Design Patterns Everyday

Anurag Hazra May 21 *Updated on May 22, 2020* • 21 min read

#javascript #designpatterns

> Photo by Flo P on Unsplash

Originally posted on my site

Hey folks! few weeks ago i started a new challenge to learn about one design pattern everyday i called it "#DesignPatternsEveryday".

Since i completed the challenge i thought i should share briefly what i learned about design patterns, let's get started.

I will be going over most of the patterns and going to explain them in my own words, so if you find any mistakes or misinformation please let me know. i'm not a design patterns expert.

Table of Contents

What are Design Patterns?

"Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you
can customize to solve a particular design problem in your code." - **refactoring.guru**

There are 3 categories of design patterns, we are going to cover them one by one.

- **Creational**

  Provides a way to create new objects which increases the flexibility and reusability.
- **Structural**

  Helps to structure & assemble objects and classes while making them flexible.
- **Behavioural**

  Helps to communicate between objects and concerned with responsiblities between objects.

Also note that, One thing is really important for utilizing design patterns in your project.

Never start with the mindset of "okay I'm going to use {this pattern} in the codebase"

Judge & analyze the code base, plan the logic and implementation first then apply design patterns to solve any
particular problem only IF NECESSARY.

> Design patterns are solution to problems, not solution finding problems.

Day 1

- **Abstract Factory Pattern**

Abstract factory is a creational design pattern which allows us to produce families of objects without specifying their
concrete classes.

Suppose you are creating a drawing app where you'll have tools like "draw box", "draw circle" but you also need rounded
variants of box & circle, in that case you can create a factory for "ShapeFactory" and "RoundedShapeFactory" which will
return respective shapes.

Use Cases

Abstract factory pattern could be helpful in scenarios while you need to have a framework work cross platform, for
example "Electronjs". i dont know how electronjs handles that probably not with factory pattern but it can be
implemented with factory pattern.

- Example

> code on github

```
class Button {
  render() {}
}

class Factory {
  createButton() {}
}

class WinButton extends Button {
  render() {
    return "<button class='windows'></button>";
  }
}

class LinuxButton extends Button {
  render() {
    return "<button class='linux'></button>";
  }
}

class WinFactory extends Factory {
  createButton() {
    return new WinButton();
  }
}
class LinuxFactory extends Factory {
  createButton() {
    return new LinuxButton();
  }
}

class AbstractFactory {
  static factory(type) {
    switch (type) {
      case 'windows':
        return new WinFactory();
```

```
      case 'linux':
        return new LinuxFactory();

      default:
        break;
    }
  }
}

let guiFactory = AbstractFactory.factory('linux');
let button = guiFactory.createButton();
console.log(button.render());
```

Day 2

- **Builder Pattern**

Builder pattern is creational design pattern which allows us to create complex objects in a step by step manner. It
allows us to create different type of objects with same code.

Real World Analogy

Think of it as car assembly line. The car will be assembled with parts gradually step by step, firstly its chassis will
be setup then the engine, radiator, wheels, seats, doors. and by modifying these steps in the assembly line we can
create different types of car models with the same assembly line.

Use Cases

Builder pattern is useful when you want to create various objects with different representation without creation
subclasses for each of them.

I implemented builder pattern in one of my previous project Evolution Aquarium to build different kind of Boids with different behaviours and traits.

- Example

> code on github

```
class Car {
  constructor(engine, fuelTank, seats) {
    this.engine = engine;
    this.fuelTank = fuelTank;
    this.seats = seats;
  }

  printSpecs() {
    console.log(this.engine, this.fuelTank, this.seats);
  }
}
class CarBuilder {
  constructor() {
    this.engine = '';
    this.seats = '';
    this.fuelTank = '';
  }

  addSeats(name) {
    this.seats = name;
    return this;
  }
  addEngine(value) {
    this.engine = value;
    return this;
  }
  addFuelTank(value) {
```

```
    this.fuelTank = value;
    return this;
  }

  build() {
    return new Car(this.engine, this.fuelTank, this.seats);
  }
}

let truck = new CarBuilder()
  .addSeats(8)
  .addEngine('v12')
  .addFuelTank('200liters')
  .build();

let sedan = new CarBuilder()
  .addSeats(4)
  .addEngine('v6')
  .addFuelTank('100liters')
  .build();
```

Day 3

- **Factory Method**

Factory Method Pattern, its similar to abstract factory method but has some subtle differences. In Abstract factory
pattern it creates Factories & Sub Factories depending on the type. (which i think is a little bit verbose) but Factory
method is pretty straight forward it has only one factory.

Use Cases

DOM API's `document.createElement` method is a Factory method. which creates different type of HTML elements depending
on the passed type.

