Design Patterns

1. **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
2. **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
3. **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.
4. **Class patterns** deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.

Object patterns deal with object relationships, which can be changed at run-time and are more dynamic

Inheritance versus Composition

- The two most common techniques for reusing functionality in object-oriented systems are **class inheritance** and **object composition.**
- **Class Inheritence** lets you define the implementation of one class in terms of another's.
- Reuse by subclassing is often referred to as **white-box reuse**.
- The term "white-box" refers to **visibility**: With inheritance, the internals of parent classes are often visible to subclasses.

- **Object composition** is an alternative to class inheritance.
- New functionality is obtained by assembling or *composing* objects to get more complex functionality.
- Object composition requires that the objects being composed have well-defined interfaces.
- This style of reuse is called **black-box reuse**, because no internal details of objects are visible. Objects appear only as "black boxes."

Class inheritance

Advantages

- Supported by programming languages, defined statically at compile-time and is straightforward to use.
- Make it easier to modify the implementation being reused, when a subclass overrides some but not all operations.

Disadvantages

- Cannot change the implementations/representations inherited from parent classes at run-time.
- Implementation dependency between a subclass and its parent class.

Object composition

Advantages

- Defined dynamically at run-time by referencing interfaces of objects.
- Access other objects though their interfaces only, do not break encapsulation.
- Fewer implementation dependencies.
- Small class hierarchies

Disadvantages

- Design based on object composition have more objects
- The system's behavior will depend on their interrelationships instead of being defined in one class

One of the principle of object-oriented design: Favor object composition over class inheritance.

How to select a Design Pattern?

- Consider how design patterns solve design problems.
- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a cause of redesign.
- Consider what should be variable in your design.

Factory Method

```
public abstract class Restaurant {
    public Burger orderBurger() {
        Burger burger = createBurger();
        burger.prepare();
        return burger;
    }

    public abstract Burger createBurger();
}
```

```
public class BeefBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new BeefBurger();
    }
}
```

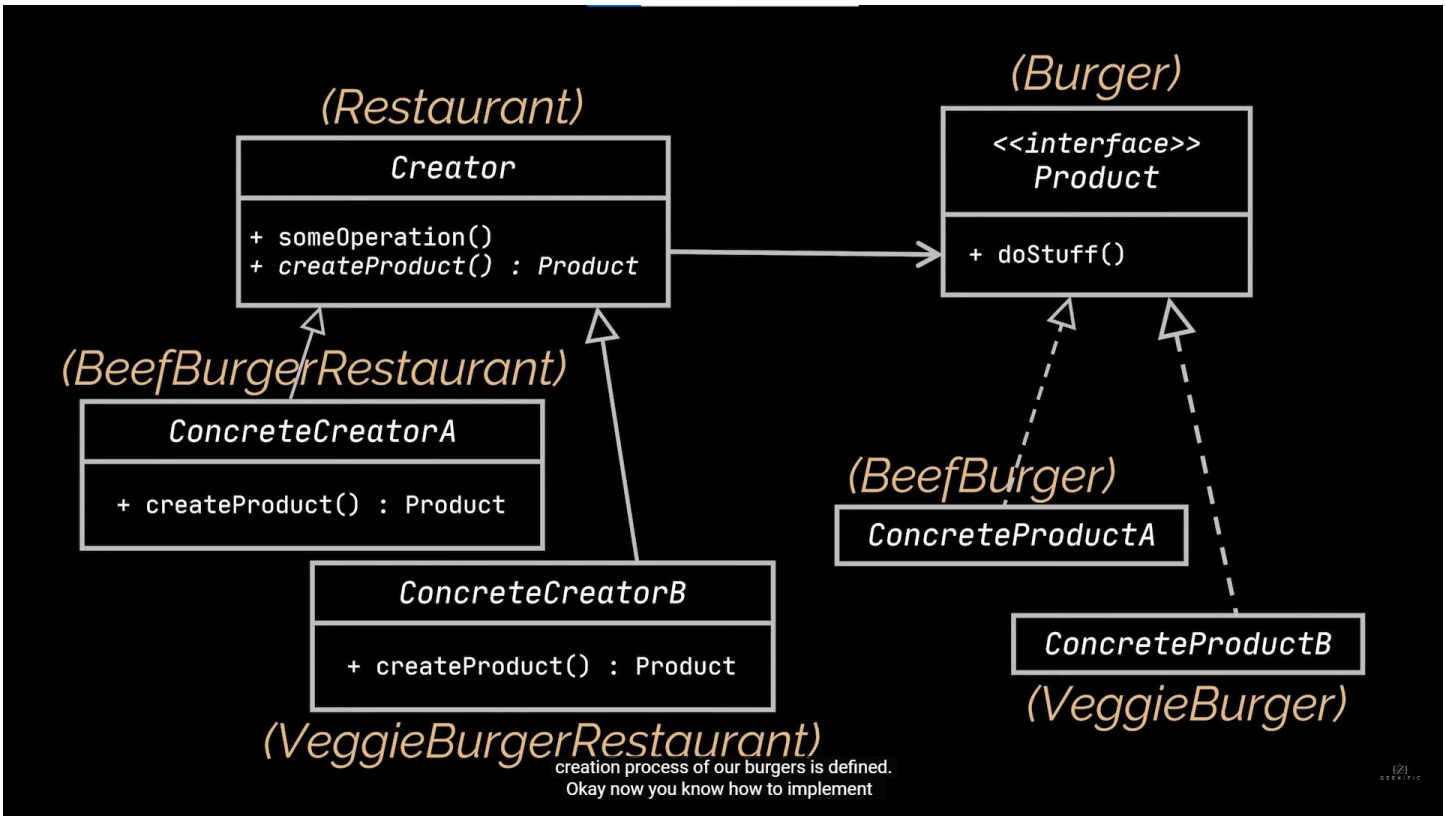
```
public class VeggieBurgerRestaurant extends Restaurant {
    @Override
    public Burger createBurger() {
        return new VeggieBurger();
    }
}
```

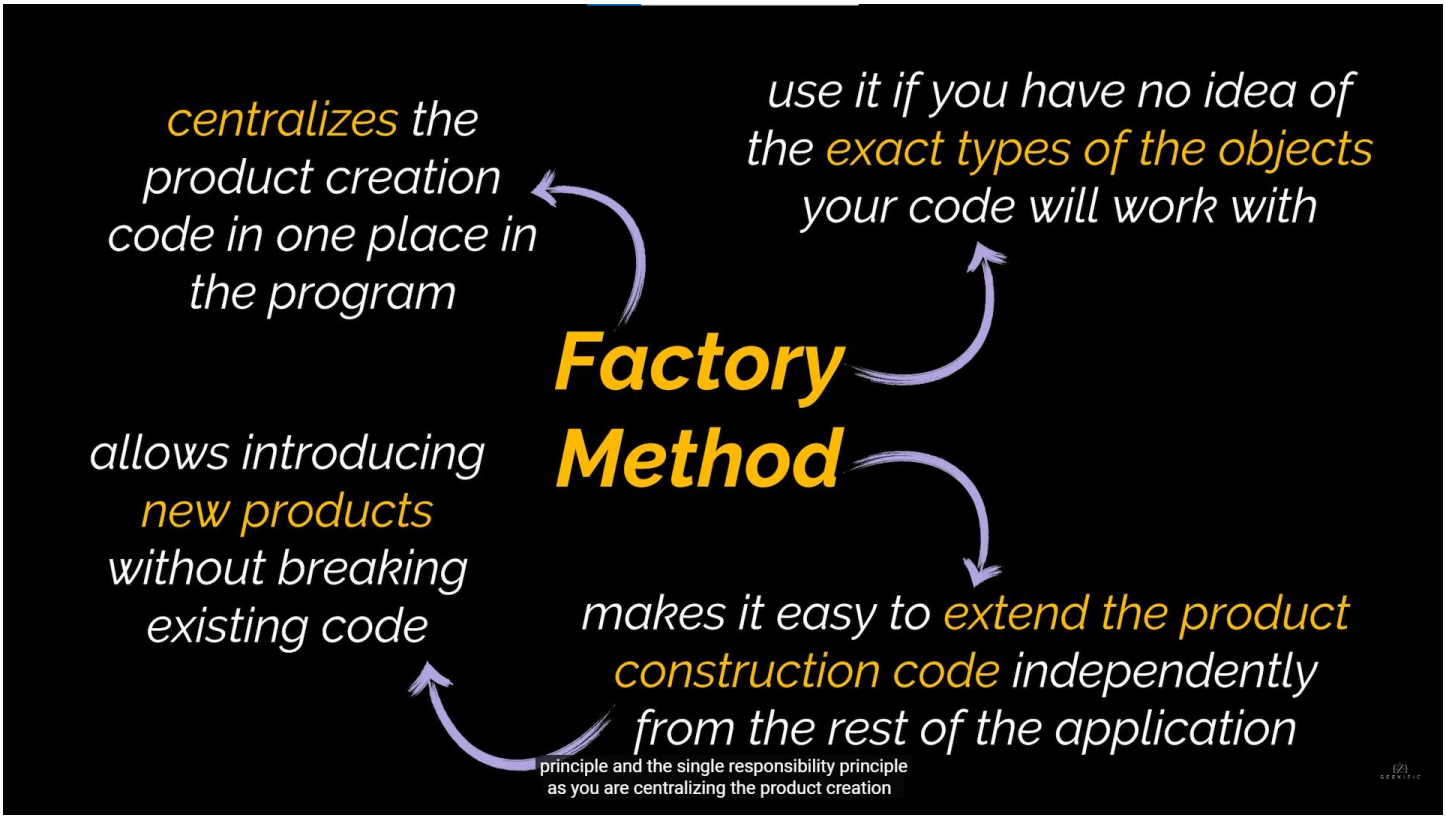
```
public interface Burger {
    void prepare();
}
```

```
public class BeefBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare beef
        // burger code
    }
}
```

```
public class VeggieBurger
        implements Burger {

    @Override
    void prepare() {
        // prepare veggie
        // burger code
    }
}
```

