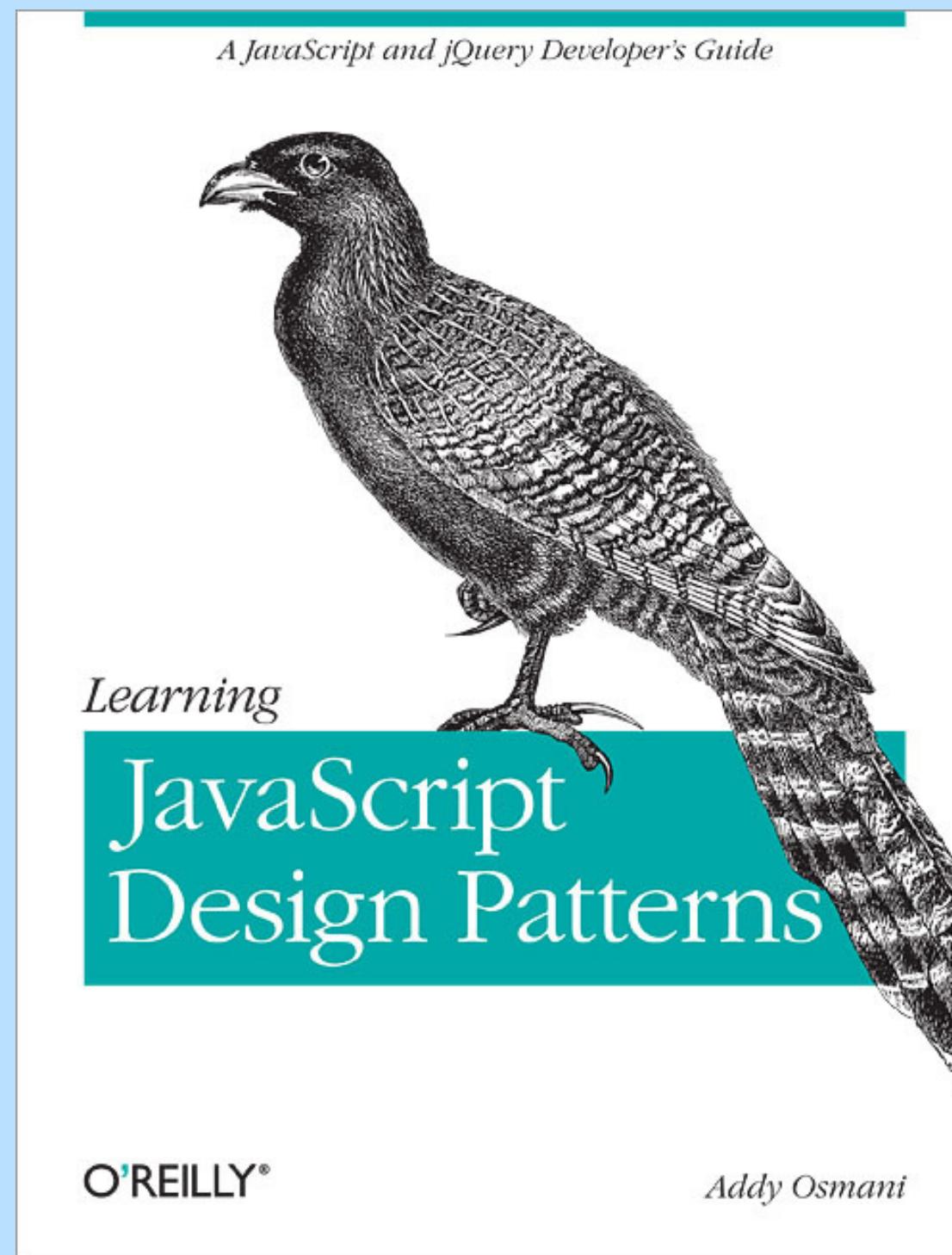


BEHAVIORAL PATTERNS

Vanessa Espitia
Diana Aviles





ÍNDICE

01 Patrones de diseño

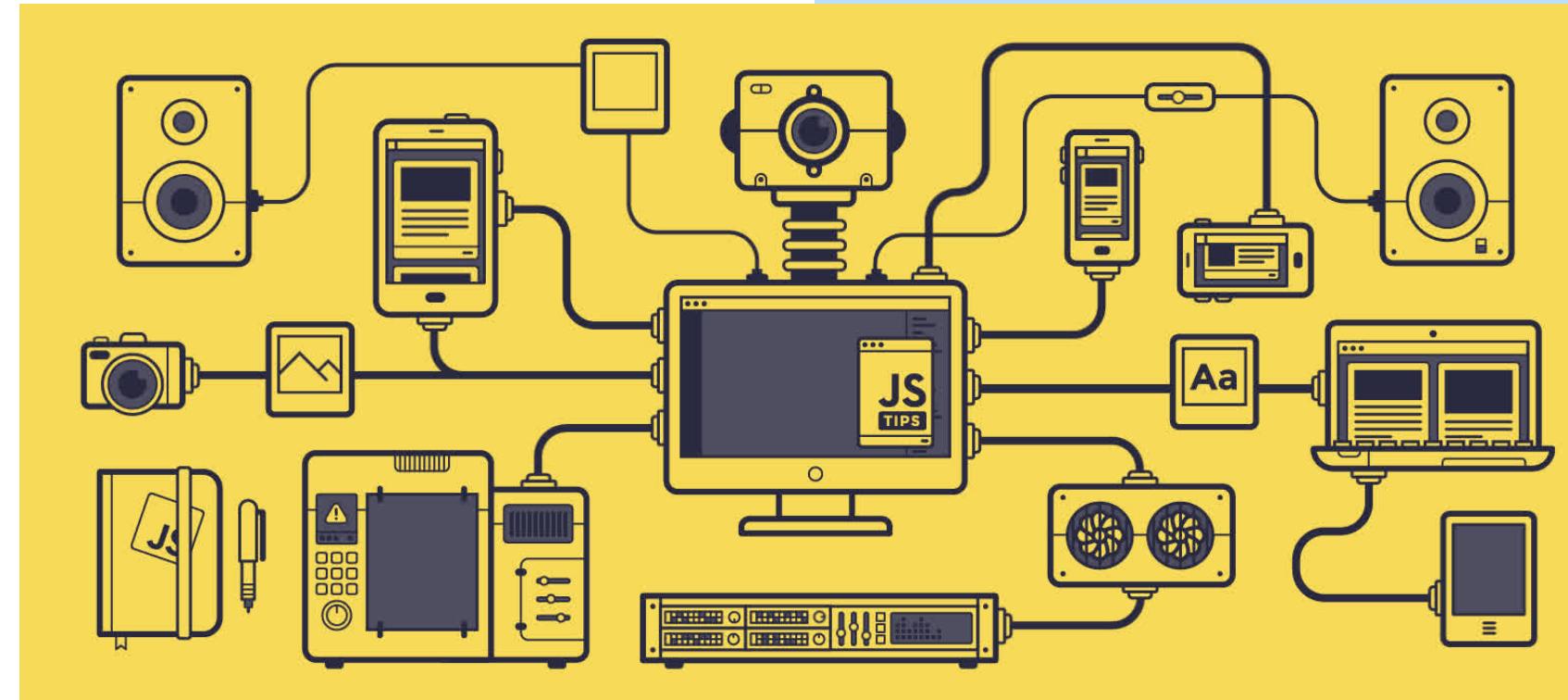
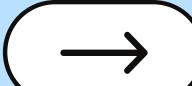
02 Behavioral Pattern

