Software Design and Algorithms



design patterns

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Contents

[1. design patterns 4](#_Toc64220392)

[1.1. history of patterns 4](#_Toc64220393)

[1.2. types of patterns 5](#_Toc64220394)

[2. creational design patterns 5](#_Toc64220395)

[2.1. abstract factory 5](#_Toc64220396)

[2.1.1. **Abstract factory structure** 6](#_Toc64220397)

[2.1.2. **Abstract factory example** 7](#_Toc64220398)

[2.1.3. **When to apply, pros and cons** 8](#_Toc64220399)

[2.2. singleton 9](#_Toc64220400)

[2.2.1. **Singleton Structure** 9](#_Toc64220401)

[2.2.2. **Singleton Example** 10](#_Toc64220402)

[2.2.3. **When to apply, pros and cons** 10](#_Toc64220403)

[2.3. builder 11](#_Toc64220404)

[2.3.1. **Builder Structure** 11](#_Toc64220405)

[2.3.2. Builder Example 14](#_Toc64220406)

[2.3.3. **When to apply, pros and cons** 15](#_Toc64220407)

[3. structural design patterns 16](#_Toc64220408)

[3.1. façade 16](#_Toc64220409)

[3.1.1. Façade structure 16](#_Toc64220410)

[3.1.2. Façade example 17](#_Toc64220411)

[3.1.3. When to apply, pros and cons 18](#_Toc64220412)

[3.2. decorator 19](#_Toc64220413)

[3.2.1. **Decorator structure** 19](#_Toc64220414)

[3.2.2. **Decorator example** 21](#_Toc64220415)

[3.2.3. **When to apply, pros and cons** 22](#_Toc64220416)

[3.3. proxy 22](#_Toc64220417)

[3.3.1. **Proxy structure** 23](#_Toc64220418)

[3.3.2. **Proxy example** 24](#_Toc64220419)

[3.3.3. **When to apply, pros and cons** 25](#_Toc64220420)

[4. behavioral design patterns 26](#_Toc64220421)

[4.1. template method 26](#_Toc64220422)

[4.1.1. **Template Method structure** 26](#_Toc64220423)

[4.1.2. **Template Method example** 28](#_Toc64220424)

[4.1.3. **When to apply, pros and cons** 29](#_Toc64220425)

[4.2. strategy 29](#_Toc64220426)

[4.2.1. **Strategy Structure** 30](#_Toc64220427)

[4.2.2. **Strategy Example** 31](#_Toc64220428)

[4.2.3. **When to apply, pros and cons** 32](#_Toc64220429)

[4.3. visitor 33](#_Toc64220430)

[4.3.1. **Visitor structure** 33](#_Toc64220431)

[4.3.2. **Visitor example** 35](#_Toc64220432)

[4.3.3. **When to apply, pros and cons** 36](#_Toc64220433)

[4.4. command 37](#_Toc64220434)

[4.4.1. **Command real life example** 38](#_Toc64220435)

[4.4.2. **Command structure** 38](#_Toc64220436)

[4.4.3. **Command example** 40](#_Toc64220437)

[4.4.4. **When to apply, pros and cons** 42](#_Toc64220438)

[Conclusion 43](#_Toc64220439)

# design patterns

Design pattern is a commonly used solution for a specific problem in software design. Unlike developed functions or libraries, pattern cannot be just copied to the code. It’s not an already made piece of software, but a general conception of solution for the problem, which has to be adjusted and implemented in your software.

Patterns are often misconcepted as algorithms, since both concepts represent common solutions of some known issues. But algorithm is just a defined set of actions unlike pattern, which is a high-level description of a solution. And different programs may have their own implementation of this solution.

There is an analogy for better understanding the difference between these two concepts. Just imagine that algorithm is a cooking recipe with predefined steps and measures while design pattern is a blueprint or scheme of the solution without any direct steps or instructions.

## history of patterns

Who has invented patterns? It’s a good but not quite accurate question, since patterns are not being invented. More likely, that they are being explored. Patterns are not extraordinary solutions, but opposite. Most typical, often used solutions for common problems.

Historically the concept of patterns was not about software design. It came from a book about buildings. It was firstly described in book “A Pattern Language. Towns. Buildings. Construction” as a language of environment design. Here patterns were used for architectural questions: how high windows should be, how many floors should be in the building, how much place should be taken for trees and lawn.

This idea was adopted by authors Erich Gamma, Richard Helm, Ralph Johnson and John Vissides. In 1994 they published a book “Design Patterns: Elements of Reusable Object-Oriented Software”, which consisted of 23 patterns for different commonly occurring software design problems. Long title of the patterns book was hard to remember and soon they started calling it “Book by the gang of four” and later just “GoF book”. Since then, dozens of  object-oriented design patterns were discovered.