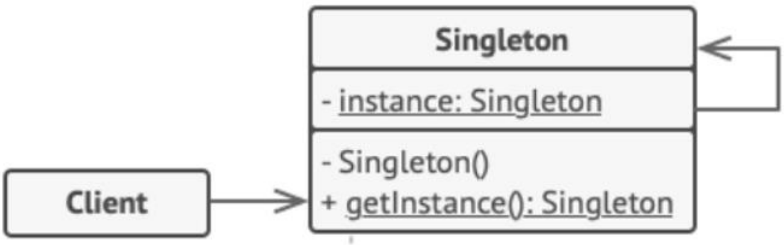
need it as the users of our restaurant
can now directly instantiate and invoke the





Singleton Method

Strcuture



1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

SINGLETON

*let you access your object from **anywhere** in your application*

*guarantee that only **one instance** of this class will be available at any point in time*

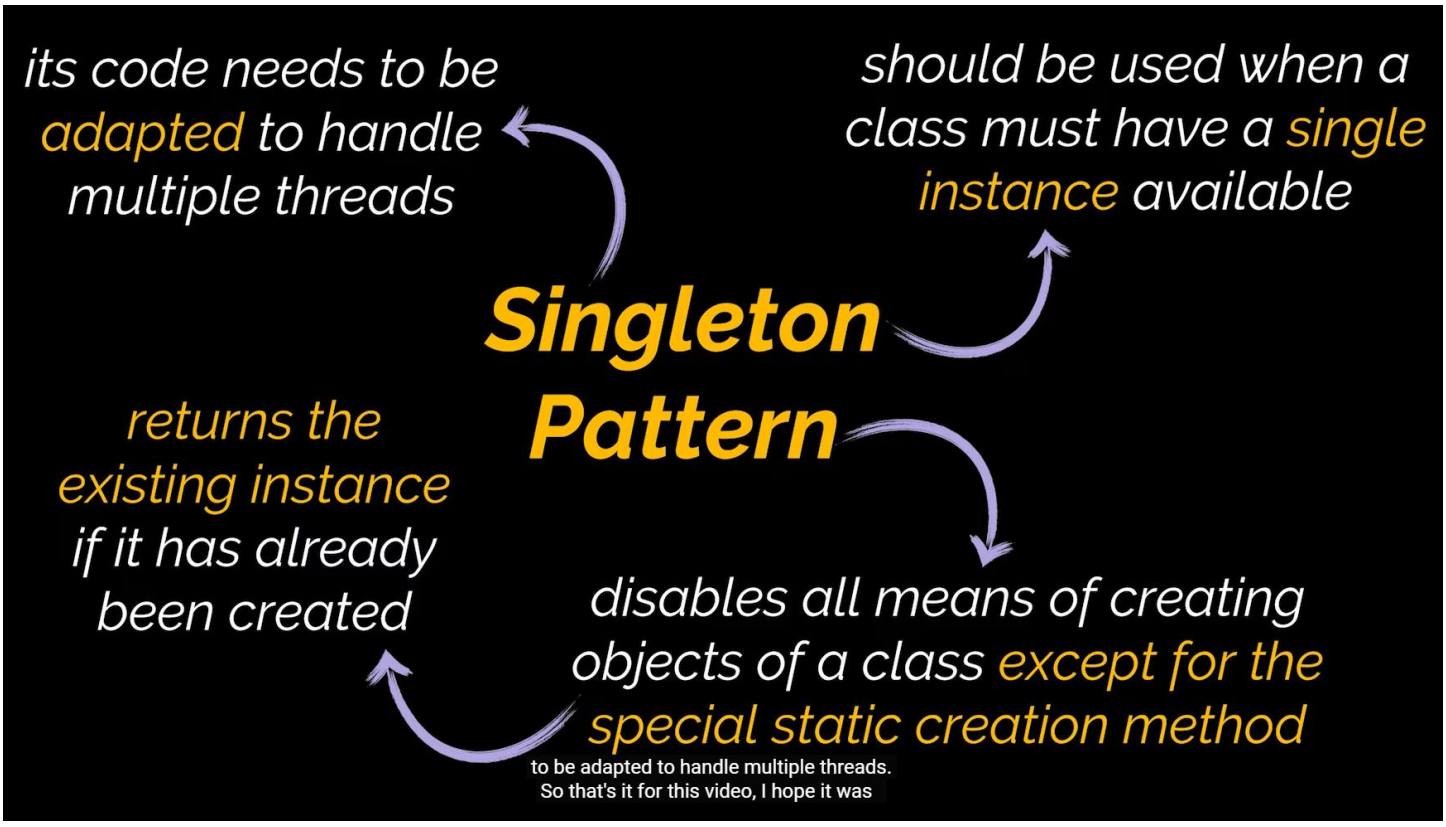
```java
public class Singleton {

    private static Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    public static Singleton getInstance(String data) {
        if (instance == null) {
            instance = new Singleton(data);
        }
        return instance;
    }

}
```

*nothing in this code prevents **two threads** from accessing this piece of code **at the same time***

```java
public class Singleton {

    private static volatile Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    public static Singleton getInstance(String data) {
        Singleton result = instance;
        if (result == null) {
            synchronized (Singleton.class) {
                result = instance;
                if (result == null) {
                    instance = result = new Singleton(data);
                }
            }
        }
        return result;
    }

}
```

*the usage of such a local variable can **improve the method overall performance** by as much as 40%!*

to avoid multiple direct reads of the memory.



TEMPLATE METHOD DESIGN PATTERN

1. Break down the algorithm into a series of methods
2. Put a series of calls to these methods or steps inside a single “template method”
3. The steps may either be abstract, or have some default implementation inside the parent class
4. To use the algorithm, the client must provide its own subclass and implement all abstract steps

```java
public abstract class BaseGameLoader {
    public void load() {
        byte[] data = loadLocalData();
        createObjects(data);
        downloadAdditionalFiles();
        cleanTempFiles();
        initializeProfiles();
    }

    abstract byte[] loadLocalData();
    abstract void createObjects(byte[] data);
    abstract void downloadAdditionalFiles();
    abstract void initializeProfiles();

    protected void cleanTempFiles() {
        System.out.println("Cleaning temporary files...");
        // Some Code...
    }
}
```

this will allow us to get rid of code duplication

```java
public class WorldOfWarcraftLoader extends BaseGameLoader {
    @Override
    byte[] loadLocalData() {
        System.out.println("Loading local WoW files...");
        // Some Warcraft Code...
    }

    @Override
    void createObjects(byte[] data) {
        System.out.println("Creating WoW objects...");
        // Some Warcraft Code...
    }

    @Override
    void downloadAdditionalFiles() {
        System.out.println("Downloading WoW sounds...");
        // Some Warcraft Code...
    }

    @Override
    void initializeProfiles() {
        System.out.println("Loading WoW profiles...");
        // Some Warcraft Code...
    }
}
```

```java
public class DiabloLoader extends BaseGameLoader {
    @Override
    byte[] loadLocalData() {
        System.out.println("Loading Diablo files...");
        // Some Diablo Code...
    }

    @Override
    void createObjects(byte[] data) {
        System.out.println("Creating Diablo objects...");
        // Some Diablo Code...
    }

    @Override
    void downloadAdditionalFiles() {
        System.out.println("Downloading Diablo sounds...");
        // Some Diablo Code...
    }

    @Override
    void initializeProfiles() {
        System.out.println("Loading Diablo profiles...");
        // Some Diablo Code...
    }
}
```

by doing this, each step will contain the actual behavior needed by this specific game

