



DESIGN PATTERNS IN JAVASCRIPT

by Pavel Yukhnovich

DESIGN PATTERNS

**CREATIONAL
PATTERNS**

**STRUCTURAL
PATTERNS**

**BEHAVIORAL
PATTERNS**

CREATIONAL PATTERNS

provide various object creation mechanisms,
which increase flexibility and reuse
of existing code

PROTOTYPE

SINGLETON

**FACTORY
METHOD**

**ABSTRACT
FACTORY**

BUILDER

SINGLETON

How often is used?



Code complexity:




What it is?

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

When?


There must be only one instance of a class

SINGLETON



```
const exampleObject1 = {  
  exampleProp: 'unique value',  
}  
  
const exampleObject2 = {  
  exampleProp: 'unique value',  
}  
  
> exampleObject1 == exampleObject2  
< false  
  
> exampleObject1 === exampleObject2  
< false
```

SINGLETON



```
export default class Robot {  
  constructor() {  
    if (typeof Robot.instance === 'object') {  
      return Robot.instance;  
    }  
  
    Robot.instance = this;  
  
    return this;  
  }  
}
```


Where to use

- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Use the Singleton pattern when you need stricter control over global variables.

SINGLETON



Pros:

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



Cons:

- Violates the Single Responsibility Principle. The pattern solves two problems at the time.
- The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

PROTOTYPE

How often is used?



Code complexity:




What it is?

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

When?

Classes to instantiate are available only in runtime

PROTOTYPE



```
class Robot {  
  constructor(color, capacity) {  
    this.color = color;  
    this.capacity = capacity;  
  }  
}  
  
let r100 = new Robot('red', 100);  
  
let r200 = new Robot('white', 1000);
```

Where to use

- Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.
- Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.

PROTOTYPE



Pros:

- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex objects.



Cons:

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

FACTORY METHOD

How often is used?



Code complexity:



What it is?

Factory method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

When?

A class wants its subclasses to decide which object to create

FACTORY METHOD

```
class Robot {  
  constructor(color, capacity) {  
    this.color = color;  
    this.capacity = capacity;  
  }  
}  
  
class RobotFactory {  
  createRobot(type) {  
    switch (type) {  
      case 'r100':  
        return new Robot('red', 100)  
      case 'r200':  
        return new Robot('white', 1000)  
    }  
  }  
}  
  
const factory = new RobotFactory();  
  
const newR200 = factory.createRobot('r200');
```


Where to use

- Use the Factory Method when you don't know beforehand the exact types and dependencies of the objects your code should work with.
- Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
- Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

FACTORY METHOD



Pros:

- You avoid tight coupling between the creator and the concrete products.
- **Single Responsibility Principle.** You can move the product creation code into one place in the program, making the code easier to support.
- **Open/Closed Principle.** You can introduce new types of products into the program without breaking existing client code.



Cons:

- The code **may become more complicated** since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

ABSTRACT FACTORY

How often is used?



Code complexity:



What it is?

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

When?

System should be independent of how what it is producing is structured or represented.

ABSTRACT FACTORY

```
class Robot {
  constructor(color, capacity) {
    this.color = color;
    this.capacity = capacity;
  }
}

class RobotFactory {
  createRobot(type) {
    switch (type) {
      case 'r100':
        return new Robot('red', 100)
      case 'r200':
        return new Robot('white', 1000)
    }
  }
}

class RobotPRO {
  constructor(color, capacity) {
    this.color = color;
    this.capacity = capacity;
  }
}

class RobotPROFactory {
  createRobot(type) {
    switch (type) {
      case 'r100PRO':
        return new Robot('red', 200)
      case 'r200PRO':
        return new Robot('white', 2000)
    }
  }
}
```

```
const robotFactory = new RobotFactory();
const robotPROFactory = new RobotPROFactory();

const robotCreator = (editionType, type) => {
  switch (editionType) {
    case 'robot':
      return robotFactory.createRobot(type);
    case 'robotPRO':
      return robotPROFactory.createRobot(type);
  }
}

const newR100 = robotCreator('robot', 'r100');
const newR200PRO = robotCreator('robotPRO', 'r200');
```

Where to use

- Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

ABSTRACT FACTORY



Pros:

- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- Single Responsibility Principle. You can extract the product creation code into one place, making the code easier to support.
- Open/Closed Principle. You can introduce new variants of products without breaking existing client code.



