**DESIGN PATTERNS**

Are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

You can’t just find a pattern and copy it into your program, that way you can with off-the shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your program.

Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is more a high-level description of a solution. The code of the same pattern applied to different programs may be different.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Here are the sections that are usually present in a pattern description:

* **Intent** of the briefly describes both the problem and the solution.
* **Motivation** further explains the problem and the solution the pattern makes possible.
* **Structure** of classes shows each part of the pattern and how they are related
* **Code** example in one of the popular programming languages makes it easier to grasp the idea behind the pattern

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns.

**Why should I learn patterns?**

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern.

* Design patterns are a toolkit of tried and tested solutions to common problems in software design. Even if you never encounter these problems, knowing pattern is still useful because it teaches you how to solve all sort of problems using principles of object-oriented design.
* Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that”, and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern.

**Classification of patterns**

Design patterns differ by their complexity, level of detail and scale of applicability to the entire system being designed.

The most basic and low-level patterns are often called idioms. They usually apply only to a single programming language.

The most universal and high-level patterns are architectural patterns. Developers can implement these patterns in virtually any language. Unlike other patterns, they can be used to design the architecture of an entire application.

Mainly patterns are classified as:

* **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
* **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
* **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

**Creational patterns.**

* **Factory method**
* **Abstract factory**
* **Builder**
* **Prototype**
* **Singleton**

**Structural patterns.**

* **Adapter**
* **Bridge**
* **Composite**
* **Decorator**
* **Facade**
* **Flyweight**
* **Proxy**

**Behavioral patterns.**

* **Chain of Responsibility**
* **Command**
* **Memento**
* **Observer**
* **Iterator**
* **Mediator**
* **State**
* **Strategy**
* **Template Method**
* **Visitor**

**FACTORY METHOD**

**Intent:** Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

Suggests that you replace direct object construction calls with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.

This pattern allows you to override the factory method in a subclass and change the class of products being created by the method.

There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

*Applicability*

* **Use it when you don’t know beforehand the exact types and dependencies of the objects your code should work with**.

Separates 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.

* **Use it when you want to provide users of your library or framework with a way to extend its internal components.**

Inheritance is probably the easiest way to extend the default behavior of a library or framework. But how would the framework recognize that your subclass should be used instead of a standard component itself.

* **Use it when you want to save system resources by reusing existing objects instead of rebuilding them each time.**

You often experience this need when dealing with large, resource-intensive objects such as database connections, file systems, and network resources.

*Pros*

You avoid tight coupling between the creator and the concrete products.

Single responsibility principle. You can move the product creation code into one place in the program, making code easier to support.

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**

**Intent.** Is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

*Applicability*

* **Use it 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.** This pattern provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don’t have to worry creating the wrong variant of a product which doesn’t match the products already created by your app.

Consider implementing it 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.*

You can be sure that the products you’re getting from a factory are compatible with each other.

You avoid tight coupling between concrete products and client code.

Single responsibility principle. You can extract the product creation code into one place, making the code easier to support.

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**

**Intent.** It lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**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.

This pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

To create an object, you execute a series of these steps on a builder object. The important part is that you don’t need to call all the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

You can go further and extract a series of calls to the builder steps, you use to construct a product into a separate class called **director**. The director class defines the order which to execute the building steps, while the builder provides the implementation for those steps.

Having a director class in your program isn’t’ strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across the program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

*Applicability*

* **Use it to get a rid of “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. After applying the pattern, you don’t have to cram dozens of parameters into your constructors anymore, this is because, the pattern uses only the steps that you really need to create objects.

* **Use it when you want your code to be able to create different representations of some product.** Can be applied when construction of various representations of the product involves similar steps that differ only in the details. The base builder interface defines all possible construction steps, and concrete builders implement these steps to construct particular representations of the product. Meanwhile, the director class guides the order of construction.
* **Use it to construct Composite trees or other complex objects.**

This 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*

You can construct objects step-by-step, defer construction steps or run steps recursively.

You can reuse the same construction code when building various representations of products.

Single responsibility principle. You can isolate complex construction code from the business logic of the product.

*Cons*

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

**PROTOTYPE**

**Intent.** Let’s you copy existing objects without making your code dependent on their classes.

**Problem**. Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

Nice! But there’s a catch. Not all objects can be copied that way because some of the object’s fields may be private and not visible from outside of the object itself.

There’s one more problem with the direct approach. Since you have to know the object’s class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn’t scare you, there’s another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.