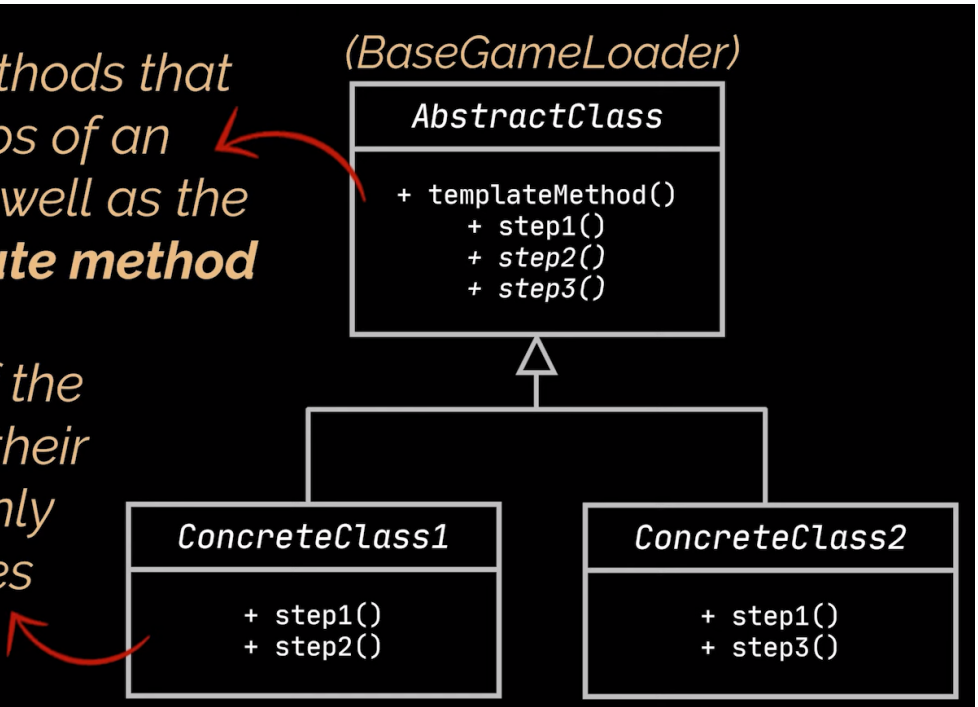
TEMPLATE METHOD DESIGN PATTERN

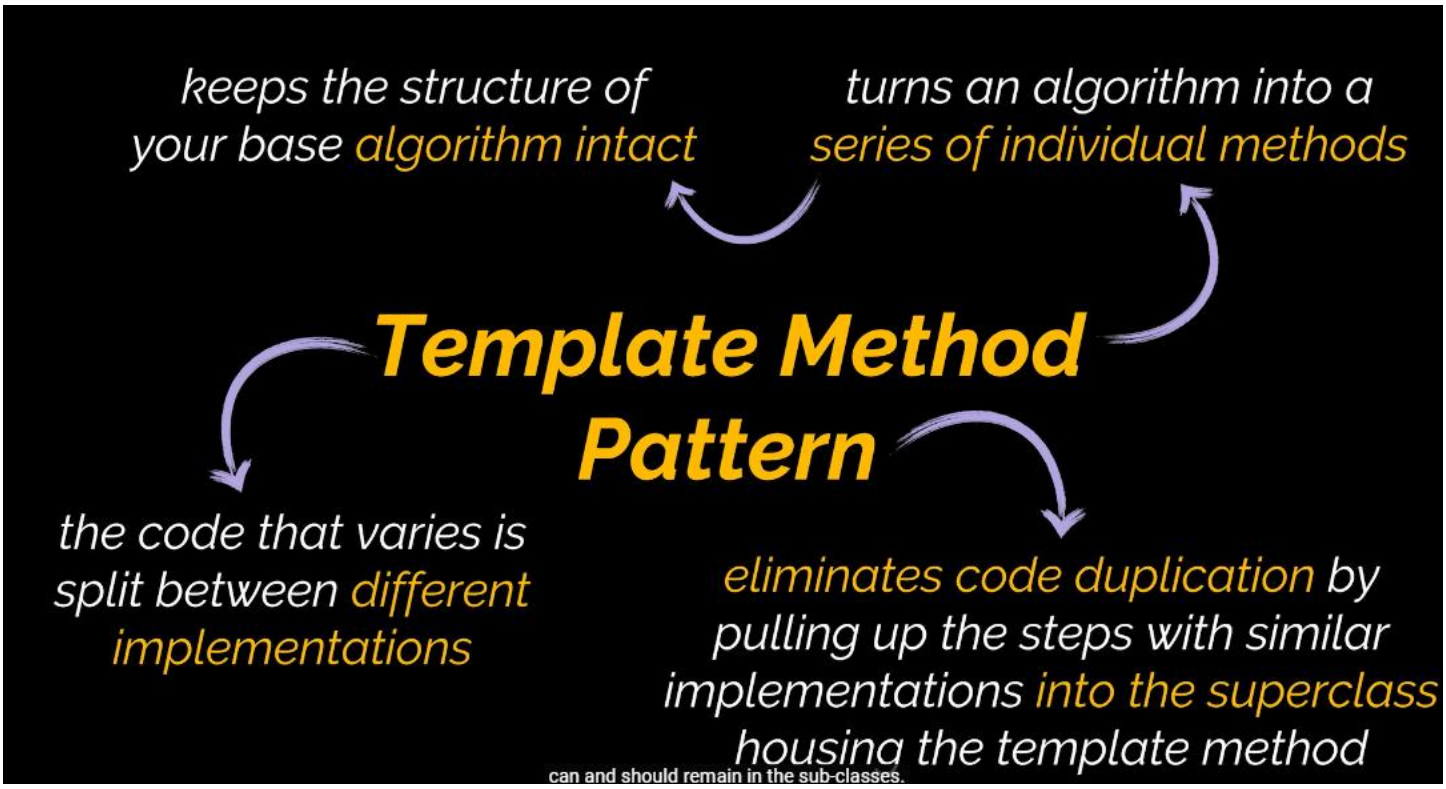
*defines the **skeleton** of an algorithm in the **superclass** but let's subclasses **override specific steps** of the algorithm without changing its structure*

***behavioral** design pattern*

declares** methods that act as steps of an algorithm, as well as the actual **template method

*can **override** all of the steps provided by their parent class or only the **abstract** ones*



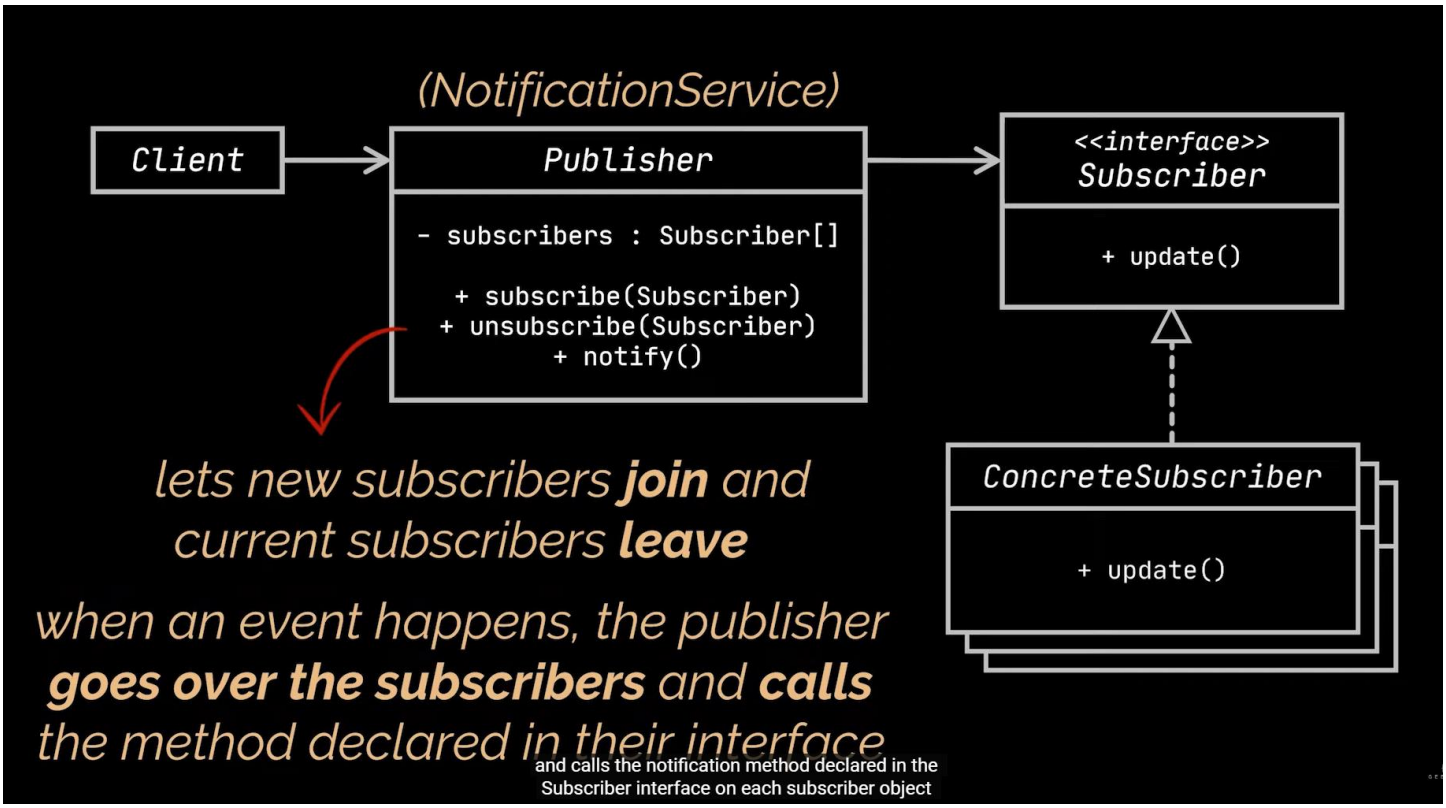

keeps the structure of your base algorithm intact
turns an algorithm into a series of individual methods
Template Method Pattern
the code that varies is split between different implementations
eliminates code duplication by pulling up the steps with similar implementations into the superclass housing the template method
can and should remain in the sub-classes.

