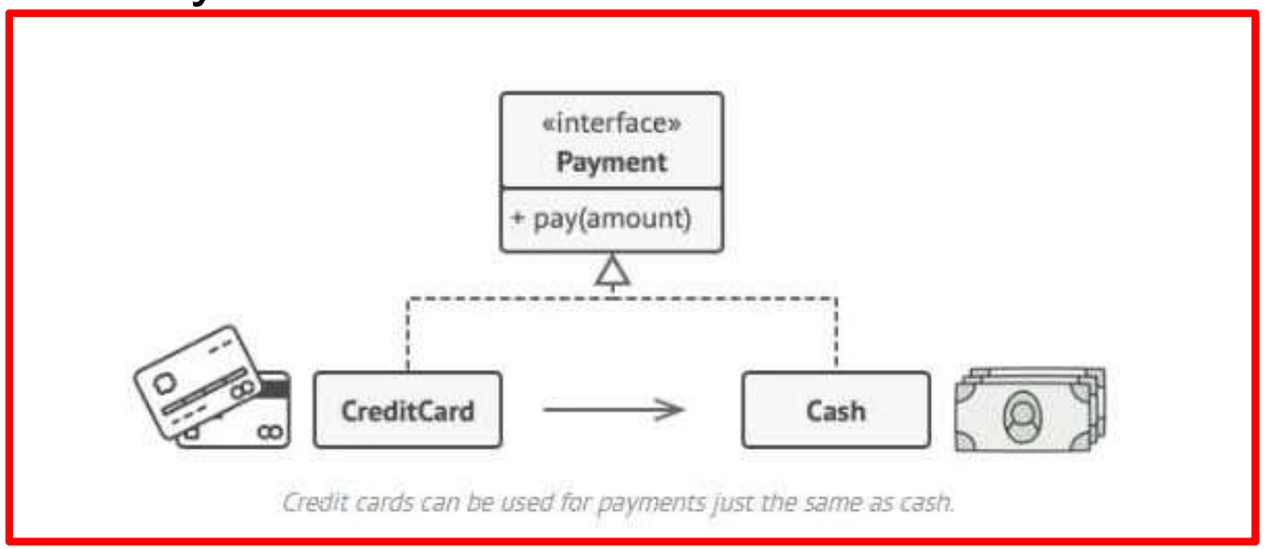
- The ***Proxy pattern*** suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object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



The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

Real-World Analogy

- A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment.
- A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.



Credit cards can be used for payments just the same as cash.

Complexity: ★★☆

Popularity: ★☆☆

Usage examples: While the Proxy pattern isn't a frequent guest in most Java applications, it's still very handy in some special cases. It's irreplaceable when you want to add some additional behaviors to an object of some existing class without changing the client code.

Some examples of proxies in standard Java libraries:

- `java.lang.reflect.Proxy`
- `java.rmi.*`
- `javax.ejb.EJB` (see comments)
- `javax.inject.Inject` (see comments)
- `javax.persistence.PersistenceContext`

Example

```
// Driver code
    public static void main(String args[])
    {
        int arr[] = { 2, 3, 4, 10, 40 };
        int x = 10;

        // Function call
        int result = search(arr, x);
        if (result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index “ + result);
    }
}
```

Applicability

1. Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

2. Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.

3. Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.

Pros and Cons

- ✓ You can control the service object without clients knowing about it.
- ✓ You can manage the lifecycle of the service object when clients don't care about it.
- ✓ The proxy works even if the service object isn't ready or is not available.
- ✓ *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

- ✗ The code may become more complicated since you need to introduce a lot of new classes.
- ✗ The response from the service might get delayed.

Summary

In this lesson, you should have learned how to:

- Analysis of Structural DP
- Exploration of Adapter Pattern, Composite Pattern
- Proxy Pattern , Flyweight Pattern
- Facade Pattern, Bridge Pattern
- Decorator Pattern