# Design patterns in JavaScript

**This is about all 23 design patterns from the Gang of Four book** "Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional", **in JavaScript**.

**Design patterns are similar to algorithms, they are are more general, but can still be seen as machines, or ways to make machines**.

These are patterns for the object oriented programming, but they can also be implemented in languages that are not object oriented. More than that, these are general patterns, and here after every pattern I wrote an example from the real world. These patterns can also occur in biology or in human society, in many ways in the real world.

**What concerns the names, then the terms in programming are often confusing, they are often too general, when they really mean only some particular thing. So when you look at these design patterns, look at what they are really about, don't look at their names**.

**There is a reference to the pdf used in this video in the description, so you can read it and also run the code directly by clicking on the links, as long as the code remains in OneCompiler**.

Umbrello was used for UML diagrams.

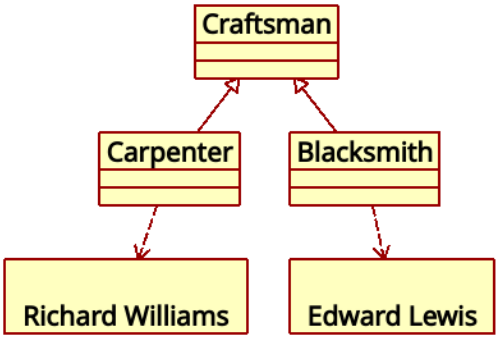
# Table of Contents

# Abstract Factory

https://onecompiler.com/javascript/3zvvtv3ba

**The Abstract Factory design pattern is about inheritance, thus it is a language construct, rather than a separate design pattern**. Factory means that some see creating instances of the classes like making objects in the factory, by blueprints. Several classes are inherited from the base class, we create objects from all these classes, and we can use these objects in the same way, by calling the same functions from these objects.

```
/* Abstract Factory, creational pattern */

class Craftsman {
	constructor(name) {
		this.name = name;
	}
	say = () => console.log("I am " + this.trade + " " +
this.name);
};

class Carpenter extends Craftsman {
	trade = "carpenter";
};

class Blacksmith extends Craftsman {
	trade = "blacksmith";
};

const carpenterFactory = name => new Carpenter(name);
const blacksmithFactory = name => new Blacksmith(name);

const craftsmen = [];
craftsmen.push(carpenterFactory("Richard Williams"));
craftsmen.push(blacksmithFactory("Edward Lewis"));
for (let craftsman of craftsmen) craftsman.say();
```



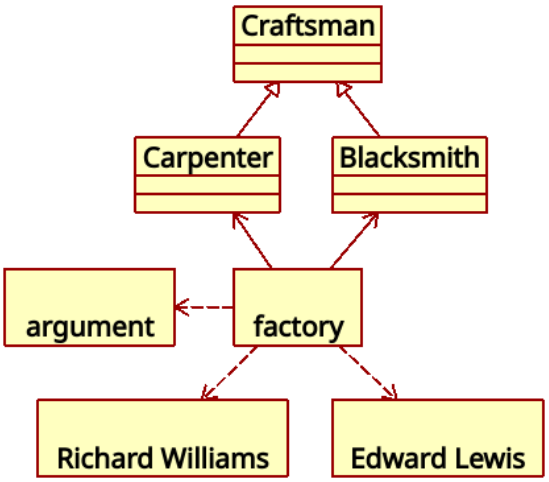

“Like different hammers, all made the same way, but have different weight.”

# Factory Method

https://onecompiler.com/javascript/3zvwrexzx

**The Factory Method design pattern is like Abstract Factory, except that there we decide by the argument of the factory() function, what class to use to create the object**. It is good in that there we use conditional statements only when creating the objects, and later in the code we no longer have to use the conditional statements, we just use the objects and use them in the same way.

```
/* Factory Method, creational pattern */

class Craftsman {
        constructor(name) {
        this.name = name;
        }
        say = () => console.log("I am " + this.trade + " " +
this.name);
};

class Carpenter extends Craftsman {
        trade = "carpenter";
};

class Blacksmith extends Craftsman {
        trade = "blacksmith";
};

const factory = (trade, name) => {
        if (trade == "carpenter") return new Carpenter(name);
        if (trade == "blacksmith") return new Blacksmith(name);
        return null;
}

const craftsmen = [];
craftsmen.push(factory("carpenter", "Richard Williams"));
craftsmen.push(factory("blacksmith", "Edward Lewis"));
for (let craftsman of craftsmen) craftsman.say();
```



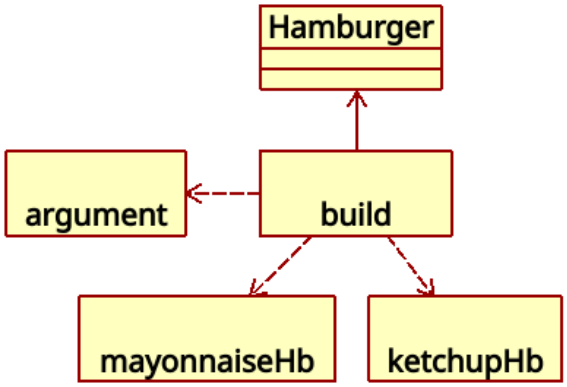

“Choose the right thing from the beginning, instead of deciding every time what to use.”