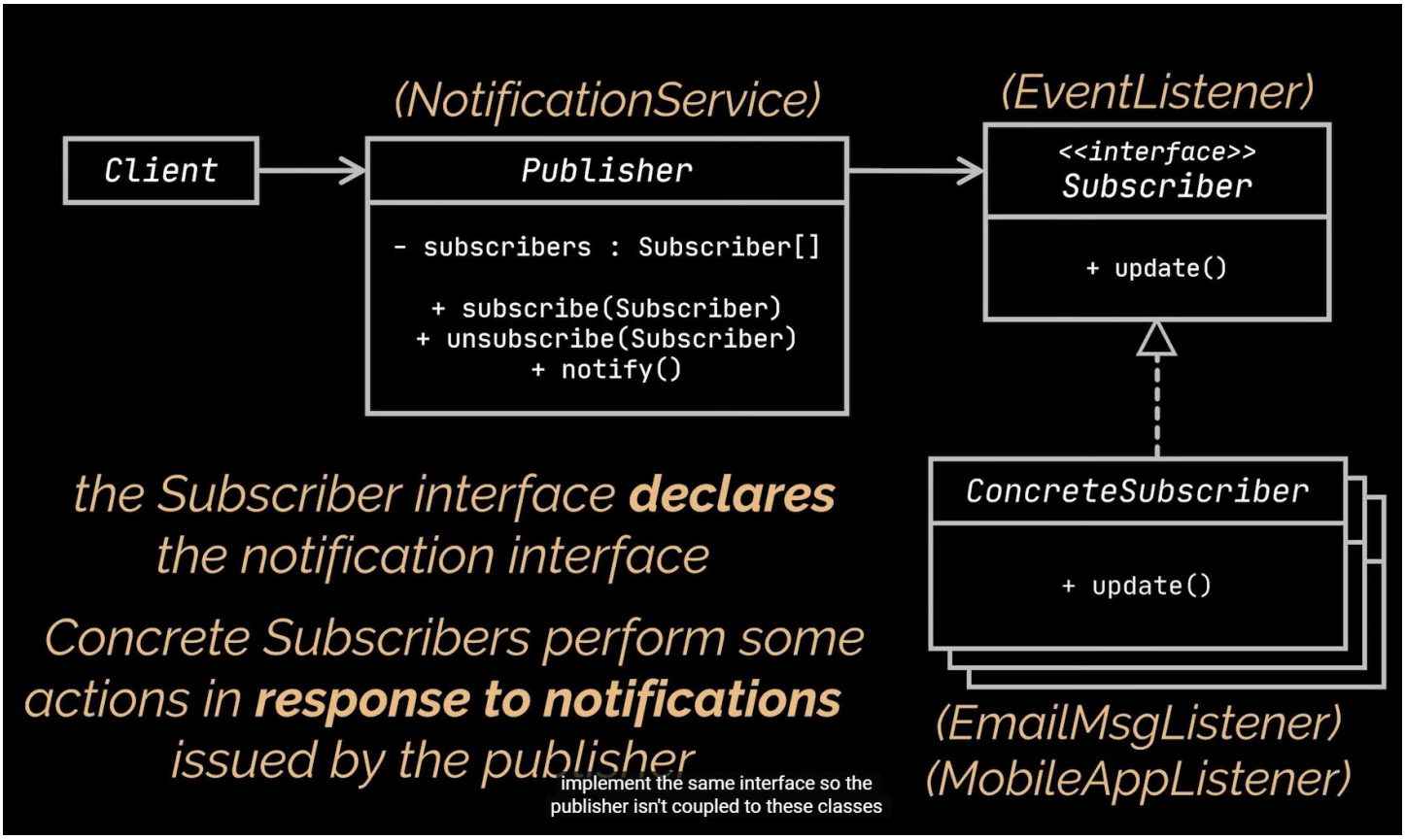

OBSERVER DESIGN PATTERN
notifies multiple objects, or subscribers, about any events that happen to the object they're observing, or publisher
behavioral design pattern

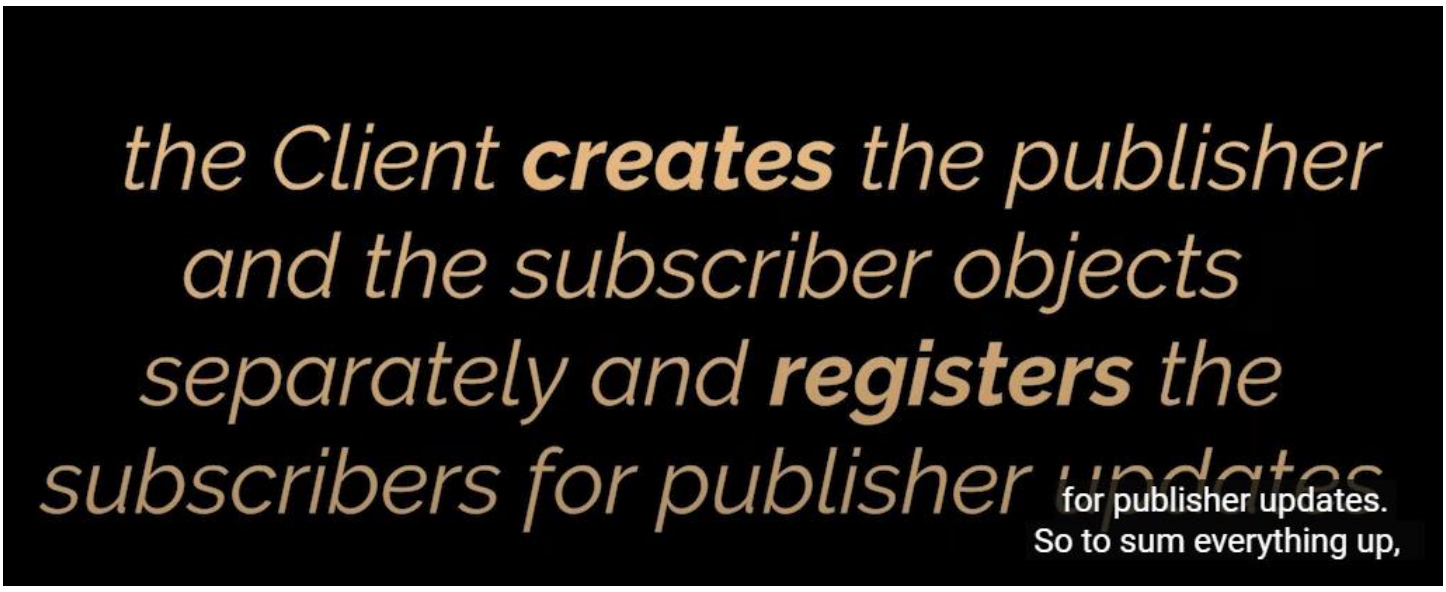
```java
public class Store {
    private final NotificationService notificationService;

    public Store() {
        notificationService = new NotificationService();
    }

    public void newItemPromotion() {
        notificationService.notify();
    }

    public NotificationService getService() {
        return notificationService;
    }
}
```

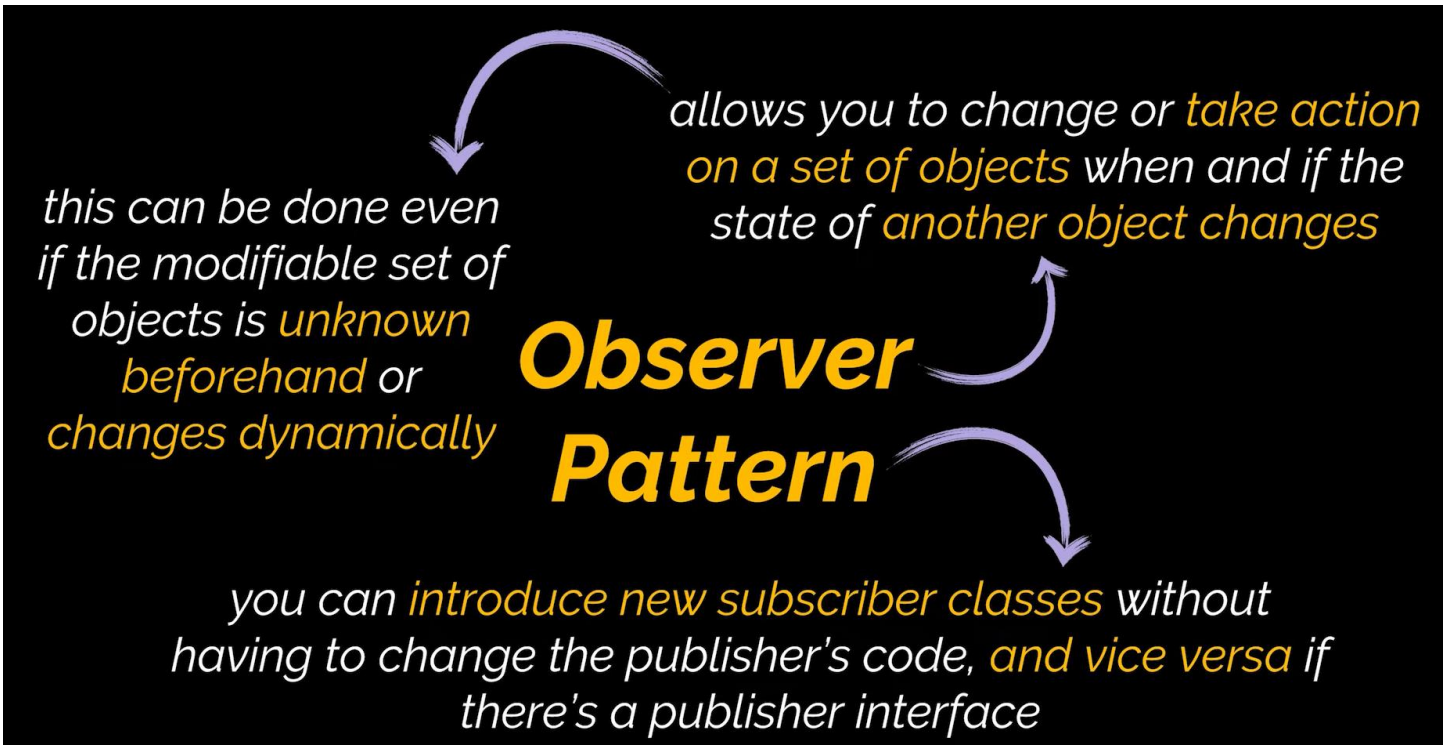
```java
public class NotificationService {
    private final List<EmailMsgListener> customers;

    public NotificationService() {
        customers = new ArrayList<>();
    }

    public void subscribe(EmailMsgListener listener) {
        customers.add(listener);
    }

    public void unsubscribe(EmailMsgListener listener) {
        customers.remove(listener);
    }

    public void notify() {
        customers.forEach(listener -> listener.update());
    }
```

```java
public class EmailMsgListener {
    private final String email;

    public EmailMsgListener(String email) {
        this.email = email;
    }

    public void update() {
        // Actually send the mail
    }
}
```

