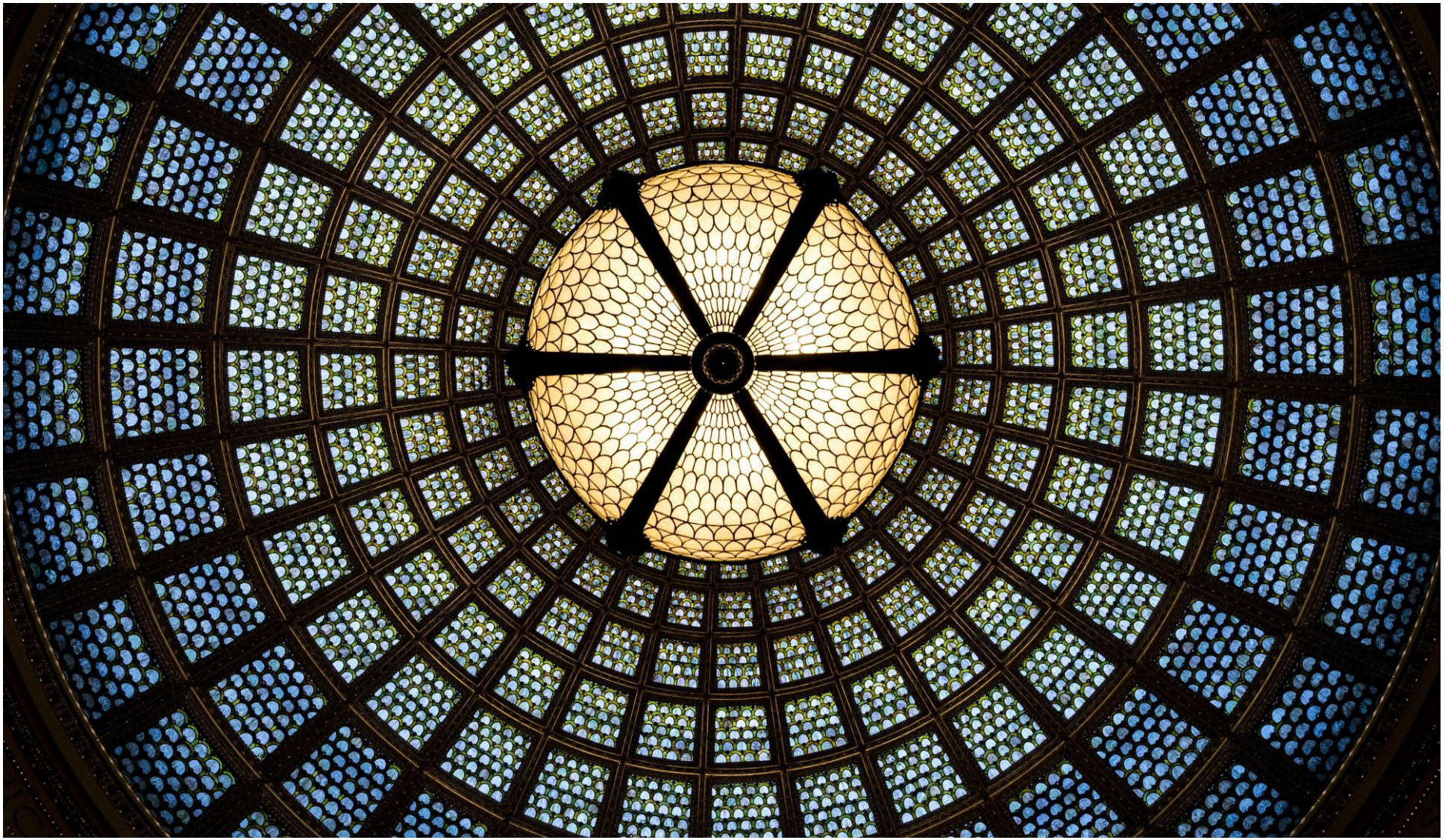
JUNE 22, 2022 / #JAVASCRIPT

JavaScript Design Patterns – Explained with Examples

Germán Cocca



useful.

We'll also go through some of the most popular design patterns out there and give examples for each of them. Let's go!

Table of Contents

What Are Design Patterns?

Design patterns were popularized by the book "Design Patterns: Elements of Reusable Object-Oriented Software", published in 1994 by a group of four C++ engineers.

