## Structural Design Patterns:

Structural design patterns are design patterns that deal with the composition of classes and objects to form larger structures. They help to simplify the relationships between classes and objects, and make the code more flexible and reusable.

Here are some commonly used structural design patterns:

1. Adapter Pattern: This pattern allows two incompatible interfaces to work together by creating a bridge between them. It acts as a translator between two different interfaces and allows them to communicate with each other.
2. Bridge Pattern: This pattern separates an abstraction from its implementation, allowing them to vary independently. It involves creating two separate class hierarchies - one for the abstraction and one for the implementation - and then using a bridge to connect them.
3. Composite Pattern: This pattern allows you to treat a group of objects as a single object. It involves creating a tree-like structure of objects, where each object can have one or more child objects. The composite object can then be treated as a single object, while still allowing access to the individual objects within it.
4. Decorator Pattern: This pattern allows you to add behavior to an object dynamically, without affecting the behavior of other objects in the same class. It involves creating a wrapper around an object, which can then be used to add new functionality to the original object.
5. Facade Pattern: This pattern provides a simplified interface to a complex system, making it easier to use. It involves creating a single interface that hides the complexity of a system, and provides a simple way to access its functionality.
6. Flyweight Pattern: This pattern is used to minimize memory usage by sharing data between similar objects. It involves creating a pool of objects that can be reused, rather than creating new objects each time they are needed.
7. Proxy Pattern: This pattern provides a surrogate or placeholder for another object, allowing you to control access to it. It involves creating an object that acts as a stand-in for another object, and can be used to provide additional functionality or control access to the original object.

These are just a few examples of the many structural design patterns that are available. By using these patterns, you can make your code more modular, flexible, and reusable, and improve its overall quality and maintainability.

### Adapter Design Pattern:

The adapter pattern is a structural design pattern that allows two incompatible interfaces to work together by creating a bridge between them. It acts as a translator between two different interfaces and allows them to communicate with each other.

#### Steps to implement Adapter Design Pattern:

1. Find out all your requirements that you need to implement in your code base.
2. Create an adapter interface and for each requirement put a method in that interface.
3. Figure out how to implement that requirement via 3rd party or legacy api’s.

#### Here's an example scenario when the adapter pattern can be useful:

1. Database Adapter: Suppose you have a legacy database that has an incompatible interface with your new application. You can create an adapter that translates the interface of the legacy database to a common interface that your application can use. This allows you to reuse the existing data in your legacy database without having to modify your application.
2. File Format Adapter: Suppose you have an application that can read and write data in a specific file format, but you need to add support for a new file format. You can create an adapter that translates the new file format to the common interface used by your application. This allows you to add support for the new file format without having to modify your existing code.
3. Networking Adapter: Suppose you have an application that needs to communicate with a web service that has an incompatible interface. You can create an adapter that translates the web service interface to a common interface used by your application. This allows your application to communicate with the web service without having to modify your existing code.
4. GUI Widget Adapter: Suppose you have a GUI library that has a specific set of widgets, but you need to use a widget from another library that has an incompatible interface. You can create an adapter that translates the widget interface to a common interface used by your GUI library. This allows you to use the widget from the other library without having to modify your existing code.
5. Legacy Adapter: Suppose you have a client application that needs to communicate with a legacy system, but the legacy system uses a different interface than the client application. Instead of modifying the client application to work with the legacy system's interface, you can use the adapter pattern to create an adapter that translates the requests and responses between the two interfaces.

The adapter pattern can also be used when you want to reuse an existing class that doesn't meet the requirements of your current project. By creating an adapter, you can modify the behavior of the existing class to meet the requirements of your project without changing its code.

In summary, the adapter pattern is useful when you need to make two incompatible interfaces work together, or when you want to reuse an existing class that doesn't meet the requirements of your current project.

##### Example 1:

Suppose you have an e-commerce website that allows customers to purchase products using various payment gateways, such as PayPal, Stripe, and Square. Each payment gateway has its own API and interface for processing payments, which makes it difficult to integrate them into your website without modifying the existing code.