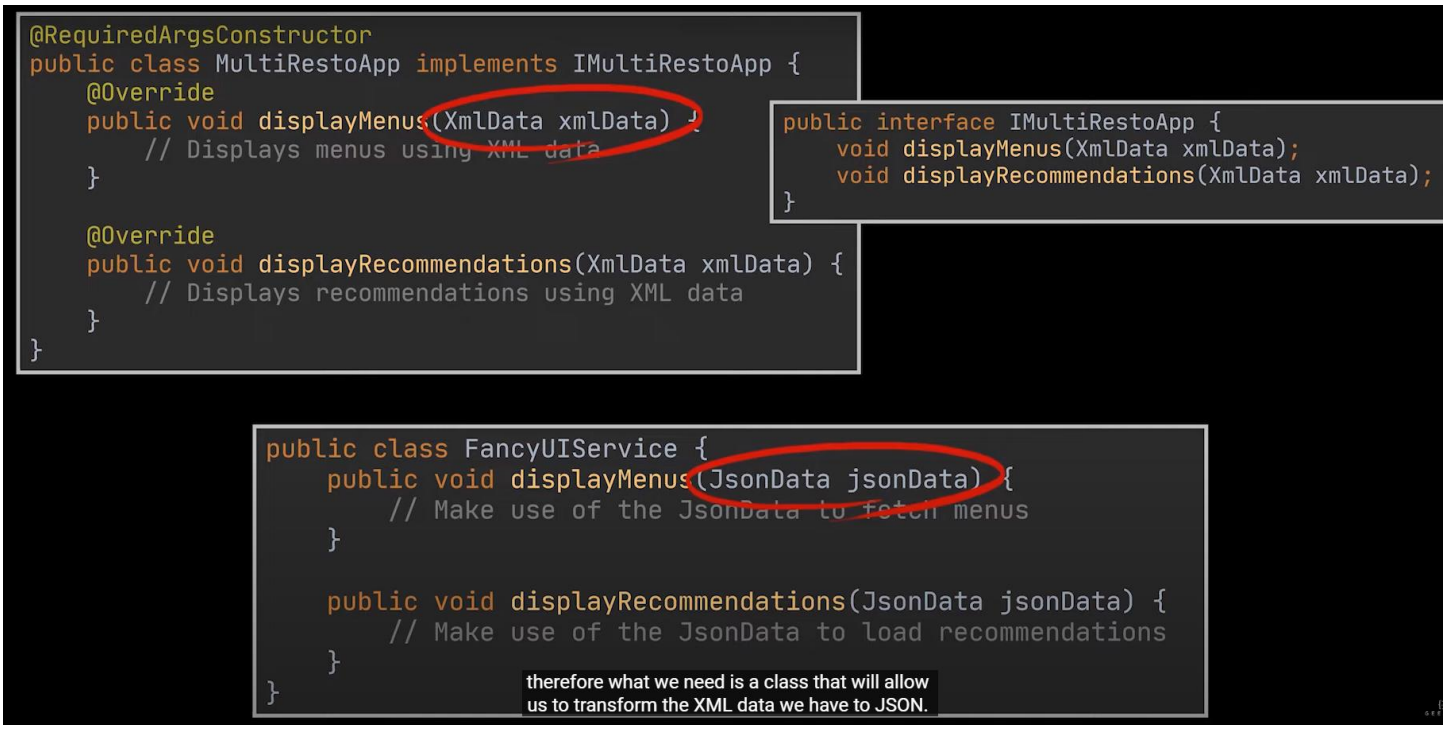



(NotificationService)
(EventListener)
Client
Publisher
- subscribers : Subscriber[]
+ subscribe(Subscriber)
+ unsubscribe(Subscriber)
+ notify()
<<interface>>
Subscriber
+ update()
ConcreteSubscriber
+ update()
the Subscriber interface declares the notification interface
Concrete Subscribers perform some actions in response to notifications issued by the publisher
implement the same interface so the publisher isn't coupled to these classes
(EmailMsgListener)
(MobileAppListener)


the Client creates the publisher and the subscriber objects separately and registers the subscribers for publisher updates.
for publisher updates.
So to sum everything up,


allows you to change or take action on a set of objects when and if the state of another object changes
this can be done even if the modifiable set of objects is unknown beforehand or changes dynamically
Observer Pattern
you can introduce new subscriber classes without having to change the publisher's code, and vice versa if there's a publisher interface


ADAPTER
DESIGN PATTERN
allows objects with incompatible interfaces to collaborate with one another
structural design pattern


@RequiredArgsConstructor
public class MultiRestoApp implements IMultiRestoApp {
@Override
public void displayMenus(XmlData xmlData) {
// Displays menus using XML data
}
@Override
public void displayRecommendations(XmlData xmlData) {
// Displays recommendations using XML data
}
}
public interface IMultiRestoApp {
void displayMenus(XmlData xmlData);
void displayRecommendations(XmlData xmlData);
}
public class FancyUIService {
public void displayMenus(JsonData jsonData) {
// Make use of the JsonData to fetch menus
}
public void displayRecommendations(JsonData jsonData) {
// Make use of the JsonData to load recommendations
}
}
therefore what we need is a class that will allow
us to transform the XML data we have to JSON.

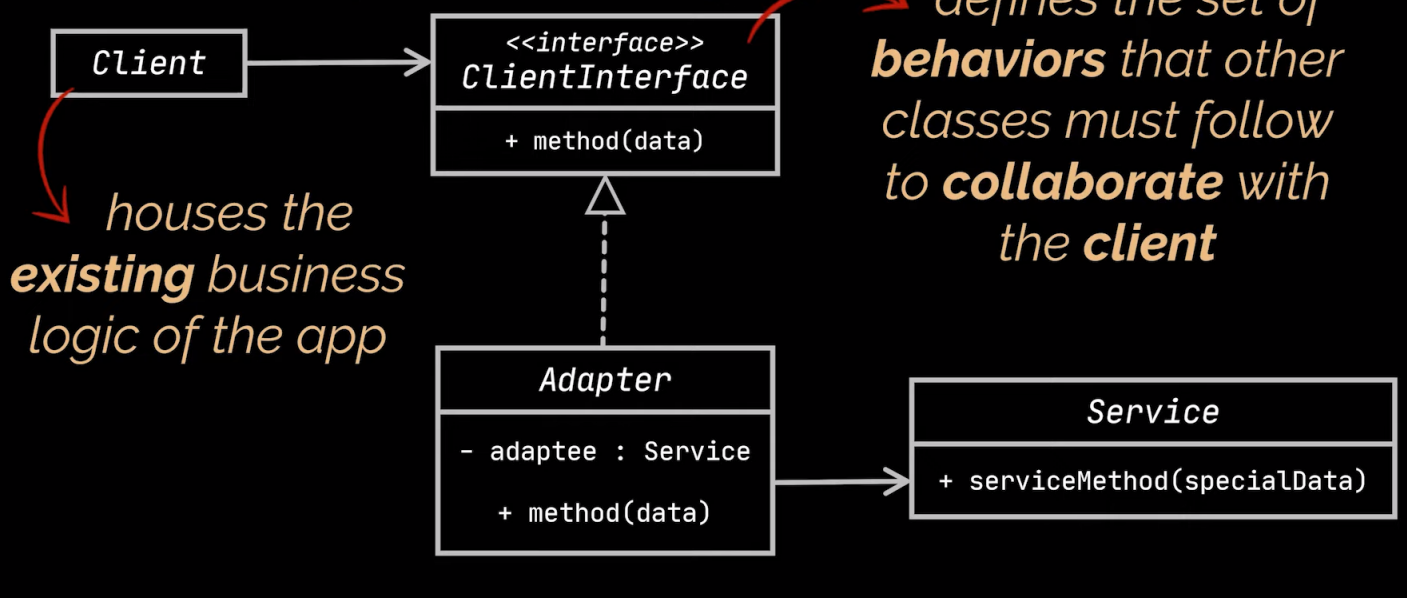

public class FancyUIServiceAdapter implements IMultiRestoApp {
private final FancyUIService fancyUIService;
public FancyUIServiceAdapter() {
fancyUIService = new FancyUIService();
}
@Override
public void displayMenus(XmlData xmlData) {
JsonData jsonData = convertXmlToJson(xmlData);
fancyUIService.displayMenus(jsonData);
}
@Override
public void displayRecommendations(XmlData xmlData) {
JsonData jsonData = convertXmlToJson(xmlData);
fancyUIService.displayRecommendations(jsonData);
}
private JsonData convertXmlToJson(XmlData xmlData) {
// Convert XmlData to JsonData and return it
}
}
us to call its respective methods.
Now, when you want to make