## types of patterns

There are several kinds of patterns:

**Creational patterns –**provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

**Structural patterns –**are related to class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

**Behavioral patterns –** are used for efficient and safe communication between objects and distribution of responsibility among them. For this purpose, both inheritance and composition-based mechanisms can be used.

Patterns are software development tools. Like real-world tools, that you can simply buy in a hardware store, they have specific purpose and are not equally useful in different scenarios.

Some of them are similar to a trusty hummer, that has very wide range of use. Others can be associated with very specific measurement tool, which is great to have on rare occasion.

We are not going to cover all gang of four design patterns. We will target the most common and most useful. And by the way, most of them are not quite easy to implement in plain JavaScript and therefore we will use Typescript in examples.

# creational design patterns

Creational patterns as we have already mentioned, provide various object creation mechanisms. We are going to cover three of them: abstract factory, singleton and builder.

## abstract factory

The Abstract Factory  or in simple words – factory of factories.

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

### **Abstract factory structure**

Diagram

Description automatically generated

1. Abstract Products declare interfaces for a set of distinct but related products which make up a product family as chair or sofa.
2. Concrete Products are various implementations of abstract products, grouped by variants. In this case we implement each abstract product (chair or sofa) in all given variants (Victorian or Modern).
3. The Abstract Factory interface declares a set of methods for creating each of the abstract products.
4. Concrete Factories implement creation methods of the abstract factory. Each concrete factory creates only those product variants, that corresponds to a specific variant of products.
5. Signatures of concrete factories creation methods must return corresponding abstract products. In this case the client code that uses a factory doesn’t get coupled to the specific variant of the product. The Client can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

### **Abstract factory example**

Here we have 2 kinds of products: door and door fitting master for each type of doors.

interface Door {  
 getDescription();  
}  
  
class WoodenDoor implements Door {  
 public getDescription() {  
 return 'I am a wooden door';  
 }  
}  
  
class IronDoor implements Door {  
 public getDescription() {  
 return 'I am a iron door';  
 }  
}

interface DoorFittingExpert {  
 getDescription();  
}  
  
class Welder implements DoorFittingExpert {  
 public getDescription() {  
 return 'I can only fit iron doors';  
 }  
}  
  
class Carpenter implements DoorFittingExpert {  
 public getDescription() {  
 return 'I can only fit wooden doors';  
 }  
}

We declare abstract and concrete products.

Next, our abstract factory that would allow us to create a family of related objects:

* wooden door and a carpenter from wooden door factory
* iron door and a welder from iron door factory

As you can see, the wooden doors factory contains carpenter's creation and a wooden door, and iron doors factory contains an iron door and a welder.

interface DoorFactory {  
 makeDoor(): Door;  
 makeFittingExpert(): DoorFittingExpert;  
}  
  
class WoodenDoorFactory implements DoorFactory {  
 public makeDoor(): Door {  
 return new WoodenDoor();  
 }  
  
 public makeFittingExpert(): DoorFittingExpert {  
 return new Carpenter();  
 }  
}  
  
class IronDoorFactory implements DoorFactory {  
 public makeDoor(): Door {  
 return new IronDoor();  
 }  
  
 public makeFittingExpert(): DoorFittingExpert {  
 return new Welder();  
 }  
}

In this case we make sure, that a client who would use a door factory will get a proper specialist for each of the created doors.

### **When to apply, pros and cons**

We should get benefits from usage of the Abstract Factory while working with various families of related products. In this case we are removing the dependency on concrete classes of these products and allowing future extensibility

The abstract factory incapsulates the details of object creation. But client code can still work with all types of created objects, since their interface is initially defined.

Pros

* ensures compatibility of products
* gets rid of coupling
* extracts the product creation code into one place. (Single responsibility)
* introducing new variants without breaking existing code. (Open/Closed)

Cons

* code becomes more complicated after introducing lots of new interfaces
* after extending abstract factory interface all concrete factories will need to be updated to implement it

## singleton

Singleton is a creational pattern, which means, that the class has only one instance and it is accessible from any part of application.

### **Singleton Structure**

Diagram

Description automatically generated

Singleton defines static method getInstance(), which returns the only instance of its class. The constructor of singleton should be hidden from client code. Calling the getInstance() method should be the only way of getting the Singleton object.

In other words, the Singleton 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.

Singleton solves two tasks at a time, which violates the single responsibility principle.