The book explores the capabilities and pitfalls of object-oriented programming, and describes 23 useful patterns that you can implement to solve common programming problems.

These patterns are **not algorithms or specific implementations**. They are more like **ideas, opinions, and abstractions** that can be useful in certain situations to solve a particular kind of problem.

for our problem.

This being said, keep in mind these patterns were thought up with OOP C++ programming in mind. When it comes to more modern languages like JavaScript or other programming paradigms, these patterns might not be equally useful and might even add unnecessary boilerplate to our code.

Nevertheless, I think it's good to know about them as general programming knowledge.

Side comment: If you're not familiar with programming paradigms or OOP, I recently wrote two articles about those topics. 😉

Anyway... Now that we've gotten the introduction out of the way, design patterns are classified into three main categories: **creational, structural, and behavioral patterns**. Let's briefly explore each of them. 🧐

Creational Design Patterns

Creational patterns consist of different mechanisms used to create objects.

Singleton Pattern

We want to have some immutable single *point of truth* for our application.

Let's say for example we want to have all of our app's configuration in a single object. And we want to disallow any duplication or modification of that object.

Two ways of implementing this pattern are using object literals and classes:

```
const Config = {
  start: () => console.log('App has started'),
  update: () => console.log('App has updated'),
}

// We freeze the object to prevent new properties being added and existing properties being modified or removed
Object.freeze(Config)

Config.start() // "App has started"
Config.update() // "App has updated"

Config.name = "Robert" // We try to add a new key
console.log(Config) // And verify it doesn't work: { start: [Function: start], update: [Function: update] }
```

Using an object literal

```
    constructor() {}
    start(){ console.log('App has started') }
    update(){ console.log('App has updated') }
}

const instance = new Config()
Object.freeze(instance)
```

Using classes

Factory Method Pattern

The **Factory method** pattern provides an interface for creating objects that can be modified after creation. The cool thing about this is that the logic for creating our objects is centralized in a single place, simplifying and better organizing our code.

This pattern is used a lot and can also be implemented in two different ways, via classes or factory functions (functions that return an object).

```
class Alien {
    constructor (name, phrase) {
```

```
    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
    sayPhrase = () => console.log(this.phrase)
}

const alien1 = new Alien("Ali", "I'm Ali the alien!")
console.log(alien1.name) // output: "Ali"
```

Using classes

```
function Alien(name, phrase) {
    this.name = name
    this.phrase = phrase
    this.species = "alien"
}

Alien.prototype.fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")
Alien.prototype.sayPhrase = () => console.log(this.phrase)

const alien1 = new Alien("Ali", "I'm Ali the alien!")

console.log(alien1.name) // output "Ali"
console.log(alien1.phrase) // output "I'm Ali the alien!"
alien1.fly() // output "Zzzzzziiiiiinnnnnggggg"
```

Abstract Factory Pattern

The **Abstract Factory** pattern allows us to produce families of related objects without specifying concrete classes. It's useful in situations where we need to create objects that share only some properties and methods.

The way it works is by presenting an abstract factory the client interacts with. That **abstract factory** calls the corresponding **concrete factory** given the corresponding logic. And that concrete factory is the one that returns the end object.

Basically it just adds an abstraction layer over the factory method pattern, so that we can create many different types of objects, but still interact with a single factory function or class.

So let's see this with an example. Let's say we're modeling a system for a car company, which builds cars of course, but also motorcycles and trucks.

```
// We have a class or "concrete factory" for each vehicle type
class Car {
    constructor () {
        this.name = "Car"
        this.wheels = 4
```

```
class Truck {
    constructor () {
        this.name = "Truck"
        this.wheels = 8
    }
    turnOn = () => console.log("RRRRRRRRUUUUUUUUUMMMMMMMMM!!")
}

class Motorcycle {
    constructor () {
        this.name = "Motorcycle"
        this.wheels = 2
    }
    turnOn = () => console.log("sssssssssssssssssssssssssssshhhhhhhhhham!!")
}

// And and abstract factory that works as a single point of interaction for our clients
// Given the type parameter it receives, it will call the corresponding concrete factory
const vehicleFactory = {
    createVehicle: function (type) {
        switch (type) {
            case "car":
                return new Car()
            case "truck":
                return new Truck()
            case "motorcycle":
                return new Motorcycle()
            default:
                return null
        }
```

```
const truck = vehicleFactory.createVehicle("truck") // Truck { turnOn: [Function: turnOn], name: 'Truck', wheels:
const motorcycle = vehicleFactory.createVehicle("motorcycle") // Motorcycle { turnOn: [Function: turnOn], name: 'M
```

Builder Pattern

The **Builder** pattern is used to create objects in "steps". Normally we will have functions or methods that add certain properties or methods to our object.