- Example

> code on github

```
class Document {
  render() {
    return null;
  }
}
class Div extends Document {
  render() {
    return '<div />';
  }
}
class Section extends Document {
  render() {
    return '<section />';
  }
}
class DOMFactory {
  createElement(type) {
    switch (type) {
      case 'div':
        return new Div();
      case 'section':
        return new Section();
      default:
        break;
    }
  }
}

let domFactory = new DOMFactory();
let div = domFactory.createElement('div');
let section = domFactory.createElement('section');
```

Day 4

- **Singleton**

Singleton design pattern is a creational design pattern that ensures that a class will only have one instance.

Real World Analogy

A good real world analogy for singleton is Government, a country can only have one government regardless of how many
person it consists of it be always titled as "Government of {Country}"

- Example

> code on github

```
class Singleton {
  static instance = new Singleton();
  static getInstance() {
    return this.instance;
  }

  showMessage() {
    console.log('Hello singleton');
  }
}

let instance1 = Singleton.getInstance();
let instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // true

instance2.showMessage();
```

Starting Structural Design Patterns

Day 5

- **Adapter Pattern**

Adapter pattern is a structural design pattern which acts like a translator between two different interfaces/apis.

Use Cases

This pattern could be useful in cases where you have two different APIs and you want an universal interface to handle
them both.

let's take an example. suppose you are building a 2D Renderer for web which supports both WebGL & CanvasAPI you can make
an universal rendering api and use adapter pattern to fill the gaps between them.

- Example

> Code on github

```
class Python {
  print(msg: string) {
    return console.log(msg);
  }
```

```
}
class Javascript {
  console(msg: string) {
    return console.log(msg);
  }
}

class LoggerAdapter {
  adapter: any;
  constructor(type: string) {
    if (type === 'py') {
      this.adapter = new Python();
    } else if (type === 'js') {
      this.adapter = new Javascript();
    }
  }

  log(type: string, msg: string) {
    if (type === 'py') {
      this.adapter.print(msg);
    } else if (type === 'js') {
      this.adapter.console(msg);
    }
  }
}

class Logger {
  adapter: any;
  log(type: string, msg: string) {
    this.adapter = new LoggerAdapter(type);
    this.adapter.log(type, msg);
  }
}

const logger = new Logger();

logger.log('js', 'Hello world js');
logger.log('py', 'Hello world py');
```

Day 6

- **Bridge pattern**

""Decouple an abstraction from its implementation so that the two can vary independently"" what?

Well, it's confusing but it's interesting to see how useful this pattern could be.

Basically Bridge pattern allows us to separate the platform depended logic from platform independent logic.

This could be useful for building User Interfaces where you want to make different views depending on different
resources and doing that in traditional manner will force you to implement each and every view and their resource implementation separately and will exponentially grow the number of complex coupled classes.

but with bridge we can solve this issue by having a Uniform Resource Interface to talk with an abstract view class.

- Example

> code on github

```
interface IResource {
  title: "() => string;"
  body: () => string;
  link: () => string;
  image: () => string;
```

```
}

abstract class View {
  resource: IResource;
  constructor(resource: IResource) {
    this.resource = resource;
  }

  render(): string {
    return '';
  }
}

class DetailedView extends View {
  render() {
    return `
    <div>
      <h2>${this.resource.title()}</h2>
      <img src="${this.resource.image()}" />
      <div>${this.resource.body()}</div>
      <a href="${this.resource.link()}">readmore</a>
    </div>
    `;
  }
}
class MinimalView extends View {
  render() {
    return `
    <div>
      <h2>${this.resource.title()}</h2>
      <a href="${this.resource.link()}">readmore</a>
    </div>
    `;
  }
}

class ArtistResource implements IResource {
  artist: any;
  constructor(artist: any) {
    this.artist = artist;
  }
```

```
  title() {
    return this.artist.name;
  }
  body() {
    return this.artist.bio;
  }
  image() {
    return this.artist.image;
  }
  link() {
    return this.artist.slug;
  }
}

class SongResource implements IResource {
  song: any;
  constructor(song: any) {
    this.song = song;
  }

  title() {
    return this.song.name;
  }
  body() {
    return this.song.lyrics;
  }
  image() {
    return this.song.coverImage;
  }
  link() {
    return this.song.spotifyLink;
  }
}

const artist = new ArtistResource({
  name: 'Anurag',
  bio: '404 not found',
  image: '/img/mypic.png',
  slug: '/u/anuraghazra',
});
const song = new SongResource({
  name: 'Cant belive i can fly',
```

```
  lyrics: 'la la la la la',
  coverImage: '/img/cover.png',
  spotifyLink: '/s/song/132894',
});

const artist_detail_view = new DetailedView(artist);
const artist_minimal_view = new MinimalView(artist);

const song_detail_view = new DetailedView(song);
const song_minimal_view = new MinimalView(song);

console.log(artist_detail_view.render());
console.log(song_detail_view.render());
console.log(artist_minimal_view.render());
console.log(song_minimal_view.render());
```

Day 7

- **Composite Design Pattern**

Composite patterns allows us to compose objects which have hierarchical tree structure.