* guarantees the existence of single instance of class. It is often useful for accessing common resource, e.g., a database
* provide a global access point to that instance

### **Singleton Example**

All implementations of the Singleton in general do the same – they make default constructor private and create public static method, which controls the lifecycle of singleton object. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the saved object.

class President {  
 private static *president*: President;  
 private constructor() {}  
 public static getInstance(): President {  
 if (!President.*president*) {  
 President.*president* = new President();  
 }  
 return President.*president*;  
 }  
}  
  
const *president1* = President.getInstance();  
const *president2* = President.getInstance();  
  
*president1* === *president2* // true

The President is an excellent example of the Singleton pattern. This class doesn’t have a public constructor, so the only way to get its instance – is call getInstance() method. This method saves the first created object and will return it in further calls.

### **When to apply, pros and cons**

* if 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
* if you need stricter control over global variables

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
* pattern can hide bad design
* requires special handling in a multithreaded environment
* you will need to think of a creative way to mock the singleton

## builder

Builder is a creational design pattern that 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. In other words: Builder pattern allows to create different parts of object, avoiding overload of constructor. Builder pattern can be used, when object should be built with several parts, or if objects creation takes lots of steps and each of these steps should be configurable.

Think of it as dividing an object creation into several steps with different parameters.

### **Builder Structure**

Imagine a complex object that requires step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a huge constructor with lots of parameters.  Often most of the parameters will be unused, making the constructor calls  ugly.

The Builder pattern suggests you extract the object construction code out of its own class and move it to separate objects called *builders*. The pattern organizes object construction into a set of steps. To create an object, you execute a series of these steps on a builder object. The important part is that you do not need to call all the steps at a time. 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 in which to execute the building steps, while the builder provides the implementation for those steps.

Having a director class in your program is not necessary. You can always call the building steps in a specific order directly from the client code.

Diagram

Description automatically generated

1. The Builder interface declares product construction steps that are common to all types of builders.
2. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that do not follow the common interface.
3. Products are resulting objects. Products constructed by different builders do not have to belong to the same class hierarchy or interface.
4. The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.
5. The Client must associate one of the builder objects with the director. Usually, it is done just once, via parameters of the director’s constructor. Then the director uses that builder object for all further construction. However, there is an alternative approach when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

### Builder Example

This example of the Builder pattern illustrates how you can reuse the same object construction code when building diverse types of products, such as cars, and create the corresponding manuals for them.

class Car {}  
  
interface Builder {  
 reset();  
 setSeats(n: Number);  
 setEngine(n: String);  
 setTripComputer(n: Boolean);  
 setGPS(n: Boolean);  
}  
  
class CarBuilder implements Builder {  
 private car: Car;  
  
 constructor() {  
 this.reset();  
 }  
  
 reset() { this.car = new Car(); }  
 setSeats() {}  
 setEngine() {}  
 setTripComputer() {}  
 setGPS() {}  
  
 getProduct(): Car {  
 const product = this.car;  
 this.reset();  
 return product;  
 }  
}

A car is a complex object that can be constructed in a hundred diverse ways. Instead of bloating the Car class with a huge constructor, we extract the car assembly code into a separate car builder class. This class has a set of methods for configuring various parts of a car.

class Director {  
 private builder: Builder;  
  
 setBuilder(b: Builder) {  
 this.builder = b;  
 }  
  
 makeSportCar(b: Builder = this.builder) {  
 b.reset();  
 b.setSeats(2);  
 b.setEngine('V12');  
 b.setTripComputer(true);  
 b.setGPS(true);  
 }  
}  
  
const *director* = new Director();  
const *builder* = new CarBuilder();  
*director*.makeSportCar(*builder*);

If the client code needs to assemble a special, fine-tuned model of a car, it can work with the builder directly. On the other hand, the client can delegate the assembly to the director class, which knows how to use a builder to construct several of the most popular models of cars.

### **When to apply, pros and cons**

* use to get rid of a “telescopic constructor” – the pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you do not have to cram dozens of parameters into your constructors anymore
* use to construct complex objects – a builder does not expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result

Pros

* constructing objects step-by-step, defer construction steps or run steps recursively
* reusing the same construction code when building various representations of products
* isolating complex construction code from the business logic of the product

Cons

* increases complexity of the code since the pattern requires creating multiple new classes
* client will be bound to concrete builder classes, since builders interface does not have a method for fetching the result of the construction

# structural design patterns

Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient. To achieve that, both inheritance and composition can be applied.

## façade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

### Façade structure

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you will have your first facade.

