1/1/2022

Ateet Agarwal

ateet1989@gmail.com

Design Patterns

Explanation with code examples

Table of Contents

[Creational Patterns 3](#_Toc92977682)

[Factory Method 4](#_Toc92977683)

[Problem 4](#_Toc92977684)

[Existing Project Example 4](#_Toc92977685)

[Applicability 4](#_Toc92977686)

[Pros 4](#_Toc92977687)

[Cons 4](#_Toc92977688)

[Abstract Factory 4](#_Toc92977689)

[Problem 4](#_Toc92977690)

[Existing Project Example 4](#_Toc92977691)

[Applicability 4](#_Toc92977692)

[Pros 4](#_Toc92977693)

[Cons 4](#_Toc92977694)

[Builder 4](#_Toc92977695)

[Problem 4](#_Toc92977696)

[Existing Project Example 4](#_Toc92977697)

[Applicability 4](#_Toc92977698)

[Pros 4](#_Toc92977699)

[Cons 4](#_Toc92977700)

[Prototype 4](#_Toc92977701)

[Problem 4](#_Toc92977702)

[Existing Project Example 4](#_Toc92977703)

[Applicability 4](#_Toc92977704)

[Pros 4](#_Toc92977705)

[Cons 4](#_Toc92977706)

[Singleton 4](#_Toc92977707)

[Problem 4](#_Toc92977708)

[Existing Project Example 4](#_Toc92977709)

[Applicability 4](#_Toc92977710)

[Pros 4](#_Toc92977711)

[Cons 4](#_Toc92977712)

[Structural Pattern 3](#_Toc92977713)

[Adapter 3](#_Toc92977714)

[Problem 4](#_Toc92977715)

[Existing Project Example 4](#_Toc92977716)

[Applicability 4](#_Toc92977717)

[Pros 4](#_Toc92977718)

[Cons 4](#_Toc92977719)

[Bridge 3](#_Toc92977720)

[Problem 4](#_Toc92977721)

[Solution 4](#_Toc92977722)

[Existing Project Example 4](#_Toc92977723)

[Applicability 4](#_Toc92977724)

[Pros 4](#_Toc92977725)

[Cons 4](#_Toc92977726)

[Composite 3](#_Toc92977727)

[Problem 4](#_Toc92977728)

[Existing Project Example 4](#_Toc92977729)

[Applicability 4](#_Toc92977730)

[Pros 4](#_Toc92977731)

[Cons 4](#_Toc92977732)

[Decorator 3](#_Toc92977733)

[Problem 4](#_Toc92977734)

[Solution 4](#_Toc92977735)

[Existing Project Sample 4](#_Toc92977736)

[Applicability 4](#_Toc92977737)

[Pros 4](#_Toc92977738)

[Cons 4](#_Toc92977739)

[Façade 3](#_Toc92977740)

[Problem 3](#_Toc92977741)

[Solution 3](#_Toc92977742)

[Existing Project Example 4](#_Toc92977743)

[Applicability 4](#_Toc92977744)

[Pros 4](#_Toc92977745)

[Cons 4](#_Toc92977746)

# Creational Patterns

These patterns provide various object creation mechanisms which increase the flexibility and the reuse of code.

## Factory Method

This pattern provides an interface for creating objects (product object) in a superclass(creator class) but allows subclasses (concrete creator class) to change the type of objects (product object) that will be created.

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

In simple term, it is used to created multiple object types in a single interface.

### Problem

Suppose there exist a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the *Truck* class.

After a while, your app becomes popular. Each day you receive dozens of requests from sea transportation companies to incorporate *sea* logistics into the app.

But at present, most of your code is coupled to the Truck class. Adding Ships into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all these changes again.

### Existing Project Example

For *TechDoc* webpart handling data for multiple content types using legacy methodology – switch and case logic and that make the code unstructured and length and difficult to read.

Incorporating this pattern, separate classes of content type (as a product) can be created and implementing their functionality in their own class. However, calling of concrete creator classes will return the object the content type (product). But identification of which concrete class to call based on selected content type (from user end – UI) need to be done at code level or they can be wrapped in concrete creator classes.