The cool thing about this pattern is that we separate the creation of properties and methods into different entities.

If we had a class or a factory function, the object we instantiate will always have all the properties and methods declared in that class/factory. But using the builder pattern, we can create an object and apply to it only the "steps" we need, which is a more flexible approach.

This is related to object composition, a topic I've talked about here.

```
// We declare our objects
const bug1 = {
```

```
const bug2 = {
    name: "Martiniano Buggland",
    phrase: "Can't touch this! Na na na na..."
}

// These functions take an object as parameter and add a method to them
const addFlyingAbility = obj => {
    obj.fly = () => console.log(`Now ${obj.name} can fly!`)
}

const addSpeechAbility = obj => {
    obj.saySmthg = () => console.log(`${obj.name} walks the walk and talks the talk!`)
}

// Finally we call the builder functions passing the objects as parameters
addFlyingAbility(bug1)
bug1.fly() // output: "Now Buggy McFly can fly!"

addSpeechAbility(bug2)
bug2.saySmthg() // output: "Martiniano Buggland walks the walk and talks the talk!"
```

Prototype Pattern

The **Prototype** pattern allows you to create an object using another object as a blueprint, inheriting its properties and methods.

The end result is very similar to what we get by using classes, but with a little more flexibility since properties and methods can be shared between objects without depending on the same class.

```
// We declare our prototype object with two methods
const enemy = {
    attack: () => console.log("Pim Pam Pum!"),
    flyAway: () => console.log("Flyyyy like an eagle!")
}

// We declare another object that will inherit from our prototype
const bug1 = {
    name: "Buggy McFly",
    phrase: "Your debugger doesn't work with me!"
}

// With setPrototypeOf we set the prototype of our object
Object.setPrototypeOf(bug1, enemy)

// With getPrototypeOf we read the prototype and confirm the previous has worked
console.log(Object.getPrototypeOf(bug1)) // { attack: [Function: attack], flyAway: [Function: flyAway] }

console.log(bug1.phrase) // Your debugger doesn't work with me!
console.log(bug1.attack()) // Pim Pam Pum!
console.log(bug1.flyAway()) // Flyyyy like an eagle!
```

Structural patterns refer to how to assemble objects and classes into larger structures.

Adapter Pattern

The **Adapter** allows two objects with incompatible interfaces to interact with each other.

Let's say, for example, that your application consults an API that returns XML and sends that information to another API to process that information. But the processing API expects JSON. You can't send the information as it's received since both interfaces are incompatible. You need to *adapt it* first. 😉

We can visualize the same concept with an even simpler example. Say we have an array of cities and a function that returns the greatest number of habitants any of those cities have. The number of habitants in our array is in millions, but we have a new city to add that has its habitants without the million conversion:

```
// Our array of cities
const citiesHabitantsInMillions = [
    { city: "London", habitants: 8.9 },
    { city: "Rome", habitants: 2.8 },
```

```
// The new city we want to add
const BuenosAires = {
    city: "Buenos Aires",
    habitants: 3100000
}

// Our adapter function takes our city and converts the habitants property to the same format all the other cities
const toMillionsAdapter = city => { city.habitants = parseFloat((city.habitants/1000000).toFixed(1)) }

toMillionsAdapter(BuenosAires)

// We add the new city to the array
citiesHabitantsInMillions.push(BuenosAires)

// And this function returns the largest habitants number
const MostHabitantsInMillions = () => {
    return Math.max(...citiesHabitantsInMillions.map(city => city.habitants))
}

console.log(MostHabitantsInMillions()) // 8.9
```

Decorator Pattern

The **Decorator** pattern lets you attach new behaviors to objects by placing them inside wrapper objects that contain the behaviors. If you're somewhat familiar with React and higher order

Memo we can see that we're passing a component as a child to this HOC, and thanks to that this child component is able to access certain features.