When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you a simple voice interface to the ordering system, payment gateways, and various delivery services.

Diagram

Description automatically generated

1. The Facade provides convenient access to a particular part of the subsystem’s functionality. It knows where to direct the client’s request and how to operate all the moving parts.
2. An Additional Facade class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.
3. The Complex Subsystem consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem’s implementation details, such as initializing objects in the correct order and supplying them with data in the proper format. Subsystem classes aren’t aware of the facade’s existence. They operate within the system and work with each other directly.
4. The Client uses the facade instead of calling the subsystem objects directly.

### Façade example

Here we have a Computer class with lots of unclear methods.

class Computer {  
 public getElectricShock() { return 'Ouch!'; }  
 public makeSound() { return 'Beep beep!'; }  
 public showLoadingScreen() { return 'Loading..'; }  
 public bam() { return 'Ready to be used!'; }  
 public closeEverything() { return 'Zzzzzz bup'; }  
 public sooth() { return 'shhshh'; }  
}

And a façade with 2 straightforward methods turnOn and turnOff.

class ComputerFacade {  
 protected computer: Computer;  
  
 constructor(c: Computer) {  
 this.computer = c;  
 }  
  
 public turnOn() {  
 this.computer.getElectricShock();  
 this.computer.makeSound();  
 this.computer.showLoadingScreen();  
 this.computer.bam();  
 }  
  
 public turnOff() {  
 this.computer.closeEverything();  
 this.computer.sooth();  
 }  
}  
  
const *computer* = new ComputerFacade(new Computer());  
*computer*.turnOn();  
*computer*.turnOff();

### When to apply, pros and cons

* when need a limited but straightforward interface to a complex subsystem
* if want to structure a subsystem into layers: create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades

Pros

* you can isolate your code from the complexity of a subsystem

Cons

* a facade can become a god object coupled to all classes of an app

## decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

### **Decorator structure**

Imagine you are developing a coffee brewing system. Initially only black coffee brewing is required. Then single black coffee option become not enough. Now you need to implement white coffee and black with cream. Therefore, you reasonably made a Coffee abstract class and implemented it in black, white and black with cream classes. Soon you got request for a white coffee with cream and for coffees with different amount of sugar. Thus you came to the point, that inheritance is not the case.

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: it is static, and it does not let a class inherit behaviors of multiple classes at the same time.

So, apparently, we need a composition here.

“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.

When does a simple wrapper become the real decorator? As I mentioned, 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 allow you to cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

Diagram

Description automatically generated

1. The Component declares the common interface for both wrappers and wrapped objects.
2. Concrete Component is a class of objects being wrapped. It defines the basic behaviour, which can be altered by decorators.
3. The Base Decorator class has a field for referencing a wrapped object. The field’s type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
4. Concrete Decorators define extra behaviours that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behaviour either before or after calling the parent method.
5. The Client can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

### **Decorator example**

interface Coffee {  
 getCost();  
 getDescription();  
}  
  
class SimpleCoffee implements Coffee {  
 public getCost() { return 10; }  
 public getDescription() { return 'Simple coffee' }  
}  
  
class CoffeeDecorator implements Coffee {  
 protected wrappee: Coffee;  
  
 constructor(coffee: Coffee) {  
 this.wrappee = coffee;  
 }  
  
 public getCost() { return this.wrappee.getCost(); }  
 public getDescription() { return this.wrappee.getDescription(); }  
}

The decorators and the data source class implement the same interface, which makes them all interchangeable in the client code.

class MilkCoffeeDecorator extends CoffeeDecorator {  
 public getCost(): any { return this.wrappee.getCost() + 2; }  
 public getDescription(): any { return this.wrappee.getDescription() + 'with milk'; }  
}  
  
class WhipCoffeeDecorator extends CoffeeDecorator {  
 public getCost(): any { return this.wrappee.getCost() + 3; }  
 public getDescription(): any { return this.wrappee.getDescription() + 'and with whip'; }  
}  
  
let *someCoffee* = new SimpleCoffee();  
*someCoffee*.getCost(); // 10  
*someCoffee*.getDescription(); // Simple Coffee  
*someCoffee* = new MilkCoffeeDecorator(*someCoffee*);  
*someCoffee*.getCost(); // 12  
*someCoffee* = new WhipCoffeeDecorator(*someCoffee*);  
*someCoffee*.getDescription(); // Simple Coffee with milk and whip

### **When to apply, pros and cons**

* when you need to be able to assign extra behaviours to objects at runtime without breaking the code that uses these objects – The Decorator 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
* when it’s awkward or not possible to extend an object’s behaviour using inheritance. For example, you are working with a library, and you don’t have access to alter its behavior, but what you can do is extending it with Decorator

Pros

* extending an object’s behavior without making a new subclass
* adding or remove responsibilities from an object at runtime
* combining several behaviors by wrapping an object into multiple decorators
* dividing a monolithic class that implements many possible variants of behavior into several smaller classes

Cons

* hard to remove a specific wrapper from the wrappers stack
* hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack
* initial configuration code of layers might look overcomplicated

## proxy

Proxy is a structural design pattern that 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.