# Builder

https://onecompiler.com/javascript/3zvwkecfv

The Builder design pattern is about making something by adding things separately and selectively. What to add is determined by the argument of the build() function. **The Builder design pattern is somewhat similar to the Factory Method, except that we build the objects, instead of instantiating them from different classes**.

```
/* Builder, creational pattern */

class Hamburger {
	constructor () {
		this.bun = true;
		console.log("== New hamburger ==");
	}

	addKetchup = () => {
		this.ketchup = true;
		console.log("Ketchup added")
	}

	addMayonnaise = () => {
		this.mayonnaise = true;
		console.log("Mayonnaise added")
	}

	addPatty = () => {
		this.patty = true;
		console.log("Patty added")
	}

	addTomato = () => {
		this.tomato = true;
		console.log("Tomato added")
	}
};

const build = sauce => {
	hamburger = new Hamburger();
	if (sauce == "ketchup")
		hamburger.addKetchup();
	else
		hamburger.addMayonnaise();
	hamburger.addPatty();
	hamburger.addTomato();
	return hamburger;
}

const mayonnaiseHb = build("mayonnaise");
const ketchupHb = build("ketchup");
```



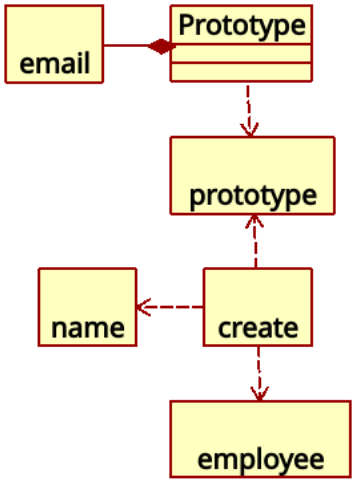

“Like we make hamburger, step by step.”

# Prototype

https://onecompiler.com/javascript/3zvwsfgr7

**The Prototype design pattern is implemented using language constructs for delegation, thus it is also properly not a separate design pattern**. It is about creating some object based on another object. This is how we can create some object with some fields already filled by default, but it can also used in various other ways as a delegation.

```
/* Prototype, creational pattern */

class Prototype {
	constructor(email) {
		this.email = email;
		this.name = "none";
	}
	setName = name => this.name = name;
	say = () => console.log("Name: " + this.name + ", Email: " +
		this.email);
};

const create = (name, prototype) => {
	const employee = Object.create(prototype);
	employee.setName(name);
	return employee;
}

const prototype = new Prototype("contact@company.com");
const employee = create("Edward Lewis", prototype);
employee.say();
```



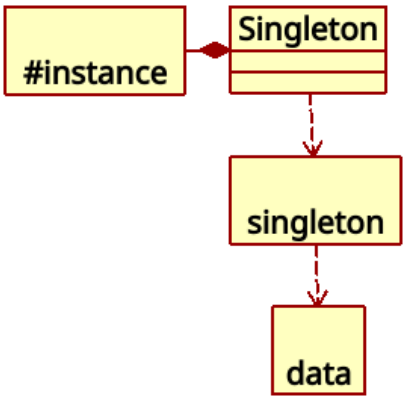

“Only certain kind of plants grow from certain kind of seeds.”

# Singleton

https://onecompiler.com/javascript/42684hknb

The Singleton design pattern allows to instantiate class only once. The class has a static private property #instance, and the constructor throws an error if there already is an instance. The private property #data contains the data provided when the class was instantiated, after which it cannot be changed. **Thus Singleton design pattern enables to store data once, after which it is like a constant, and cannot be changed in any way**.

```
/* Singleton, creational pattern */

class Singleton {
	constructor(data) {
		if (Singleton.#instance)
			throw new Error("Can only be instantiated once");
		Singleton.#instance = this;
		this.#data = data;
	}
	#data = null;
	static #instance = null;
	static getInstance = () => this.#instance;
	getData = () => this.#data;
};

let singleton = new Singleton({x: 100, y: 120, width: 128,
	height: 128});
singleton = Singleton.getInstance();
console.log(singleton.getData());
```



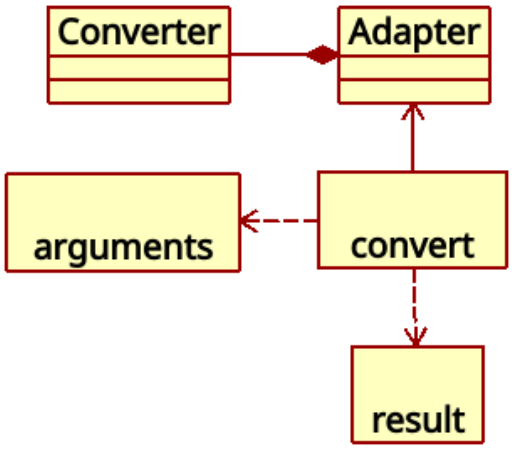

"Land marker, once put there, is expected not to be moved."