03 Patrones de comportamiento que incluyen

04 Resumen

05 Ejercicio Final

PATRÓN DE DISEÑO

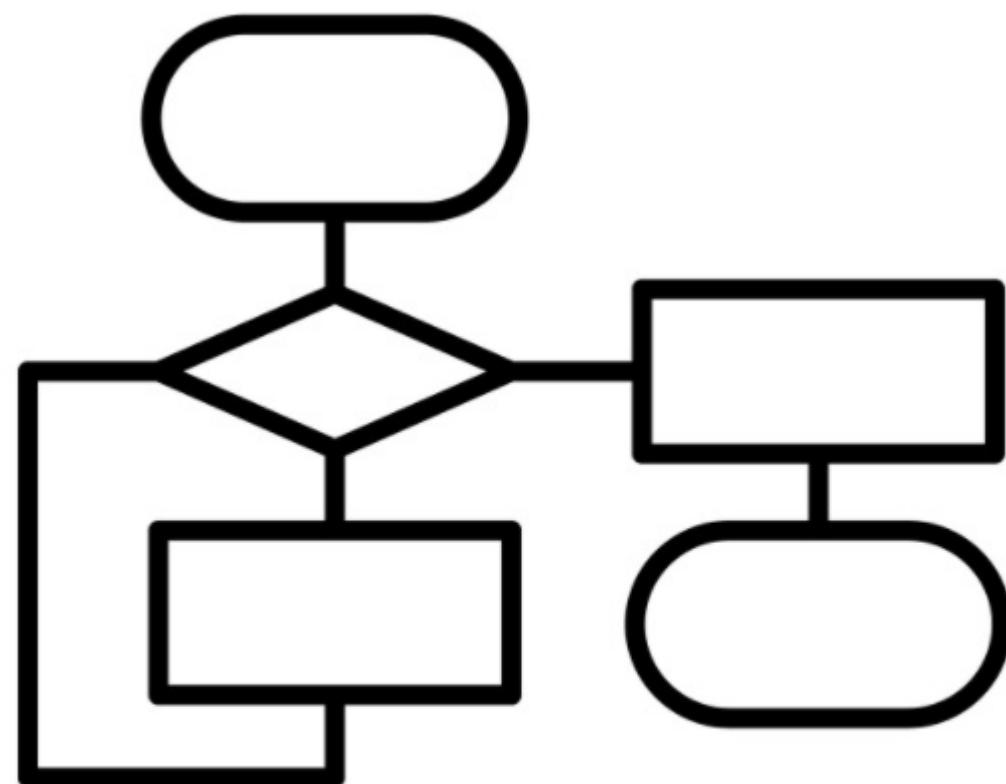
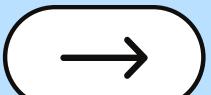


- Nombra, resume e identifica los aspectos clave
- Identifica clases participantes y sus instancias, sus roles y colaboraciones, y la distribución de responsabilidades

Se basan en soluciones prácticas que se han implementado en varios lenguajes



PATRÓN DE DISEÑO



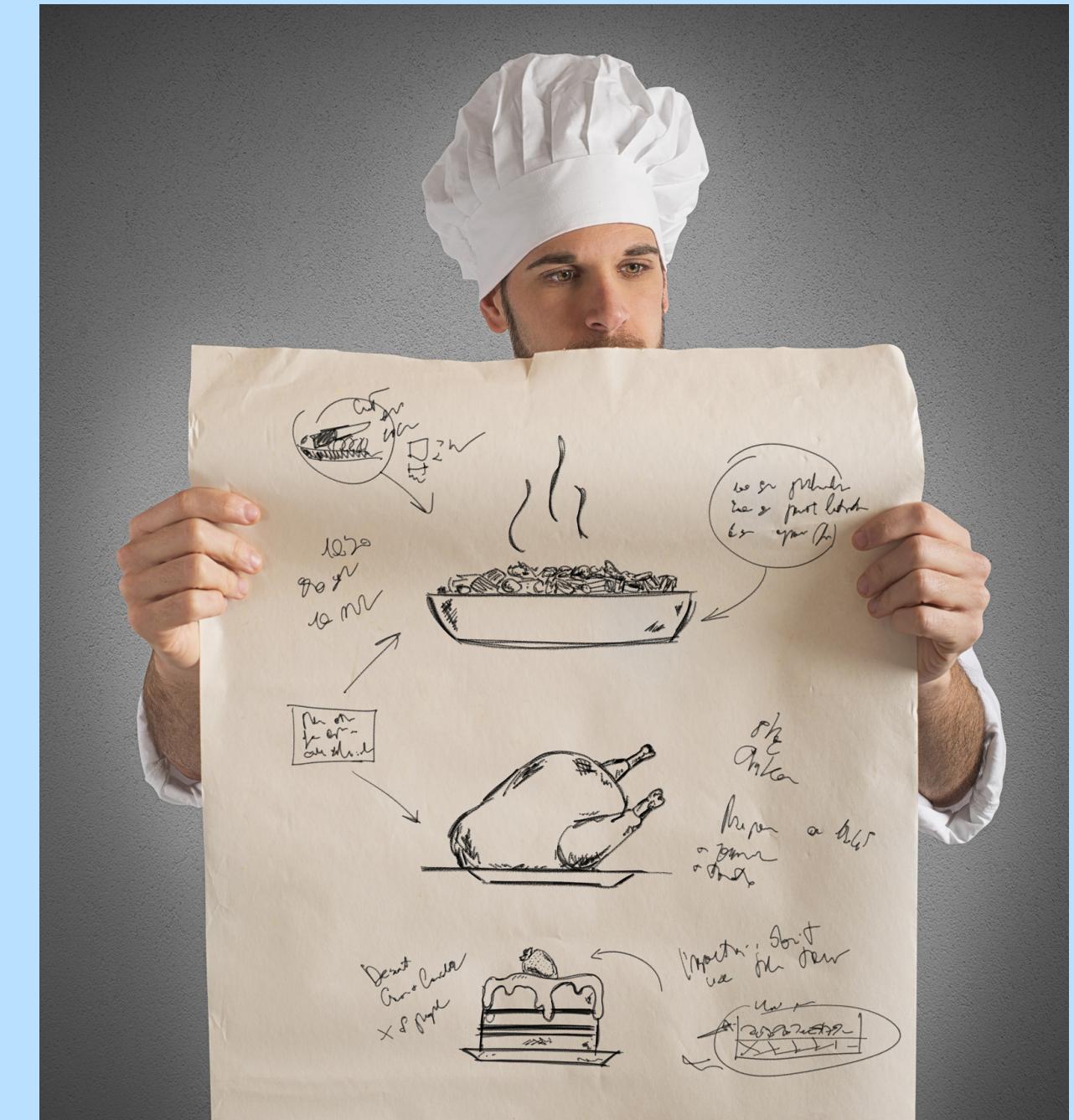
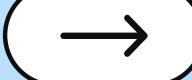
ALGORITMO



PATRÓN DE DISEÑO



ALGORITMO



CATEGORÍAS

- **PATRONES DE CREACIÓN**

Proporcionan mecanismos de creación de objetos.

Mariana y Steve

- **PATRONES ESTRUCTURALES**

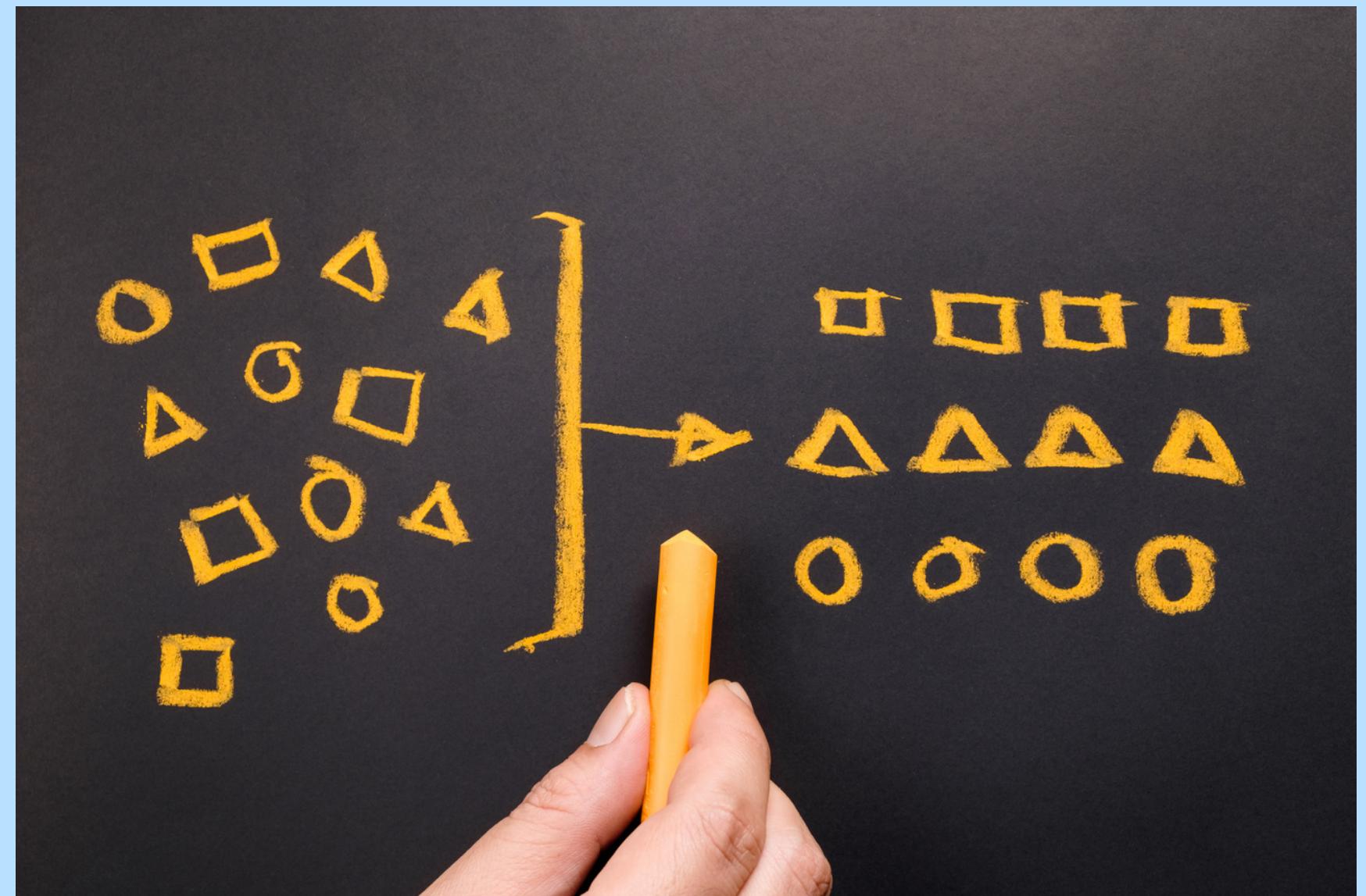
Explican cómo ensamblar objetos y clases