Use Cases

Nice use cases of this pattern i can see is that you can easily make composable Layering & Grouping System, like photoshop where you have a Layer() class you'll push Circle/Shape classes to the Layer and those shapes will get relatively positioned and parented to that Layer.

- CodeSandbox example

```
const rootLayer = new Layer('rootlayer');
const shapesLayer = new Layer('my layer');
const circle = new Shape(100, 100, 'red');
const box = new Shape(200, 100, 'red');

layer.add(circle);
layer.add(box);
rootLayer.add(shapesLayer);
```

- Example

> code on github

As you can see i have a FileNode, FolderNode if i were to implement this without composite pattern then i have to do
extra checks to see if the type of the passed component is Folder and then recursively go through the childs and make
the whole tree.

```
interface Component {
  remove?: (c: Component) => void;
  add?: (c: Component) => void;
  ls: () => string;
}

class FolderNode implements Component {
  name: string;
  childrens: Component[];
  constructor(name: string) {
    this.name = name;
    this.childrens = [];
  }
```

```
  add(component: Component) {
    this.childrens.push(component);
  }

  remove(component: Component) {
    this.childrens = this.childrens.filter((c: Component) => c !== compo
  }

  ls() {
    let str = '\n---' + this.name;
    this.childrens.forEach(child => {
      str += child.ls();
    });
    return str;
  }
}

class FileNode implements Component {
  name: string;
  constructor(name: string) {
    this.name = '\n------' + name;
  }

  ls() {
    return this.name;
  }
}

let root = new FolderNode('root');
let src = new FolderNode('src');
let lib = new FolderNode('lib');

let jsFile = new FileNode('app.js');
let htmlFile = new FileNode('index.html');
let cssFile = new FileNode('style.css');
let mainFile = new FileNode('index.js');

src.add(jsFile);
src.add(htmlFile);
src.add(cssFile);
lib.add(mainFile);
```

```
root.add(src);
root.add(lib);

console.log(root.ls());
```

Day 8

- **Decorator Pattern**

Decorator pattern allows us to enhance any class/object with extra behaviour without having to define any subclasses. I really like the flexibility and composability powers which decorators provide me.

Use Cases

Decorator pattern is extremely useful and we have already used it in many places. Angular devs uses @Decorator syntax very often. and React also make use of HigherOrder Functions (decorators), and libraries like MobX take advantage of decorator patterns very cleverly.

Javascript will also have native @Decorators support some point in future the @Decorator proposal is right now on 'Stage 2' so we might see some changes, i'm excited for it. but we can use typescript/babel to compile down those to todays js and use them right now.

- Example

> more examples on github

```
// simplified example
function loggerDecorator(wrapped) {
  return function(...args) {
    console.log('******');
    console.log(wrapped.apply(this, args));
    console.log('******');
  };
}
function mapper(arr: any[], add: number) {
  return arr.map(i => i + add);
}

loggerDecorator(mapper)([1, 2, 3], 10);
```

Day 9

- **Facade Pattern**

Facade pattern provides a consistent and unified API for any complicated API/Subsystem, making it easier to use for the client.

It basically works as a bootstrapper where it abstracts away all the complicated setups and provides a straight forward simple interface.

- Example

Example is a little big so check it out on github

Day 10

- **Proxy Design Pattern**

Proxy is an object which works as a placeholder or substitute to any other object. proxy provides a similar interface to
original object but extends the behaviour of how the object will react to changes.

There are mainly 5 types of proxies.

- Remote Proxy
- Virtual Proxy
- Cache Proxy
- Protection Proxy
- Smart Proxy

▶️ Remote proxy acts as a translator between two remote origins and you can do stuff like logging the requests.

▶️ Cache proxy improves performance by caching any long running operation's results and serving the cached results
instead of request the data every time from the original source.

▶️ Virtual proxy object is a default placeholder proxy that can be lazily initiated, we can think of it as a skeleton
object which acts as the original object until the data loads.

▶️ Protection proxies mainly acts as a authentication layer for the original object. restricting unauthorized access to

the object.