# Adapter

https://onecompiler.com/javascript/3zvwga744

**The Adapter design pattern is about composition, thus it is about a language construct and is not properly a separate design pattern**. We have a class where things are done in a certain way, and we have to create a new class, using an object of that class inside it, and do it there in some other way, that fits our needs.

```
/* Adapter, structural pattern */

class Converter {
	kmToMiles = l => l * 0.6213712;
	milesToKm = l => l * 1.609344;
};

class Adapter {
	converter = new Converter();
	convert = (l, operation) => {
		if (operation == "to miles")
			return this.converter.kmToMiles(l);
		return this.converter.milesToKm(l);
	}
};

const convert = (l, operation) => {
	const adapter = new Adapter();
	return adapter.convert(l, operation);
}

console.log(convert(7, "to miles"));
```



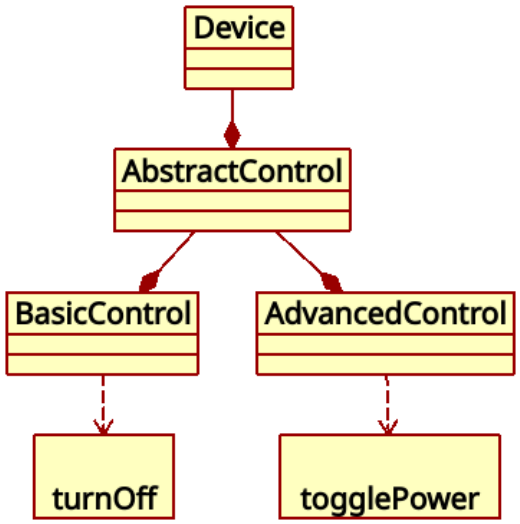

"Like horseshoe, horses are not animals that can well walk on paved roads, but they can with horseshoes."

# Bridge

https://onecompiler.com/javascript/3zvwk8vej

**The Bridge design pattern is like an Adapter for something abstract. Thus it is also not properly a separate design pattern**. There can be several adapters for the abstract base class. We use an object of that abstract class in other classes, just as we did in Adapter, but because the object is abstract and general, we can extend it in several different ways.

```
/* Bridge, structural pattern */

class Device {
	turnOn = () => console.log("Device is ON");
	turnOff = () => console.log("Device is OFF");
};

class AbstractControl {
	constructor(device) {
		this.device = new Device();
	}
	turnOn = () => this.device.turnOn();
	turnOff = () => this.device.turnOff();
};

class BasicControl {
	constructor(device) {
		this.control = new AbstractControl(device);
	}
	turnOn = () => this.control.turnOn();
	turnOff = () => this.control.turnOff();
};

class AdvancedControl {
	constructor(device) {
		this.control = new AbstractControl();
		this.on = false;
	}
	togglePower = () => {
		if (this.on)
			this.control.turnOff();
		else
			this.control.turnOn();
	}
};

const device = new Device();
const basicControl = new BasicControl(device);
basicControl.turnOff();
const advancedControl = new AdvancedControl(device);
advancedControl.togglePower();
```



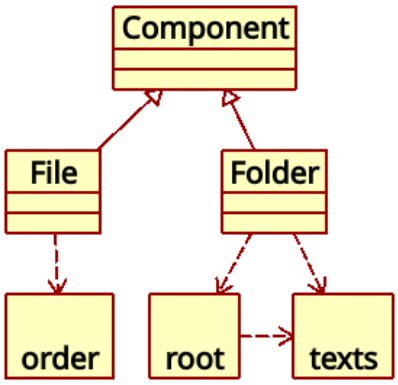

"Like adapter but extends something abstract, so there can be multiple adapters."

# Composite

https://onecompiler.com/javascript/3zvwqf837

**The Composite design pattern is a tree-like structure**. The branches of the tree can be handled in the same way, so we can add branches and to these branches we can likewise add other branches. Finally we have so-called leaf, here File is the leaf, that cannot be further extended. Here a file structure is implemented, we create a folder “root”, to that we add a folder “texts”, and to that we add a file.

```
/* Composite, structural pattern */

class Component {
	constructor(name) {
		this.name = name;
		this.files = [];
		this.type = "Node";
	}
	print = () => {
		console.log(this.type + " " + this.name);
		for (let file of this.files) file.print();
	}
};

class Folder extends Component {
	constructor(name) {
		super(name);
		this.type = "Folder";
	}
	add = file => this.files.push(file);
	delete = file => {
		const index = this.files.indexOf(file);
		if (index !== -1) this.files.splice(index, 1);
	}
};

class File extends Component {
	constructor(name) {
		super(name);
		this.type = "File";
	}
};

const order = new File("order.txt");
const texts = new Folder("texts");
texts.add(order);
const root = new Folder("root");
root.add(texts);
root.print();
```



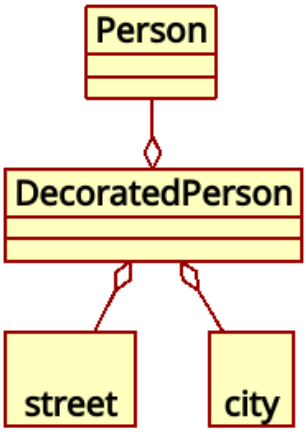

“One knows how many tree-like things there are everywhere.”

