Decorator Pattern

The **Decorator Pattern** is a structural design pattern that allows behavior to be added to an individual object, dynamically, **without affecting the behavior of other objects from the same class**.

It's useful when you want to **add features to objects without modifying their code**—especially when inheritance is not feasible or would lead to a large number of subclasses.

**🛒 Retail Example: Building a Product with Add-ons**

Let’s say you are working on a **retail e-commerce system**, and you are designing a **product customization** feature for a T-shirt

**✅ Basic Requirement**

A user can buy a **T-shirt**. They can also add **customizations**, like:

* Gift Wrap
* Express Delivery
* Discount Coupon

You don’t want to create new subclasses for every combination (like TShirtWithGiftWrapAndExpressDeliveryAndDiscountCoupon). Instead, you can use decorators!

**✅ Advantages of Decorator Pattern in Retail**

* You can **add features dynamically** without modifying original product classes.
* Promotes **composition over inheritance**.
* Makes code **more flexible and readable**.

In The **Decorator Pattern**, the **base component (like** TShirt**) is mandatory**, because decorators are designed to **wrap around a core object** to enhance its behavior. They **cannot exist independently** — they extend or modify the behavior of an existing component.

**💡 Think of it like this:**

A **decorator** is not a full product on its own. It’s like an **addon** — it *needs* something to decorate.

So in our retail example:

* GiftWrap, ExpressDelivery, and DiscountCoupon **require a Product to decorate.**
* If there's no TShirt or other Product, then decorating doesn’t make sense.