**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 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.

**FLYWEIGHT**

**Intent**. This pattern lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object.

*Applicability*

* **Use it only when your program must support a huge number of objects which barely fit into available RAM**

The benefit of applying the pattern depends heavily on how and where it’s used. It’s most useful when:

* an application needs to spawn a huge number of similar objects
* this drains all available RAM on a target device
* the objects contain duplicate states which can be extracted and shared between multiple objects

*How to implement*

* Divide fields of a class that will become a flyweight into two parts:

The intrinsic state: the fields that contain unchanging data duplicated across many objects

The extrinsic state: the fields that contain contextual data unique to each object

* Leave the fields that represent the intrinsic state in the class, but make sure they’re immutable. They should take their initial values only inside the constructor.
* Go over methods that use fields of the extrinsic state. For each field used in the method, introduce a new parameter, and use it instead of the field.
* Optionally, create a factory class to manage the pool of flyweights. It should check for an existing flyweight before creating a new one. Once the factory is in place, clients must only request flyweights through it. They should describe the desired flyweight by passing its intrinsic state to the factory.
* The client must store or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects. For the sake of convenience, the extrinsic state along with the flyweight-referencing field may be moved to a separate context class.

*Pros*

* You can save lots of RAM, assuming your program has tons of similar objects.

*Cons*

* You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.
* The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.

**PROXY**

**Intent.** This patter lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

*Applicability*

* **Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.** Instead of creating the object when the app launches, you can delay the object’s initialization to a time when it’s really needed.
* **Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).** The proxy can pass the request to the service object only if the client’s credentials match some criteria.
* **Local execution of a remote service (remote proxy). This is when the service object is located on a remote server**. In this case, the proxy passes the client request over the network, handling all the nasty details of working with the network.
* **Logging requests (logging proxy). This is when you want to keep a history of requests to the service object**. The proxy can log each request before passing it to the service.
* **Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.** The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.
* **Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.** The proxy can keep track of clients that obtained a reference to the service object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources. The proxy can also track whether the client had modified the service object. Then the unchanged objects may be reused by other clients.

*How to implement*

* If there’s no pre-existing service interface, create one to make proxy and service objects interchangeable. Extracting the interface from the service class isn’t always possible, because you’d need to change all of the service’s clients to use that interface. Plan B is to make the proxy a subclass of the service class, and this way it’ll inherit the interface of the service.
* Create the proxy class. It should have a field for storing a reference to the service. Usually, proxies create and manage the whole life cycle of their services. On rare occasions, a service is passed to the proxy via a constructor by the client.
* Implement the proxy methods according to their purposes. In most cases, after doing some work, the proxy should delegate the work to the service object.
* Consider introducing a creation method that decides whether the client gets a proxy or a real service. This can be a simple static method in the proxy class or a full-blown factory method.
* Consider implementing lazy initialization for the service object.

*Pros*

* You can control the service object without clients knowing about it.
* You can manage the lifecycle of the service object when clients don’t care about it.
* The proxy works even if the service object isn’t ready or is not available.
* Open/Closed Principle. You can introduce new proxies without changing the service or clients.

*Cons*

* The code may become more complicated since you need to introduce a lot of new classes.
* The response from the service might get delayed.

**CHAIN OF RESPONSIBILITY**

**Intent.** This pattern lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

*Applicability*

* **Use it when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.** The pattern lets you link several handlers into one chain and, upon receiving a request, “ask” each handler whether it can process it. This way all handlers get a chance to process the request.
* **Use the pattern when it’s essential to execute several handlers in a particular order.** Since you can link the handlers in the chain in any order, all requests will get through the chain exactly as you planned.
* **Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.** If you provide setters for a reference field inside the handler classes, you’ll be able to insert, remove or reorder handlers dynamically.

*How to implement.*

Declare the handler interface and describe the signature of a method for handling requests.

Decide how the client will pass the request data into the method. The most flexible way is to convert the request into an object and pass it to the handling method as an argument.

To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.

This class should have a field for storing a reference to the next handler in the chain. Consider making the class immutable. However, if you plan to modify chains at runtime, you need to define a setter for altering the value of the reference field.

You can also implement the convenient default behavior for the handling method, which is to forward the request to the next object unless there’s none left. Concrete handlers will be able to use this behavior by calling the parent method.