▶️ Smart proxies adds extra behaviours to the original object, for example sending the data to any third-party API or logging the data

- Example

more examples on gihtub

```
// EXAMPLE OF PROTECTION PROXY
interface IServer {
  request(url: string): void;
}
class Server implements IServer {
  request(url: string) {
    console.log('------');
    console.log('loading:    ', url);
    console.log('completed:  ', url);
    console.log('------');
  }
}

class ProtectedServer implements IServer {
  api: IServer;
  bannedWebsites: string[];
  constructor() {
    this.api = new Server();
    this.bannedWebsites = ['https://fakesite.com', 'https://spamming.com
  }
  request(url: string) {
    if (this.bannedWebsites.includes(url)) {
      console.log('------');
      console.log('BANNED:    ', url);
      console.log('------');
    } else {
      this.api.request(url);
```

```
    }
  }
}

const server = new ProtectedServer();
console.log('EXAMPLE-1 Protected Proxy');
server.request('https://google.com');
server.request('https://fakesite.com');
server.request('https://facebook.com');
```

Starting Behavioural Design Pattern

Day 11

- **Chain of Responsibility. (CoR)**

CoR is a behavioural design pattern which we know of as middlewares. CoR lets us delegate the individual logic as a handler and passing it onto the next handler.

Real World Analogy

A good real world analogy would be call centers or tech support channels.. when you call them, firstly you are prompted with an automated voice asking you to do some steps in order to talk to a real person, then they pass your call to a real person and if they can't help you they will again pass your call to a technician.

Use Cases

Express.js heavily make use of CoR or Middleware pattern, it passed the next() handler to the next middleware performing some checks and doing some operation in between.

CoR can be beneficial when you wanted to your logic to be reusable delegate the logic to multiple handlers. CoR also helps minimizing the complexity of a tightly coupled system by making sure that every chunk of handler does some specific work and passes the data to the next handler.

- Example

> code on github

```
// Chain of responsibility
import { consoleColor } from '../utils';

interface IHandler {
  addMiddleware(h: IHandler): IHandler;
  get(url: string, callback: (data: any) => void): void;
}

abstract class AbstractHandler implements IHandler {
  next: IHandler;
  addMiddleware(h: IHandler) {
    this.next = h;
    return this.next;
  }

  get(url: string, callback: (data: any) => void) {
    if (this.next) {
      return this.next.get(url, callback);
    }
  }
```

```
}

class Auth extends AbstractHandler {
  isAuthenticated: boolean;
  constructor(username: string, password: string) {
    super();

    this.isAuthenticated = false;
    if (username === 'anuraghazra' && password === 'password123') {
      this.isAuthenticated = true;
    }
  }

  get(url: string, callback: (data: any) => void) {
    if (this.isAuthenticated) {
      return super.get(url, callback);
    } else {
      throw new Error('Not Authorized');
    }
  }
}

class Logger extends AbstractHandler {
  get(url: string, callback: (data: any) => void) {
    consoleColor('green', '/GET Request to: ', url);
    return super.get(url, callback);
  }
}

class Route extends AbstractHandler {
  url: string;
  URLMaps: {};
  constructor() {
    super();
    this.URLMaps = {
      '/api/todos': [{ title: 'hello' }, { title: 'world' }],
      '/api/random': Math.random(),
    };
  }

  get(url: string, callback: (data: any) => void) {
    super.get(url, callback);
```

```
    if (this.URLMaps.hasOwnProperty(url)) {
      callback(this.URLMaps[url]);
    }
  }
}

const route = new Route();
route.addMiddleware(new Auth('anuraghazra', 'password123')).addMiddlewa

route.get('/api/todos', data => {
  consoleColor('blue', JSON.stringify({ data }, null, 2));
});
route.get('/api/random', data => {
  console.log(data);
});
```

Day 12

- **Command pattern**

Command pattern is a behavioural design pattern which lets us decouple the business logic from the client implementation.

Real World Analogy

Think of it like when you go to a restaurant you, call the waiter and command him to place your order and waiter passes that command to the chief, and after chief completed the order it gets back to you.

Use Cases

Command pattern also lets you do undo and redo operations. Suppose you are making a text editor and you wanted to implement undo, redo feature, it is can advantageous. And Command pattern also provides a nice interface to implement modular GUI Actions allowing us to seperate the UI Layer from the logic of the code.

Traditionally if you have a CopyText feature, you might face scenarios like when you want to allow users to trigger that CopyText function from the ContextMenu and the Toolbar both, in this scenario Command Pattern can be very useful.

- Example

check out the code on github

```
interface ICommand {
  undo?(payload?: any): any;
  execute(payload?: any): any;
}

abstract class Command implements ICommand {
  calc: Calculator;
  constructor(calc?: Calculator) {
    this.calc = calc;
  }
  execute() {}
}

class Calculator {
  currentValue: number;
  history: CommandHistory;
  constructor() {
    this.history = new CommandHistory();
    this.currentValue = 0;
```

```
  }

  getValue(): number {
    return this.currentValue;
  }

  execute(command: ICommand) {
    this.currentValue = command.execute(this.currentValue);
    this.history.add(command);
  }

  undo() {
    let lastCommand = this.history.remove();
    if (lastCommand) {
      this.currentValue = lastCommand.undo(this.currentValue);
    }
  }
}

class CommandHistory {
  commands: ICommand[];
  constructor() {
    this.commands = [];
  }

  add(command: ICommand) {
    this.commands.push(command);
  }

  remove() {
    return this.commands.pop();
  }
}

class AddCommand {
  value: number;
  constructor(value: number) {
    this.value = value;
  }
  execute(value: number) {
    return value + this.value;
  }
```

```
  undo(value: number) {
    return value - this.value;
  }
}

const calc = new Calculator();
calc.execute(new AddCommand(50));
calc.undo(); // undo last command
```

Day 13

- **Iterator pattern**

Iterator pattern is a behavioural design pattern which lets us traverse any complex data structure without exposing the underlying implementation to the client.