Cons:

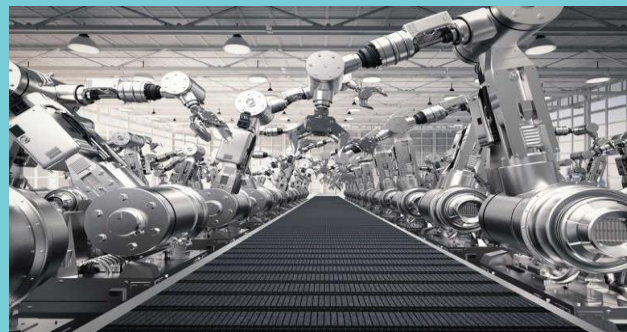
- The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.

BUILDER

How often is used?



Code complexity:



What it is?

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.

When?

Algorithm of creation is independent of the parts of the object.

BUILDER

```
class SimpleRobot {  
  constructor() {  
    this.color = '';  
    this.capacity = '';  
  }  
}  
  
class RobotBuilder {  
  constructor() {  
    this.robot = new SimpleRobot();  
  }  
  
  updateColor(color) {  
    this.robot.color = color;  
    return this;  
  }  
  
  updateCapacity(capacity) {  
    this.robot.capacity = capacity;  
    return this;  
  }  
  
  equipWithAdditionalFeature(feature) {  
    this.robot.feature = feature;  
    return this;  
  }  
  
  build() {  
    return this.robot;  
  }  
}
```

```
let brandNewRobot = new RobotBuilder()  
    .updateColor('blue')  
    .updateCapacity('75')  
    .equipWithAdditionalFeature('makes coffee');  
  
let yetAnotherRobot = new RobotBuilder()  
    .updateColor('red')  
    .updateCapacity('170');  
  
let andOneMoreRobot = new RobotBuilder();
```


Where to use

- Use the Builder pattern to get rid of a “telescopic constructor”.
- Use the Builder pattern when you want your code to be able to create different representations of some product.
- Use the Builder to construct Composite trees or other complex objects.



Pros:

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.



Cons:

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

STRUCTURAL PATTERNS

explain how to assemble objects and classes
into larger structures
while keeping these structures flexible and efficient

ADAPTER

BRIDGE

COMPOSITE

DECORATOR

FACADE

FLYWEIGHT

PROXY

ADAPTER

How often is used?



Code complexity:



What it is?

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

When?

You want to use existing class but its interface does not match the one you need.

ADAPTER

```
class RobotAnalysis {
  initiateSystemAnalysis() { console.log('... slow system scan ...') }
}

class RobotAnalysisPRO {
  initiateSystemAnalysisPRO() { console.log('super fast scan!') }
}

class SystemAdapter {
  constructor(system) {
    this.system = system;
  }

  initiateSystemAnalysis() {
    this.system.initiateSystemAnalysisPRO();
  }
}

class Robot {
  startSystem(system) {
    system.initiateSystemAnalysis()
  }
}
```

```
let brandNewRobot = new Robot();

let regularSystemCheck = new RobotAnalysis();
brandNewRobot.startSystem(regularSystemCheck);
<- ... slow system scan ...

let superFastSystemCheck= new RobotAnalysisPRO();
brandNewRobot.startSystem(superFastSystemCheck);
<- TypeError ...

let adaptedSuperFastSystemCheck = new SystemAdapter( new RobotAnalysisPRO());
brandNewRobot.startSystem(adaptedSuperFastSystemCheck);
<- super fast scan!
```

Where to use

- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

ADAPTER



Pros:

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



Cons:

- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

BRIDGE

How often is used?



Code complexity:



What it is?

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

When?

You want to avoid binding between abstraction and its implementation if, for example, each of them must be selected in runtime.

Where to use

- Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).
- Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.
- Use the Bridge if you need to be able to switch implementations at runtime.

BRIDGE



Pros:

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



Cons:

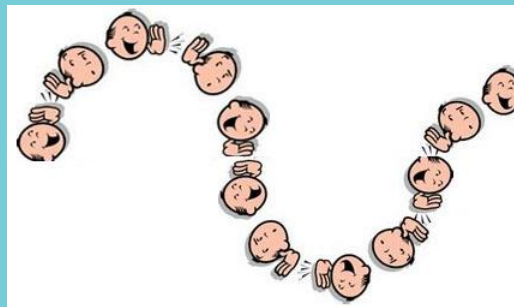
- You might make the code extremely complicated by applying the pattern to a highly cohesive class.
- Almost never used in JS applications.