**Solution.** This pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class to that object. Usually, such an interface contains just a single clone method.

The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Here’s how it works: you create a set of objects, configured in various ways. When you need an object like the one you’ve configured, you just clone a prototype instead of constructing a new object from scratch.

*Applicability*

* **Use it when your code shouldn’t depend on the concrete classes of objects that you need to copy.**  This happens a lot when your code works with objects passed to you from 3rd party code via some interface. The concrete classes of these objects unknown, and you couldn’t depend on them even if you wanted to.

This pattern provides the client code with general interface for working with all objects that supporting cloning. This interface makes the client code independent from the concrete classes of objects that it clones.

* **Use it when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.**Suppose you have a complex class that requires a laborious configuration before it can be used. There are several common ways to configure this class, and this code is scattered through your app. To reduce the duplication, you create several subclasses and put every common configuration code into their constructors. You solved the duplication problem, but now you have lots of dummy subclasses.

This pattern lets you use a set of pre-built objects configured in various ways as prototypes. Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

*Pros*

You can clone objects without coupling to their concrete classes.

You can get rid of repeated initialization code in favor of cloning pre-built prototypes.

You can produce complex objects more conveniently.

You get an alternative to inheritance when dealing with configuration presets for complex objects.

*Cons*

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

**SINGLETON**

**Intent.** This lets you ensure that a class has only one instance.

**Problem.** This pattern solves two problems at the same time, violating the single responsibility principle:

1.Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

Here’s how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you’ll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

2.**Provide a global access point to that instance**. Remember those global variables that you (all right, me) used to store some essential objects? While they’re very handy, they’re also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

There’s another side to this problem: you don’t want the code that solves problem #1 to be scattered all over your program. It’s much better to have it within one class, especially if the rest of your code already depends on it.

Nowadays, the Singleton pattern has become so popular that people may call something a *singleton* even if it solves just one of the listed problems.

**Solution.**

All implementations of the Singleton have these two steps in common:

* Make the default constructor private, to prevent other objects from using the new operator with the singleton class.
* Create a static creation method that acts as a constructor to create an object and saves it in static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So, whenever that method is called, the same object is always returned.

*Applicability*

* **Use the 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.** This pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.
* **Use it when you need stricter control over global variables.** Unlike global variables, the Singleton pattern guarantees that there’s just one instance of a class. Nothing, except for the Singleton class itself, can replace the cached instance.

Note that you can always adjust this limitation and allow creating any number of Singleton instances.

*How to implement.*

* Add a private static field to the class for storing the singleton instance.
* Declare a public static creation method for getting the singleton instance.
* Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
* Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
* Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.

*Pros*

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

*Cons*

* Violates the single responsibility principle. The pattern solves two problems at the time.
* The singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
* The pattern requires special treatment in a multithreaded environment so that multiple threads won’t create a singleton object several times.
* 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 DESIGN PATTERNS**

**ADAPTER.**

**Intent.** Allows objects with incompatible interfaces to collaborate.

**Problem.** Imagine that you’re creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there’s a catch: the analytics library only works with data in JSON format.

You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library’s source code in the first place, making this approach impossible.

**Solution.** You can create an adapter. This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all the data to imperial units such as feet and miles.

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here’s how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter’s methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Sometimes it’s even possible to create a two-way adapter that can convert the calls in both directions.

*Applicability*

* **Use it when you want to use some existing class, but its interface isn’t compatible with the rest of your code.** This 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.
* **Use it when you want to reuse several existing subclasess that lack some common functionality that can’t be added to the superclass.** You could extend each subclass and put the missing functionality into new child classes. However, you’ll need to duplicate the code across all these new classes, which smells really bad. 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.

*How to implement*

* Make sure that you have at least two classes with incompatible interfaces:
  + A useful service class, which you can’t change (often 3rd-party, legacy or with lots of existing dependencies).
  + One or several client classes that would benefit from using the service class.
* Declare the client interface and describe how clients communicate with the service.
* Create the adapter class and make it follow the client interface. Leave all the methods empty for now.
* Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it’s more convenient to pass it to the adapter when calling its methods.
* One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.
* Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

*Pros*

* Single responsibility principle. You can separate the interface or data conversion code from the primary business logic of the program.
* Open/closed principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

*Cons*

* 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**

**Intent.** This lets you split a large class or a set of closely related classes into two separate hierarchies -abstraction and implementation- which can be developed independently of each other.

**Solution.** The Bridge pattern attempts to solve problems 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 its state and behaviors within one class.

