A Guide to 10 Common JavaScript Design Patterns

By AI Generated Published October 3, 2025 12 min read

Top 10 JavaScript Design Patterns

Software **design patterns** are proven, reusable templates for solving common problems in software design. They encapsulate best practices and provide a shared vocabulary for structuring code. In fact, a design pattern can be seen as “a general reusable solution to a commonly occurring problem” in software engineering ((Source: ozgurozkok.com)) ((Source: handwiki.org)). Even though many patterns originated in object-oriented languages, they apply to JavaScript just as well ((Source: ozgurozkok.com)). In JavaScript, these patterns help organize code, improve maintainability, and reduce duplication as applications grow in complexity. The sections below cover ten widely used JavaScript design patterns, explaining each with examples and key points.

1. Constructor Pattern

The **Constructor Pattern** uses constructor functions and the `new` keyword to create objects. In this pattern, a function acts like a class, initializing properties on `this`. Each call with `new` produces a new object instance with its own properties. Developers often attach methods to the function's prototype so all instances share behavior. This is JavaScript's way of mimicking classical classes. The constructor pattern is simple and maps closely to how objects are instantiated:

- Uses a function (e.g. `function Person(name) { ... }`) and the `new` keyword to create objects.
- Inside the constructor, `this.property = value` sets instance-specific data.
- Methods are usually added to `Constructor.prototype` so all instances share the same function (saving memory).
- Forgetting to use `new` can lead to bugs (e.g., polluting the global object).

```
// Constructor function example
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.sayHello = function() {
  console.log(`Hello, I'm ${this.name} and I'm ${this.age} years old.`);
};

// Usage
const alice = new Person('Alice', 30);
alice.sayHello(); // Hello, I'm Alice and I'm 30 years old.
```

Key benefits: Groups related data into clear object instances. It's easy to see how each object is created and to share methods via `prototype`. *Use when:* You need multiple similar objects and want to use classical-style instantiation.

2. Module Pattern

The **Module Pattern** leverages JavaScript's functions and closures to encapsulate private state and expose a public API. It usually takes the form of an Immediately Invoked Function Expression (IIFE) that returns an object. Inside the IIFE, local variables and helper functions are "private," inaccessible from

outside. The returned object exposes only the methods or properties you choose, creating a clean interface. This pattern helps avoid global variables and keeps code organized:

- Encapsulates variables and functions inside a function scope (often an IIFE) to create privacy.
- Returns an object literal with methods that can access those private variables via closure.
- Avoids polluting the global namespace by exposing just one global object.
- Often called the *Revealing Module Pattern* when you explicitly map private functions to public names.

```
// Module pattern example (using an IIFE)
const CounterModule = (function() {
  let count = 0; // private variable

  function increment() {
    count++;
    console.log(`Count is now ${count}`);
  }

  function reset() {
    count = 0;
    console.log('Count has been reset');
  }

  // Public API
  return {
    increment,
    reset
  };
})();

// Usage
CounterModule.increment(); // Count is now 1
CounterModule.increment(); // Count is now 2
CounterModule.reset();     // Count has been reset
```

Key benefits: Creates private state and well-defined public methods. It avoids naming collisions and groups related functionality. *Use when:* You want to bundle related functions/variables together with controlled visibility.

3. Singleton Pattern

The **Singleton Pattern** ensures that only one instance of an object exists and provides a global point of access to it. In JavaScript, this is typically implemented by having a module or object manage a single instance internally (often via closures or a static property). Whenever you need the instance, you go through a `getInstance` or similar method. Use cases include configuration objects or logging components where having multiple copies would cause conflicts:

- Keeps a private reference to a single instance. Calling the constructor or initializer multiple times returns the same instance.
- Often implemented with an IIFE or closure that stores the instance after the first creation.
- Ensures global coordination (e.g., one event bus or one cache).
- Overuse can lead to hidden dependencies or harder testing (since state is shared).

```
// Singleton pattern example
const Logger = (function() {
  let instance;

  function createInstance() {
    const obj = { logs: [] };
    obj.log = function(message) {
      this.logs.push(message);
      console.log(`Log entry: ${message}`);
    };
    return obj;
  }

  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

// Usage
const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();
console.log(logger1 === logger2); // true
logger1.log('First message');      // Log entry: First message
```

Key benefits: Provides a single, globally accessible resource or state. *Use when:* Exactly one instance is needed (e.g. a single database connection, logger, or configuration manager).