Use Cases

We can traverse graphs, lists, trees with iterator pattern easily. Javascript internally uses Iterator Protocol to implement [...spread] spread operators and loops.

- Example

code on github

```
interface IIterator {
  next(): any;
  hasMore(): any;
}

interface ICounter {
  getIterator(): IIterator;
```

```
}

class Counter implements ICounter {
  collection: any;
  constructor(data: any) {
    this.collection = data;
  }
  getIterator() {
    return new CounterIterator(this.collection);
  }
}

class CounterIterator implements IIterator {
  current: number;
  collection: any;
  constructor(data: any) {
    this.collection = data;
    this.current = 0;
  }

  next() {
    return this.collection[this.current++];
  }

  prev() {
    return this.collection[this.current - 1];
  }

  hasMore() {
    return this.collection.length > this.current;
  }
}

let iterator = new Counter([1, 2, 3, 4, 5]).getIterator();
while (iterator.hasMore()) {
  console.log(iterator.next());
}
```

Day 14

- **Mediator Design Pattern**

Mediator design is a behavioural design pattern which determines how set of objects will interact with each other.
mediator pattern encourages loose coupling between components because it prevents objects from directly referencing each other. thus reducing the overall complexity.

Mediator acts as an middle-man between different objects and all other objects will communicate through the mediator only.

Real World Analogy

A nice real world analogy would be Air Traffic Controller. While landing and taking off airplanes does not talk to each other directly instead they talk to the air traffic controllers to get information about other airplanes and the control tower tell them when to land/takeoff.

Use Cases

I think this pattern has some use cases, for example when building ChatRooms you can implement mediator pattern to simplify the relation between different members of the chatroom and send them messages though the Mediator.

You can also use Mediator pattern as a global event manager in front end applications where components talk to each other by the mediator instead of passing callbacks/props.

- Example

> code on github

```
// mediator pattern
import { consoleColor } from '../utils';

interface IMediator {
  sendMessage(msg: string, from: any, to?: any): void;
}

class Chatroom implements IMediator {
  members: { [x: string]: Member };
  constructor() {
    this.members = {};
  }

  addMember(member: Member) {
    member.chatroom = this;
    this.members[member.name] = member;
  }

  sendMessage(msg: string, from: Member, to?: Member) {
    Object.keys(this.members).forEach(name => {
      if (!to && name !== from.name) {
        this.members[name].receive(msg, from);
        return;
      }
      if (to && name == to.name) {
        this.members[name].receive(msg, from);
      }
    });
  }
}

class Member {
  name: string;
  chatroom: Chatroom;
```

```
  constructor(name: string) {
    this.name = name;
    this.chatroom = null;
  }

  send(msg: string, to?: any) {
    this.chatroom.sendMessage(msg, this, to);
  }

  receive(msg: string, from: Member) {
    consoleColor('magenta', `-------`);
    consoleColor('cyan', `${from.name} says to ${this.name} : `);
    consoleColor('green', `${msg}`);
    consoleColor('magenta', `-------`);
  }
}

const chatroom = new Chatroom();

let anurag = new Member('Anurag');
let hitman = new Member('hitman');
let jonathan = new Member('John Wick');
chatroom.addMember(anurag);
chatroom.addMember(hitman);
chatroom.addMember(jonathan);

anurag.send("I'm more dangerous than you hitman");
hitman.send('Sorry brother forgive me! pls', anurag);
jonathan.send('Hey hey hey hitman, nerver ever mess with Anurag', hitmar
```

Day 15

- **Observer Design Pattern**

Observer design pattern is a behavioural design pattern which is a subscription system which notifies multiple objects about any changes to the object they are observing.

▶️ The cool thing about observer pattern is that it decouples the State from the actual business logic. in terms of UIs
you can separate the State from the actual rendering of the UI and if that State updates the UI will automatically react
to it.

Suppose you have a some Todos in your state you can decouple the data from the Ui and implement the render logic
entirely differently. You can have a DOMRenderer and a ConsoleRenderer and both will react and update to the changes made to the Todos. Heres a good example
https://github.com/anuraghazra/VanillaMVC

Real world Analogy

You can compare Observer pattern with daily newspaper subscriptions, If you subscribe to any newspaper you don't need to
go to the store everyday and get the newspaper instead the Publisher sends the newspaper to your home.