To solve this problem, you can use the adapter pattern to create adapters for each payment gateway that translate their API and interface to a common interface that your website can use. The adapters act as a bridge between your website and the payment gateways, allowing them to work together seamlessly.

Here's an example implementation of the adapter pattern for payment gateways in Java:

| // This is the interface that defines the common payment gateway interface public interface PaymentGateway {  public void processPayment(double amount); }  // This is the adapter for the PayPal payment gateway public class PayPalAdapter implements PaymentGateway {  private PayPalAPI paypalAPI;    public PayPalAdapter(PayPalAPI paypalAPI) {  this.paypalAPI = paypalAPI;  }    public void processPayment(double amount) {  paypalAPI.makePayment(amount);  } }  // This is the adapter for the Stripe payment gateway public class StripeAdapter implements PaymentGateway {  private StripeAPI stripeAPI;    public StripeAdapter(StripeAPI stripeAPI) {  this.stripeAPI = stripeAPI;  }    public void processPayment(double amount) {  stripeAPI.charge(amount);  } }  // This is the adapter for the Square payment gateway public class SquareAdapter implements PaymentGateway {  private SquareAPI squareAPI;    public SquareAdapter(SquareAPI squareAPI) {  this.squareAPI = squareAPI;  }    public void processPayment(double amount) {  squareAPI.processPayment(amount);  } } |
| --- |

In this example, we define a PaymentGateway interface that defines the common payment gateway interface. We then create three different adapters - PayPalAdapter, StripeAdapter, and SquareAdapter - that implement the PaymentGateway interface and translate the API and interface of each payment gateway to the common interface. This allows us to use any of these payment gateways interchangeably in our e-commerce website, without modifying the existing code.

Overall, using the adapter pattern for payment gateways can simplify the integration of multiple payment gateways into an e-commerce website and make it more flexible and scalable.

##### Example 2:

Suppose you have a calendar app that needs to display events from different calendars, such as Google Calendar, Apple Calendar, and Microsoft Outlook Calendar. Each calendar has its own API and interface for retrieving event data, which makes it difficult to integrate them into the calendar app without modifying the existing code.

To solve this problem, you can use the adapter pattern to create adapters for each calendar that translate their API and interface to a common interface that your calendar app can use. The adapters act as a bridge between your calendar app and the different calendars, allowing them to work together seamlessly.

Here's an example implementation of the adapter pattern for calendars in Java:

| // This is the interface that defines the common calendar event interface public interface CalendarEvent {  public String getTitle();  public String getLocation();  public Date getStartTime();  public Date getEndTime(); }  // This is the adapter for the Google Calendar public class GoogleCalendarAdapter implements CalendarEvent {  private GoogleCalendarAPI googleCalendarAPI;  private GoogleCalendarEvent googleCalendarEvent;    public GoogleCalendarAdapter(GoogleCalendarAPI googleCalendarAPI, GoogleCalendarEvent googleCalendarEvent) {  this.googleCalendarAPI = googleCalendarAPI;  this.googleCalendarEvent = googleCalendarEvent;  }    public String getTitle() {  return googleCalendarEvent.getSummary();  }    public String getLocation() {  return googleCalendarEvent.getLocation();  }    public Date getStartTime() {  return googleCalendarEvent.getStart().getDate();  }    public Date getEndTime() {  return googleCalendarEvent.getEnd().getDate();  } }  // This is the adapter for the Apple Calendar public class AppleCalendarAdapter implements CalendarEvent {  private AppleCalendarAPI appleCalendarAPI;  private AppleCalendarEvent appleCalendarEvent;    public AppleCalendarAdapter(AppleCalendarAPI appleCalendarAPI, AppleCalendarEvent appleCalendarEvent) {  this.appleCalendarAPI = appleCalendarAPI;  this.appleCalendarEvent = appleCalendarEvent;  }    public String getTitle() {  return appleCalendarEvent.getTitle();  }    public String getLocation() {  return appleCalendarEvent.getLocation();  }    public Date getStartTime() {  return appleCalendarEvent.getStartDate();  }    public Date getEndTime() {  return appleCalendarEvent.getEndDate();  } }  // This is the adapter for the Microsoft Outlook Calendar public class OutlookCalendarAdapter implements CalendarEvent {  private OutlookCalendarAPI outlookCalendarAPI;  private OutlookCalendarEvent outlookCalendarEvent;    public OutlookCalendarAdapter(OutlookCalendarAPI outlookCalendarAPI, OutlookCalendarEvent outlookCalendarEvent) {  this.outlookCalendarAPI = outlookCalendarAPI;  this.outlookCalendarEvent = outlookCalendarEvent;  }    public String getTitle() {  return outlookCalendarEvent.getSubject();  }    public String getLocation() {  return outlookCalendarEvent.getLocation();  }    public Date getStartTime() {  return outlookCalendarEvent.getStart();  }    public Date getEndTime() {  return outlookCalendarEvent.getEnd();  } } |
| --- |