One by one create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:

Whether it’ll process the request.

Whether it’ll pass the request along the chain.

The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.

The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.

Due to the dynamic nature of the chain, the client should be ready to handle the following scenarios:

The chain may consist of a single link.

Some requests may not reach the end of the chain.

Others may reach the end of the chain unhandled.

*Pros*

* You can control the order of request handling.
* Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations.
* Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.

*Cons*

* Some requests may end up unhandled.

**COMMAND**

**Intent.** This pattern turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request’s execution, and support undoable operations.

*Applicability*

* **Use it when you want to parametrize objects with operations.** The Command pattern can turn a specific method call into a stand-alone object. This change opens a lot of interesting uses: you can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc.
* **Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.** As with any other object, a command can be serialized, which means converting it to a string that can be easily written to a file or a database. Later, the string can be restored as the initial command object. Thus, you can delay and schedule command execution. But there’s even more! In the same way, you can queue, log or send commands over the network.
* **Use the Command pattern when you want to implement reversible operations.** Although there are many ways to implement undo/redo, the Command pattern is perhaps the most popular of all. To be able to revert operations, you need to implement the history of performed operations. The command history is a stack that contains all executed command objects along with related backups of the application’s state.

This method has two drawbacks. First, it isn’t that easy to save an application’s state because some of it can be private. This problem can be mitigated with the [Memento](https://refactoring.guru/design-patterns/memento) pattern.

Second, the state backups may consume quite a lot of RAM. Therefore, sometimes you can resort to an alternative implementation: instead of restoring the past state, the command performs the inverse operation. The reverse operation also has a price: it may turn out to be hard or even impossible to implement.

*How to implement*

* Declare the command interface with a single execution method.
* Start extracting requests into concrete command classes that implement the command interface. Each class must have a set of fields for storing the request arguments along with a reference to the actual receiver object. All these values must be initialized via the command’s constructor.
* Identify classes that will act as senders. Add the fields for storing commands into these classes. Senders should communicate with their commands only via the command interface. Senders usually don’t create command objects on their own, but rather get them from the client code.
* Change the senders so they execute the command instead of sending a request to the receiver directly.
* The client should initialize objects in the following order:
  + Create receivers.
  + Create commands and associate them with receivers if needed.
  + Create senders and associate them with specific commands.

*Pros*

* Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations.
* Open/Closed Principle. You can introduce new commands into the app without breaking existing client code.
* You can implement undo/redo.
* You can implement deferred execution of operations.
* You can assemble a set of simple commands into a complex one.

*Cons*

* The code may become more complicated since you’re introducing a whole new layer between senders and receivers.

**ITERATOR**

**Iterator.** This pattern lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.)

**Problem.** Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs, and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On **the other hand**, the client code that’s supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

**Solution**. The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn’t return anything, which means that the iterator has traversed all the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm if there’s a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

*Applicability*

* **Use it when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).** The iterator encapsulates the details of working with a complex data structure, providing the client with several simple methods of accessing the collection elements. While this approach is very convenient for the client, it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.
* **Use the pattern to reduce duplication of the traversal code across your app.** The code of non-trivial iteration algorithms tends to be very bulky. When placed within the business logic of an app, it may blur the responsibility of the original code and make it less maintainable. Moving the traversal code to designated iterators can help you make the code of the application leaner and cleaner.
* **Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.** The pattern provides a couple of generic interfaces for both collections and iterators. Given that your code now uses these interfaces, it’ll still work if you pass it various kinds of collections and iterators that implement these interfaces.

*How to implement*

* Declare the iterator interface. At the very least, it must have a method for fetching the next element from a collection. But for the sake of convenience you can add a couple of other methods, such as fetching the previous element, tracking the current position, and checking the end of the iteration.
* Declare the collection interface and describe a method for fetching iterators. The return type should be equal to that of the iterator interface. You may declare similar methods if you plan to have several distinct groups of iterators.
* Implement concrete iterator classes for the collections that you want to be traversable with iterators. An iterator object must be linked with a single collection instance. Usually, this link is established via the iterator’s constructor.
* Implement the collection interface in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class. The collection object must pass itself to the iterator’s constructor to establish a link between them.
* Go over the client code to replace all of the collection traversal code with the use of iterators. The client fetches a new iterator object each time it needs to iterate over the collection elements.