Another analogy would be Youtube, well you might know pretty well that subscribing to youtube channels means you will
get notification about new videos. Observer pattern also works the same. you as a user will subscribe to events you
choose to get notifications.

Use Cases

Observer pattern has lot of use cases. (a lot) Vuejs's Reactivity system relies on Observer pattern. The whole idea of

RxJs is based upon observers. MobX also uses Observer design pattern effectively.

From User Interfaces to Data Reactivity, Observer pattern is really handy when some changes/events in a particular
object has to be reflected on other objects

- Example

> code on github

```
import { consoleColor } from '../utils';

interface IPublisher {
  addSubscriber(subscriber: any): void;
  removeSubscriber(subscriber: any): void;
  notifyObservers(): void;
}
interface IObserver {
  notify(payload: any): void;
}

class Publisher implements IPublisher {
  subscribers: IObserver[];
  state: any;
  constructor(state: any = {}) {
    this.subscribers = [];
    this.state = state;
  }

  addSubscriber(subscriber: IObserver) {
    if (this.subscribers.includes(subscriber)) return;
    this.subscribers.push(subscriber);
  }
```

```
  removeSubscriber(subscriber: IObserver) {
    if (!this.subscribers.includes(subscriber)) return;
    let index = this.subscribers.indexOf(subscriber);
    this.subscribers.splice(index, 1);
  }

  notifyObservers() {
    this.subscribers.forEach(subs => {
      subs.notify(this.state);
    });
  }

  setState(newState: any) {
    this.state = newState;
    this.notifyObservers();
  }
}

class UserInterface implements IObserver {
  renderTodos(todos) {
    console.clear();
    todos.forEach(todo => {
      consoleColor('cyan', '-----');
      consoleColor(todo.isCompleted ? 'green' : 'red', `${todo.title} ${
      consoleColor('cyan', '-----');
    });
  }

  notify(state: any) {
    this.renderTodos(state.todos);
  }
}

const store = new Publisher({
  todos: [
    { title: 'hello', isCompleted: false, id: 1 },
    { title: 'world', isCompleted: false, id: 2 },
  ],
});

const userInterface = new UserInterface();
store.addSubscriber(userInterface);
```

```
// add todo
store.setState({
  todos: [...store.state.todos, { title: 'new item', id: Math.random() }]
});

// remove todo
store.setState({
  todos: store.state.todos.filter(t => t.id !== 2),
});
```

Day 16

- **State Pattern**

State pattern is a behavioural design pattern which lets objects change its behaviour based on its internal state.

If you wanted to see the code, i have three examples on state-pattern in my github repo:
https://github.com/anuraghazra/design-patterns-everyday

▶️ State pattern can be correlated with State-Machines where at certain point of time an application can only be in one state or in a limited finite number of states.

Where State-Machines heavily relays on if statements and switch cases which leads to Complex logic and unmaintainable code when codebase gets larger. State pattern changes behaviour methods based on the current state.

Real World Analogy

Suppose you have a Music Player and that music player has 2 buttons "UP" & "DOWN"

- When you are playing a song those "UP" & "DOWN" button will change the song volume.
- And when you are on a playlist menu the Up and Down buttons will scroll up and down in the list.

Use Cases

A good real world use case would be a any drawing app / text editor or anything where you have some class which changes
its behaviour based on some state.

for example: if are building a drawing app the app will have a pain brush tool which would draw in different color/size
based on the selected color/size.

Another example would be text editor, where you have a class for writing the text on screen but depending on the Uppercase/bold/lowercase buttons you write appropriate character to the screen

- Example

> code on github

```
/* SIMPLE TOGGLE */
interface IToggleState {
```

```
  toggle(state: IToggleState): void;
}

class ToggleContext {
  currentState: any;
  constructor() {
    this.currentState = new Off();
  }

  setState(state: IToggleState) {
    this.currentState = state;
  }

  toggle() {
    this.currentState.toggle(this);
  }
}

class Off implements IToggleState {
  toggle(ctx: ToggleContext) {
    console.log('OFF');
    ctx.setState(new On());
  }
}
class On implements IToggleState {
  toggle(ctx: ToggleContext) {
    console.log('ON');
    ctx.setState(new Off());
  }
}

let button = new ToggleContext();
button.toggle();
button.toggle();
```

Day 17

- **Strategy Design Pattern**

Strategy design pattern is behavioural design pattern which lets us define different algorithms for doing a particular action and interchange them as we wish. basically means you can switch between different types of behaviour and implementation.

Strategy design pattern is very similar to state design pattern. strategy pattern is an extension to state pattern, but strategy pattern completely makes the subclasses independent from each other.