In this example, we define a CalendarEvent interface that defines the common calendar event interface. We then create three different adapters - GoogleCalendarAdapter, AppleCalendarAdapter, and OutlookCalendarAdapter - that implement the CalendarEvent interface and translate the API and interface of each calendar to the common interface. This allows us to display events from different calendars interchangeably in our calendar app, without modifying the existing code.

Overall, using the adapter pattern for calendars can simplify the integration of multiple calendars into a calendar app and make it more flexible and scalable.

##### Example 3:

Suppose you have a legacy database that uses a different interface than the one required by your application. You want to reuse the data in the legacy database in your new application without having to modify your application's code.

To solve this problem, you can create a database adapter that translates the interface of the legacy database to the one used by your application. Here's an example implementation in Java:

| // Interface used by the application public interface Database {  public void connect(String username, String password);  public ResultSet executeQuery(String query);  public void disconnect(); }  // Legacy database interface public interface LegacyDatabase {  public void open(String username, String password);  public Object runQuery(String query);  public void close(); }  // Adapter that implements the application's interface public class DatabaseAdapter implements Database {  private LegacyDatabase legacyDatabase;    public DatabaseAdapter(LegacyDatabase legacyDatabase) {  this.legacyDatabase = legacyDatabase;  }    public void connect(String username, String password) {  this.legacyDatabase.open(username, password);  }    public ResultSet executeQuery(String query) {  Object result = this.legacyDatabase.runQuery(query);  // Convert the legacy result to a ResultSet object  ResultSet resultSet = convertLegacyResult(result);  return resultSet;  }    public void disconnect() {  this.legacyDatabase.close();  }    // Helper method to convert the legacy result to a ResultSet object  private ResultSet convertLegacyResult(Object result) {  // Implementation omitted for brevity  return null;  } } |
| --- |

In this example, Database is the interface used by the application, LegacyDatabase is the legacy database interface, and DatabaseAdapter is the adapter that translates the legacy database interface to the application's interface.

The DatabaseAdapter takes a LegacyDatabase object in its constructor and implements the methods of the Database interface by calling the corresponding methods of the LegacyDatabase interface. The executeQuery method also converts the result returned by the legacy database to a ResultSet object used by the application.

With this adapter, your application can use the legacy database without having to modify its code.

### Decorator Design Pattern:

The decorator design pattern addresses the problem of adding new functionality to an object dynamically without changing its original implementation or class hierarchy. In other words, the decorator pattern allows us to modify the behavior of an object by wrapping it with additional behavior or functionality at runtime.

One of the main problems that the decorator pattern solves is the limitation of inheritance. When we use inheritance to extend the behavior of a class, we create a subclass that is tightly coupled to its parent class. This can make it difficult to add new behavior or to modify existing behavior in the future without affecting the entire class hierarchy.