Below are the mappings –

1. Single interface for all content type, say,

* ITechDocContentType

1. Separate class for each content type implement the interface, say,

* InstallationGuide implement ITechDocContentType
* CADDownloads implement ITechDocContentType
* ProductSpecSubmittals implement ITechDocContentType

1. Single Creator abstract class having abstract factory method of interface type.
2. Multiple ConcreteCreator classes inheriting Creator class and implement abstract method to return the product (Content Type) object, say,

* ConcreteCreatorInstallationGuide creating object for InstallationGuide class
* ConcreateCreatorCADDownloads creating object for CADDownloads class

### Applicability

1. Use this pattern when don’t know beforehand the exact types and dependencies of the objects your code work with 🡪 Its separate product construction code from the code that uses the product. Therefore, it's easier to extend the product construction code independently from the rest of the code.

For example, to add a new product type to the app, you’ll only need to create a new creator subclass and override the factory method in it.

1. Use this pattern when you want to save the system resources by reusing existing objects instead of rebuilding them each time.
2. Use this pattern when you want to provide users of your library or framework with a way to extend its internal components.

### Pros

1. It avoids tight coupling between the creator and the concrete products.
2. Follow *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
3. Follow *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.

### Cons

The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best-case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

## Abstract Factory

This pattern that lets you produce families of related objects without specifying their concrete classes.

It suggests to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa, or coffee table). Then you can make all variants of products follow those interfaces.

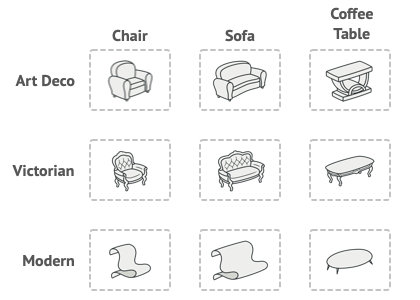
It says that just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. **That means Abstract Factory lets a class returns a factory of classes**. So, this is the reason that **Abstract Factory Pattern is one level higher than the Factory Pattern**.

It is also known as **Kit.**

### Problem

Suppose there exists a family of related product, say: *Chair* + *Sofa* + *CoffeeTable*. Several variants of this family, for example, products *Chair* + *Sofa* + *CoffeeTable* are available in these variants: *Modern*, *Victorian*, *ArtDeco*.



You need a way to create individual furniture objects so that they match other objects of the same family. Customers get quite mad when they receive non-matching furniture.

Text, application

Description automatically generated

A Modern-style sofa doesn’t match Victorian-style chairs.

Also, you don’t want to change existing code when adding new products or families of products to the program. Furniture vendors update their catalogs very often, and you wouldn’t want to change the core code each time it happens.

### Existing Project Example

Consider *Lutron Products – Athena, Vive, Quantum* etc. Allthese products share same family but have different page layout/structure based on their variants like *Desktop*, *Mobile* and *Fullbleed*.

Below are the mappings –

1. Distinct Products (of same family) 🡪 Athena, Vive, Quantum etc. which belong to same product family. So, each one will have separate interfaces.
2. Product Variants 🡪 These will be served as individual classes implementing the product interfaces. Like,

* ViveDesktop implement IVive
* ViveMobile implement IVive
* ViveFullBleed implement IVive
* QuantumDesktop implement IQuantum
* QuantumMobile implement IQuantum
* QuantumFullBleed implement IQuantum

1. There will be Abstract Factory interface declaring methods of interface type, say,

* IVive CreateViveFeedbackForm()
* IQuantum CreateQuantumFeedbackForm()

1. There will be ConcreteFactory classes which will implement the AbstractFactory interface. Multiple concrete factory classes will be created as number of variants, say,

* ConcreteFactoryDesktop implement AbstractFactory
* ConcreteFactoryMobile implement AbstractFactory
* ConcreteFactoryFullBleed implement AbstractFactory

Accordingly, client create the ConcreteFactory object on the basis of variant it required and called the required product function.

### Applicability

Use the Abstract Factory when your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products—they might be unknown beforehand, or you simply want to allow for future extensibility.

Consider implementing the Abstract Factory when you have a class with a set of Factory Methods that blur its primary responsibility.

In a well-designed program each class is responsible only for one thing. When a class deals with multiple product types, it may be worth extracting its factory methods into a stand-alone factory class or a full-blown Abstract Factory implementation.