# Decorator

https://onecompiler.com/javascript/3zvwr3zb2

**The Decorator design pattern is about adding things to an object**. Here we have a class Person, we add some properties to the object of that class, and override the function.

```
/* Decorator, structural pattern */

class Person {
	constructor(name) {
		this.name = name;
	}
	say = () => console.log("Name: " + this.name);
};

class DecoratedPerson {
	constructor(person, street, city) {
		this.person = person;
		this.name = person.name;
		this.street = street;
		this.city = city;
	}
	say = () => {
		this.person.say();
		console.log(this.street + ", " + this.city);
	}
};

const person = new Person("Brian");
const decoratedPerson =
	new DecoratedPerson(person, "Church Street", "Orlando");
decoratedPerson.say();
```



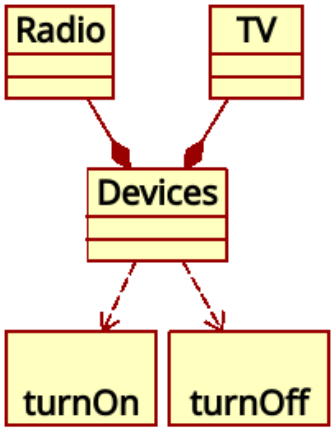

“We add things to our house, like new furniture.”

# Facade

https://onecompiler.com/javascript/3zvwr9yww

**The Facade design pattern is about doing some complex things in a simple way**. Like here in the class Devices we use only two methods to turn on and off two different devices. **In essence this is also about composition, and a use of a language construct**.

```
/* Facade, structural pattern */

class Radio {
	turnOn = () => console.log("Radio turned on");
	turnOff = () => console.log("Radio turned off");
};

class TV {
	turnOn = () => console.log("TV turned on");
	turnOff = () => console.log("TV turned off");
};

class Devices {
	#radio = new Radio();
	#tv = new TV();

	turnOn = () => {
		this.#radio.turnOn();
		this.#tv.turnOn();
	}

	turnOff = () => {
		this.#radio.turnOff();
		this.#tv.turnOff();
	}
};

const devices = new Devices();
devices.turnOn();
devices.turnOff();
```



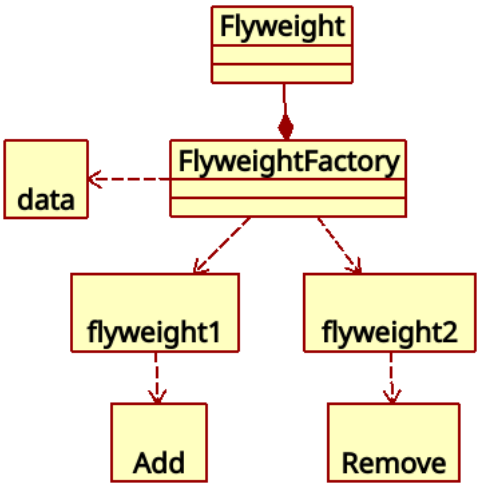

"When traffic light goes red, many things happen."

# Flyweight

https://onecompiler.com/javascript/3zvwrkbkj

The Flyweight design pattern is about many similar objects, and they all are created and accessed in the same way. There are intrinsic properties that we provide when creating the objects. The extrinsic properties are about the operations that we are going to do to these objects. **The Flyweight design pattern is in essence also about using composition, but for many objects that are usually kept in an array or in some other data structure**.

```
/* Flyweight, structural pattern */

class Flyweight {
	constructor(sharedData) {
		this.sharedData = sharedData;
	}
	op = uniqueData => console.log("Intrinsic: " +
		this.sharedData, " Extrinsic: " + uniqueData);
};

class FlyweightFactory {
	flyweights = {};
	getData = sharedData => {
		if (!this.flyweights[sharedData])
			this.flyweights[sharedData] =
				new Flyweight(sharedData);
		return this.flyweights[sharedData];
	}
	count = () => Object.keys(this.flyweights).length;
};

const factory = new FlyweightFactory();
const flyweight1 = factory.getData("A");
flyweight1.op("Add");
const flyweight2 = factory.getData("B");
flyweight2.op("Remove");
console.log("Number of flyweights: " + factory.count());
```



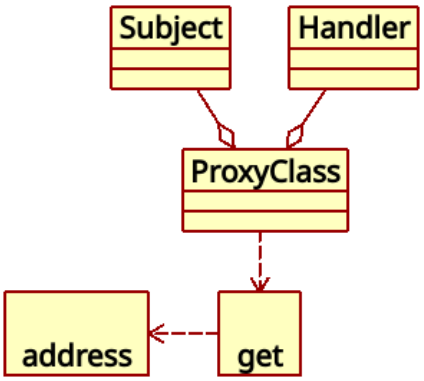

“Like many pencils that have different colors, but can be used in the same way.”

