Decorator Design Pattern Explained

In the **Decorator Design Pattern**, we separate extra features/behaviours into their **own decorator classes**.

Core Concept

- You have a **base interface or abstract class** (e.g., INotification).
- You have a **concrete implementation** (e.g., EmailNotification).
- Each **additional feature** is put into a **separate decorator class** (e.g., SlackNotificationDecorator, SMSNotificationDecorator, etc.).
- Decorators **wrap** objects of the same interface and **add behavior on top**.

Why separate features?

To achieve **single responsibility** and **composable behaviour**.

Each decorator class:

- Adds only ONE feature
- Wraps another object
- Doesn't modify the original class
- Can be stacked in any order

Example Structure:

```
public interface INotification
{
   void Notify(string message);
}

// Core/Base behaviour
public class EmailNotification : INotification
```

```
{
    public void Notify(string message)
    {
        Console.WriteLine("Email: " + message);
    }
}
//Decorators
public abstract class NotificationDecorator : INotification
{
    protected INotification _notification;

    public NotificationDecorator(INotification notification)
    {
        _notification = notification;
    }

    public virtual void Notify(string message)
    {
        _notification.Notify(message);
    }
}

//decorator 1
public class SMSNotification : NotificationDecorator
{
    public SMSNotification(INotification notification) : base(notification) { }
```

```
    public override void Notify(string message)
    {
        base.Notify(message);
        Console.WriteLine("SMS: " + message);
    }
}

//decorator 2
public class SlackNotification : NotificationDecorator
{
    public SlackNotification(INotification notification) : base(notification) { }

    public override void Notify(string message)
    {
        base.Notify(message);
        Console.WriteLine("Slack: " + message);
    }
}

//stacking decorator
INotification notification =
    new SlackNotification(
        new SMSNotification(
            new EmailNotification()
        )
```

);

notification.Notify("Hello!");

Summary:

Concept	**In Decorator Pattern**
Are features separated?	**Yes** — each new feature goes in its own class
Do decorators modify the real object?	No, they *wrap* it
Can features be combined?	Yes — decorators are stackable
Why do this?	Flexibility, Open/Closed Principle, reusable behaviors

When to Prefer Decorator Over Inheritance

Here are the **clear, real-world reasons**:

✓ 1. When you need dynamic behavior changes

Decorators allow you to add/remove features at runtime.

Example:

var notification = new SMSNotification(new EmailNotification());

// Later...

notification = new SlackNotification(notification);

Inheritance cannot change behavior at runtime.

✓ 2. When you need combinable features

Decorators allow stacking:

- Email + SMS
- Email + Slack
- Email + SMS + Slack
- Email only
- SMS only

Inheritance would require many combinations:

EmailAndSMSNotification

EmailSMSAndSlackNotification

EmailSlackNotification

...

Explosion of subclasses = ❌

✓ 3. When you must follow Single Responsibility Principle

Each decorator does ONE small thing (e.g., add SMS).

Inheritance tends to grow into a "God class" with too many features.

✓ 4. When you need the Open/Closed Principle

You can add new decorators **without modifying existing code**.

Inheritance often forces rewriting base classes or adding more subclasses.

✓ 5. When subclassing is not possible

Sometimes the class is **sealed**, or you don’t own the source code.

Decorator works even when the class cannot be extended.

✓ 6. When you want to avoid deep inheritance trees

Instead of:

BaseNotification

|- EmailNotification

|- EmailPlusSMSNotification

|- EmailSMSPlusSlackNotification

You just keep wrapping.

❌ When NOT to use Decorator

- When the feature **must be mandatory**
- When too many decorators make debugging difficult
- When the creation of object graph becomes too complex
- When order of decorators matters (and it becomes messy)

Negative Scenario (WITHOUT Decorator Pattern)

what the same system *would look like if we did NOT use the Decorator pattern*.

We need optional features:

- Compression
- Encryption
- Caching

Assume we try to solve this **using only inheritance** or building many variants manually.

Approach 1: Giant Conditional Class (God Class)

You create one class that tries to handle everything:

```
public class FileDataStream
{
    public bool EnableCompression { get; set; }
    public bool EnableEncryption { get; set; }
    public bool EnableCaching { get; set; }

    public void Write(string data)
    {
        if (EnableCaching)
        {
            Cache(data);
        }

        if (EnableCompression)
        {
            data = Compress(data);
        }

        if (EnableEncryption)
        {
            data = Encrypt(data);
        }
```

```
        Console.WriteLine("Writing to file: " + data);
    }

    private void Cache(string data) { ... }
    private string Compress(string data) { ... }
    private string Encrypt(string data) { ... }
}
```

Problems with this approach

1. Violates Single Responsibility Principle

This class handles:

- File writing
- Caching
- Compression
- Encryption

All mixed into one place 💥

---

2. Violates Open/Closed Principle

If you want to add:

- Chunking
- Signing
- Logging
- Retry policy

You must MODIFY this class again and again.

Approach 2: Inheritance Explosion (the worst scenario)

Trying to use inheritance instead of decorators:

FileDataStream

|- CompressedFileDataStream

|- EncryptedFileDataStream

|- CachedFileDataStream

|- CompressedEncryptedFileDataStream

|- CachedEncryptedFileDataStream

|- CachedCompressedFileDataStream

|- CachedCompressedEncryptedFileDataStream

This is just **8 classes** for 3 features.

Add 1 more feature = 16 classes
Add another = 32 classes

➡️ **CLASS EXPLOSION**
➡️ Impossible to maintain
➡️ Impossible to test
➡️ Order-of-operations issues

```
// what one class looks like
public class CompressedEncryptedFileDataStream : FileDataStream
{
    public override void Write(string data)
    {
        string compressed = Compress(data);
        string encrypted = Encrypt(compressed);
```

```
        base.Write(encrypted);
    }
}
```

Now imagine writing **7 more classes** with similar code.
And updating them every time you add a new feature.

This is **code duplication hell**.