In other words by using proxy, one class provides functionality of another class.

Unlike Decorator, a *Proxy* usually manages the life cycle of its service object on its own.

### **Proxy structure**

Diagram

Description automatically generated

1. The Service Interface declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.
2. The Service is a class that provides some useful business logic.
3. The Proxy class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.
4. Usually, proxies manage the full lifecycle of their service objects.
5. The Client should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

### **Proxy example**

// Presume we have an input filed with an ID of inputname:  
const *el* = `<input type="text" id="inputname" value="" />`;  
  
// We also have a JS object named myUser with  
// an id property which references this input  
const *myUser* = {  
 id: 'inputname',  
 name: ''  
};  
  
// Our first objective is to update myUser.name  
// when a user changes the input value. This can be achieved  
// with an onchange event handler on the field:  
(function *inputChange*(myObject) {  
 if (!myObject || !myObject.id) { return; }  
  
 const input = *document*.getElementById(myObject.id);  
 input.addEventListener('onchange', function (e) {  
 myObject.name = input.value;  
 });  
})(*myUser*);

There is a JavaScript native class Proxy, which allows to catch get/set calls of objects and perform additional actions. Here we use this feature to create a primitive but operating bi-directional data-binding.

// create proxy  
const *myUserProxy* = new *Proxy*(*myUser*, {  
 set: function(target, prop, newValue) {  
 if (prop === 'name' && target.id) {  
 // update object property  
 target[prop] = newValue;  
  
 // update input field value  
 *document*.getElementById(target.id).value = newValue;  
 return true;  
 }  
  
 return false;  
 }  
});  
  
// set a new name  
*myUserProxy*.name = 'Craig';  
*console*.log(*myUserProxy*.name); // Craig  
*console*.log(*document*.getElementById('inputname').value);

### **When to apply, pros and cons**

* lazy initialization (virtual proxy) of a heavyweight service object that wastes system resources by being always up, but is needed 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) – for letting 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 of 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

Pros

* controlling the service object without clients knowing about it
* managing 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
* introducing new proxies without changing the service or clients

Cons

* overcomplicated code since you need to introduce a lot of new classes
* the response from the service might get delayed

# behavioral design patterns

Behavioral design patterns are related to algorithms and the assignment of responsibilities between objects. They can be based on both – inheritance and composition.

## template method

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but allows subclasses to override specific steps of the algorithm without changing its structure.

### **Template Method structure**

The Template Method pattern suggests 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).

There are three types of steps:

* *abstract steps* must be implemented by every subclass
* *optional steps* already have default implementation, but can be overridden if needed
* *hooks* are optional steps with an empty body. A template method would work even if a hook is not overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points for an algorithm.
* Diagram

  Description automatically generated

1. The Abstract Class declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared abstract or have some default implementation.
2. Concrete Classes can override all the steps, but not the template method itself.

Template method for making tea and coffee:

boilWater();

brew();

pourInCup()

addCondiments();

### **Template Method example**

abstract class Builder {  
 // Template method  
 public build() {  
 this.test();  
 this.lint();  
 this.assemble();  
 this.deploy();  
 }  
  
 abstract test();  
 abstract lint();  
 abstract assemble();  
 abstract deploy();  
}

Note that final keyword used to prevent overriding of the build method and thus changing order of steps for a given algorithm, is not supported in TS at the moment.

class AndroidBuilder extends Builder {  
 public test() { return 'Running android tests'; }  
 public lint() { return 'Linting android code'; }  
 public assemble() { return 'Assembling android build'; }  
 public deploy() { return 'Deploying android build to server'; }  
}  
  
class IosBuilder extends Builder {  
 public test() { return 'Running ios tests'; }  
 public lint() { return 'Linting ios code'; }  
 public assemble() { return 'Assembling ios build'; }  
 public deploy() { return 'Deploying ios build to server'; }  
}  
  
const *androidBuilder* = new AndroidBuilder();  
*androidBuilder*.build();  
// Running android tests  
// Linting android code  
// Assembling android build  
// Deploying android build to server  
  
const *iosBuilder* = new IosBuilder();  
*androidBuilder*.build();  
// Running ios tests  
// Linting ios code  
// Assembling ios build  
// Deploying ios build to server

### **When to apply, pros and cons**

* 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
* 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 move the steps with similar implementations into a superclass, eliminating code duplication. Code that varies between subclasses can remain in subclasses

Pros

* letting clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm
* move 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

## strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. Allows to switch between algorithms or strategies depending on situation.