# Proxy

https://onecompiler.com/javascript/3zvwsnvwu

**The Proxy design pattern is like Adapter, that can access different data**. The data is provided as an argument when instantiating a proxy. Here the data is in the class Subject. Also provided as an argument, is an object of the class Handler. Similar to the model-view pattern, handlers can provide data in different formats, thus different handler can be provided for proxy to access data in different format. The ProxyClass has the method get(), with which we can access all the elements of the data.

```
/* Proxy, structural pattern */

class Subject {
	cities = {
		"Kansas City": "39°05′59″N 94°34′42″W",
		"Philadelphia": "39°57′10″N 75°09′49″W",
		"Orlando": "28°32′24″N 81°22′48″W"
	};
	get = address => this.cities[address];
};

class Handler {
	cache = {};
	get = (subject, address) => {
		if (!this.cache[address])
			this.cache[address] = subject.get(address);
		console.log(address + ": " + this.cache[address]);
		return this.cache[address];
	}
	 count = () => Object.keys(this.cache).length;
};

class ProxyClass {
	constructor(subject, handler) {
		this.subject = subject;
		this.handler = handler;
	}
	get = address =>
		this.handler.get(this.subject, address);
	count = () => this.handler.count();
};

const subject = new Subject();
const handler = new Handler();
const proxy = new ProxyClass(subject, handler);
proxy.get("Philadelphia");
proxy.get("Orlando");
proxy.get("Kansas City");
proxy.get("Kansas City");
proxy.get("Orlando");
console.log("Cache length: " + proxy.count());
```



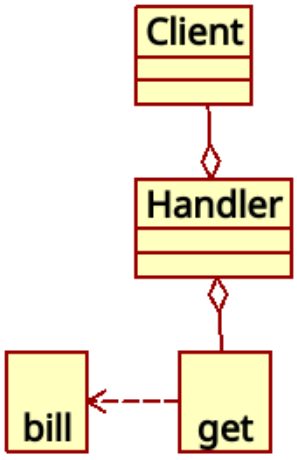

“It is like adapter, but more specific, mostly used to hide some access.”

# Chain of Responsibility

https://onecompiler.com/javascript/3zvwkur7c

**In the Chain of Responsibility design pattern, a sequence of operations are done on the data provided by the client**. The processing is done by Handler and its method get(). Here the sequence of operations is provided by fluent interface. In the get() method we first calculate how many of the bills, provided as an argument, go into the sum, and when none goes, then we do nothing. Thus we only apply this method when it needs to be applied.

```
/* Chain of Responsibility, behavioral pattern */

class Client {
	constructor(sum) {
		this.sum = sum;
		console.log("Request $" + sum);
	}
};

class Handler {
	constructor(client) {
		this.sum = client.sum;
	}
	get = bill => {
		const count = Math.floor(this.sum / bill);
		if (!count) return this;
		this.sum -= count * bill;
		console.log("Paid " + count + " $" + bill + " bills");
		return this;
	}
};

const client = new Client(246);
const handler = new Handler(client);
handler.get(100).get(50).get(20).get(10).get(5).get(1);
```



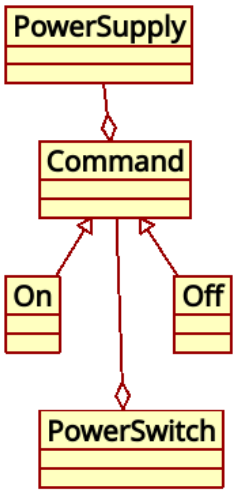

“We go pass the shelves in the store, and where is something that we need, we pick it.”

# Command

https://onecompiler.com/javascript/3zvwkzsh6

**The Command design pattern is about dependency injection, that is not properly a separate design pattern**. We have the base class Command, and we extend it in different ways, creating different commands. When we do set on, then the on there is really an object, and off is also an object, that do certain things, in this case they switch on and off the power.

```
/* Command, behavioral pattern */

class Command {
	constructor(power) {
		this.power = power;
	}
};

class On extends Command {
	execute = () => this.power.on();
};

class Off extends Command {
	execute = () => this.power.off();
};

class PowerSupply {
	on = () => console.log("Power is on");
	off = () => console.log("Power is off");
};

class PowerSwitch {
	set = command => command.execute();
};

const power = new PowerSupply();
const on = new On(power);
const off = new Off(power);
const powerSwitch = new PowerSwitch();
powerSwitch.set(on);
powerSwitch.set(off);
```



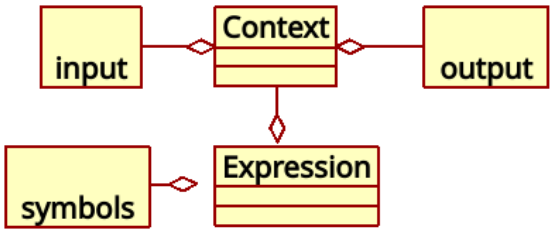

“We push the break pedal, and many things happen that make the car to stop.”