Abstraction (also called interface) is a high-level control layer for some entity. This layer isn’t supposed to do any real work on its own. It should delegate the work to the implementation layer (also called platform).

Note that we’re not talking about interfaces or abstract classes from your programming language. These aren’t the same things.

*Applicability*

* **Use it 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).**

The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change. The changes made to one of the variations of functionality may require making changes across the whole class, which often results in making errors or not addressing some critical side effects.

The Bridge pattern lets you split the monolithic class into several class hierarchies. After this, you can change the classes in each hierarchy independently of the classes in the others. This approach simplifies code maintenance and minimizes the risk of breaking existing code.

* **Use it when you need to extend a class in several orthogonal (independent) dimensions.**

This pattern 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.

* **Use it 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](https://refactoring.guru/design-patterns/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.

*How to implement.*

* Identify the orthogonal dimensions in your classes. These independent concepts could be: abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation.
* See what operations the client needs and define them in the base abstraction class.
* Determine the operations available on all platforms. Declare the ones that the abstraction needs in the general implementation interface.
* For all platforms in your domain create concrete implementation classes, but make sure they all follow the implementation interface.
* Inside the abstraction class, add a reference field for the implementation type. The abstraction delegates most of the work to the implementation object that’s referenced in that field.
* If you have several variants of high-level logic, create refined abstractions for each variant by extending the base abstraction class.
* The client code should pass an implementation object to the abstraction’s constructor to associate one with the other. After that, the client can forget about the implementation and work only with the abstraction object.

*Pros*

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

*Cons*

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

**COMPOSITE**

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

**Problem.** Using composite pattern makes sense only when the core model of your app can be represented as a tree.

*Applicability*

* **Use it when you have to implement a tree-like object structure**

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.

* **Use it 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.

*How to implement.*

* Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.
* Declare the component interface with a list of methods that make sense for both simple and complex components.
* Create a leaf class to represent simple elements. A program may have multiple different leaf classes.
* Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and containers, so make sure it’s declared with the component interface type.
* While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.
* Finally, define the methods for adding and removal of child elements in the container.

Keep in mind that these operations can be declared in the component interface. This would violate the *Interface Segregation Principle* because the methods will be empty in the leaf class. However, the client will be able to treat all the elements equally, even when composing the tree.

*Pros*

* You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
* 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*

* 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**

**Intent.** This pattern lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

**Solution.** Extending a class is 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.

* 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.
* 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. Both alternatives work almost the same way: one object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

With this new approach you can easily substitute the linked “helper” object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work. 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 it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

The wrapper implements the same interface as the wrapped object. That’s why from the client’s perspective these objects are identical. Make the wrapper’s reference field accept any object that follows that interface. This will let you cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

*Applicability*

* **Use it when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.**

This lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way since they all follow a common interface.

* **Use it 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.

*How to implement*

* Make sure your business domain can be represented as a primary component with multiple optional layers over it.
* Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.
* Create a concrete component class and define the base behavior in it.
* Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.
* Make sure all classes implement the component interface.
* Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behavior before or after the call to the parent method (which always delegates to the wrapped object).
* The client code must be responsible for creating decorators and composing them in the way the client needs.

*Pros*

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

*Cons*

* It’s hard to remove a specific wrapper from the wrappers stack.
* 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.
* The initial configuration code of layers might look pretty ugly.

**FACADE**

**Intent.** This pattern provides a simplified interface to a library, a framework, or any other complex set of classes.

**Problem.** Imagine that you must make your code with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you’d need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, 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.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

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.

*Applicability*

* **Use it 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. A subsystem may become more flexible and easier to reuse in various contexts, but the amount of configuration and boilerplate code it demands from a client grows ever larger. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.
* **Use it 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.

For example, let’s return to our video conversion framework. It can be broken down into two layers: video- and audio-related. For each layer, you can create a facade and then make the classes of each layer communicate with each another via those facades. This approach looks very similar to the [Mediator](https://refactoring.guru/design-patterns/mediator) pattern.

*How to implement*

* Check whether it’s possible to provide a simpler interface than what an existing subsystem already provides. You’re on the right track if this interface makes the client code independent from many of the subsystem’s classes.
* Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
* To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.
* If the facade becomes [too big](https://refactoring.guru/smells/large-class), consider extracting part of its behavior to a new, refined facade class.

*Pros*

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

*Cons*

* A facade can become [a god object](https://refactoring.guru/antipatterns/god-object) coupled to all classes of an app.