The decorator pattern can be used in various scenarios where we need to add new functionality or behavior to an object dynamically without changing its original implementation or class hierarchy. Here are some common scenarios where the decorator pattern can be used:

1. Text formatting: As explained earlier, the decorator pattern can be used to add different font styles to a text editor dynamically at runtime.
2. Input/output stream processing: The decorator pattern can be used to add functionality to an input or output stream, such as encryption, compression, or buffering.
3. GUI components: The decorator pattern can be used to add new features or behavior to GUI components, such as buttons or text fields, without modifying their original implementation.
4. Logging: The decorator pattern can be used to add logging functionality to an object, without changing its original behavior.
5. Caching: The decorator pattern can be used to add caching functionality to an object, without modifying its original implementation.
6. Security: The decorator pattern can be used to add security features to an object, such as authentication or authorization, without changing its original behavior.

In general, the decorator pattern is useful in scenarios where we need to add new behavior or functionality to an object dynamically and selectively, without modifying its original implementation or class hierarchy.

##### Example 1:

| public interface Text {  String getContent(); }  public class PlainText implements Text {  private String content;   public PlainText(String content) {  this.content = content;  }   public String getContent() {  return content;  } }  public abstract class TextDecorator implements Text {  private Text text;   public TextDecorator(Text text) {  this.text = text;  }   public String getContent() {  return text.getContent();  } }  public class BoldText extends TextDecorator {  public BoldText(Text text) {  super(text);  }   public String getContent() {  return "<b>" + super.getContent() + "</b>";  } }  public class ItalicText extends TextDecorator {  public ItalicText(Text text) {  super(text);  }   public String getContent() {  return "<i>" + super.getContent() + "</i>";  } }  public class TextEditor {  public static void main(String[] args) {  Text text = new PlainText("Hello, world!");  text = new BoldText(text);  text = new ItalicText(text);  System.out.println(text.getContent()); // Outputs "<i><b>Hello, world!</b></i>"  } } |
| --- |

##### Example 2:

| public abstract class StreamDecorator extends InputStream {  protected InputStream inputStream;   public StreamDecorator(InputStream inputStream) {  this.inputStream = inputStream;  }   public int read() throws IOException {  return inputStream.read();  }   public void close() throws IOException {  inputStream.close();  } }  public class BufferedInputStreamDecorator extends StreamDecorator {  private byte[] buffer = new byte[1024];  private int position = 0;   public BufferedInputStreamDecorator(InputStream inputStream) {  super(inputStream);  }   public int read() throws IOException {  if (position >= buffer.length) {  fillBuffer();  }   return buffer[position++];  }   private void fillBuffer() throws IOException {  int bytesRead = inputStream.read(buffer, 0, buffer.length);   if (bytesRead == -1) {  return;  }   position = 0;  } }  public class EncryptedInputStreamDecorator extends StreamDecorator {  public EncryptedInputStreamDecorator(InputStream inputStream) {  super(inputStream);  }   public int read() throws IOException {  int b = inputStream.read();  return (b == -1 ? b : b ^ 0xff);  } }  public class StreamProcessor {  public static void main(String[] args) throws Exception {  InputStream inputStream = new FileInputStream("input.txt");  inputStream = new BufferedInputStreamDecorator(inputStream);  inputStream = new EncryptedInputStreamDecorator(inputStream);   int b;  while ((b = inputStream.read()) != -1) {  System.out.print((char) b);  }   inputStream.close();  } } |
| --- |

### Facade Design Pattern: The Facade Design Pattern is a structural design pattern that provides a simplified interface to a complex system of classes, interfaces, and objects. The facade pattern is used when we need to simplify and unify the interactions between a client and a set of complex subsystems by providing a single interface that encapsulates the complexity of the subsystems.

In essence, the Facade Design Pattern provides a simplified interface to a complex system of classes, interfaces, and objects, making it easier for clients to use the system. The pattern accomplishes this by creating a new class that acts as a facade, or an interface, to the complex subsystem. This new class shields the client from the complexity of the subsystem by providing a simplified interface that the client can use to interact with the subsystem.