### Pros

1. You can be sure that the products you're getting from a factory are compatible with each other.
2. You avoid tight coupling between concrete products and client code.
3. *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
4. *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.

### Cons

The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

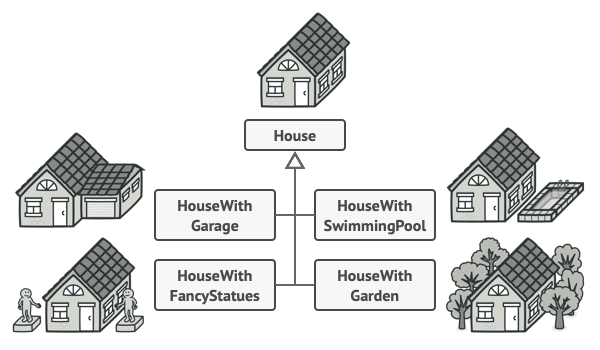
## Builder

This pattern allows constructing complex objects step by step. Unlike other creational patterns, here products don’t require to have a common interface which make it possible to produce different products using same construction process.

It can be recognized in a class, which has a single creation method and several methods to configure the resulting object.

### Problem

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.



You might make the program too complex by creating a subclass for every possible configuration of an object.

It can be possible to have single constructor having lots of parameters to handle all kinds of configuration but that make the **constructor call pretty ugly**.

### Existing Project Example

Consider an SSOT and DPL jobs, instead of creating two solutions for two jobs lets create a single project in single solution to implement both jobs using this pattern. Below are the mappings –

1. IBuilder interface 🡪 Declarations of set of common methods like –

* ConnectCosmosDB
* GetDrupalData
* PostDrupalData

1. ConcreteBuilder implement IBuilder 🡪 Implements all functions
2. Director 🡪 Create public property *Builder and* create some methods to call series of methods to build a product, in our case it is SSOT and DPL jobs.
3. ProductSSOT and ProductDPL 🡪Write only business logic of individual jobs.
4. Main class 🡪 Call through Director class or directly through Products classes to execute or perform a particular task.

### Applicability

1. Use it to get rid of a “telescopic constructor” 🡪 Say you have a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. These constructors still refer to the main one, passing some default values into any omitted parameters.
2. Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).
3. Use the Builder to construct **Composite** trees or other complex objects 🡪 The Builder pattern lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.   
   A builder doesn’t expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

### Pros

1. You can construct objects step-by-step, defer construction steps or run steps recursively.
2. You can reuse the same construction code when building various representations of products.
3. *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

### Cons

1. The overall complexity of the code increases since the pattern requires creating multiple new classes.

## Prototype

This pattern lets you copy existing objects without making your code dependent on their classes.

All prototype classes should have a common interface that makes it possible to copy objects even if their concrete classes are unknown. Prototype objects can produce full copies since objects of the same class can access each other's private fields.

### Problem

Suppose there is need to create a copy of existing object, so one must create a new object of the same class then one has to go through all the fields of the original object and copy their values over to the new object.

In such case if object has private fields, then one would not have access to those ones through the original object. Also, there is one more problem one must know the object’s class to create a duplicate which make it dependent on that class.

### Existing Project Example

SSOT and DPL have complex DTOs/Entities where it needs to create a duplicate object to perform some action without impacting the original ones then implementing this pattern to create a prototype or clone of original object will help to avoid duplicate code.

### Applicability

1. Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.
2. Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration

### Pros

1. You can clone objects without coupling to their concrete classes.
2. You can get rid of repeated initialization code in favour of cloning pre-built prototypes.
3. You can produce complex objects more conveniently.
4. You get an alternative to inheritance when dealing with configuration presets for complex objects.

### Cons

1. Cloning complex objects that have circular references might be very tricky.

## Singleton

This pattern helps one to ensure that a class has only one instance, while providing a global access point to this instance.

### Problem

It solves two problems at a time, violating the *Single Responsibility Principle*.

1. Ensure that a class has just single instance 🡪 This is to control access to some shared resources – for example, a database or a file.
2. Provide a global access point to that instance 🡪 Global object are used to store some essential objects that can be shared throughout the application but are unsafe since any code can potentially overwrite the contents of those variables and can crash the app.