In this example we can see that the ContextProvider component is receiving children as props:

```
import { useState } from 'react'
import Context from './Context'

const ContextProvider: React.FC = ({children}) => {

    const [darkModeOn, setDarkModeOn] = useState(true)
    const [englishLanguage, setEnglishLanguage] = useState(true)

    return (
        <Context.Provider value={{
            darkModeOn,
            setDarkModeOn,
            englishLanguage,
            setEnglishLanguage
        }} >
            {children}
        </Context.Provider>
    )
}
```

Then we wrap the whole application around it:

```
export default function App() {
  return (
    <ContextProvider>
      <Router>

        <ErrorBoundary>
          <Suspense fallback={<></>}>
            <Header />
          </Suspense>

          <Routes>
              <Route path='/' element={<Suspense fallback={<></>}><AboutPage /></Suspense>}/>

              <Route path='/projects' element={<Suspense fallback={<></>}><ProjectsPage /></Suspense>}/>

              <Route path='/projects/helpr' element={<Suspense fallback={<></>}><HelprProject /></Suspense>}/>

              <Route path='/projects/myWebsite' element={<Suspense fallback={<></>}><MyWebsiteProject /></Suspense

              <Route path='/projects/mixr' element={<Suspense fallback={<></>}><MixrProject /></Suspense>}/>

              <Route path='/projects/shortr' element={<Suspense fallback={<></>}><ShortrProject /></Suspense>}/>

              <Route path='/curriculum' element={<Suspense fallback={<></>}><CurriculumPage /></Suspense>}/>
```

```
          </Routes>
        </ErrorBoundary>

      </Router>
    </ContextProvider>
  )
}
```

And later on, using the `useContext` hook I can access the state defined in the Context from any of the components in my app.

```
const AboutPage: React.FC = () => {

    const { darkModeOn, englishLanguage } = useContext(Context)

    return (...)
}

export default AboutPage
```

Facade Pattern

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

Well...we can probably come out with lots of examples for this, right? I mean, React itself or any of the gazillion libraries out there used for pretty much anything related to software development. Specially when we think about declarative programming, it's all about providing abstractions that hide away complexity from the eyes of the developer.

A simple example could be JavaScript's `map`, `sort`, `reduce` and `filter` functions, which all work like good 'ol `for` loops beneath the hood.

Another example could be any of the libraries used for UI development nowadays, like MUI. As we can see in the following example, these libraries offer us components that bring built-in features and functionalities that help us build code faster and easier.

But all this when compiled turns into simple HTML elements, which are the only thing browsers understand. These components are only abstractions that are here to make our lives easier.

A facade...

```
import * as React from 'react';
import Table from '@mui/material/Table';
import TableBody from '@mui/material/TableBody';
import TableCell from '@mui/material/TableCell';
import TableContainer from '@mui/material/TableContainer';
import TableHead from '@mui/material/TableHead';
import TableRow from '@mui/material/TableRow';
import Paper from '@mui/material/Paper';

function createData(
  name: string,
  calories: number,
  fat: number,
  carbs: number,
  protein: number,
) {
```

```
  createData('Frozen yoghurt', 159, 6.0, 24, 4.0),
  createData('Ice cream sandwich', 237, 9.0, 37, 4.3),
  createData('Eclair', 262, 16.0, 24, 6.0),
  createData('Cupcake', 305, 3.7, 67, 4.3),
  createData('Gingerbread', 356, 16.0, 49, 3.9),
];

export default function BasicTable() {
  return (
    <TableContainer component={Paper}>
      <Table sx={{ minWidth: 650 }} aria-label="simple table">
        <TableHead>
          <TableRow>
            <TableCell>Dessert (100g serving)</TableCell>
            <TableCell align="right">Calories</TableCell>
            <TableCell align="right">Fat (g)</TableCell>
            <TableCell align="right">Carbs (g)</TableCell>
            <TableCell align="right">Protein (g)</TableCell>
          </TableRow>
        </TableHead>
        <TableBody>
          {rows.map((row) => (
            <TableRow
              key={row.name}
              sx={{ '&:last-child td, &:last-child th': { border: 0 } }}
            >
              <TableCell component="th" scope="row">
                {row.name}
              </TableCell>
              <TableCell align="right">{row.calories}</TableCell>
```

```
          ))}
        </TableBody>
      </Table>
    </TableContainer>
  );
}
```

Proxy Pattern

The **Proxy** pattern provides a substitute or placeholder for another object. The idea is to control access to the original object, performing some kind of action before or after the request gets to the actual original object.