# Interpreter

https://onecompiler.com/javascript/3zw4394pw

**The Interpreter design pattern is like a translator, that translates one sequence into another**. Here is the translation of Roman numbers. We go through the sequence of a Roman number. We iterate through a list of subsequences, called symbols, that are simple forms of expressions, and every time we create an object of class Expression, that represents the symbol. That class contains an object of class Context, that contains the input that is not yet processed, and the current state of output, that will be the final number. The Expression contains the method interpret(), that repeatedly finds whether the context input starts with the symbol, and if it does, adds the corresponding value to the context output.

```
/* Interpreter, behavioral pattern */

class Context {
	constructor(input) {
		this.input = input;
		this.output = 0;
	}
	match = str => {
		const next = this.input.substr(0, str.length);
		if (next != str) return false;
		this.input = this.input.substr(next.length);
		return true;
	}
};

class Expression {
	constructor(context, symbol, value) {
		this.context = context;
		this.symbol = symbol;
		this.value = value;
	}
	interpret = () => {
		while (this.context.match(this.symbol))
			this.context.output += this.value;
	}
};

const roman = "MCMXXVIII"
const context = new Context(roman);
const symbols = {"M": 1000, "CM": 900, "D": 500, "CD": 400, "C": 100,
  "XC": 90, "L": 50, "XL": 40, "X": 10, "IX": 9, "V": 5, "IV": 4,
  "I": 1};
for (let key in symbols) {
	const expression = new Expression(context, key, symbols[key]);
	expression.interpret();
}
console.log(roman + " = " + context.output);
```



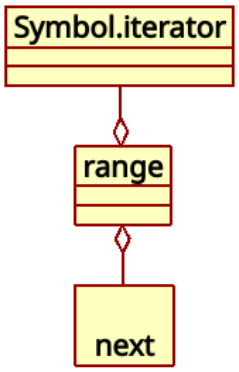

“An RNA codes a protein molecule.”

# Iterator

https://onecompiler.com/javascript/3zvwrvptz

**The Iterator design pattern is a language construct, and is not properly a separate design pattern**. It is about how to implement iterator in a programming language. Here is the implementation in JavaScript, so that we can use the function range(), as an iterator. Important there is the function next, that provides two things, the next value, and whether we finished the iteration or not.

```
/* Iterator, behavioral pattern */

const range = (start, end, step = 1)  => {
        return {
                [Symbol.iterator]() {
                        return this;
                },

                next() {
                        if (start >= end) return {value: end, done: true};
                        start += step;
                        return {value: start, done: false};
                }
        }
}

for (const n of range(0, 20, 5)) console.log(n);
```



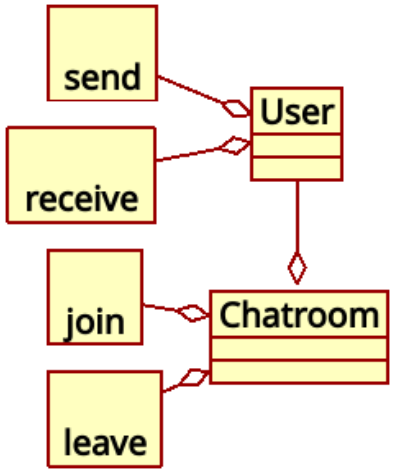

"Like a ladder, every next step brings us to new height."

# Mediator

https://onecompiler.com/javascript/3zvws2qpg

**The Mediator design pattern is about something central, that deals with many. Important there is that these many are independent objects, that use mediator for certain purposes**. The example here is an implementation of a chatroom. Users can join and leave the room, and send messages. The task of the chatroom as a mediator, is to send messages to all users that are in the room at any particular time.

```
/* Mediator, behavioral pattern */

class User {
	constructor(name) {
		this.name = name;
		this.chatroom = null;
	}
	send = message => this.chatroom.send(message, this);
	receive = (message, from) => console.log(from.name + " to " +
		this.name + ": " + message);
};

class Chatroom {
	users = {};
	join = user => {
		this.users[user.name] = user;
		user.chatroom = this;
	}
	leave = user => delete this.users[user.name];
	send = (message, from) => {
		for (let key in this.users)
			if (this.users[key] !== from)
				this.users[key].receive(message, from);
	}
};

const robert = new User("Robert");
const donald = new User("Donald");
const paul = new User("Paul");
const george = new User("George");
const chatroom = new Chatroom();
chatroom.join(robert);
chatroom.join(donald);
chatroom.join(paul);
chatroom.join(george);
chatroom.leave(donald);
robert.send("Hello everyone");
```



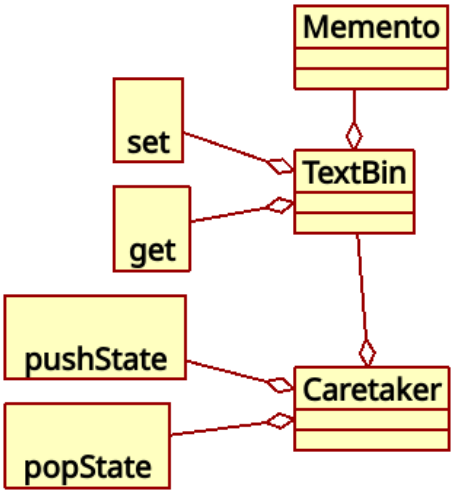

"Everything that deals with many, such as a store."