Real World Analogy

I think a good real world analogy would be a match of football. the coach (context) decides a

strategy every situation the game flows through and switch between them depending on the situation. for example if the opposition is playing defensive then coach changes the strategy to playing aggressive. and when the team is leading 1 goal coach changes the strategy to semi-defensive.

Use cases

If you every used passportjs then you already used Strategy design pattern. passportjs uses strategy pattern to easily change/add new Authentication providers and the system becomes more flexible to use and extend on.

- Example

> code on github

```
// Strategy pattern
interface IStrategy {
  authenticate(...args: any): any;
}

class Authenticator {
  strategy: any;
  constructor() {
    this.strategy = null;
  }

  setStrategy(strategy: any) {
    this.strategy = strategy;
  }

  authenticate(...args: any) {
    if (!this.strategy) {
      console.log('No Authentication strategy provided');
      return;
    }
    return this.strategy.authenticate(...args);
  }
}

class GoogleStrategy implements IStrategy {
  authenticate(googleToken: string) {
    if (googleToken !== '12345') {
      console.log('Invalid Google User');
      return;
    }
    console.log('Authenticated with Google');
  }
}

class LocalStrategy implements IStrategy {
  authenticate(username: string, password: string) {
    if (username !== 'johnwick' && password !== 'gunslotsofguns') {
```

```
      console.log('Invalid user. you are `Excommunicado`');
      return;
    }
    console.log('Authenticated as Baba Yaga');
  }
}

const auth = new Authenticator();

auth.setStrategy(new GoogleStrategy());
auth.authenticate('invalidpass');

auth.setStrategy(new GoogleStrategy());
auth.authenticate('12345');

auth.setStrategy(new LocalStrategy());
auth.authenticate('anurag', '12345');

auth.setStrategy(new LocalStrategy());
auth.authenticate('johnwick', 'gunslotsofguns');
```

Day 18

- **Template Method**

Template method is a behavioural design pattern which defines skeleton of an algorithm in a step by step manner and lets subclasses override them.

Basically what template method does is that it requires you to split an algorithm into a smaller chunks and make
separate methods for them, and then call each of the methods sequentially in a series. that way you can override any
step of an algorithm in a subclass.

Real World Analogy

A good real world analogy would be the concept of house construction, while making a house it requires some steps such as building the roof, floor, walls electricity supply etc etc. and the client (or the owner) can customize these components and get different type of house.

Use Cases

Template method is widely used in frameworks, lets take Reactjs

React's Class Component is a implements template method where it has placeholder methods of componentDidMount, componentWillUnmount etc etc and the client will override and customize these methods as per their needs.

Btw fun fact, these kind of inversion of control is called "the Hollywood principle" ("don't call us, we'll call you".)

- Example

code on github

```
// template method
import fs from 'fs';

abstract class DataParser {
  data: string;
  out: any;
  constructor() {
    this.data = '';
```

```
    this.out = null;
  }

  parse(pathUrl: string) {
    this.readFile(pathUrl);
    this.doParsing();
    this.postCalculations();
    this.printData();
  }

  readFile(pathUrl: string) {
    this.data = fs.readFileSync(pathUrl, 'utf8');
  }

  doParsing() {}
  postCalculations() {}
  printData() {
    console.log(this.out);
  }
}

class DateParser extends DataParser {
  doParsing() {
    let dateRegx = /(0?[1-9]|[12][0-9]|3[01])[\/\-](0?[1-9]|1[012])[\/\·
    this.out = this.data.match(dateRegx);
  }
}

class CSVParser extends DataParser {
  doParsing() {
    this.out = this.data.split(',');
  }
}

class MarkupParser extends DataParser {
  doParsing() {
    this.out = this.data.match(/<\w+>.*<\/\w+>/gim);
  }

  postCalculations() {
    this.out = this.out.reverse();
  }
```

```
}

const dataUrl = '../../behavioral/data.csv';

new DateParser().parse(dataUrl);
new CSVParser().parse(dataUrl);
new MarkupParser().parse(dataUrl);
```

Day 19

- **Visitor Pattern**

Visitor design pattern is a behavioural design pattern which lets you define new operations/behaviours without changing the classes.

Real world Analogy

A good real world analogy is given by refactoring.guru where it says imagine a insurance agent who is eager to get new customers, he will visit every building on the region and

- if it’s a residential building, he sells medical insurance,
- If it’s a bank, he sells theft insurance.
- If it’s a shop, he sells fire and flood insurance.

Use cases

Visitor pattern is very useful when it comes with extending existing behaviours without changing the base class.

If you every wrote a GraphQL directive you've used visitor pattern.

GraphQL server exposes a "SchemaDirectiveVisitor" class which has methods like "visitFieldDefinition" & "visitEnumValue"
it implements the visitor pattern to add extra behaviours to the schemas.