Raúl y César

- **PATRONES DE COMPORTAMIENTO**

Mejoran o simplifican la comunicación entre objetos

Vanessa y Diana

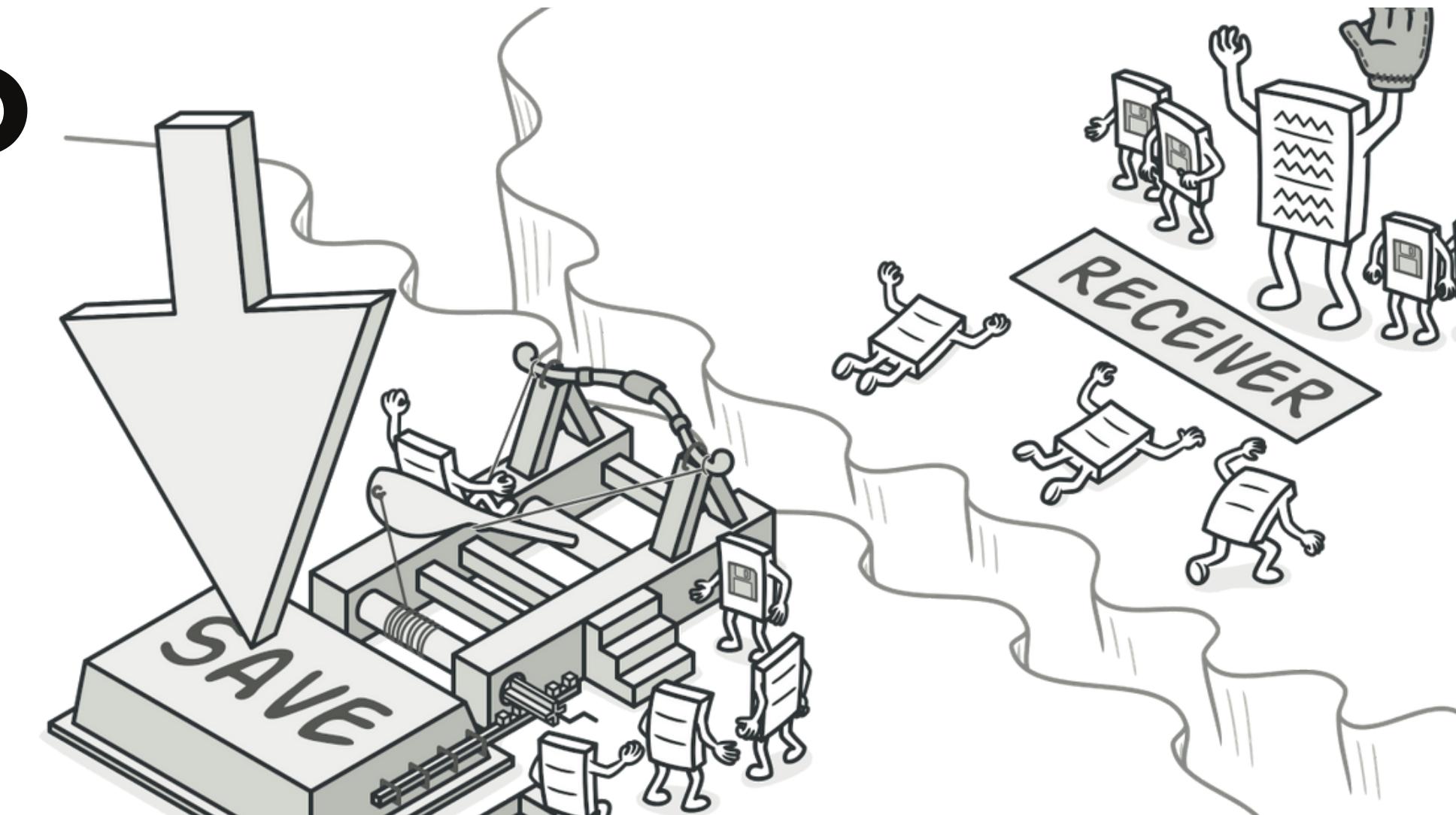


● ○ ✕ ●

+ ✗ ● ○

Patrones de comportamiento

Los patrones de comportamiento se preocupan de proporcionar soluciones con respecto a la interacción de objetos



Están relacionados con algoritmos y la asignación de responsabilidades entre objetos



JavaScript

Patrones que lo conforman

Patrones de comportamiento

- Cadena de responsabilidad
- Comando
- Iterador
- Mediador
- Memento
- Estado
- Estrategia
- Método de plantilla
- Visitante