### **Strategy Structure**

The Strategy pattern suggests you take a class that does something specific in a lot of diverse 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 is not responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context does not 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.

Diagram

Description automatically generated

1. The Context keeps a reference to one of the concrete strategies and communicates with this object only via the strategy interface.
2. The Strategy interface is unified for all concrete strategies. It declares a method the context uses to execute a strategy.
3. Concrete Strategies implement different variations of an algorithm the context uses.
4. The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context does not know what type of strategy it works with or how the algorithm is executed.
5. The Client creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

### **Strategy Example**

// The strategy interface declares operations common  
// to all supported versions of some algorithm.  
interface Strategy {  
 execute(a: number, b: number): number;  
}  
// Concrete strategies implement the algorithm while following  
// the base strategy interface. The interface makes them  
// interchangable in the context.  
class ConcreteStrategyAdd implements Strategy {  
 execute(a, b) {  
 return a + b;  
 }  
}  
class ConcreteStrategySubstract implements Strategy {  
 execute(a, b) {  
 return a - b;  
 }  
}  
class ConcreteStrategyMultiply implements Strategy {  
 execute(a, b) {  
 return a \* b;  
 }  
}

We have common interface and 3 concrete classes implementing this interface. The Context in not aware of what specific strategy he works with. Context knows that it can send the execute message. Thus the client decides when and which strategy should be used. It can switch them on the fly, getting different behavior, without changing the interface.

// The context defines the interface of interest to clients.  
class Context {  
 private strategy: Strategy;  
  
 setStrategy(s: Strategy) {  
 this.strategy = s;  
 }  
 // The context delegates some work to the strategy object  
 // instead of implementing multiple versions of the  
 // algorithm on its own.  
 executeStrategy(a: number, b: number) {  
 return this.strategy.execute(a, b);  
 }  
}  
  
let *ctx* = new Context();  
  
*ctx*.setStrategy(new ConcreteStrategyAdd());  
*ctx*.executeStrategy(5, 2); // 7

### **When to apply, pros and cons**

* 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 diverse ways
* 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
* 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
* when your class has a massive conditional operator that switches between different variants of the same algorithm

Pros

* swapping algorithms used inside an object at runtime
* isolating the implementation details of an algorithm from the code that uses it
* replacing inheritance with composition
* introducing new strategies without having to change the context

Cons

* if you have only a couple of algorithms and they rarely change, there is 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

## visitor

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

### **Visitor structure**

Imagine that your team develops an app which works with geographic information structured as one colossal graph. Each node of the graph may represent a complex entity such as a city, but also more granular things like industries, sightseeing areas, etc. The nodes are connected with others if there is a relation between the real objects that they represent. Under the hood, each node type is represented by its own class, while each specific node is an object.    
At some point, you got a task to implement exporting the graph into XML format. At first, the job seemed straightforward. You planned to add an export method to each node class and then leverage recursion to go over each node of the graph, executing the export method. The solution was simple and elegant: thanks to polymorphism, you were not coupling the code which called the export method to concrete classes of nodes. The Visitor pattern suggests you put the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes. The original object that has to perform the behavior is now passed to one of the visitor’s methods as an argument, providing the method access to all necessary data contained within the object.

Now, what if that behavior can be executed over objects of different classes? For example, in our case with XML export, the actual implementation will probably be a little bit different across various node classes. Thus, the visitor class may define not one, but a set of methods, each of which could take arguments of different types.

Diagram

Description automatically generated

1. The Visitor interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the program is written in a language that supports overloading, but the type of their parameters must be different.
2. Each Concrete Visitor implements several versions of the same behaviors, tailored for different concrete element classes.
3. The Element interface declares a method for “accepting” visitors. This method should have one parameter declared with the type of the visitor interface.
4. Each Concrete Element must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor’s method corresponding to the current element class. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.
5. The Client usually represents a collection or some other complex object (for example, a Composite tree). Usually, clients are not aware of all the concrete element classes because they work with objects from that collection via some abstract interface.

### **Visitor example**

The element interface declares an `accept` method that takes the base visitor interface as an argument. Each concrete element class must implement the `accept` method in such a way that it calls the visitor's method that corresponds to the element's class.