Again, if you're familiar with ExpressJS this probably rings a bell for you. Express is a framework used to develop NodeJS APIs, and one of the features it has is the use of Middlewares. Middlewares are nothing more than pieces of code we can make execute before, in the middle, or after any request reaches our endpoints.

Let's see this in an example. Here I have a function that validates an authentication token. Don't pay much attention to how it does that. Just know that it receives the token as parameter, and once it's done it calls the `next()` function.

```
module.exports = function authenticateToken(req, res, next) {
    const authHeader = req.headers['authorization']
    const token = authHeader && authHeader.split(' ')[1]

    if (token === null) return res.status(401).send(JSON.stringify('No access token provided'))

    jwt.verify(token, process.env.TOKEN_SECRET, (err, user) => {
      if (err) return res.status(403).send(JSON.stringify('Wrong token provided'))
      req.user = user
      next()
    })
}
```

This function is a middleware, and we can use it in any endpoint of our API in the following way. We just place the middleware after the endpoint address and before declaration of the endpoint function:

```
router.get('/:jobRecordId', authenticateToken, async (req, res) => {
  try {
    const job = await JobRecord.findOne({_id: req.params.jobRecordId})
    res.status(200).send(job)

  } catch (err) {
```

In this way, if no token or a wrong token is provided, the middleware will return the corresponding error response. If a valid token is provided, the middleware will call the `next()` function and the endpoint function will get executed next.

We could've just written the same code within the endpoint itself and validated the token in there, without worrying about middlewares or anything. But the thing is now we have an abstraction we can reuse in many different endpoints. 😉

Again, this might not have been the precise idea the authors had in mind, but I believe it's a valid example. We're controlling an object's access so we can perform actions at a particular moment.

Behavioral Design Patterns

Behavioral patterns control communication and the assignment of responsibilities between different objects.

Chain of Responsibility Pattern

For this pattern we could use the same exact example as before, as middlewares in Express are somehow handlers that either process a request or pass it to the next handler.

If you'd like another example, think about any system in which you have certain information to process along many steps. At each step a different entity is in charge of performing an action, and the information only gets passed to another entity if a certain condition is met.

A typical front-end app that consumes an API could work as an example:

- We have a function responsible for rendering a UI component.
- Once rendered, a another function makes a request to an API endpoint.
- If the endpoint response is as expected, the information is passed to another function that sorts the data in a given way and stores it in a variable.
- Once that variable stores the needed information, another function is responsible of rendering it in the UI.

We can see how here we have many different entities that collaborate to execute a certain task. Each of them is responsible for a single "step" of that task, which helps with code modularity and separation of concerns. 👌 👌

The **iterator** is used to traverse elements of a collection. This might sound trivial in programming languages used nowadays, but this wasn't always the case.

Anyway, any of the JavaScript built in functions we have at our disposal to iterate over data structures (`for` , `forEach` , `for...of` , `for...in` , `map` , `reduce` , `filter` , and so on) are examples of the iterator pattern.

Same as any traversing algorithm we code to iterate through more complex data structures like trees or graphs.

Observer Pattern

The **observer** pattern lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing. Basically, it's like having an event listener on a given object, and when that object performs the action we're listening for, we do something.

React's useEffect hook might be a good example here. What useEffect does is execute a given function at the moment we declare.

The hook is divided in two main parts, the executable function and an array of dependencies. If the array is empty, like in the following example, the function gets executed each time the component is

```
useEffect(() => { console.log('The component has rendered') }, [])
```

If we declare any variables within the dependency array, the function will execute only when those variables change.

```
useEffect(() => { console.log('var1 has changed') }, [var1])
```

Even plain old JavaScript event listeners can be thought of as observers. Also, reactive programming and libraries like RxJS, which are used to handle asynchronous information and events along systems, are good examples of this pattern.

Roundup

If you'd like to know more about this topic, I recommend this great Fireship video and this awesome website where you can find very detailed explanations with illustrations to help you understand each pattern.