# Memento

https://onecompiler.com/javascript/3zvws7rwp

**The Memento design pattern is about remembering something, so that it can be restored later**. This is often used for the undo operation in different applications. In the Caretaker we have the history, where we can push the state represented by Memento. We can also pop the state, then the content goes back to what it was when it was pushed. Here we set the changed text, but after we pop the state, we really get back the first text.

```
/* Memento, behavioral pattern */

class TextBin {
        content = "";
        newMemento = () => new Memento(this.content);
        restore = memento => this.content = memento.get();
        set = content => this.content = content;
        get = () => this.content;
};

class Memento {
        constructor(content) {
                this.content = content;
        }
        get = () => this.content;
};

class Caretaker {
        history = [];
        pushState = textBin => this.history.push(textBin.newMemento());
        popState = textBin => textBin.restore(this.history.pop());
}

const textBin = new TextBin();
const caretaker = new Caretaker();
textBin.set("First text");
caretaker.pushState(textBin);
textBin.set("Changed text");
caretaker.popState(textBin);
console.log(textBin.get());
```



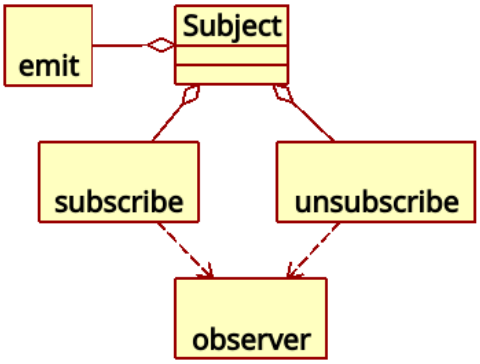

"You make and eat a dinner, and then restore everything in the kitchen to how it was before."

# Observer

https://onecompiler.com/javascript/3zvwsbjsk

**The Observer design pattern is about a number of observers as functions that observe the events, it provides subscribing and unsubscribing the observers, and processing the event**. Here the observer is implemented by the class Subject. When an event occurs, then the observers are called, and an observer does a certain operation that is necessary to handle the event or for whatever else.

```
/* Observer, behavioral pattern */

class Subject {
	observers = {};
	subscribe = (key, observer) => this.observers[key] = observer;
	unsubscribe = key => delete this.observers[key];
	emit = e => {
		for (let key in this.observers)
			this.observers[key](e);
	}
}

const observer1 = e => console.log("Observer 1 received: " + e);
const observer2 = e => console.log("Observer 2 received: " + e);
const subject = new Subject();
subject.subscribe("observer 1", observer1);
subject.subscribe("observer 2", observer2);
subject.emit("event 1");
subject.unsubscribe("observer 2");
subject.emit("event 2");
subject.subscribe("observer 2", observer2);
subject.emit("event 3");
```



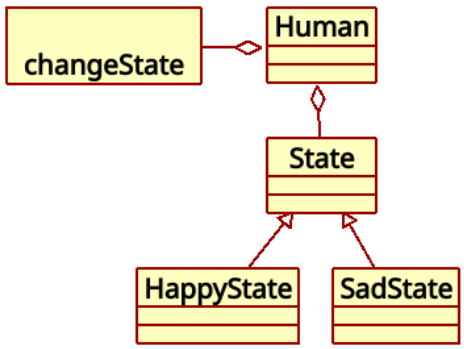

“A number of groundhogs observe events, when predator comes, they go to their burrows.”

# State

https://onecompiler.com/javascript/3zvwt7ny8

**The State design pattern is another example of dependency injection, that is not properly a separate design pattern**. It is about the state that everything has at any particular time. Here is the base class State and we can extend it to different kind of states. Then we can change the states of an object and use these states.

```
/* State, behavioral pattern */

class State {
	think = () => "";
};

class HappyState extends State {
	think = () => "I am happy :)";
};

class SadState extends State {
	think = () => "I am sad :(";
};

class Human {
	state = new HappyState();
	think = () => this.state.think();
	changeState = state => this.state = state;
}

const happy = new HappyState();
const sad = new SadState();
const human = new Human();
console.log(human.think());
human.changeState(sad);
console.log(human.think());
```



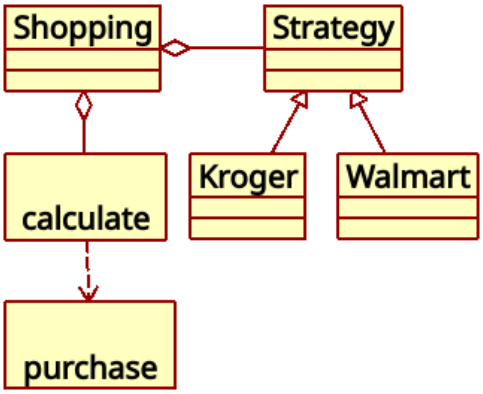

“It is sunny, or it is raining, these are states of the weather.”