The Facade pattern is helpful when a system is complex and difficult to understand. It provides a simple interface to the client, which hides the complexities of the system. The client does not need to know about the underlying classes and interfaces used by the system.

##### Example 1:

For example, imagine you are building a music player application that can play music from different sources such as a CD, MP3 player, and streaming services. Each source has its own set of classes and interfaces, and the client code would need to interact with each of them differently. By using the Facade Design Pattern, you could create a new MusicPlayerFacade class that acts as a simplified interface to the different music sources, shielding the client from the complexity of interacting with each source separately.

| // Complex subsystem interface interface MusicSource {  void play();  void pause();  void stop(); }  // Concrete subsystems class CdPlayer implements MusicSource {  public void play() {  System.out.println("Playing CD...");  }   public void pause() {  System.out.println("Pausing CD...");  }   public void stop() {  System.out.println("Stopping CD...");  } }  class Mp3Player implements MusicSource {  public void play() {  System.out.println("Playing MP3...");  }   public void pause() {  System.out.println("Pausing MP3...");  }   public void stop() {  System.out.println("Stopping MP3...");  } }  class StreamingService implements MusicSource {  public void play() {  System.out.println("Playing streaming music...");  }   public void pause() {  System.out.println("Pausing streaming music...");  }   public void stop() {  System.out.println("Stopping streaming music...");  } }  // Facade class class MusicPlayerFacade {  private MusicSource cdPlayer;  private MusicSource mp3Player;  private MusicSource streamingService;   public MusicPlayerFacade() {  this.cdPlayer = new CdPlayer();  this.mp3Player = new Mp3Player();  this.streamingService = new StreamingService();  }   public void playCd() {  cdPlayer.play();  }   public void pauseCd() {  cdPlayer.pause();  }   public void stopCd() {  cdPlayer.stop();  }   public void playMp3() {  mp3Player.play();  }   public void pauseMp3() {  mp3Player.pause();  }   public void stopMp3() {  mp3Player.stop();  }   public void playStreamingMusic() {  streamingService.play();  }   public void pauseStreamingMusic() {  streamingService.pause();  }   public void stopStreamingMusic() {  streamingService.stop();  } }  // Client code public class Client {  public static void main(String[] args) {  MusicPlayerFacade musicPlayer = new MusicPlayerFacade();   // Play CD  musicPlayer.playCd();   // Play MP3  musicPlayer.playMp3();   // Play streaming music  musicPlayer.playStreamingMusic();  } } |
| --- |

##### Example 2:

| public class Computer {  private CPU cpu;  private Memory memory;  private HardDrive hardDrive;   public Computer() {  this.cpu = new CPU();  this.memory = new Memory();  this.hardDrive = new HardDrive();  }   public void startComputer() {  cpu.freeze();  memory.load(0, hardDrive.read(0, 100));  cpu.jump(0);  cpu.execute();  } }  public class CPU {  public void freeze() { /\* ... \*/ }  public void jump(long position) { /\* ... \*/ }  public void execute() { /\* ... \*/ } }  public class Memory {  public void load(long position, byte[] data) { /\* ... \*/ } }  public class HardDrive {  public byte[] read(long lba, int size) { /\* ... \*/ } } |
| --- |

In this example, the Computer class is a facade that provides a simple interface to the client. It hides the complexity of the CPU, Memory, and HardDrive subsystems, and provides a single startComputer method that the client can use to start the computer. The CPU, Memory, and HardDrive classes are the underlying subsystems that are used by the Computer class to start the computer.

Using the Facade pattern in this example makes it easy for the client to start the computer without having to understand the complexities of the CPU, Memory, and HardDrive subsystems.

### Flyweight Design Pattern:

The Flyweight design pattern is a structural design pattern that aims to minimize the memory usage and improve performance by sharing as much data as possible with similar objects.