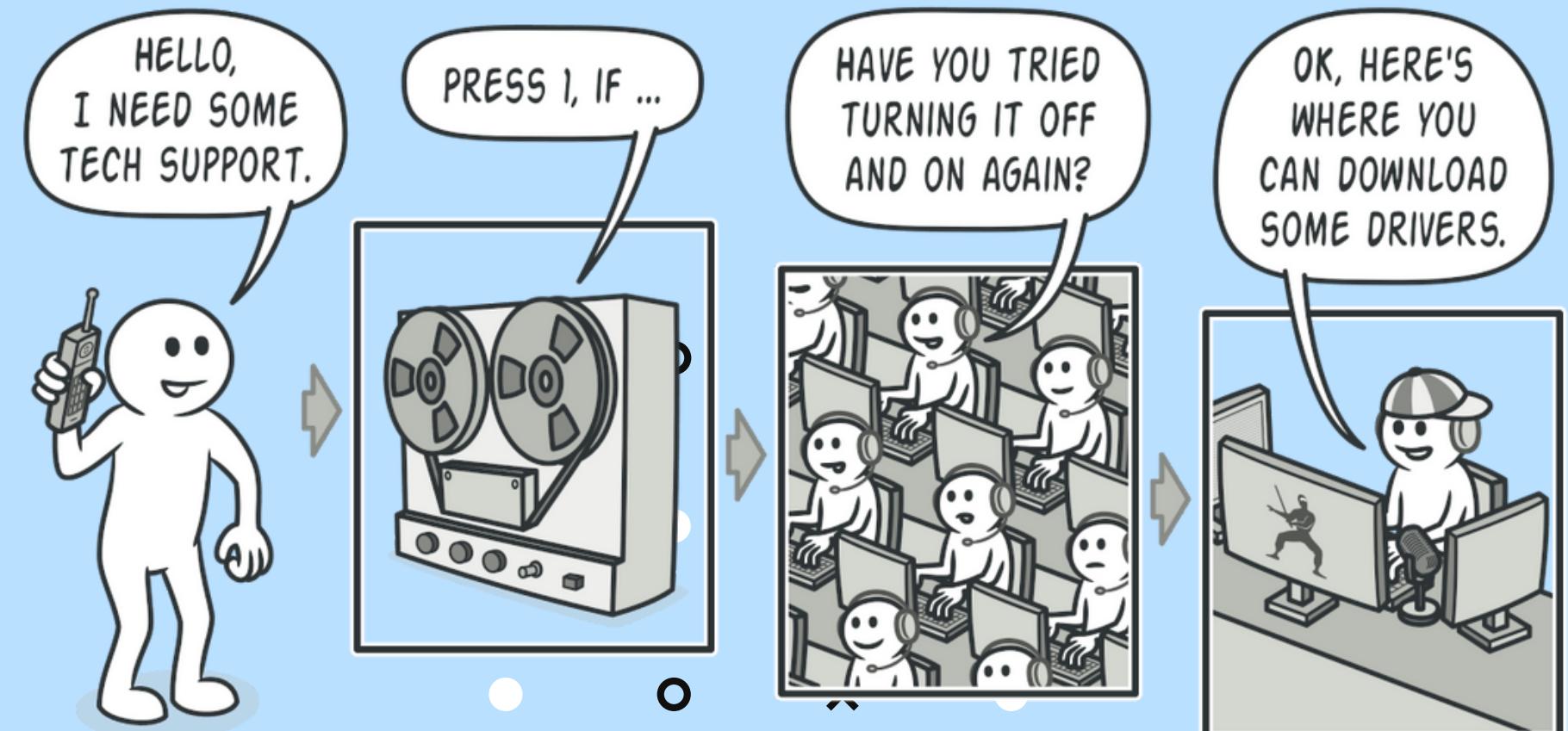


Chain of Responsibility

Lo que es

Permite pasar solicitudes a lo largo de una cadena de controladores. Al recibir una solicitud, cada controlador decide procesar la solicitud o pasarla al siguiente en la cadena.

Se basa en transformar comportamientos particulares en objetos independientes llamados "controladores"



+ ✗ ● ○
○ ● ○ ✗

Es una forma de pasar una solicitud entre un grupo de diferentes objetos para encontrar el que pueda con la solicitud

Ejemplo

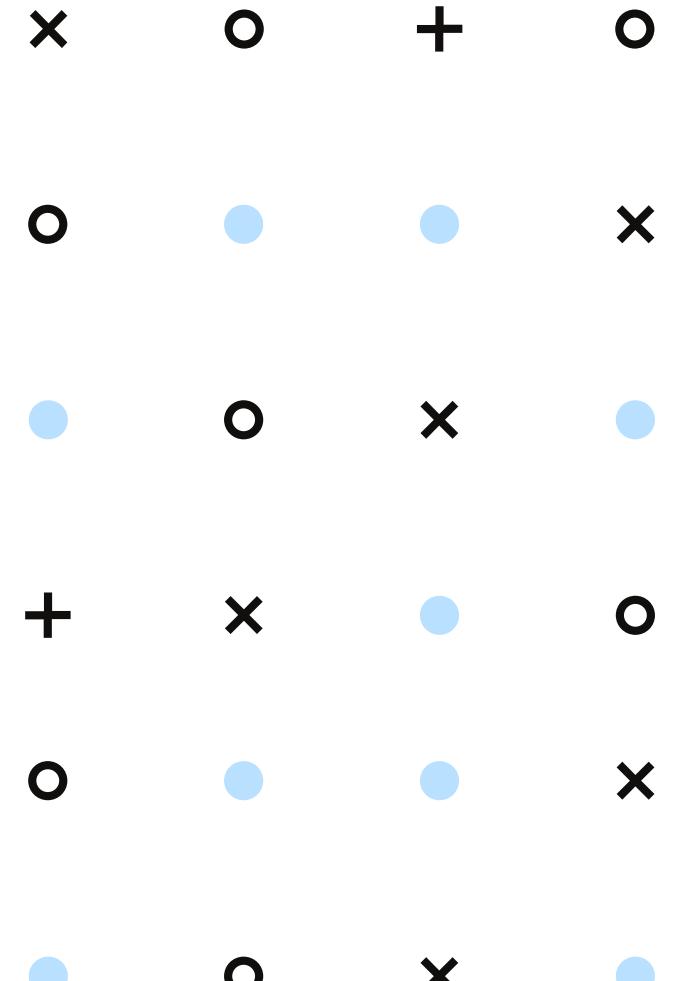
Cadenas de responsabilidad



Problema

El cliente necesita una cierta cantidad de dinero de un ATM
¿Cuál es la combinación de billetes de banco que satisface
esta solicitud?

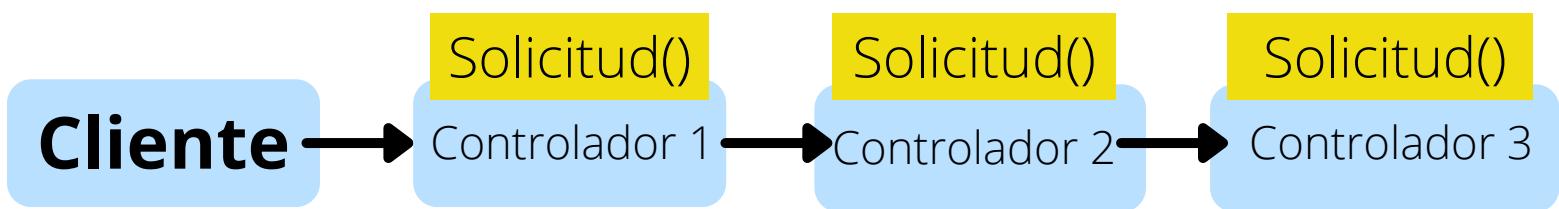
COMBINACIÓN DE BILLETES



Ejemplo

Cadenas de responsabilidad

Diagrama



Objetos participantes:

Cliente: Inicia la solicitud a una cadena de objetos de controlador

Controlador: Define una interfaz para manejar las solicitudes e implementa un enlace sucesor con THIS

```
//CREANDO EL OBJETO CLIENTE -> LA SOLICITUD
let Request = function(amount) {
    this.amount = amount;
    log.add(`Monto Solicitado: $ ${amount}`);
}

Request.prototype = {
    get: function(bill) {
        let count = Math.floor(this.amount / bill);
        this.amount -= count * bill;
        log.add(`Repartir ${count} billetes de ${bill}`);
        return this;
    }
}

//Mostrar el alert
let log = (function() {
    let log = "";
    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }();
})()

function run() {
    let request = new Request(489);
    //Se encadenan las llamadas get, dependiendo la denominación
    request.get(100).get(50).get(20).get(10).get(5).get(1);
    log.show();
}
```

Este código JavaScript ilustra la implementación del patrón de diseño Cadena de responsabilidad. Se define una clase "Request" que gestiona el monto solicitado y logra la impresión de los billetes repartidos. El prototipo incluye un método "get" que reduce el monto total y registra el número de billetes repartidos. Una función anónima crea un objeto "log" para mostrar mensajes de desarrollo. Finalmente, se ejecuta la función "run" que crea un objeto "Request" con un monto de 489 y lo procesa a través de una cadena de llamadas "get" para billetes de 100, 50, 20, 10, 5 y 1.

Monto Solicitado: \$ 489
Repartir 4 billetes de \$100
Repartir 1 billetes de \$50
Repartir 1 billetes de \$20
Repartir 1 billetes de \$10
Repartir 1 billetes de \$5
Repartir 4 billetes de \$1

Aceptar



Command



Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.

Esta transformación le permite pasar solicitudes como argumentos de método, retrasar o poner en cola la ejecución de una solicitud y admitir operaciones que se pueden deshacer.