4. Prototype Pattern

The **Prototype Pattern** is fundamental in JavaScript since JS natively uses prototypal inheritance. In this pattern, objects inherit directly from other objects. You create a new object specifying an existing object as its prototype. This pattern is useful for cloning objects or sharing behavior without classes. JavaScript's `Object.create()` is a direct way to implement it. Key ideas:

- Defines an object (the prototype) and creates new objects that delegate to that prototype for shared properties/methods.
- Promotes sharing of structure: changing the prototype's properties affects all derived objects.
- Contrasts with creating new instances via a constructor – here you often clone an existing object as a baseline.
- Common use: polyfills or object cloning.

```
// Prototype pattern example
const vehicle = {
  wheels: 4,
  start() {
    console.log(`Starting a vehicle with ${this.wheels} wheels`);
  }
};

// Create a new object with vehicle as prototype
constcar = Object.create(vehicle);
car.wheels = 6;
car.start(); // Starting a vehicle with 6 wheels

// Another derived object
const bike = Object.create(vehicle);
bike.wheels = 2;
bike.start(); // Starting a vehicle with 2 wheels
```

Key benefits: Easy way to create objects that share behavior without redundant code. *Use when:* You have a base object and want multiple objects that use or override parts of it.

5. Factory Pattern

The **Factory Pattern** abstracts object creation, letting a function or class decide which specific type of object to instantiate. Instead of calling constructors directly, you call a factory function that internally chooses and returns an object (often based on input type or configuration). This decouples the caller from the concrete classes. In JavaScript, it's usually a function that returns different objects. Important points:

- Provides a function (`factory`) that takes in parameters (e.g., type) and returns one of several new object instances.
- Encapsulates the logic of choosing a class or value so the rest of the code just calls the factory.
- Facilitates adding new types: you just update the factory logic.
- Often used in frameworks (e.g. a component factory).

```
// Factory pattern example
class Sedan {
  drive() { console.log("Driving a sedan"); }
}
class Suv {
  drive() { console.log("Driving an SUV"); }
}

function carFactory(type) {
  if (type === 'sedan') {
    return new Sedan();
  } else if (type === 'suv') {
    return new Suv();
  }
  // default
  throw new Error('Unknown car type');
}

// Usage
const myCar = carFactory('suv');
myCar.drive(); // Driving an SUV
```

Key benefits: Centralizes object creation logic and reduces coupling between code and concrete classes.
Use when: You need a flexible way to instantiate different types of objects based on conditions.

6. Decorator Pattern

The **Decorator Pattern** adds new functionality to an object dynamically by wrapping it. Instead of modifying the original object, you create a wrapper (or decorator) that extends behavior. In JavaScript, this often means creating a function or object that takes another object and adds properties or methods. This is different from ES7 decorators on classes; here it's a structural pattern to compose features. Key points:

- Wraps a core object with another object that adds (decorates) additional behavior.
- Keeps the original object unchanged; enhancement is transparent to callers.
- Allows stacking: you can apply multiple decorators in layers.
- Useful for extending objects without subclassing or editing the original code.

```
// Decorator pattern example
function Car() {
  this.cost = function() {
    return 20000;
  };
}

function withLeatherSeats(car) {
  const originalCost = car.cost();
  car.cost = function() {
    return originalCost + 1500;
  };
  return car;
}

function withSportPackage(car) {
  const originalCost = car.cost();
  car.cost = function() {
    return originalCost + 3000;
  };
  return car;
}

// Usage
let myCar = new Car();
myCar = withLeatherSeats(myCar);
myCar = withSportPackage(myCar);
console.log(myCar.cost()); // 24500 (20000 + 1500 + 3000)
```

Key benefits: Flexibly extend objects without altering their structure. *Use when:* You want to add responsibilities on-the-fly or combine behaviors in a modular way.

7. Observer (Pub/Sub) Pattern

The **Observer Pattern** (also known as Publish/Subscribe) defines a dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. JavaScript's event system is a natural example of this. Often you implement a simple pub/sub mechanism where functions (listeners) subscribe to certain events or subjects. Highlights:

- Maintains a list of subscribers (callbacks) and notifies them (usually by calling each function) when an event occurs.
- Decouples event producers from consumers: the subject doesn't need to know what observers do with the event.
- Common in UI (e.g. DOM events) and libraries (e.g. Node's `EventEmitter`).
- Helps build reactive, event-driven architectures.

```
// Observer pattern (simple Pub/Sub) example
class EventEmitter {
  constructor() {
    this.subscribers = {};
  }
  subscribe(event, handler) {
    if (!this.subscribers[event]) {
      this.subscribers[event] = [];
    }
    this.subscribers[event].push(handler);
  }
  emit(event, data) {
    const handlers = this.subscribers[event] || [];
    handlers.forEach(h => h(data));
  }
}

// Usage
const emitter = new EventEmitter();
emitter.subscribe('message', (msg) => console.log('Received:', msg));
emitter.emit('message', 'Hello, Observers!'); // Received: Hello, Observers!
```

Key benefits: Loose coupling between components. *Use when:* Many parts of your code need to respond to certain events or state changes.