// The component interface declares an 'accept' method that  
// takes the base visitor interface as an argument.  
interface Shape {  
 move(x, y);  
 draw();  
 accept(v: Visitor);  
}  
// Each concrete component class must implement the 'accept' method  
class Dot implements Shape {  
 accept(v: Visitor) { v.visitDot(this); }  
 move(x, y) {}  
 draw() {}  
}  
class Circle implements Shape {  
 accept(v: Visitor) { v.visitCircle(this); }  
 move(x, y) {}  
 draw() {}  
}  
class Rectangle implements Shape {  
 accept(v: Visitor) { v.visitRectangle(this); }  
 move(x, y) {}  
 draw() {}  
}

The Visitor interface declares a set of visiting methods that correspond to element classes. The signature of a visiting method lets the visitor identify the exact class of the element that it's dealing with.

// The Visitor interface declares a set of visiting methods that  
// correspond to component classes.  
interface Visitor {  
 visitDot(d: Dot);  
 visitCircle(c: Circle);  
 visitRectangle(r: Rectangle);  
}  
class JSONExportVisitor implements Visitor {  
 visitDot(d: Dot) {  
 // Export the dot's ID and coordinates.  
 }  
 visitCircle(d: Circle) {  
 // Export the circle's ID, center coordinates and radius.  
 }  
 visitRectangle(d: Rectangle) {  
 // Export the rectangle's ID, left-top coordinates, width and height.  
 }  
}  
  
// The client code can run visitor operations over any set of  
// elements without figuring out their concrete classes. The  
// 'accept' operation directs a call to the appropriate operation  
// in the visitor object.  
const *allShapes* = [new Dot(), new Circle(), new Rectangle()];  
const *exportVisitor* = new JSONExportVisitor();  
  
*allShapes*.forEach(shape => shape.accept(*exportVisitor*));

Concrete visitors implement several versions of the same algorithm, which can work with all concrete element classes.

### **When to apply, pros and cons**

* when you need to perform an operation on all elements of a complex object structure (for example, an object tree). The Visitor pattern allows you to execute an operation over a set of objects with different classes by having a visitor object implementing several variants of the same operation, which correspond to all target classes
* to clean up the business logic of auxiliary behaviors. The pattern allows you to make the primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes
* 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

Pros

* introducing new behavior that can work with objects of different classes without changing these classes (Open/Closed)
* moving multiple versions of the same behavior into the same class (Single Responsibility)
* 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 are supposed to work with

## command

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation allows you to parameterize methods with different requests, delay or queue a request’s execution, and support undoable operations. In other words – it allows to incapsulate actions in objects. The main idea of Command pattern is to provide a way of separating client from receiver.

### **Command real life example**

Remote control example – we’ve got on and off buttons, client configures each of these buttons for particular commands. At the same time buttons are not aware of which command are they assigned to. Another example is a friendly waiter taking your order, writing it down on a piece of paper. Then he goes to the kitchen and sticks the order on the wall. After a while, the order gets to the chef, who reads it and cooks the meal accordingly. The cook places the meal on a tray along with the order. The waiter discovers the tray, checks the order to make sure everything is as you wanted it, and brings everything to your table.

### **Command structure**

Good software design is often based on the *principle of separation of concerns*, which usually results in breaking an app into layers. The most common example: a layer for the graphical user interface and another layer for the business logic. The GUI layer is responsible for rendering a beautiful picture on the screen, capturing any input and showing results of what the user and the app are doing. However, when it comes to doing something important, like calculating the trajectory of the moon or composing an annual report, the GUI layer delegates the work to the underlying layer of business logic. In the code it might look like this: a GUI object calls a method of a business logic object, passing it some arguments. This process is usually described as one object sending another a *request*.

The Command pattern suggests that GUI objects should not send these requests directly. Instead, you should extract all the request details, such as the object being called, the name of the method and the list of arguments into a separate *command* class with a single method that triggers this request.

Command objects serve as links between various GUI and business logic objects. From now on, the GUI object does not need to know what business logic object will receive the request and how it will be processed. The GUI object just triggers the command, which handles all the details.

The next step is make your commands to implement the same interface. Usually it has just a single execution method that takes no parameters. This interface lets you use various commands with the same request sender, without coupling it to concrete classes of commands. As a bonus, now you can switch command objects linked to the sender, effectively changing the sender’s behavior at runtime.