Encapsula un comando request como un objeto para habilitar, registrar y poner en cola las solicitudes. Proporciona manejo de errores para las tareas no asignadas

Ejemplo

Mando



OPERACIONES BÁSICAS



Situación

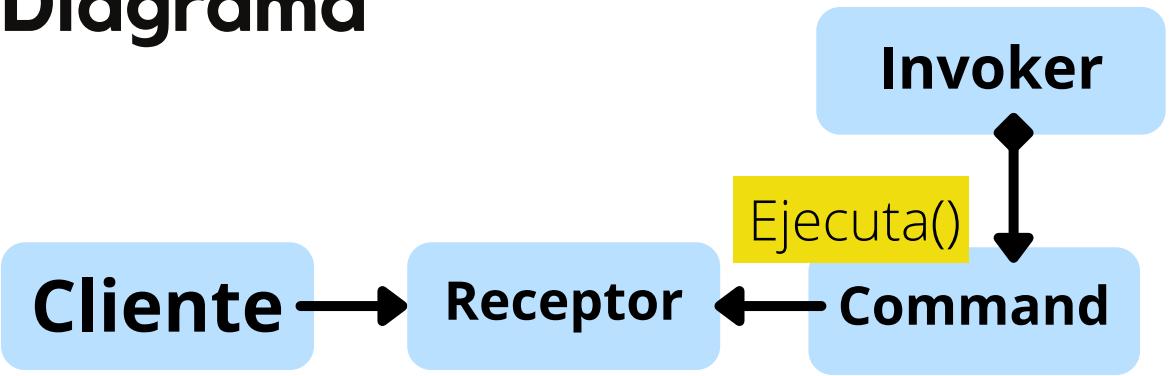
Calculadora básica con sus cuatro operaciones básicas, cada operación se encapsula con un command. Observa como se mantiene la pila de comandos

x	o	+	o
o	●	●	×
●	○	×	●
+	×	●	○
○	●	●	×

Ejemplo

Mando

Diagrama



Objetos participantes:

Cliente: Referencia al objeto Receptor

Receptor: Sabe como realizar la operación asociada al comando

Command: Mantiene información sobre la acción que se tomará

Invoker: Pide realizar la solicitud

```
let Calculator = function () {
  let current = 0;
  let commands = [];
  function action(command) {
    let name = command.execute.toString().substr(9, 3);
    return name.charAt(0).toUpperCase() + name.slice(1);
  }

  return {
    execute: function (command) {
      current = command.execute(current, command.value);
      commands.push(command);
      log.add(`${action(command)}: ${command.value}`);
    },
    undo: function () {
      let command = commands.pop();
      current = command.undo(current, command.value);
      log.add(`Deshacer ${action(command)}: ${command.value}`);
    },
    getCurrentValue: function () {
      return current;
    }
  };
}
```

```
let Command = function (execute, undo, value) {
    this.execute = execute;
    this.undo = undo;
    this.value = value;
}

let AddCommand = function (value) {
    return new Command(add, sub, value);
};

let SubCommand = function (value) {
    return new Command(sub, add, value);
};

let MulCommand = function (value) {
    return new Command(mul, div, value);
};

let DivCommand = function (value) {
    return new Command(div, mul, value);
};
```

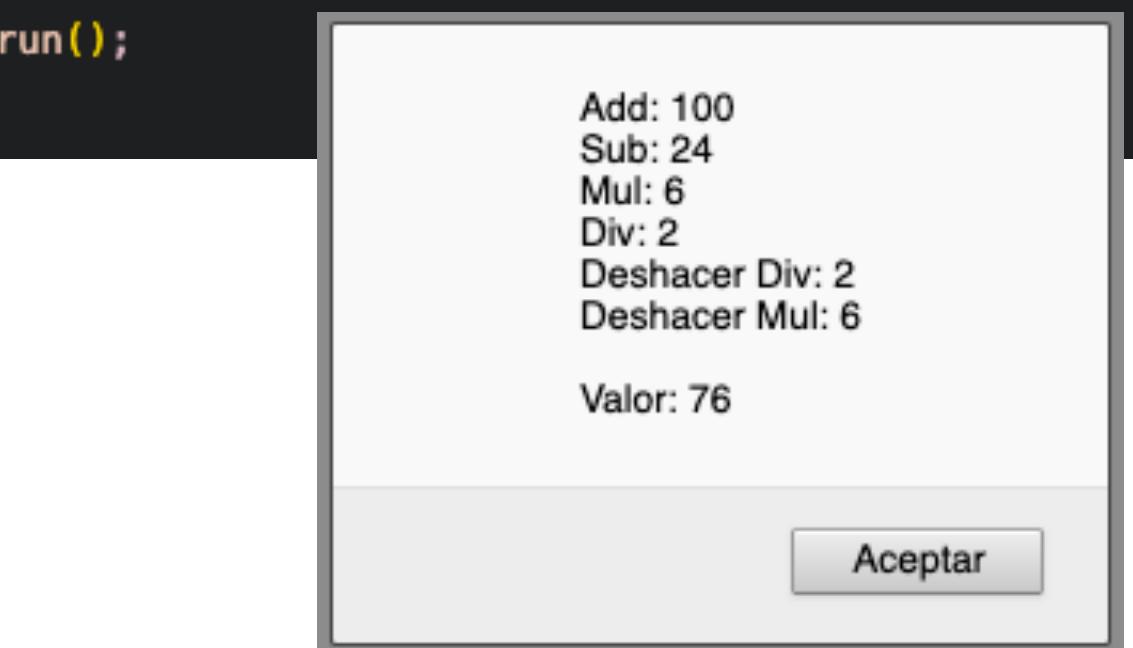
```
function run() {
    let calculator = new Calculator();

    // Solicitudes
    calculator.execute(new AddCommand(100));
    calculator.execute(new SubCommand(24));
    calculator.execute(new MulCommand(6));
    calculator.execute(new DivCommand(2));

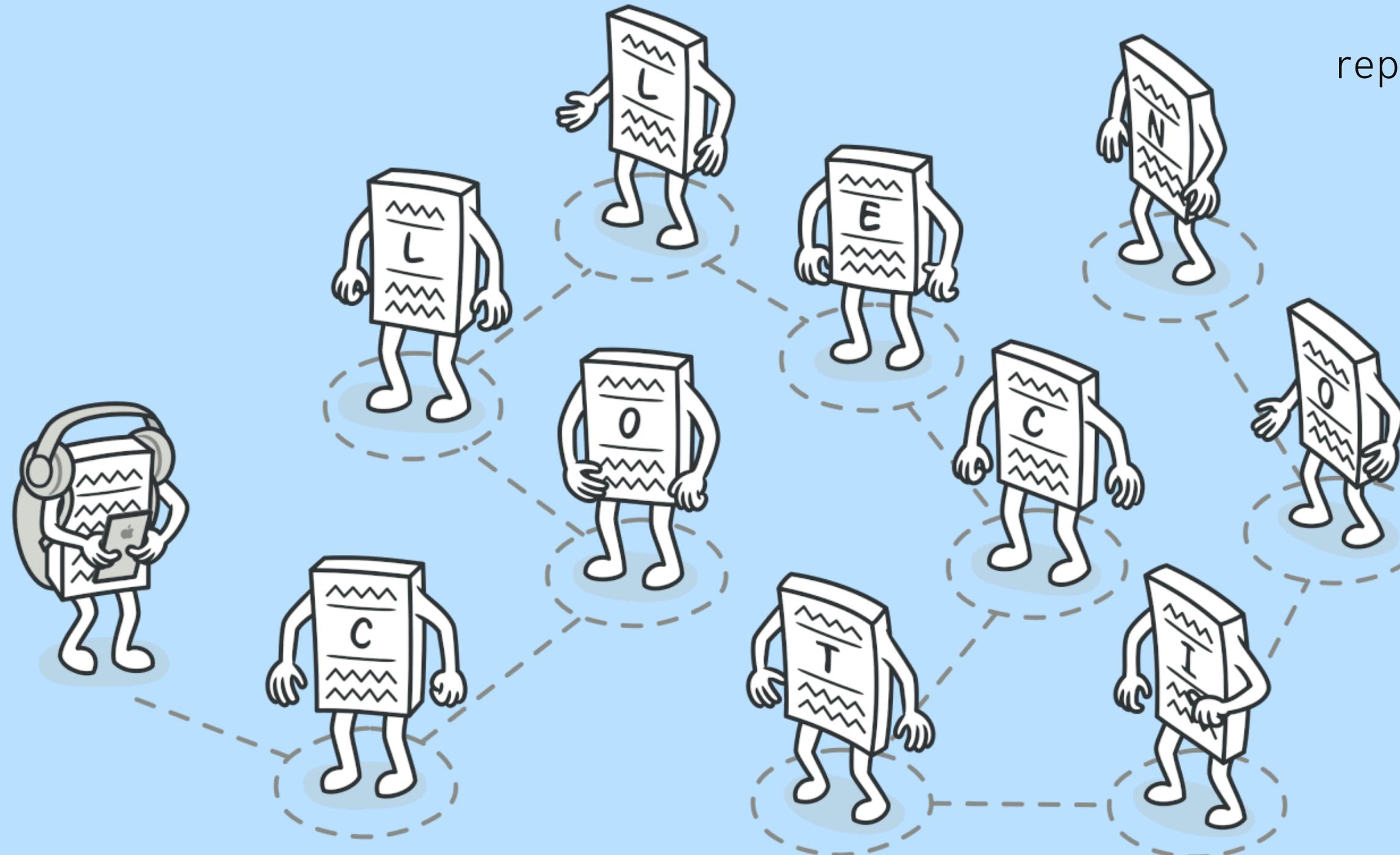
    // invertir los últimos dos comandos
    calculator.undo();
    calculator.undo();

    log.add("\nValor: " + calculator.getCurrentValue());
    log.show();
}

run();
```