8. Strategy Pattern

The **Strategy Pattern** defines a family of interchangeable algorithms or behaviors and lets you switch between them at runtime. In JavaScript, strategies are often simple objects or functions representing different approaches. You pass the chosen strategy into a context object, and the context uses it instead of hard-coding the logic. This makes your code more flexible and easier to maintain. Main points:

- Encapsulates each algorithm/behavior as a separate function or object (a “strategy”).
- The client or calling code chooses which strategy to use.
- The context code calls the strategy’s interface (often a method or function), without knowing the details.
- Simplifies adding new behaviors: just create a new strategy.

```
// Strategy pattern example
const travelByWalking = {
  travel() { return "Walking"; }
};
const travelByCar = {
  travel() { return "Driving a car"; }
};
function travelContext(strategy) {
  console.log("Traveling by: " + strategy.travel());
}

// Usage
travelContext(travelByWalking); // Traveling by: Walking
travelContext(travelByCar);     // Traveling by: Driving a car
```

Key benefits: Allows switching out functionality without changing the client. *Use when:* You need a family of algorithms (like sorting methods, compression strategies, etc.) and want the best one to be selected at runtime.

9. Facade Pattern

The **Facade Pattern** provides a simplified interface to a larger body of code or complex subsystem. It hides the complexities of the system and provides a single interface to client code. In JavaScript, this could be constructing an object or module that wraps multiple APIs or internal modules, so consumers only interact with the facade. Key aspects:

- The facade (wrapper object or function) exposes simpler methods, internally coordinating multiple parts.
- Clients use the facade instead of dealing with many classes or a complex API.
- Helps in refactoring: you can change inner workings without affecting client code.
- Common usage: jQuery acts as a facade to many DOM operations, or custom modules hiding low-level details.

```
// Facade pattern example
const AudioSystem = {
  playSound(filename) { console.log(`Playing ${filename}`); }
};
const VideoSystem = {
  playVideo(filename) { console.log(`Displaying ${filename}`); }
};
const HomeTheaterFacade = {
  playMovie(movieName) {
    console.log(`Starting movie: ${movieName}`);
    AudioSystem.playSound(`${movieName}-audio.mp3`);
    VideoSystem.playVideo(`${movieName}-video.mp4`);
  }
};

// Usage
HomeTheaterFacade.playMovie('Avatar');
// Output:
// Starting movie: Avatar
// Playing Avatar-audio.mp3
// Displaying Avatar-video.mp4
```

Key benefits: Simplifies the interface for clients and promotes a cleaner separation of concerns. *Use when:* You have a complex subsystem and you want to expose only what's necessary to the user, streamlining common operations.

10. Command Pattern

The **Command Pattern** turns requests or actions into stand-alone objects. A command object encapsulates all the information needed to perform an action (such as the action itself, its receiver, and any parameters). This decouples the object that invokes the operation from the one that knows how to perform it. In JavaScript, commands are often simple objects or functions representing actions (for example, button handlers, operations that can be queued or undone). Important points:

- You create command objects (or functions) with an `execute()` method (or direct function call).
- The `Invoker` holds and calls commands without knowing details; the `Receiver` does the actual work.
- Supports features like undo/redo (by storing past commands) or queuing commands for later execution.
- Common in GUI button handlers and task schedulers.

```
// Command pattern example
class Light {
  switchOn() { console.log("Light is on"); }
  switchOff() { console.log("Light is off"); }
}
class SwitchOnCommand {
  constructor(light) { this.light = light; }
  execute() { this.light.switchOn(); }
}
class SwitchOffCommand {
  constructor(light) { this.light = light; }
  execute() { this.light.switchOff(); }
}
class Switch {
  constructor() { this.commands = []; }
  storeAndExecute(cmd) {
    this.commands.push(cmd);
    cmd.execute();
  }
}

// Usage
const light = new Light();
const switchOn = new SwitchOnCommand(light);
const switchOff = new SwitchOffCommand(light);
const mySwitch = new Switch();

mySwitch.storeAndExecute(switchOn);  // Light is on
mySwitch.storeAndExecute(switchOff); // Light is off
```

Key benefits: Decouples command invocation from execution, and makes it easy to extend and record commands. *Use when:* You need to parameterize methods with actions, support undo operations, or queue tasks.

Summary: These ten patterns are powerful tools in a JavaScript developer's toolkit. By applying patterns like Module or Singleton, you enforce clear boundaries and reuse. Behavioral patterns like Observer and Strategy make code flexible and interactive. As one source notes, design patterns "provide a shared

vocabulary for discussing design problems and answers" ((Source: ozgurozkok.com)). Carefully choosing and implementing the right pattern can greatly improve code **reusability**, **maintainability**, and overall design quality in your JavaScript projects.