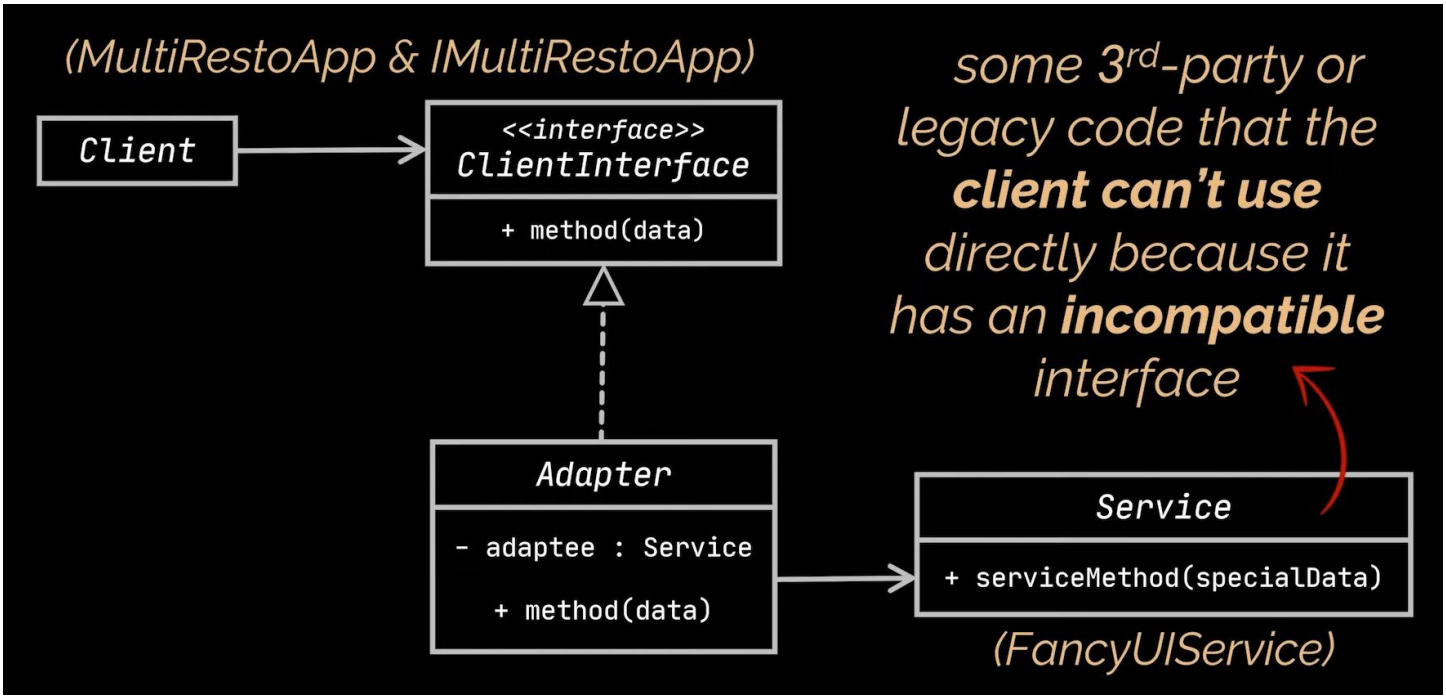
Old UI

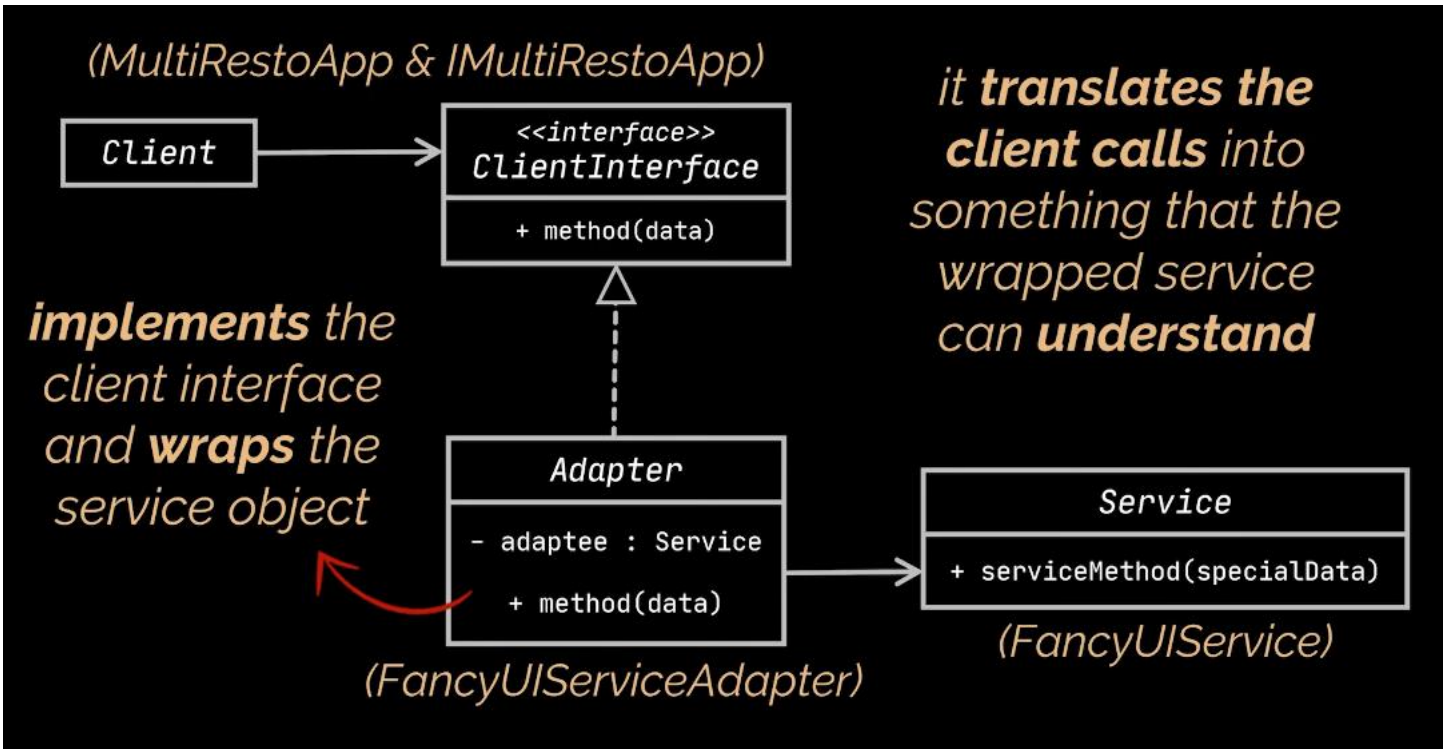
```
public static void main(String[] args) {
    IMultiRestoApp multiRestoApp = new MultiRestoApp();
    multiRestoApp.displayMenus(new XmlData());

    FancyUIServiceAdapter adapter = new FancyUIServiceAdapter();
    adapter.displayMenus(new XmlData());
}
```