*Pros*

* Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
* Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.
* You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
* For the same reason, you can delay an iteration and continue it when needed.

*Cons*

* Applying the pattern can be an overkill if your app only works with simple collections.
* Using an iterator may be less efficient than going through elements of some specialized collections directly.

**MEDIATOR**

**Intent.** This pattern lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

*Applicability*

* **Use it when it’s hard to change some of the classes because they are tightly coupled to a bunch of other classes.** The pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component from the rest of the components.
* **Use it when you can’t reuse a component in a different program because it’s too dependent on other components.** After you apply the Mediator, individual components become unaware of the other components. They could still communicate with each other, albeit indirectly, through a mediator object. To reuse a component in a different app, you need to provide it with a new mediator class.
* **Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.** Since all relations between components are contained within the mediator, it’s easy to define entirely new ways for these components to collaborate by introducing new mediator classes, without having to change the components themselves.

*How to implement*

* Identify a group of tightly coupled classes which would benefit from being more independent (e.g., for easier maintenance or simpler reuse of these classes).
* Declare the mediator interface and describe the desired communication protocol between mediators and various components. In most cases, a single method for receiving notifications from components is sufficient.

This interface is crucial when you want to reuse component classes in different contexts. As long as the component works with its mediator via the generic interface, you can link the component with a different implementation of the mediator.

* Implement the concrete mediator class. This class would benefit from storing references to all of the components it manages.
* You can go even further and make the mediator responsible for the creation and destruction of component objects. After this, the mediator may resemble a [factory](https://refactoring.guru/design-patterns/abstract-factory) or a [facade](https://refactoring.guru/design-patterns/facade).
* Components should store a reference to the mediator object. The connection is usually established in the component’s constructor, where a mediator object is passed as an argument.
* Change the components’ code so that they call the mediator’s notification method instead of methods on other components. Extract the code that involves calling other components into the mediator class. Execute this code whenever the mediator receives notifications from that component.

*Pros*

* Single Responsibility Principle. You can extract the communications between various components into a single place, making it easier to comprehend and maintain.
* Open/Closed Principle. You can introduce new mediators without having to change the actual components.
* You can reduce coupling between various components of a program.
* You can reuse individual components more easily.

*Cons*

* Over time a mediator can evolve into a [God Object](https://refactoring.guru/antipatterns/god-object).

**MEMENTO**

**Intent.** This lets you save and restore the previous state of an object without revealing the details of its implementation.

*Applicability*

* **Use it when you want to produce snapshots of the object’s state to be able to restore a previous state of the object.** The Memento pattern lets you make full copies of an object’s state, including private fields, and store them separately from the object. While most people remember this pattern thanks to the “undo” use case, it’s also indispensable when dealing with transactions (i.e., if you need to roll back an operation on error).
* **Use it when direct access to the object’s fields/getters/setters violates its encapsulation**. The Memento makes the object itself responsible for creating a snapshot of its state. No other object can read the snapshot, making the original object’s state data safe and secure.

*How to implement*

* Determine what class will play the role of the originator. It’s important to know whether the program uses one central object of this type or multiple smaller ones.
* Create the memento class. One by one, declare a set of fields that mirror the fields declared inside the originator class.
* Make the memento class immutable. A memento should accept the data just once, via the constructor. The class should have no setters.
* If your programming language supports nested classes, nest the memento inside the originator. If not, extract a blank interface from the memento class and make all other objects use it to refer to the memento. You may add some metadata operations to the interface, but nothing that exposes the originator’s state.
* Add a method for producing mementos to the originator class. The originator should pass its state to the memento via one or multiple arguments of the memento’s constructor.

The return type of the method should be of the interface you extracted in the previous step (assuming that you extracted it at all). Under the hood, the memento-producing method should work directly with the memento class.

* Add a method for restoring the originator’s state to its class. It should accept a memento object as an argument. If you extracted an interface in the previous step, make it the type of the parameter. In this case, you need to typecast the incoming object to the memento class, since the originator needs full access to that object.
* The caretaker, whether it represents a command object, a history, or something entirely different, should know when to request new mementos from the originator, how to store them and when to restore the originator with a particular memento.
* The link *between care*takers and originators may be moved into the memento class. In this case, each memento must be connected to the originator that had created it. The restoration method would also move to the memento class. However, this would all make sense only if the memento class is nested into originator or the originator class provides sufficient setters for overriding its state.