Diagram

Description automatically generated

1. The Sender class (aka *invoker*) is responsible for initiating requests. This class must have a field for storing a reference to a command object. The sender triggers that command instead of sending the request directly to the receiver. Note that the sender is not responsible for creating the command object. Usually, it gets a pre-created command from the client via the constructor.
2. The Command interface usually declares just a single method for executing the command.
3. Concrete Commands implement various kinds of requests. A concrete command is not supposed to perform the work on its own, but rather to pass the call to one of the business logic objects. However, to simplify the code, these classes can be merged.
4. Parameters required to execute a method on a receiving object can be declared as fields in the concrete command. You can make command objects immutable by only allowing the initialization of these fields via the constructor
5. The Receiver class contains some business logic. Almost any object may act as a receiver. Most commands only handle the details of how a request is passed to the receiver, while the receiver itself does the actual work.
6. The Client creates and configures concrete command objects. The client must pass all of the request parameters, including a receiver instance, into the command’s constructor. After that, the resulting command may be associated with one or multiple senders.

### **Command example**

In this example the Command pattern handles command execution history, allows to cancel them if necessary. Commands, altering the editor state (e.g. paste command), save a copy of editor state before execution.

abstract class Command {  
 protected app: Application;  
 protected editor: Editor;  
 protected backup: string;  
  
 constructor (app: Application, editor: Editor) {  
 this.app = app;  
 this.editor = editor;  
 }  
  
 saveBackup() {  
 this.backup = this.editor.text;  
 }  
  
 undo() {  
 this.editor.text = this.backup;  
 }  
  
 abstract execute();  
}

Copies of executed commands are placed into command history, where they can be accessed for cancelling.  Classes of UI, command history and others don’t depend on concrete commands, since they work via common interface with them. Thus, new commands can be added to application without changing existing code.

class CopyCommand extends Command {  
 execute() {  
 this.app.clipboard = this.editor.getSelection();  
 }  
}  
class PasteCommand extends Command {  
 execute() {  
 this.saveBackup();  
 this.editor.replaceSelection(this.app.clipboard);  
 }  
}  
class CommandHistory {  
 private history: Command[];  
  
 push(c: Command) { this.history.push(c); }  
 pop(): Command { return this.history[this.history.length -1]; }  
}  
class Editor {  
 text: string;  
  
 getSelection() { return 'some selection'; }  
 replaceSelection(clipboard) { return `some ${clipboard} selection`; }  
}

The application class sets up object relations. It acts as a sender: when something needs to be done, it creates a command object and executes it.

class Application {  
 clipboard: string;  
 editor: Editor;  
 activeEditor: Editor;  
 history: CommandHistory;  
  
 bindComands() {  
 shortcuts.onkeypress('Ctrl+C', () => {  
 return this.executeCommand(new CopyCommand(this, this.editor));  
 });  
 shortcuts.onkeypress('Ctrl+V', () => {  
 return this.executeCommand(new PasteCommand(this, this.editor));  
 });  
 }  
  
 executeCommand(command: Command) {  
 this.history.push(command);  
 command.execute();  
 }  
  
 undo() {  
 const command = this.history.pop();  
 command.undo();  
 }  
}

Take the most recent command from the history and run its undo method. Note that we don't know the class of that command. But we don't have to, since the command knows how to undo its own action.

### **When to apply, pros and cons**

* 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 up a lot of interesting uses: you can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc. Here’s an example: you’re developing a GUI component such as a context menu, and you want your users to be able to configure menu items that trigger operations when an end user clicks an item
* 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
* 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 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.

Pros

* decoupling classes that invoke operations from classes that perform these operations (Single Responsibility)
* introducing new commands into the app without breaking existing client code (Open/Closed)
* implementing undo/redo
* implementing deferred execution of operations
* assembling 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

Conclusion

You may successfully code without having an idea of what patterns are. Moreover, you might have implemented some of them without even noticing. But conscious usage of tool differs is an important step from a beginner level to professional. You may hammer a nail and you may hit it with a drill as well, if you make enough effort. Professional developer understands, that it’s not a purpose of a drill.

Software development would be much easier without changes at all. But changes are everywhere: requirements, environment, hardware and so on. The main purpose of software development is creation of systems, that will allow to localize and minimize negative influence of such changes.

That is why we should write reusable code with minimal dependency on other parts of the program and use SRP (single responsibility) and OCP (open-closed) principles. That is why we should use abstractions and depend on them rather than on realization details. And also prefer composition over inheritance.

Using design patterns, the code, which conducts these approaches and principles, can be achieved much easier. For example, Strategy pattern allows to write code, which is opened for extension, but closed for modification. Observer pattern allows create loose coupled code and so on.