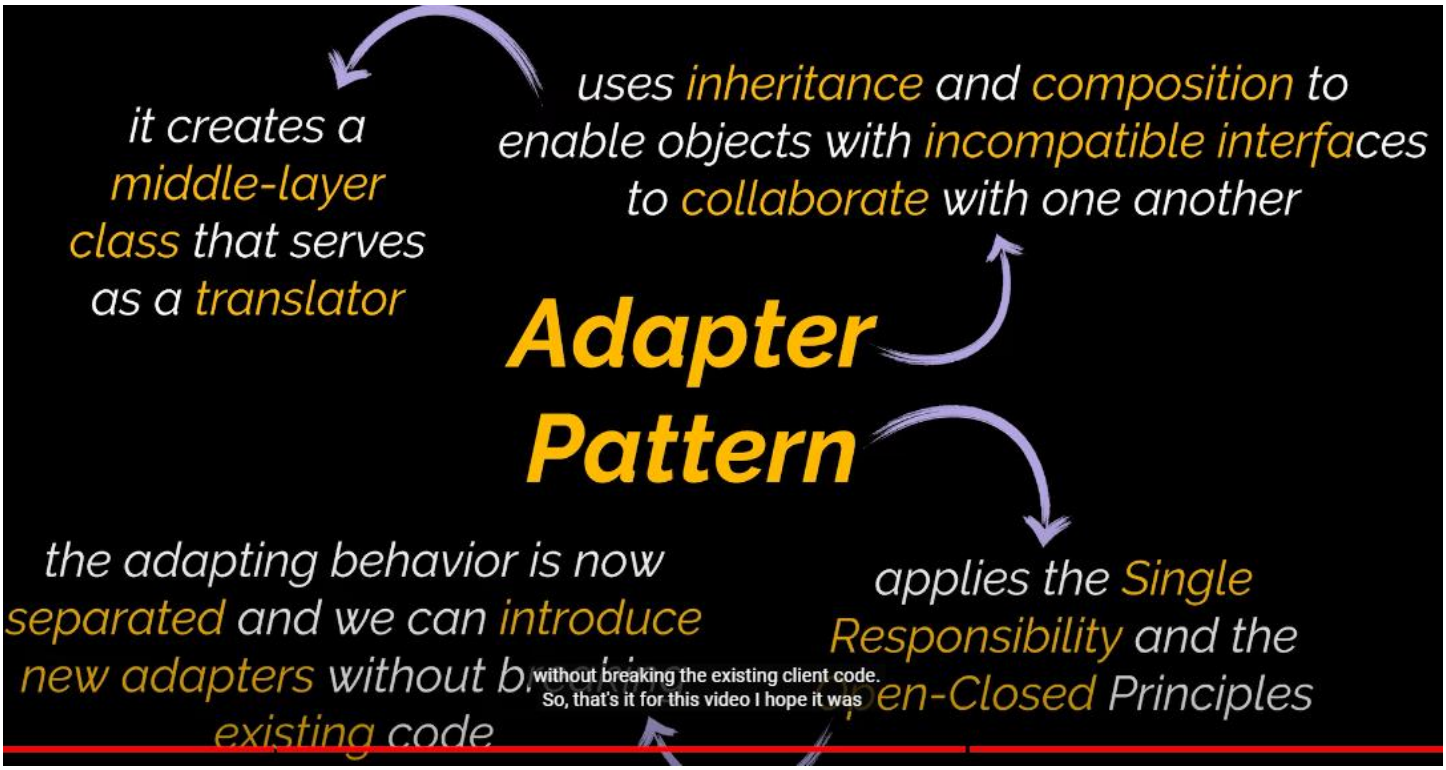
New UI

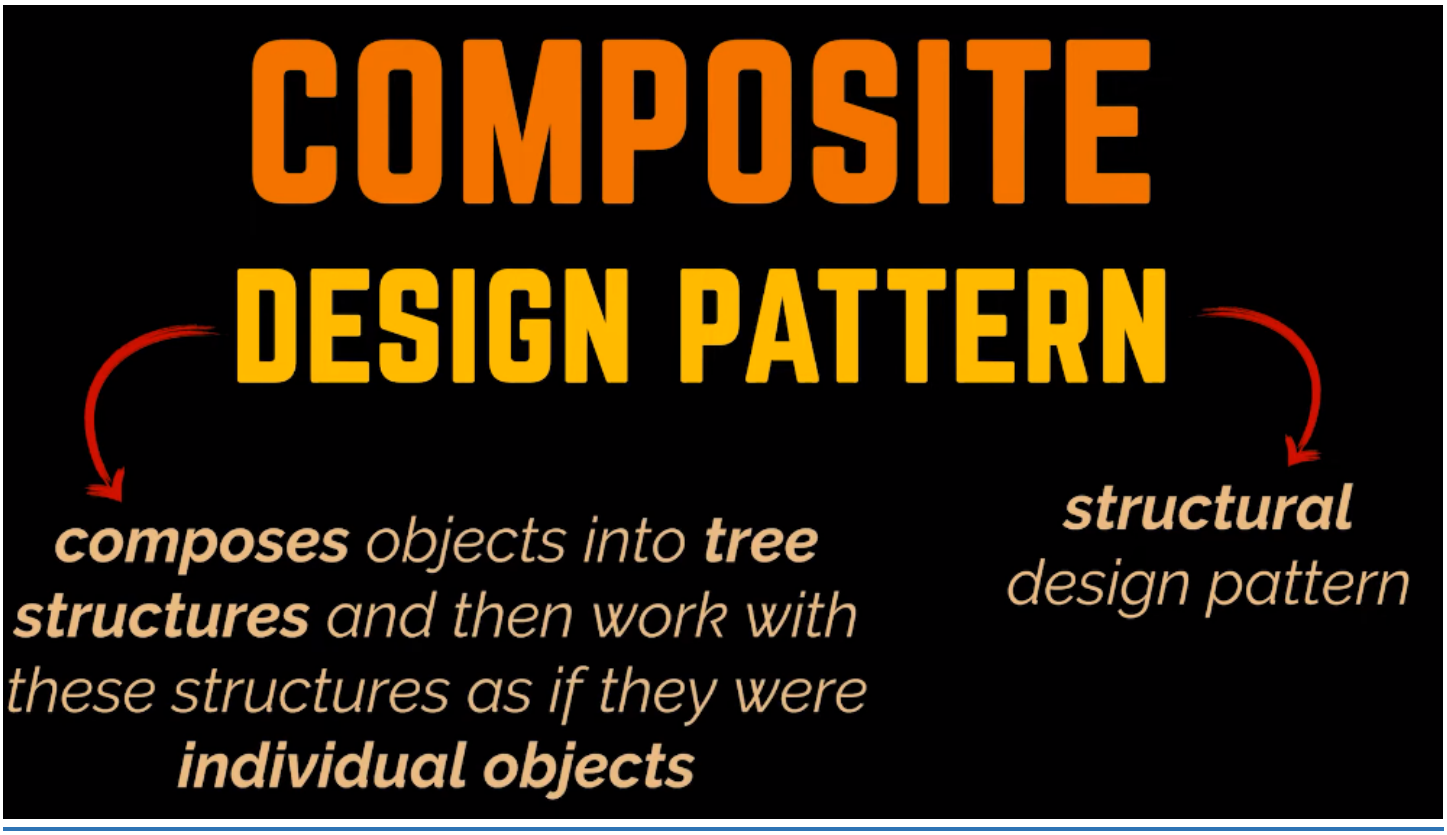
(MultiRestoApp & IMultiRestoApp)




(MultiRestoApp & IMultiRestoApp)
Client
<<interface>>
ClientInterface
+ method(data)
some 3rd-party or legacy code that the client can't use directly because it has an incompatible interface
Adapter
- adaptee : Service
+ method(data)
Service
+ serviceMethod(specialData)
(FancyUIService)


(MultiRestoApp & IMultiRestoApp)
Client
<<interface>>
ClientInterface
+ method(data)
it translates the client calls into something that the wrapped service can understand
implements the client interface and wraps the service object
Adapter
- adaptee : Service
+ method(data)
Service
+ serviceMethod(specialData)
(FancyUIService)
(FancyUIServiceAdapter)


it creates a middle-layer class that serves as a translator
uses inheritance and composition to enable objects with incompatible interfaces to collaborate with one another
Adapter Pattern
the adapting behavior is now separated and we can introduce new adapters without b
existing code
applies the Single Responsibility and the
en-Closed Principles
without breaking the existing client code.
So, that's it for this video I hope it was


COMPOSITE
DESIGN PATTERN
composes objects into tree structures and then work with these structures as if they were individual objects
structural design pattern

```java
public class CompositeBox implements Box {
    private final List<Box> children = new ArrayList<>();

    public CompositeBox(Box... boxes) {
        children.addAll(Arrays.asList(boxes));
    }

    @Override
    public double calculatePrice() {
        return children.stream()
                .mapToDouble(Box::calculatePrice)
                .sum();
    }
}
```

```java
public interface Box {
    double calculatePrice();
}
```

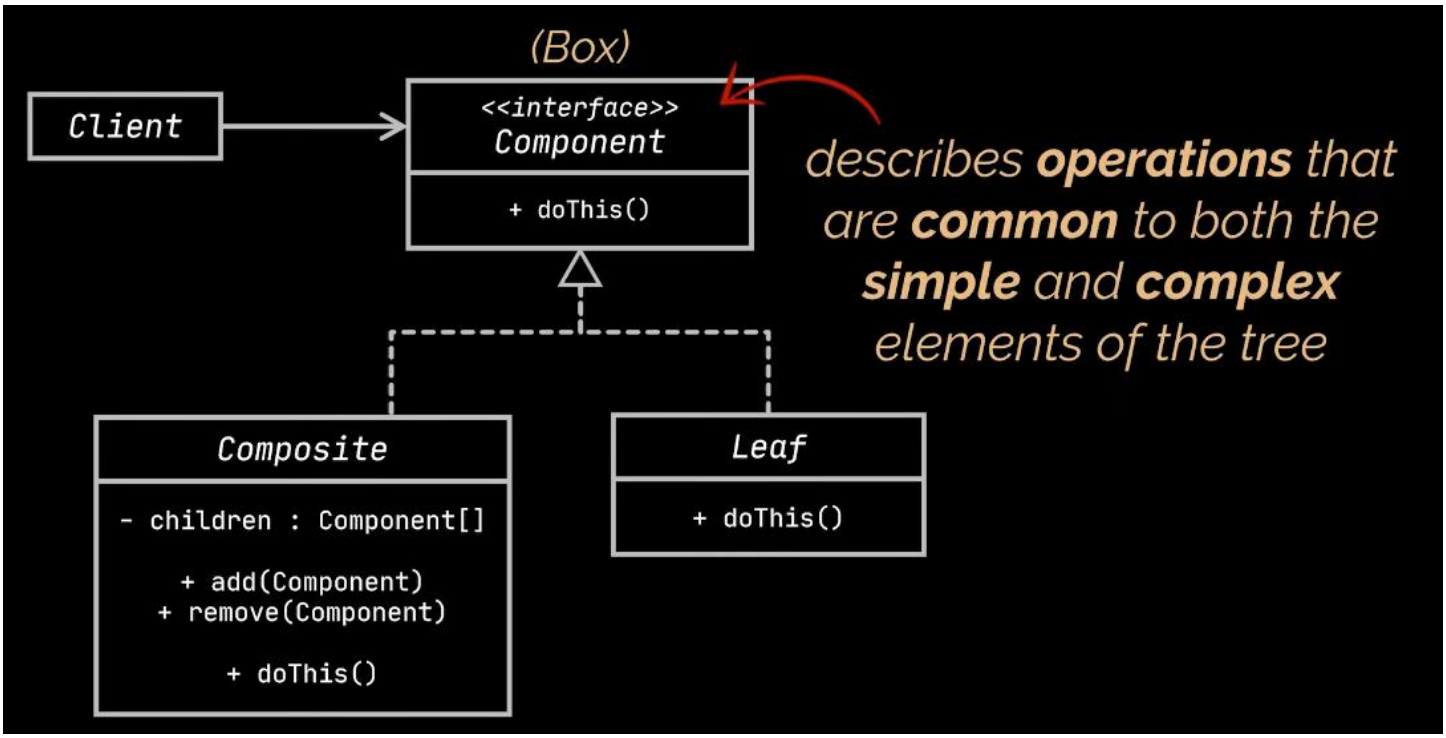
```java
@Data
public abstract class Product implements Box {
    protected final String title;
    protected final double price;
}
```

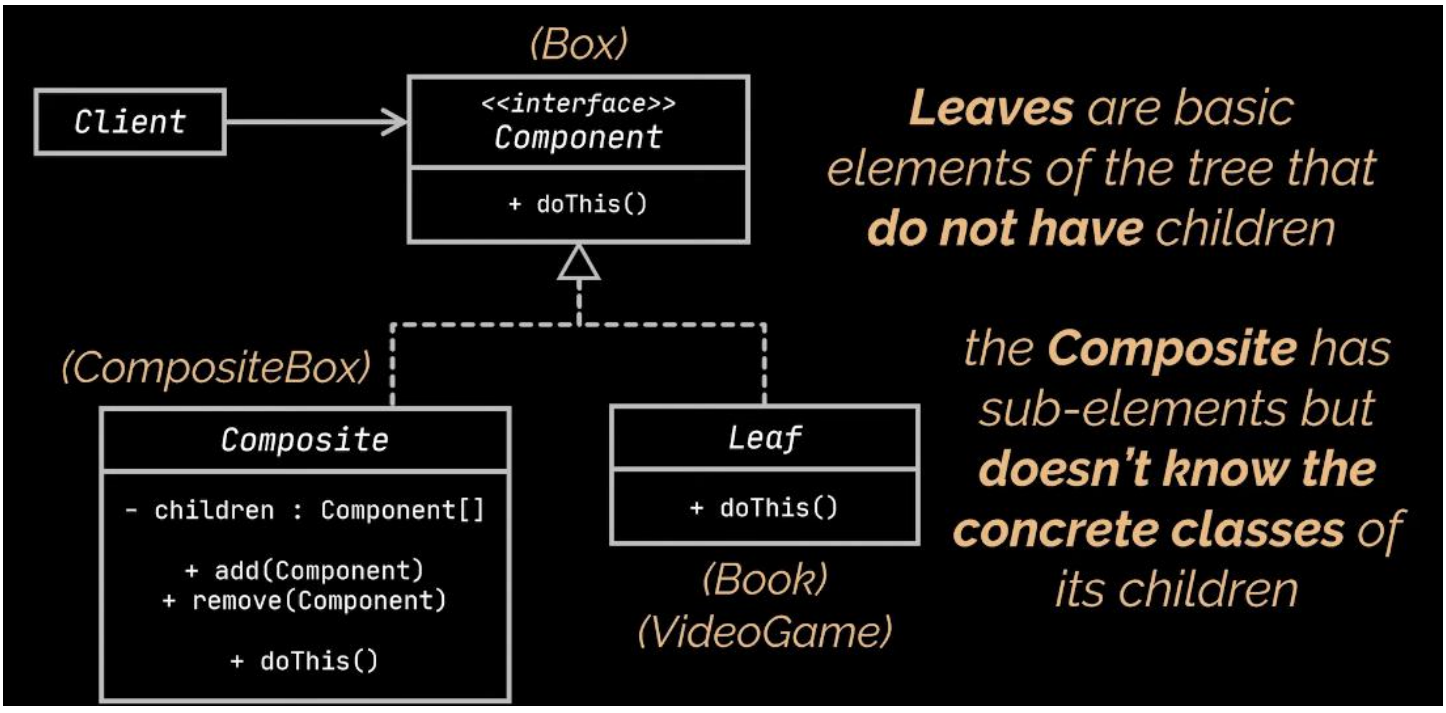
```java
public class Book extends Product {
    public Book(String title, double price) {
        super(title, price);
    }

    @Override
    public double calculatePrice() {
        return getPrice();
    }
}
```