*Pros*

* You can produce snapshots of the object’s state without violating its encapsulation.
* You can simplify the originator’s code by letting the caretaker maintain the history of the originator’s state.

*Cons*

* The app might consume lots of RAM if clients create mementos too often.
* Caretakers should track the originator’s lifecycle to be able to destroy obsolete mementos.
* Most dynamic programming languages, such as PHP, Python and JavaScript, can’t guarantee that the state within the memento stays untouched.

**OBSERVER**

**Intent.** Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

*Applicability*

* **Use it when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically**. You can often experience this problem when working with classes of the graphical user interface. For example, you created custom button classes, and you want to let the clients hook some custom code to your buttons so that it fires whenever a user presses a button.

The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects. You can add the subscription mechanism to your buttons, letting the clients hook up their custom code via custom subscriber classes.

* **Use it when some objects in your app must observe others, but only for a limited time or in specific cases**. The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.

*How to implement*

* Look over your business logic and try to break it down into two parts: the core functionality, independent from other code, will act as the publisher; the rest will turn into a set of subscriber classes.
* Declare the subscriber interface. At a bare minimum, it should declare a single update method.
* Declare the publisher interface and describe a pair of methods for adding a subscriber object to and removing it from the list. Remember that publishers must work with subscribers only via the subscriber interface.
* Decide where to put the actual subscription list and the implementation of subscription methods. Usually, this code looks the same for all types of publishers, so the obvious place to put it is in an abstract class derived directly from the publisher interface. Concrete publishers extend that class, inheriting the subscription behavior.

However, if you’re applying the pattern to an existing class hierarchy, consider an approach based on composition: put the subscription logic into a separate object, and make all real publishers use it.

* Create concrete publisher classes. Each time something important happens inside a publisher, it must notify all its subscribers.
* Implement the update notification methods in concrete subscriber classes. Most subscribers would need some context data about the event. It can be passed as an argument of the notification method.

But there’s another option. Upon receiving a notification, the subscriber can fetch any data directly from the notification. In this case, the publisher must pass itself via the update method. The less flexible option is to link a publisher to the subscriber permanently via the constructor.

* The client must create all necessary subscribers and register them with proper publishers.

*Pros*

* Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher’s code (and vice versa if there’s a publisher interface).
* You can establish relations between objects at runtime.

*Cons*

* Subscribers are notified in random order.

**STATE**

**Intent.** This pattern lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

**Problem.** This pattern is closely related to the concept of a Finite-State-Machine. The main idea is that any given moment, there’s a finite number of states which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules called transitions, are also finite predetermined.

**Solution.** The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.

To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same interface and the context itself works with these objects through that interface.

This structure may look like the [Strategy](https://refactoring.guru/design-patterns/strategy) pattern, but there’s one key difference. In the State pattern, the states may be aware of each other and initiate transitions from one state to another, whereas strategies almost never know about each other.

*Applicability*

* **Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.** The pattern suggests that you extract all state-specific code into a set of distinct classes. As a result, you can add new states or change existing ones independently of each other, reducing the maintenance cost.
* **Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class’s fields.** The State pattern lets you extract branches of these conditionals into methods of corresponding state classes. While doing so, you can also clean temporary fields and helper methods involved in state-specific code out of your main class.
* **Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine**. The State pattern lets you compose hierarchies of state classes and reduce duplication by extracting common code into abstract base classes.

*How to implement*

* Decide what class will act as the context. It could be an existing class which already has the state-dependent code; or a new class, if the state-specific code is distributed across multiple classes.
* Declare the state interface. Although it may mirror all the methods declared in the context, aim only for those that may contain state-specific behavior.
* For every actual state, create a class that derives from the state interface. Then go over the methods of the context and extract all code related to that state into your newly created class.

While moving the code to the state class, you might discover that it depends on private members of the context. There are several workarounds:

Make these fields or methods public.

Turn the behavior you’re extracting into a public method in the context and call it from the state class. This way is ugly but quick, and you can always fix it later.

Nest the state classes into the context class, but only if your programming language supports nesting classes.