In this pattern, objects are divided into intrinsic and extrinsic states. Intrinsic state is the part of the object that can be shared with other objects, while extrinsic state is the part that is unique to the object. The idea is to create a factory that will maintain a pool of shared objects and only create new objects when necessary.

##### Example 1:

| import java.util.HashMap; import java.util.Map;  public class FlyweightFactory {  private static final Map<String, Flyweight> flyweights = new HashMap<>();   public static Flyweight getFlyweight(String key) {  Flyweight flyweight = flyweights.get(key);  if (flyweight == null) {  flyweight = new ConcreteFlyweight(key);  flyweights.put(key, flyweight);  }  return flyweight;  } }  public interface Flyweight {  void operation(); }  public class ConcreteFlyweight implements Flyweight {  private final String key;   public ConcreteFlyweight(String key) {  this.key = key;  }   public void operation() {  System.out.println("ConcreteFlyweight with key " + key + " is executing.");  } }  public class Client {  public static void main(String[] args) {  Flyweight flyweight1 = FlyweightFactory.getFlyweight("key1");  flyweight1.operation();   Flyweight flyweight2 = FlyweightFactory.getFlyweight("key2");  flyweight2.operation();   Flyweight flyweight3 = FlyweightFactory.getFlyweight("key1");  flyweight3.operation();  } } |
| --- |

In this example, FlyweightFactory maintains a pool of shared flyweight objects. The ConcreteFlyweight class represents the intrinsic state of the object, and the Flyweight interface defines the operation() method that all flyweight objects must implement.

In the Client class, we obtain two different flyweight objects using different keys, and a third flyweight object using the same key as the first object. Since the flyweight objects with the same key have the same intrinsic state, only one object is actually created and shared between the first and third objects. This helps reduce memory usage and improve performance.

##### Example 2:

Let's say you're creating a video game and you need to render a large number of trees in a forest. Each tree has many properties such as location, size, and texture, and creating a new object for every tree can be resource-intensive and slow down the game.

In this case, the Flyweight pattern can be used to optimize the performance of the game by reusing objects.

You can create a Tree interface that defines the methods that every tree should have, such as draw() and setPosition(). Then you can create a TreeFactory class that acts as a Flyweight factory and manages a pool of shared Tree objects.

When the game starts, the TreeFactory creates a pool of Tree objects, each with its own unique properties. When a new tree needs to be displayed on the screen, the TreeFactory finds an existing tree with similar properties and reuses it. This can significantly reduce the number of objects created and improve the game's performance.

| // Tree interface public interface Tree {  void draw(Graphics g);  void setPosition(int x, int y); }  // Concrete tree implementation public class OakTree implements Tree {  private int x;  private int y;  private Image image;   public OakTree(Image image) {  this.image = image;  }   @Override  public void setPosition(int x, int y) {  this.x = x;  this.y = y;  }   @Override  public void draw(Graphics g) {  g.drawImage(image, x, y, null);  } }  // Flyweight factory public class TreeFactory {  private Map<String, Tree> treeMap = new HashMap<>();   public Tree getTree(String type) {  Tree tree = treeMap.get(type);   if (tree == null) {  // Create a new tree object  Image image = loadImage(type + ".png");  tree = new OakTree(image);   // Add the new tree to the pool  treeMap.put(type, tree);  }   return tree;  }   private Image loadImage(String filename) {  // Load image from file  // ...  } }  // Client code public class Game {  private TreeFactory treeFactory = new TreeFactory();   public void renderForest() {  // Render 1000 trees  for (int i = 0; i < 1000; i++) {  Tree tree = treeFactory.getTree("oak");  tree.setPosition(i \* 10, 0);  tree.draw(graphics);  }  } } |
| --- |

In this example, the TreeFactory acts as a Flyweight factory that creates and manages a pool of Tree objects. The Game class uses the TreeFactory to get Tree objects and render them on the screen. If a Tree with the same properties (e.g., image) already exists in the pool, the TreeFactory returns it instead of creating a new one. This helps reduce memory usage and improve performance.