Singleton pattern helps to create a global object and protect the instance from being overwritten by other code.

### Existing Project Example

Generally, while developing an application, one must connect to the database. When the application is multi-threaded or used by more than one user then it may create enumerable number of database instances which will eventually choke the memory of server and hence the performance issue or may crash the application.

In such cases, Singleton pattern is used to allow the creation of instance (database instance) at most once throughout the lifecycle of application to avoid any performance issues.

### Applicability

1. Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
2. Use the Singleton pattern when you need stricter control over global variables.

### Pros

1. You can be sure that a class has only a single instance.
2. You gain a global access point to that instance.
3. The singleton object is initialized only when it's requested for the first time.

### Cons

1. Violates the Single Responsibility Principle. The pattern solves two problems at the time.
2. The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
3. The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
4. It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don’t use the Singleton pattern.

# Structural Pattern

These patterns explain how to assemble objects and classes into larger structures while keeping their structures flexible and efficient.

## Adapter

It allows objects with incompatible interfaces to collaborate.

Adapter is recognizable by a constructor which takes an instance of a different abstract/interface type. When the adapter receives a call to any of its methods, it translates parameters to the appropriate format and then directs the call to one or several methods of the wrapped object.

### Problem

Imagine there is an app that want to use third party library, but that library take input in JSON format and app product data in XML format. So, how to communicate between them?

There *Adapter* pattern comes into picture which help to convert the data from XML to JSON without impacting existing app and library.

### Existing Project Example

For DTO and SSOT job, usually data is fetched and posted back to cosmos database in JSON format, but business logic is implemented on entities (object of class).

There conversion is required from JSON to Entity and Entity to JSON which can be achieve using *Adapter* pattern.

### 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.

The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.

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

The much more elegant solution would be to put the missing functionality into an adapter class. Then you would wrap objects with missing features inside the adapter, gaining needed features dynamically. For this to work, the target classes must have a common interface, and the adapter’s field should follow that interface. This approach looks very similar to the *Decorator* pattern.

### Pros

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

### Cons

1. 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.

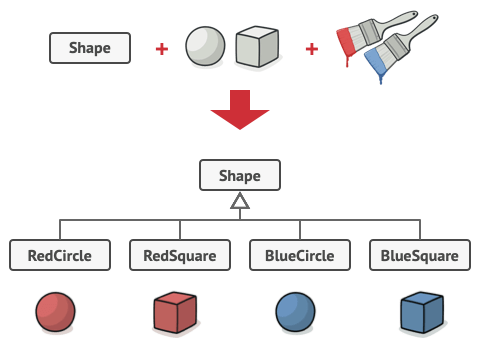
## Bridge

This pattern divides business logic or huge class into separate class hierarchies – abstractions and implementation - that can be developed independently.

One of these hierarchies (often called the Abstraction) will get a reference to an object of the second hierarchy (Implementation). The abstraction will be able to delegate some (sometimes, most) of its calls to the implementations object. Since all implementations will have a common interface, they’d be interchangeable inside the abstraction.

### Problem

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*.



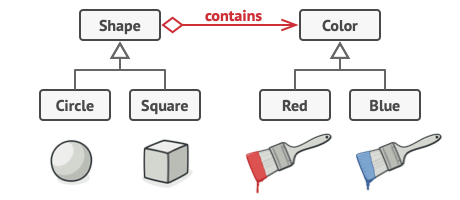
Number of class combinations grows in geometric progression.

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. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

### Solution

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.

### Existing Project Example

Consider an application with user controls, there ascx and ascx.cs files contain front-end logic and contain a reference of BL and DAL classes to call the required functionality when required. In such way, it separates the multi-dimensions of user controls into Business Layer (BL) and Data Application Layer (DAL) classes maintaining flexibility of the project structure.

### 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 🡪 It suggests that you extract a separate class hierarchy for each of the dimensions. The original class delegates the related work to the objects belonging to those hierarchies instead of doing everything on its own.
3. Use the Bridge if you need to be able to switch implementations at runtime 🡪Although it's optional, the Bridge pattern lets you replace the implementation object inside the abstraction. It’s as easy as assigning a new value to a field.