Visitor pattern is also really useful for modifying AST trees where you can visit every node and modify it one by one. i
have a example on my git repo:
https://github.com/anuraghazra/design-patterns-everyday

You can also implement visitor pattern to make Exporters as you can see on my example. i have a SVGExporter and CanvasCallsExporter.

- Example

Check out the example on github

ANNNDDD THATS IT! Phew! it was long.. **i know you probably did not read it but thats ok**, you can come back anytime
when you are stuck with a specific design pattern or confused about it.

Personally i feel like in web development world most useful patterns are:

- Observer
- Visitor
- Iterator
- Chain of responsibility

- Strategy
- Proxy
- Decorator

Links:

- Design Patterns Everyday Github Repo
- Twitter #DesignPatternsEveryday

Learning Resources:

- Refactoring Guru
- Source Making

I hope you find this post useful! Thanks for reading folks.

Did you find this article helpful?
Sign in and leave a thank you comment ❤️

Anurag Hazra + FOLLOW

javascript lover, reactjs enthusiastic, gatsbyjs fanboi. Also Canvas, Web Interactivity, Physics Simulations, P5js, UI Design. #CreativeCoding

@anuraghazra anuraghazru anuraghazra anuraghazra.github.io

Discussion

Add to the discussion

PREVIEW SUBMIT

Oziel Perez

May 22

Holy God! This needs to be an ebook!

6

REPLY

Anurag Hazra

May 23

hehe, Thank you but i learned from these sources and i think these two books deserves some kudos!

refactoring.guru/design-patterns/
sourcemaking.com/

2

REPLY

Matthew

May 21

I'm trying to write a library for scaffolding test files for JS, and I think this post is really going to help a lot in that! Thanks for posting!

5

REPLY

Anurag Hazra

May 22

Great! Happy to help. :D

1

REPLY

Matthew

May 22

Yeah man - I forked the repo and ran my tool keurig
against the creational folder and found some things I needed to change!

1

REPLY

Julian Mojico

May 21

Nice Article Anurag.. Personally I need to refresh my knowledge in patterns so I will read one per day (and leave my comments, I hope that is ok!)

Day 1:Abstract Factory Pattern:
It would be good to have a summary before the ending. Most important thing to mention here:

"The benefit of this pattern is that no matter which family/factory your `guiFactory` *instance is, you will ALWAYS implement using this code:*

```
createButton(); and button.render());
```

This is where the flexibility of pattern comes in."

Day 2: Builder pattern
Missing the final step:
_"When you have finished building your object, you should call build() to instantiate:

```
let myTruck = truck.build();
let mySedan = sedan.build();
```

"

3

REPLY

Anurag Hazra

May 22

@jmojico Opps yup! didn't noticed it the .build() part, fixed it.

2

REPLY

Anurag Hazra

May 21

Great. Thanks for the feedback Julian. 😊

1

REPLY

Larissa Iurk

May 21

I started to study design patters yesterday, and in the beging I tought it is a little bit complicated. Your article really make it more easier to understand, the code examples realy helped me! Thanks!

4 REPLY

Anurag Hazra

May 21

Btw if you want you can also take the #DesignPatternsEveryday challenge.

twitter.com/hashtag/DesignPatterns...

1 REPLY

Anurag Hazra

May 21

Wow nice to hear that. I'm glad it helped you. 😄

1 REPLY

dev-arjunan

May 21

Thanks for sharing this. One of the best posts I have come across in dev.to.keep writing 😃😃😃

2 REPLY

Anurag Hazra

May 21

Thank you arjun.. i'm glad you liked it.

1 REPLY

Cécile Fécherolle

May 22

So great to see theoretical concepts which are usually hard to grasp made simple with examples and menmonics based on real life, thanks a lot for this article!

2 REPLY


Anurag Hazra

May 22

Thanks! I'm glad to that you liked it.

1 REPLY

code of conduct - report abuse

Classic DEV Post from Jul 26 '19

🎩 JavaScript Enhanced Scss mixins! 🎩 concepts explained


Adam Crockett

In the next post we are going to explore CSS @apply to supercharge what we talk about here....

117 9

Another Post You Might Like

Consumer-Driven Contract Testing with Pact


Frank Rosner

Consumer-driven contract testing is an alternative to end-to-end tests. In this blog post we want to take a look at the basics of consumer-driven contract testing with Pact.

52 4

Another Post You Might Like

Covering these topics makes you a Javascript Interview Boss - Part 1

Abdelrhman Yousry

Here we are going to cover a lot of topics in a lite way cool features that js offers but even experienced people are afraid of it.

111 6

Reparenting is now possible with React

Paolo Longo - May 23

A better way to copy text to Clipboard in JavaScript

Hiep Bao Le - May 24

60fps JS while sorting, mapping and reducing millions of records (with idle-time coroutines)

Mike Talbot - May 24

An Open Source Maintainer's Guide to Publishing npm Packages

Kadi Kraman - May 23