COMPOSITE

How often is used?



Code complexity:



What it is?

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

When?

You want to represent hierarchies of objects.

COMPOSITE



```
$(div).addClass('active');  
$('.clickable').addClass('visited');
```

Where to use

- Use the Composite pattern when you have to implement a tree-like object structure.
- Use the pattern when you want the client code to treat both simple and complex elements uniformly.

COMPOSITE



Pros:

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.



Cons:

- It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

DECORATOR

How often is used?



Code complexity:



What it is?

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.

When?

You want to add extensions to an object in runtime without affecting other objects.

DECORATOR

```
class Robot {  
    constructor() {  
        this.capacity = 100;  
    }  
  
    getCapacity() {  
        return this.capacity;  
    }  
}  
  
class RobotPRO extends Robot {  
    constructor(capacity) {  
        super();  
        this.capacity = capacity;  
    }  
  
    getCapacity() {  
        return super.getCapacity() + 500;  
    }  
}
```

Where to use

- Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

DECORATOR



Pros:

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



Cons:

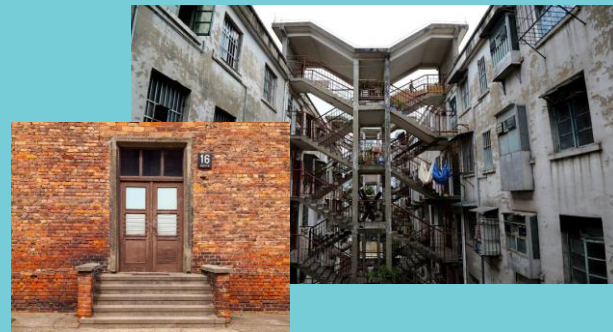
- It's hard to remove a specific wrapper from the wrappers stack.
- It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- The initial configuration code of layers might look pretty ugly.

FACADE

How often is used?



Code complexity:



What it is?

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

When?

You want to provide a simple interface to a complex subsystem

FACADE

```
class ExampleApp extends Component {  
  render() {  
    return (  
      <div>  
        <ExampleHeader />  
        <ExampleSection />  
        <ExampleFooter />  
      </div>  
    )  
  }  
}
```

```
class ExampleSection extends Component {  
  render() {  
    return (  
      <div>  
        <YetAnotherExampleComponent1 />  
        <YetAnotherExampleComponent2 />  
        <YetAnotherExampleComponent3 />  
        ...  
      </div>  
    )  
  }  
}
```

Where to use

- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- Use the Facade when you want to structure a subsystem into layers.

FACADE



Pros:

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



Cons:

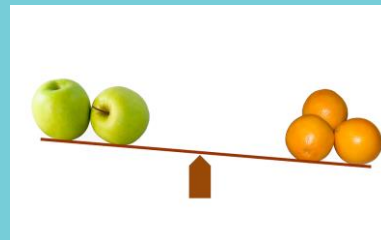
- A facade can become a god object coupled to all classes of an app.

FLYWEIGHT

How often is used?



Code complexity:



What it is?

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

When?

An application uses a lot of small objects and their storing is expensive or their identity is not important.

FLYWEIGHT

```
class Robot {  
  constructor(model) {  
    this.model = model;  
  }  
}  
  
class RobotFactory {  
  constructor(name) {  
    this.models = {};  
  }  
  
  create(name) {  
    let model = this.models[name];  
    if (model) return model;  
    this.models[name] = new Robot(name);  
    return this.models[name];  
  }  
};
```

Where to use

- Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.

FLYWEIGHT



Pros:

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



Cons:

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

PROXY

How often is used?



Code complexity:



What it is?

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.

When?

Provide a surrogate or placeholder for another object to control access to it.

PROXY

```
class Robot {
    executeTask() {
        return 'executing the task'
    }
}

class RobotPattern {
    constructor(power) {
        this.power = power
    }

    executeTask() {
        return this.power.percentage < 1
            ? 'will stop executing the task soon'
            : new Robot().executeTask()
    }
}

class Robot_POWER {
    constructor(percentage) {
        this.percentage = percentage
    }
}
```

Where to use

- Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

Instead of creating the object when the app launches, you can delay the object's initialization to a time when it's really needed.

- 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.

- 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.

PROXY



Pros:

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



Cons:

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

Thanks!