By the way, this last item is the main reason why so many people confuse the *Bridge* with the *Strategy* pattern. Remember that a pattern is more than just a certain way to structure your classes. It may also communicate intent and a problem being addressed.

### Pros

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

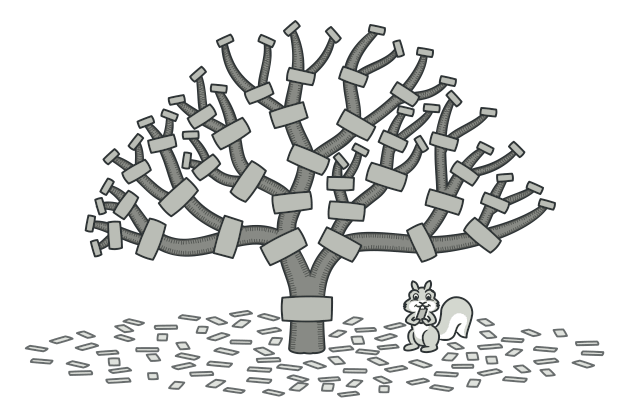
### Cons

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

## Composite

This pattern lets you to compose objects into tree structures and then work with these structures as if they were individual objects.

The Composite pattern lets you run a behaviour recursively over all components of an object tree. The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box.

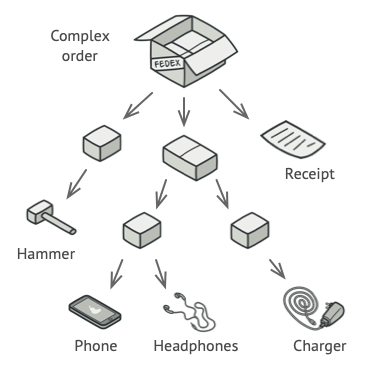


### Problem

This pattern make sense only when the core model of the application 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. You must know the classes of Products and Boxes you’re going through, the nesting level of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

### Existing Project Example

Consider an example of Sitemap in Lutron.com like the below one –

1. Component Abstract class 🡪 This class should be Abstract class and every *Leaf* and *Composite* should inherit and implement their operations.
2. Leaf class 🡪 It extend and implement the abstract and virtual methods of *Composite* class. This class is responsible to do the core operations as a standalone component.
3. Composite class 🡪 It also extend and implement the abstract and virtual methods of *Composite* class. It maintains a list or array of Component object like List<Component>. *It implements the core operation calling Composite class recursively to call the operation method of each leaf which Composite object contains.*

### Applicability

1. *Use the Composite pattern when you must implement a tree-like object structure*. The Composite pattern provides you with two basic element types that share a common interface: simple leaves and complex containers. A container can be composed of both leaves and other containers. This lets you construct a nested recursive object structure that resembles a tree.
2. *Use the pattern when you want the client code to treat both simple and complex elements uniformly*. All elements defined by the Composite pattern share a common interface. Using this interface, the client doesn’t have to worry about the concrete class of the objects it works with.

### Pros

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

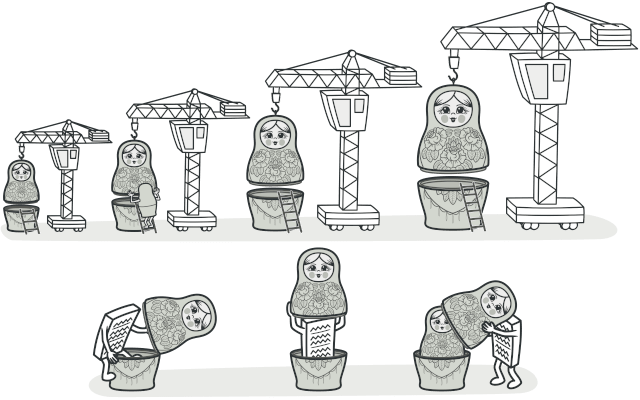
### Cons

1. It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

## Decorator

This pattern let you attach new behaviours to objects by placing these objects inside special wrapper objects that contain behaviours.

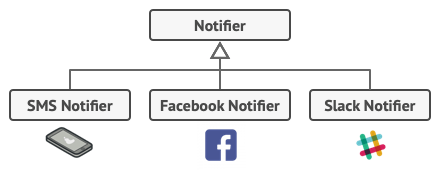
Using decorators, you can wrap objects countless number of times since both target objects and decorators follow the same interface. The resulting object will get a stacking behaviour of all wrappers.