# Strategy

https://onecompiler.com/javascript/3zvwtdvmp

**The Strategy design pattern is one more example of dependency injection, that is not properly a separate design pattern**. It is about different methods to process the same thing. Like when we set the strategy in Shopping to Walmart, the method in that class does a certain calculation. Here the calculation is not shown, but it may calculate the price somehow. We use an object of a purchase as an argument, and get the calculation results.

```
/* Strategy, behavioral pattern */

class Strategy {
	calculate = purchase => "";
};

class Walmart extends Strategy {
	calculate = purchase => "$3.26";
};

class Kroger extends Strategy {
	calculate = purchase => "$3.59";
};

class Shopping {
	strategy = null;
	setStrategy = strategy => this.strategy = strategy;
	calculate = purchase => this.strategy.calculate(purchase);
};

const purchase = {item: "milk", quantity: "1 gallon"};
const walmart = new Walmart();
const kroger = new Kroger();
const shopping = new Shopping();
shopping.setStrategy(walmart);
console.log("Walmart strategy: " + shopping.calculate(purchase));
shopping.setStrategy(kroger);
console.log("Kroger strategy: " + shopping.calculate(purchase));
```



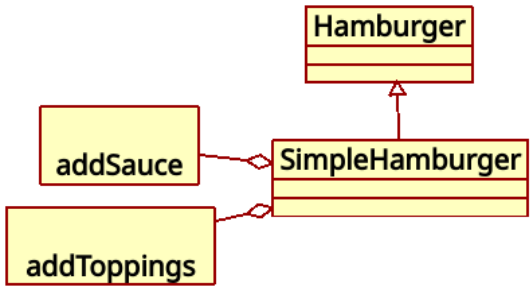

“We can use different mops to clean the floor, yet we clean the floor the same way.”

# Template Method

https://onecompiler.com/javascript/3zvwtj5b6

**The Template Method design pattern is about sequence of doing things**. This sequence is defined in the base class, together with the base abstract methods. When we extend that class to something concrete, then we add some methods to do some particular things. Then we call the method make(), that comes from the base class, that does all these things in sequence.

```
/* Template Method, behavioral pattern */

class Hamburger {
	make = () => {
		this.cutBun();
		this.addSauce();
		this.addPatty();
		this.addToppings();
		this.finish();
	}
	cutBun = () => console.log("Cutting bun to half");
	addSauce = () => {};
	addPatty = () => console.log("Add patty");
	addToppings = () => {};
	finish = () => console.log("Add top half of the bun");
};

class SimpleHamburger extends Hamburger {
	addSauce = () => console.log("Add mayonnaise");
	addToppings = () => console.log("Add tomato");
};

const simpleHamburger = new SimpleHamburger();
console.log("Making hamburger:");
simpleHamburger.make();
```



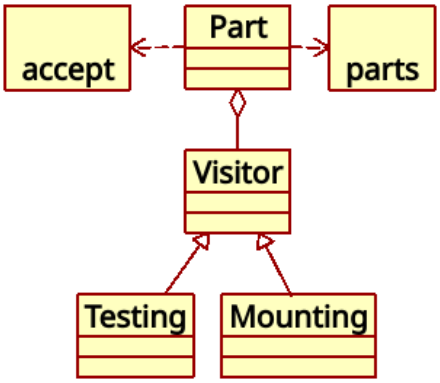

“What to eat during a day, for breakfast, for dinner, and for supper.”

# Visitor

https://onecompiler.com/javascript/3zvwtr3tj

**The Visitor design pattern is about doing things on certain sequence of objects, that may either accept or not accept the visitors**. There is an array of parts, that are provided as objects, and in the Part we can accept the visitor, that is an object that does certain operations on visited objects. The visitors here are Testing and Mounting. We use them for all the sequence of objects, but accept() in Part can also accept only some of the visitors.

```
/* Visitor, behavioral pattern */

class Visitor {
	visit = part => {};
};

class Testing extends Visitor {
	visit = part => {
		part.setPassedTest();
		console.log(part.getName() + " passed the test");
	}
};

class Mounting extends Visitor {
	visit = part => {
		part.setMounted();
		console.log(part.getName() + " is mounted");
	}
};

class Part {
	constructor(name) {
		this.name = name;
		this.passedTest = false;
		this.mounted = false;
	}
	accept = visitor => visitor.visit(this);
	getName = () => this.name;
	setPassedTest = () => this.passedTest = true;
	setMounted = () => this.mounted = true;
};

const parts = [
	new Part("Power supply"),
	new Part("Fan"),
];

const testing = new Testing();
const mounting = new Mounting();
for (let part of parts) {
	part.accept(testing);
	part.accept(mounting);
}
```




“One postman delivers letters, and the other delivers newspapers.”