* In the context class, add a reference field of the state interface type and a public setter that allows overriding the value of that field.
* Go over the method of the context again and replace empty state conditionals with calls to corresponding methods of the state object.
* To switch the state of the context, create an instance of one of the state classes and pass it to the context. You can do this within the context itself, or in various states, or in the client. Wherever this is done, the class becomes dependent on the concrete state class that it instantiates.

*Pros*

* Single Responsibility Principle. Organize the code related to particular states into separate classes.
* Open/Closed Principle. Introduce new states without changing existing state classes or the context.
* Simplify the code of the context by eliminating bulky state machine conditionals.

Cons

* Applying the pattern can be overkill if a state machine has only a few states or rarely changes.

**STRATEGY**

**Intent.** This pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

**Solution**. The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all these algorithms into separate classes called *strategies*.

The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context isn’t responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn’t know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.

*Applicability*

* **Use it when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.** The Strategy pattern lets you indirectly alter the object’s behavior at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.
* **Use it when you have a lot of similar classes that only differ in the way they execute some behavior.** The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.
* **Use it to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.** The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.
* **Use it when your class has a massive conditional operator that switches between different variants of the same algorithm.** The Strategy pattern lets you do away with such a conditional by extracting all algorithms into separate classes, all of which implement the same interface. The original object delegates execution to one of these objects, instead of implementing all variants of the algorithm.

*How to implement*

* In the context class, identify an algorithm that’s prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.
* Declare the strategy interface common to all variants of the algorithm.
* One by one, extract all algorithms into their own classes. They should all implement the strategy interface.
* In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.
* Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

*Pros*

* You can swap algorithms used inside an object at runtime.
* You can isolate the implementation details of an algorithm from the code that uses it.
* You can replace inheritance with composition.
* Open/Closed Principle. You can introduce new strategies without having to change the context.

*Cons*

* If you only have a couple of algorithms and they rarely change, there’s no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
* Clients must be aware of the differences between strategies to be able to select a proper one.
* A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you’d have used the strategy objects, but without bloating your code with extra classes and interfaces.

**TEMPLATE METHOD**

**Intent.** That pattern defines the skeleton of an algorithm in the superclass but let’s subclasses override specific steps of the algorithm without changing its structure.

**Solution.** The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single *template method.* The steps may either be abstract or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

*Applicability*

* **Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.** The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.
* **Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.** When you turn such an algorithm into a template method, you can also pull up the steps with similar implementations into a superclass, eliminating code duplication. Code that varies between subclasses can remain in subclasses.

*How to implement*

* Analyze the target algorithm to see whether you can break it into steps. Consider which steps are common to all subclasses and which ones will always be unique.
* Create the abstract base class and declare the template method and a set of abstract methods representing the algorithm’s steps. Outline the algorithm’s structure in the template method by executing corresponding steps. Consider making the template method final to prevent subclasses from overriding it.
* It’s okay if all the steps end up being abstract. However, some steps might benefit from having a default implementation. Subclasses don’t have to implement those methods.
* Think of adding hooks between the crucial steps of the algorithm.
* For each variation of the algorithm, create a new concrete subclass. It must implement all the abstract steps but may also override some of the optional ones.

*Pros*

* You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
* You can pull the duplicate code into a superclass.

*Cons*

* Some clients may be limited by the provided skeleton of an algorithm.
* You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.
* Template methods tend to be harder to maintain the more steps they have.

**VISITOR**

**Intent.** This pattern lets you separate algorithms from the objects on which they operate.

*Applicability*

* **Use it when you need to perform an operation on all elements of a complex object structure (for example, an object tree).** This pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation, which correspond to all target classes.
* **Use it to clean up the business logic of auxiliary behaviors.** This pattern lets you make the primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes.
* **Use it when a behavior makes sense only in some classes of a class hierarchy, but not in others.** You can extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty.

*How to implement*

* Declare the visitor interface with a set of “visiting” methods, one per each concrete element class that exists in the program.
* Declare the element interface. If you’re working with an existing element class hierarchy, add the abstract “acceptance” method to the base class of the hierarchy. This method should accept a visitor object as an argument.
* Implement the acceptance methods in all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object which matches the class of the current element.
* The element classes should only work with visitors via the visitor interface. Visitors, however, must be aware of all concrete element classes, referenced as parameter types of the visiting methods.
* For each behavior that can’t be implemented inside the element hierarchy, create a new concrete visitor class, and implement all of the visiting methods.