Iterator



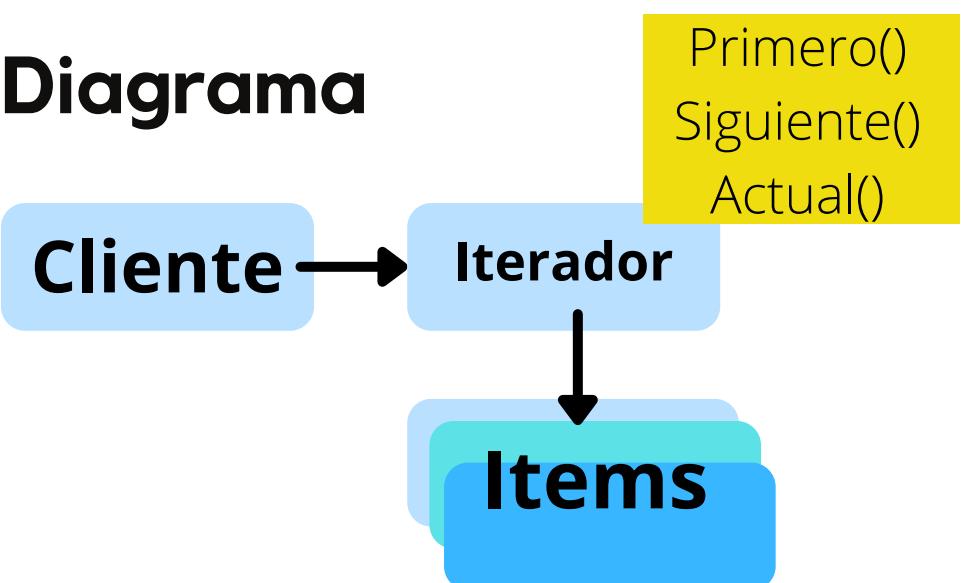
Permite atravesar elementos de una colección sin exponer sus representaciones subyacente

•
Accede secuencialmente a los elementos de una colección sin conocer el funcionamiento interno.

Ejemplo

Iterador

Diagrama



Objetos participantes:

Cliente: Invoca Iterador con colección de objetos

Iterador: Realiza un seguimiento de la posición actual al atravesar la colección

Items: Objetos de la colección

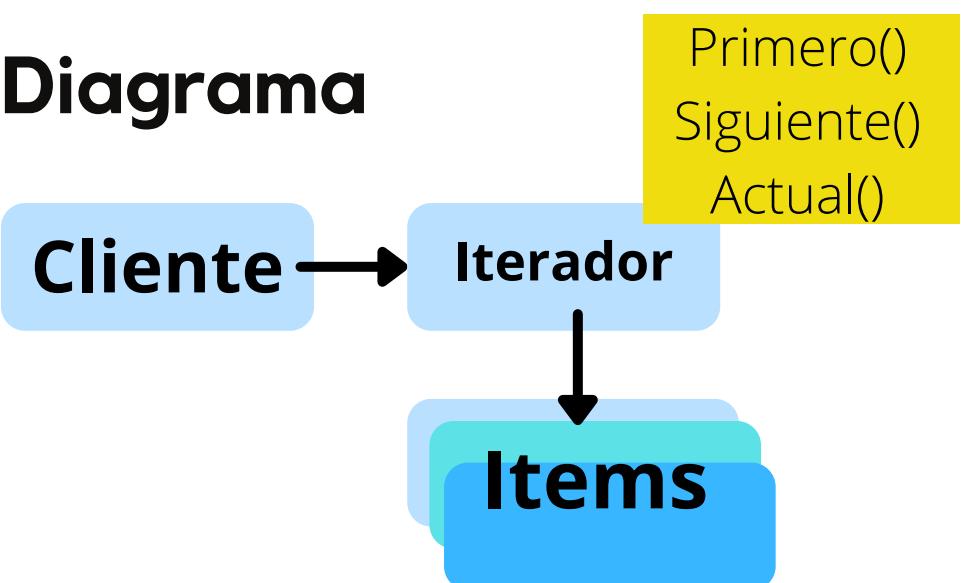
```
let Iterator = function(items) {
  this.index = 0;
  this.items = items;
}

Iterator.prototype = {
  first: function() {
    this.reset();
    return this.next();
  },
  next: function() {
    return this.items[this.index++];
  },
  hasNext: function() {
    return this.index <= this.items.length;
  },
  reset: function() {
    this.index = 0;
  },
  each: function(callback) {
    for (let item = this.first(); this.hasNext(); item = this.next()) {
      callback(item);
    }
  }
}
```

Ejemplo

Iterador

Diagrama



Objetos participantes:

Cliente: Invoca Iterador con colección de objetos

Iterador: Realiza un seguimiento de la posición actual al atravesar la colección

Items: Objetos de la colección

```
function run() {  
    let items = ["Justin", 1, "BTS", false, "IPhone"];  
    let iter = new Iterator(items);  
  
    // Usando for  
  
    for (let item = iter.first(); iter.hasNext(); item = iter.next()) {  
        log.add(item);  
    }  
    log.add("");  
  
    // Usando Each  
  
    iter.each(function(item) {  
        log.add(item);  
    });  
  
    log.show();  
}  
  
run();
```