### Problem

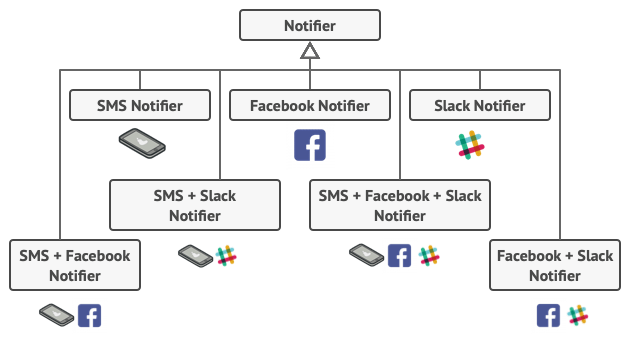
Suppose there is a notification library which lets other programs notify their users about important events.

At some point, users of library expect more than just simple email notifications. Many of them would like to receive SMS about critical updates. Some of users would like to be notified on Facebook and, corporate users would like to get Slack/Teams notifications.



Each notification type is implemented as a notifier’s subclass.

This can be easily achieved just by extending *Notifier* class and creating many subclasses.



Combinatorial explosion of subclasses.

It is quite apparent that this approach would bloat the code immensely, not only the library code but the client code as well.

### Solution

Using inheritance, extending a class 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.

1. 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.
2. 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. Aggregation/composition is the key principle behind many design patterns, including Decorator. "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.

### Existing Project Sample

Consider an application where exceptions and messages from the code to be written conditionally on below locations –

* Shared File Path
* SharePoint On-premises logs directory
* Azure Application Insights

1. Component class 🡪 Abstract class or an interface having common method “Operation” to be implemented everywhere.
2. Concrete Component 🡪 Extending or implementing Component class with definition of method “Operation”, here it will write log to “Azure Application Insights” by default.
3. Decorator class 🡪 It will have property of type “Component” whose constructor will assign the “Component” object, inheriting “Component” class and implement “Operation” method which will just call “Component” operation method.
4. ConcreteDecorators 🡪 Multiple classes implementing “Decorator” class and implementing method - “Operation” calling their base “operation” method.

### 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*. Many programming languages have the final keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

### Pros

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

### Cons

1. It's hard to remove a specific wrapper from the wrappers stack.
2. It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorator’s stack.
3. The initial configuration code of layers might look ugly.

## Façade

This pattern provides a simplified (but limited) interface to a complex system of classes, library, framework, or any other set of classes. It decreases the overall flexibility of the application and helps to move unwanted dependencies to one place.

### Problem

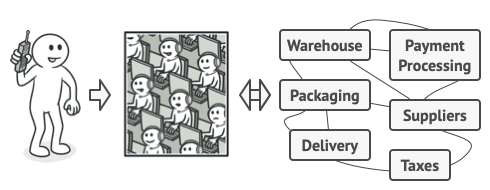
Suppose there is a project where your code works with broad set of objects that belong to a sophisticated library or framework. Usually, developer need to initialize all those objects, keep track of dependencies, execute methods in correct order, and so on.

Ultimately the business logic of project would become tightly couple to the implementation details of the 3rd party library, making it hard to comprehend and maintain.

### Solution

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.

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.



Placing orders by phone.

### Existing Project Example

Imagine a Lutron SSOT and DPL job underlying a complex sub-system for many activities like creating connection to cosmos database, CRUD operations on Cosmos, logging to Azure Application Insights, validating the process, and so on.

Each sub-system will have its own separate class which can be act as library to be re-use in both DPL and SSOT job. Here, Façade class can be created having two operation methods – one of SSOT and another is for DPL for initializing and utilizing the sub-classes in the required order.

### Applicability

1. Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem. Often, subsystems get more complex over time. Even applying design patterns typically leads to creating more classes. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.
2. Use the Facade when you want to structure a subsystem into layers. Create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades.

### Pros

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

### Cons

1. A facade can become a **god object** coupled to all classes of an app. The God object is an example of anti-pattern, and it is also known as **All-knowing** object.

## Flyweight