You might encounter a situation where the visitor will need access to some private members of the element class. In this case, you can either make these fields or methods public, violating the element’s encapsulation, or nest the visitor class in the element class. The latter is only possible if you’re lucky to work with a programming language that supports nested classes.

* The client must create visitor objects and pass them into elements via “acceptance” methods.

*Pros*

* Open/Closed Principle. You can introduce a new behavior that can work with objects of different classes without changing these classes.
* Single Responsibility Principle. You can move multiple versions of the same behavior into the same class.
* A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.

*Cons*

* You need to update all visitors each time a class gets added to or removed from the element hierarchy.
* Visitors might lack the necessary access to the private fields and methods of the elements that they’re supposed to work with.

**COLLECTIONS**

Is a framework that provides an architecture to store and manipulate the group of objects. This framework can achieve all the operations that you perform on a data such as searching, sorting, provinsertion, manipulation, and deletion.

Collections provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

*ITERATOR INTERFACE*

Iterator interface provides the facility of iterating the elements in a forward direction only.

*ITERABLE INTERFACE*

The iterable interface is the root interface for all the collection classes. The Collection interface extends the iterable interface and therefore all the subclasses of Collection interface also implement the iterable interface.

*COLLECTIONS INTERFACE*

Is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

*LIST INTERFACE*

Is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are:

* ArrayList.

This uses dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

* LinkedList.

It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

* Vector.

Uses a dynamic array to store the data elements. It is like ArrayList. However, it is synchronized and contains many methods that are not part of Collection framework.

* Stack.

Is the subclass of Vector. It implements the last-in-first-out data structure, i.e. Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

* Queue interface.

Maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

* PriorityQueue.

It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn’t allow null values to be stored in the queue.

* Deque interface.

Deque interface extends the queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

* ArrayDeque.

It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends. ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

* Set interface.

Is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn’t allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

* HashSet

It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

* LinkedHashSet.

Represents the LinkedList implementation of Set interface. It extends the HashSet class and implements Set interface. Like HashSet, it also contains unique elements. It maintains the insertion order and permits null elements.

* SortedSet interface.

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

* TreeSet.

Implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

*JAVA MAP INTERFACE*

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A map contains unique keys.

A map is useful if you must search, update or delete elements on the basis of a key.

A map doesn’t allow you duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn’t allow any null key or value.

* HashMap.

It allows us to store the null elements as well, but there should be only one null key, contains values based on the key, is non synchronized, maintains no order. The initial default capacity is 16 with a load factor of 0.75.

* LinkedHashMap.

This class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

* TreeMap.

It provides an efficient means of storing key-value pairs in sorted order.

Maintains ascending order, implements the NavigableMap interface and extends AbstractMap class.

* Hashtable.

Implements hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface. A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based in the key.

**Summarize of Collections.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Main collection classes** | **D** | **O** | **S** | **TS** |
| **ArrayList** | **Yes** | **Yes** | **No** | **No** |
| **LinkedList** | **Yes** | **Yes** | **No** | **No** |
| **Vector** | **Yes** | **Yes** | **No** | **Yes** |
| **HashSet** | **No** | **No** | **No** | **No** |
| **LinkedHashSet** | **No** | **Yes** | **No** | **No** |
| **TreeSet** | **No** | **Yes** | **Yes** | **No** |
| **HashMap** | **No** | **No** | **No** | **No** |
| **LinkedHashMap** | **No** | **Yes** | **No** | **No** |
| **Hashtable** | **No** | **No** | **No** | **Yes** |
| **TreeMap** | **No** | **Yes** | **Yes** | **No** |

* **D:** Duplicate elements is allowed?
* **O:** Elements are ordered?
* **S:** Elements are sorted?
* **TS:** The collection is thread-safe?

So, in conclusion:

* All lists allow duplicate elements which are ordered by index.
* All sets and maps do not allow duplicate elements.
* All list elements are not sorted.
* Generally, sets and maps do not sort its elements, except TreeSet and TreeMap which sort elements by natural order or by a comparator.
* Generally, elements within sets and maps are not ordered, except for:

LinkedHashSet and LinkedHashMap have elements ordered by insertion order.

TreeSet and TreeMap have elements ordered by natural order or by a comparator.

* There are only two collections are thread-safe: Vector and Hastable. The rest is not thread-safe.