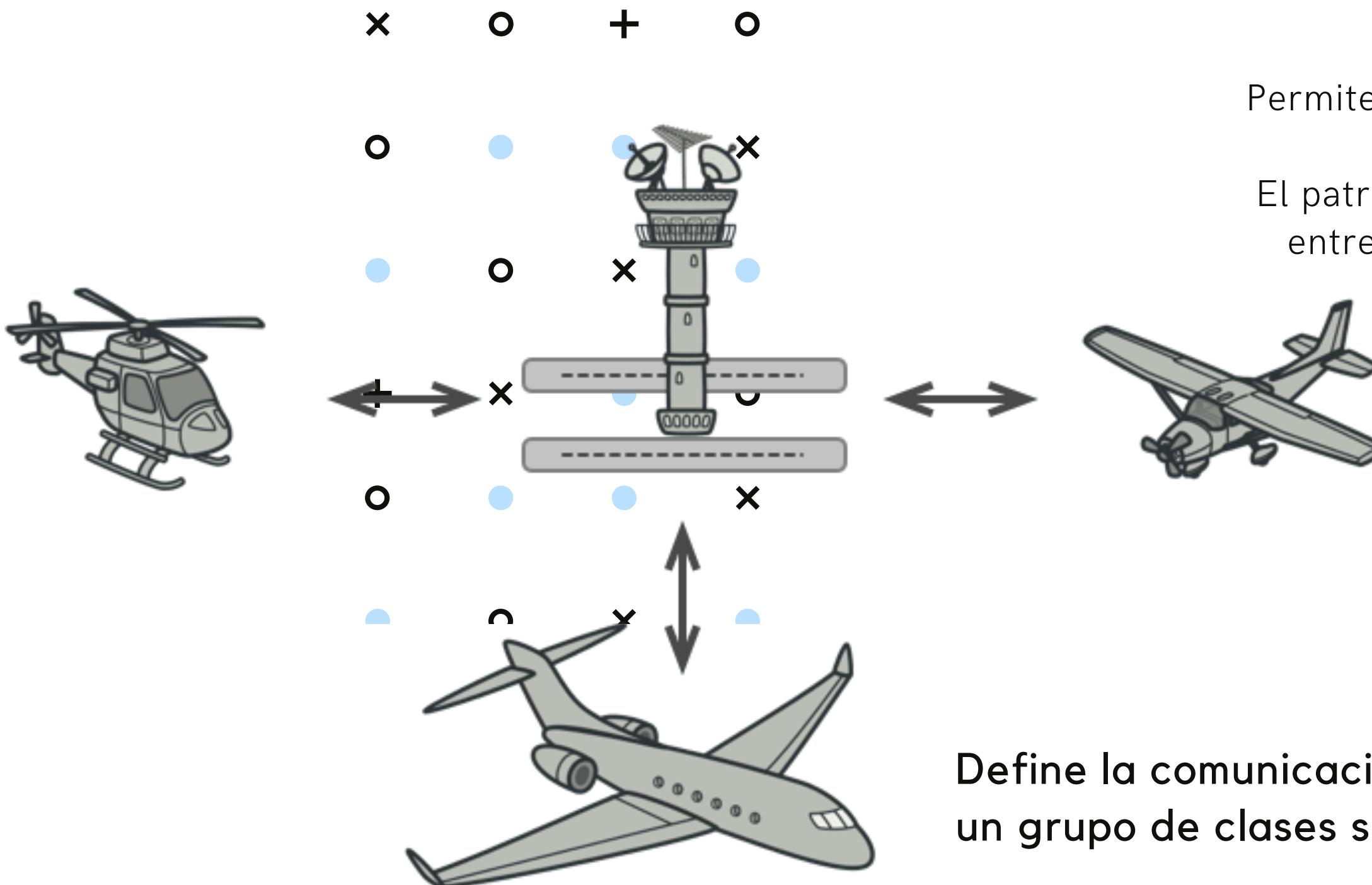
Justin
1
BTS
false
IPhone

Justin
1
BTS
false
IPhone

Aceptar



Mediator



Permite reducir las dependencias caóticas entre objetos.

El patrón restringe las comunicaciones directas entre los objetos y los obliga colaborar solo a través de un objeto mediator.

Define la comunicación simplificada entre clases para evitar que un grupo de clases se comuniquen entre si.

Ejemplo

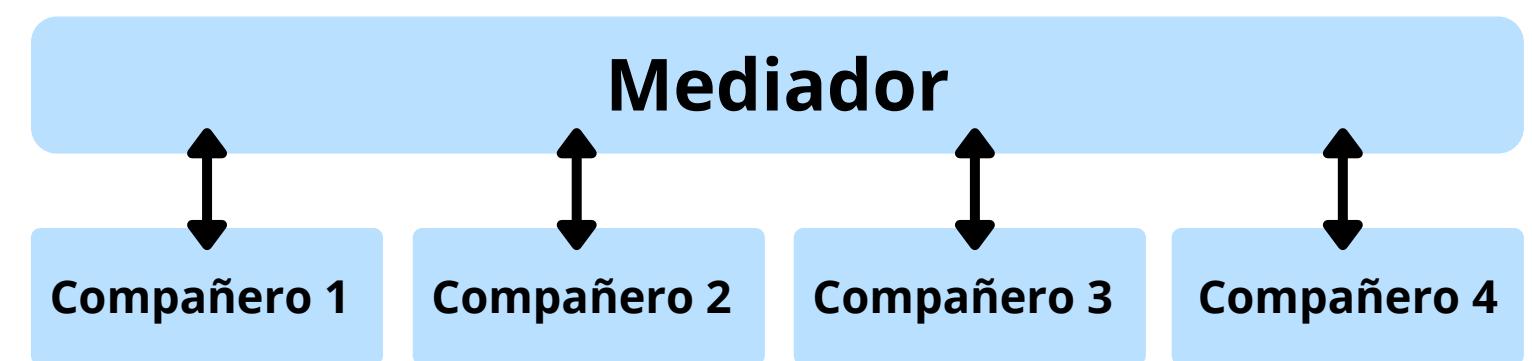
Mediador



Situación

Tenemos a cuatro personas hablando en el chat, cada uno se envía mensaje y la sala del chat se encarga de gestionar.

Diagrama



Objetos participantes:

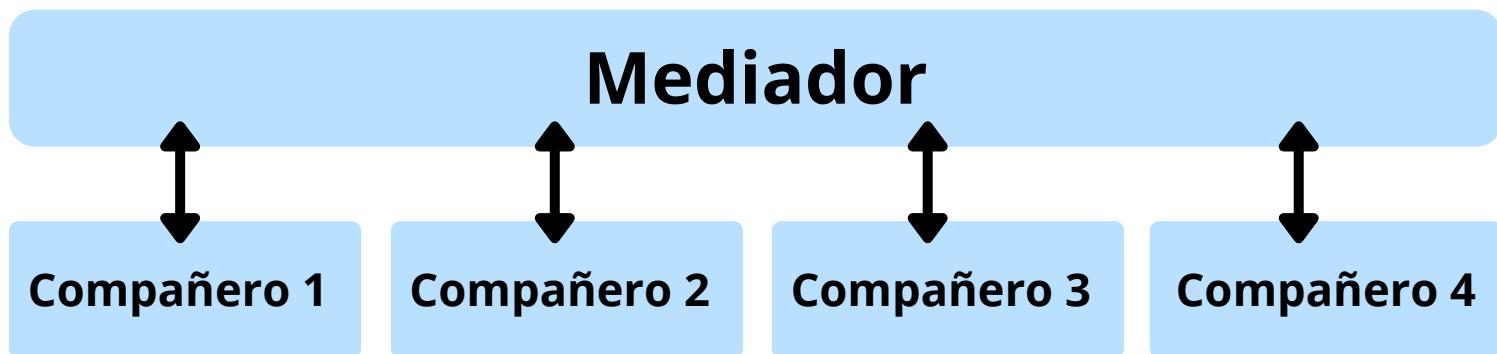
Mediador: Gestiona el control de las operaciones

Compañero: Objetos que están siendo mediados

Ejemplo

Mediador

Diagrama



Objetos participantes:

Mediador: Gestiona el control de las operaciones

Compañero: Objetos que estan siendo mediados

```
let Chatroom = function() {
  let participants = {};

  return {

    register: function(participant) {
      participants[participant.name] = participant;
      participant.chatroom = this;
    },

    send: function(message, from, to) {
      if (to) { // Mensaje en específico
        to.receive(message, from);
      } else { // Mensaje para todos
        for (key in participants) {
          if (participants[key] !== from) {
            participants[key].receive(message, from);
          }
        }
      }
    };
  };
}
```

Ejemplo

Mediador

```
let Participant = function(name) {
  ...
  this.name = name;
  this.chatroom = null;
};

Participant.prototype = {
  send: function(message, to) {
    this.chatroom.send(message, this, to);
  },
  receive: function(message, from) {
    log.add(` ${from.name} a ${this.name}: ${message}`);
  }
};
```

```
function run() {
  var Vane = new Participant("Vane");
  var Diana = new Participant("Diana");
  var Bernardo = new Participant("Bernardo");
  var Mauricio = new Participant("Mauricio");

  var chatroom = new Chatroom();
  chatroom.register(Vane);
  chatroom.register(Diana);
  chatroom.register(Bernardo);
  chatroom.register(Mauricio);

  Mauricio.send("¿Bernardo qué resultado te salio en la macro?");
  Mauricio.send("Bernardo Contesta");
  Vane.send("¡Ya contestaaa!", Bernardo);
  Diana.send("¿En que tema van?");
  Bernardo.send("Creo que esta maaal", Mauricio);

  log.show();
}

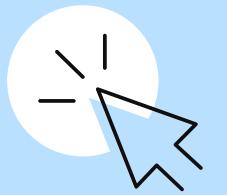
run();
```

Mauricio a Vane: ¿Bernardo qué resultado te salio en la macro?
Mauricio a Diana: ¿Bernardo qué resultado te salio en la macro?
Mauricio a Bernardo: ¿Bernardo qué resultado te salio en la macro?
Mauricio a Vane: Bernardo Contesta
Mauricio a Diana: Bernardo Contesta
Mauricio a Bernardo: Bernardo Contesta
Vane a Bernardo: ¡Ya contestaaa!
Diana a Vane: ¿En que tema van?
Diana a Bernardo: ¿En que tema van?
Diana a Mauricio: ¿En que tema van?
Bernardo a Mauricio: Creo que esta maaal

Aceptar



Memento



Permite guardar y restaurar el estado anterior de un objeto sin revelar los detalles de su implementación.