Tags: javascript, design patterns, software architecture, code quality, creational patterns, structural patterns, behavioral patterns, code examples

---

About Tapflare

Tapflare in a nutshell Tapflare is a subscription-based "scale-as-a-service" platform that hands companies an on-demand creative and web team for a flat monthly fee that starts at $649. Instead of juggling freelancers or hiring in-house staff, subscribers are paired with a dedicated Tapflare project manager (PM) who orchestrates a bench of senior-level graphic designers and front-end developers on the client's behalf. The result is agency-grade output with same-day turnaround on most tasks, delivered through a single, streamlined portal.

How the service works

1. **Submit a request.** Clients describe the task—anything from a logo refresh to a full site rebuild—directly inside Tapflare's web portal. Built-in AI assists with creative briefs to speed up kickoff.
2. **PM triage.** The dedicated PM assigns a specialist (e.g., a motion-graphics designer or React developer) who's already vetted for senior-level expertise.
3. **Production.** Designer or developer logs up to two or four hours of focused work per business day, depending on the plan level, often shipping same-day drafts.
4. **Internal QA.** The PM reviews the deliverable for quality and brand consistency before the client ever sees it.
5. **Delivery & iteration.** Finished assets (including source files and dev hand-off packages) arrive via the portal. Unlimited revisions are included—projects queue one at a time, so edits never eat into another ticket's time.

What Tapflare can create

- **Graphic design:** brand identities, presentation decks, social media and ad creatives, infographics, packaging, custom illustration, motion graphics, and more.
- **Web & app front-end:** converting Figma mock-ups to no-code builders, HTML/CSS, or fully custom code; landing pages and marketing sites; plugin and low-code integrations.
- **AI-accelerated assets (Premium tier):** self-serve brand-trained image generation, copywriting via advanced LLMs, and developer tools like Cursor Pro for faster commits.

The Tapflare portal Beyond ticket submission, the portal lets teams:

- Manage multiple brands under one login, ideal for agencies or holding companies.
- Chat in-thread with the PM or approve work from email notifications.
- Add unlimited collaborators at no extra cost.

A live status dashboard and 24/7 client support keep stakeholders in the loop, while a 15-day money-back guarantee removes onboarding risk.

Pricing & plan ladder

Plan	Monthly rate	Daily hands-on time	Inclusions
Lite	$649	2 hrs design	Full graphic-design catalog
Pro	$899	2 hrs design **+** dev	Adds web development capacity
Premium	$1,499	4 hrs design + dev	Doubles output and unlocks Tapflare AI suite

All tiers include:

- Senior-level specialists under one roof
- Dedicated PM & unlimited revisions
- Same-day or next-day average turnaround (0–2 days on Premium)
- Unlimited brand workspaces and users
- 24/7 support and cancel-any-time policy with a 15-day full-refund window.

What sets Tapflare apart

Fully managed, not self-serve. Many flat-rate design subscriptions expect the customer to coordinate with designers directly. Tapflare inserts a seasoned PM layer so clients spend minutes, not hours, shepherding projects.

Specialists over generalists. Fewer than 0.1 % of applicants make Tapflare's roster; most pros boast a decade of niche experience in UI/UX, animation, branding, or front-end frameworks.

Transparent output. Instead of vague "one request at a time," hours are concrete: 2 or 4 per business day, making capacity predictable and scalable by simply adding subscriptions.

Ethical outsourcing. Designers, developers, and PMs are full-time employees paid fair wages, yielding <1 % staff turnover and consistent quality over time.

AI-enhanced efficiency. Tapflare Premium layers proprietary AI on top of human talent—brand-specific image & copy generation plus dev acceleration tools—without replacing the senior designers behind each deliverable.

Ideal use cases

- **SaaS & tech startups** launching or iterating on product sites and dashboards.
- **Agencies** needing white-label overflow capacity without new headcount.
- **E-commerce brands** looking for fresh ad creative and conversion-focused landing pages.
- **Marketing teams** that want motion graphics, presentations, and social content at scale. Tapflare already supports 150 + growth-minded companies including Proqio, Cirra AI, VBO Tickets, and Houseblend, each citing significant speed-to-launch and cost-savings wins.

The bottom line Tapflare marries the reliability of an in-house creative department with the elasticity of SaaS pricing. For a predictable monthly fee, subscribers tap into senior specialists, project-managed workflows, and generative-AI accelerants that together produce agency-quality design and front-end code in hours—not weeks—

without hidden costs or long-term contracts. Whether you need a single brand reboot or ongoing multi-channel creative, Tapflare's flat-rate model keeps budgets flat while letting creative ambitions flare.

DISCLAIMER