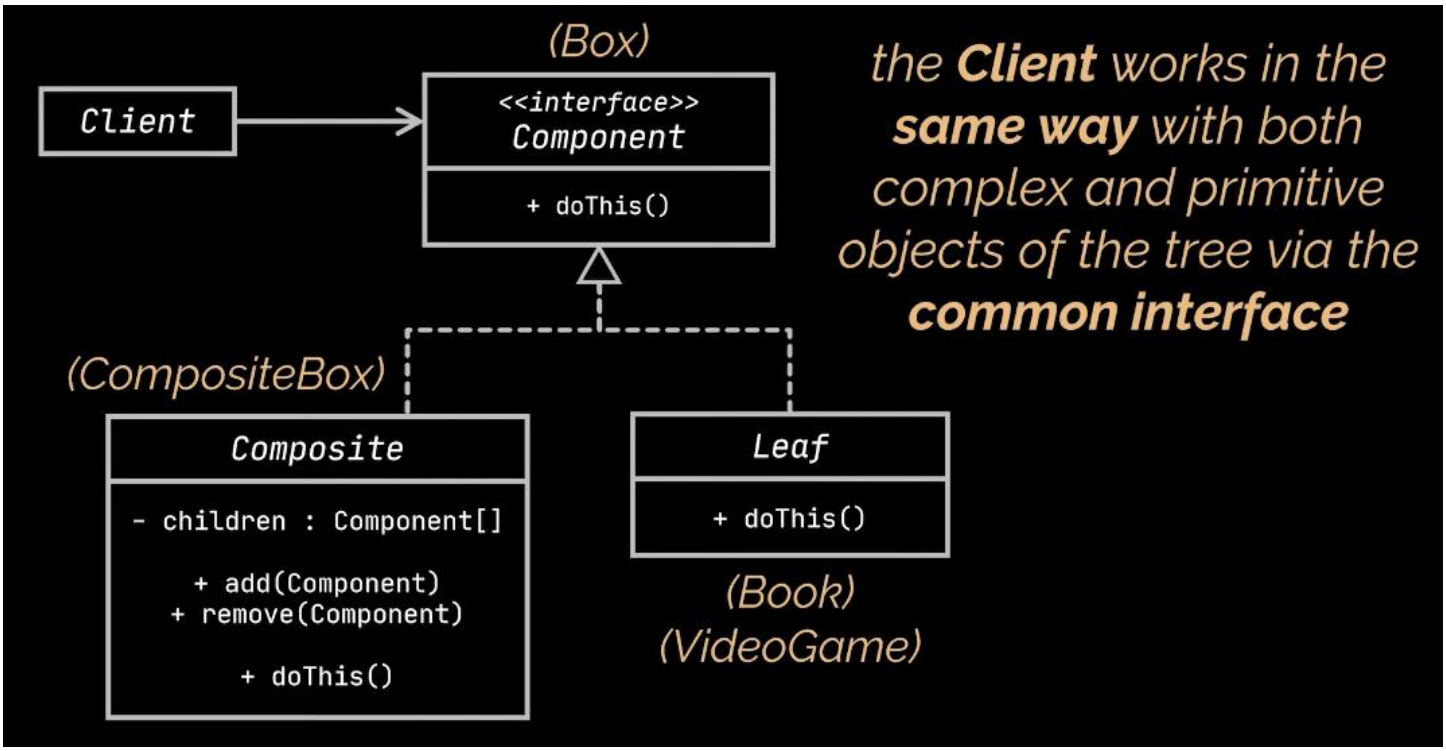
```java
public class VideoGame extends Product {
    public VideoGame(String title, double price) {
        super(title, price);
    }

    @Override
    public double calculatePrice() {
        return getPrice();
    }
}
```

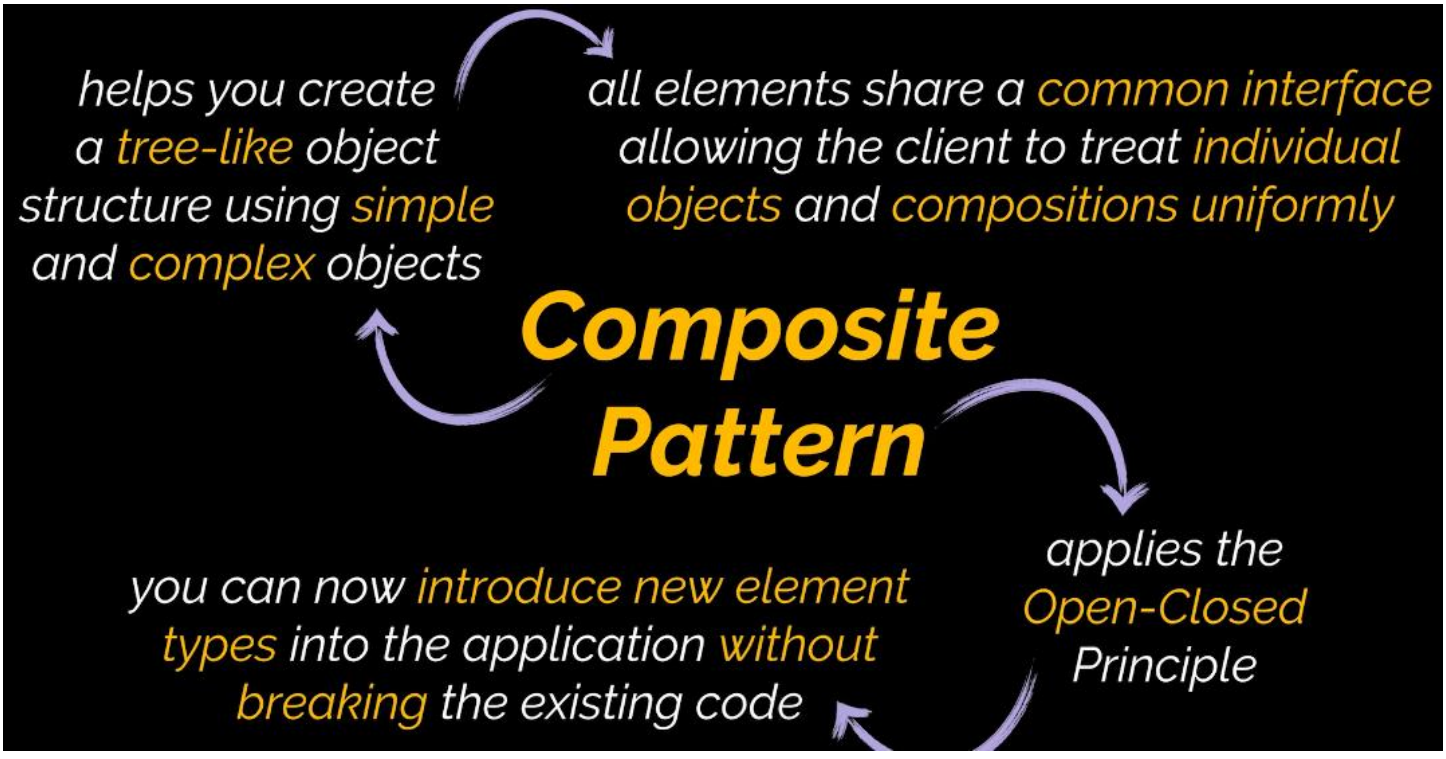
```java
public static void main(String[] args) {
    DeliveryService deliveryService = new DeliveryService();

    deliveryService.setupOrder(
            new CompositeBox(
                    new VideoGame("1", 100)
            ),
            new CompositeBox(
                    new CompositeBox(
                            new Book("2", 200),
                            new Book("3", 300)
                    ),
                    new VideoGame("4", 400),
                    new VideoGame("5", 500)
            )
    );

    deliveryService.calculateOrderPrice(); // 1500
}
```

```java
public class DeliveryService {
    private Box box;

    public DeliveryService() {
    }

    public void setupOrder(Box... boxes) {
        this.box = new CompositeBox(boxes);
    }

    public double calculateOrderPrice() {
        return box.calculatePrice();
    }
}
```


(Box)
Client
<<interface>>
Component
+ doThis()
describes operations that are common to both the simple and complex elements of the tree
Composite
- children : Component[]
+ add(Component)
+ remove(Component)
+ doThis()
Leaf
+ doThis()


(Box)
Client
<<interface>>
Component
+ doThis()
Leaves are basic elements of the tree that do not have children
(CompositeBox)
Composite
- children : Component[]
+ add(Component)
+ remove(Component)
+ doThis()
Leaf
+ doThis()
(Book)
(VideoGame)
the Composite has sub-elements but doesn't know the concrete classes of its children


(Box)
Client
<<interface>>
Component
+ doThis()
the Client works in the same way with both complex and primitive objects of the tree via the common interface
(CompositeBox)
Composite
- children : Component[]
+ add(Component)
+ remove(Component)
+ doThis()
Leaf
+ doThis()
(Book)
(VideoGame)


helps you create a tree-like object structure using simple and complex objects
all elements share a common interface allowing the client to treat individual objects and compositions uniformly
Composite Pattern
applies the Open-Closed Principle
you can now introduce new element types into the application without breaking the existing code