Captura el estado interno de un objeto cuando cambia su estado.



Ejemplo

Recuerdo

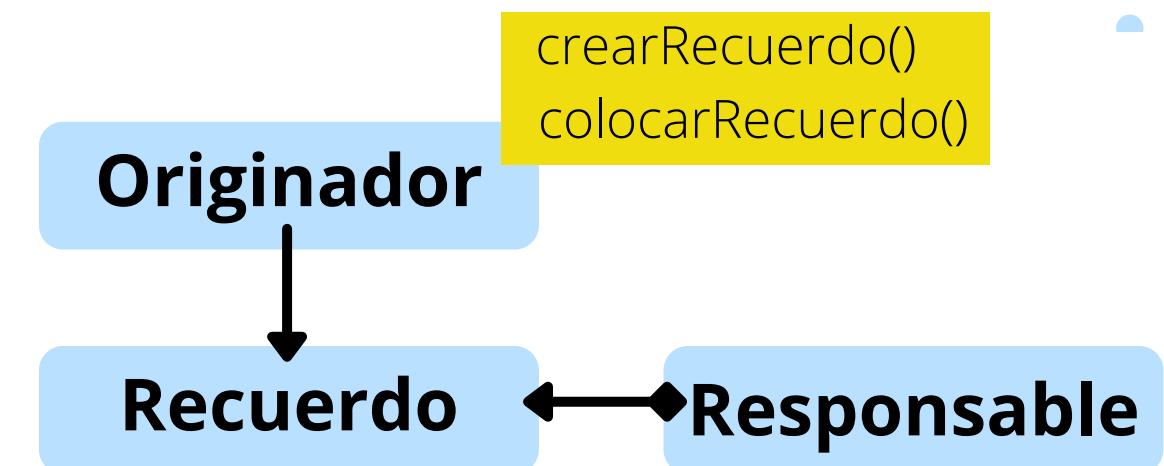


Situación

Crearemos a dos personas con sus recuerdos y les asignaremos nombres falsos antes de restaurar sus recuerdos.

¿Mantendrán sus recuerdos?

Diagrama

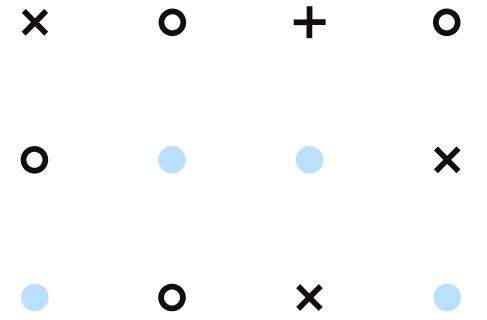


Objetos participantes:

Originador: Objeto en estado temporal, guarda y se restaura

Recuerdo: Estado interno del objeto de Originador

Responsable: Solo es un repositorio, no cambia los recuerdos



Ejemplo

Recuerdo

```
let Person = function(name, street, city, state) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
}

Person.prototype = {

    hydrate: function() {
        let memento = JSON.stringify(this);
        return memento;
    },

    dehydrate: function(memento) {
        let m = JSON.parse(memento);
        this.name = m.name;
        this.street = m.street;
        this.city = m.city;
        this.state = m.state;
    }
}
```



```
let CareTaker = function() {
    this.mementos = {};

    this.add = function(key, memento) {
        this.mementos[key] = memento;
    };

    this.get = function(key) {
        return this.mementos[key];
    }
}
```

```
function run() {
    let Superman = new Person("Superman", "1112 Main", "Dallas", "TX");
    let Batman = new Person("Batman", "48th Street", "San Jose", "CA");
    let caretaker = new CareTaker();

    // save state
    caretaker.add(1, Superman.hydrate());
    caretaker.add(2, Batman.hydrate());

    // Cambiando identidad
    Superman.name = "Clark Kent";
    Batman.name = "Bruce Wayne";

    // Restaurando
    Superman.dehydrate(caretaker.get(1));
    Batman.dehydrate(caretaker.get(2));

    log.add(Superman.name);
    log.add(Superman.street);
    log.add(Batman.name);
    log.add(Batman.street);

    log.show();
}

run();
```

Superman
1112 Main
Batman
48th Street

Aceptar

x o + o :
o ● ● x :
● o x ● :
+ x ● o :
o :
●



Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que suceda al objeto observado.

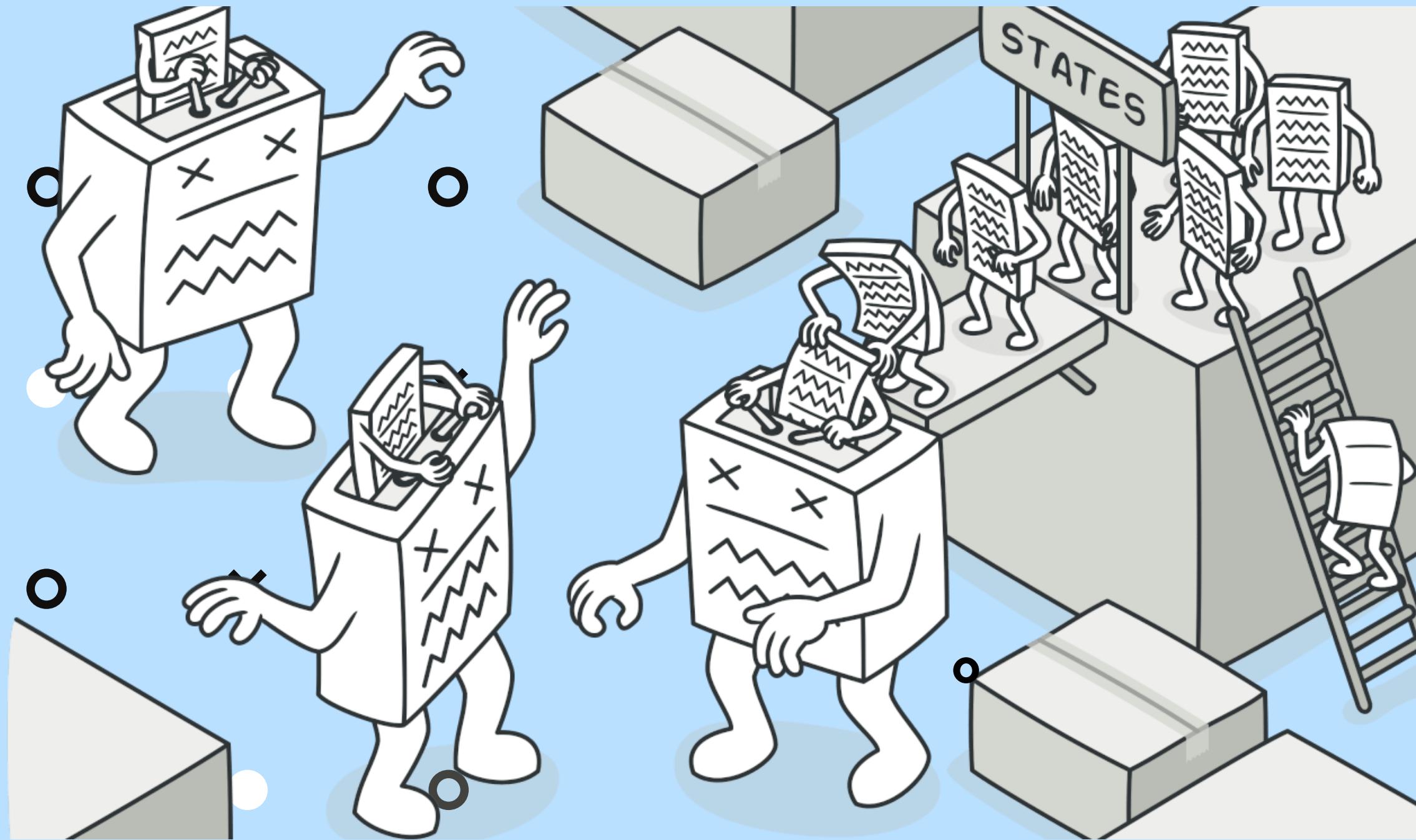
Forma de notificar cambios en una serie de clases para garantizar la coherencia entre clases.



State

Permite que un objeto a alterar cambie como de clase.

x
o
o
+
o



Altera el comportamiento de un objeto cuando cambia de estado.

Ejemplo

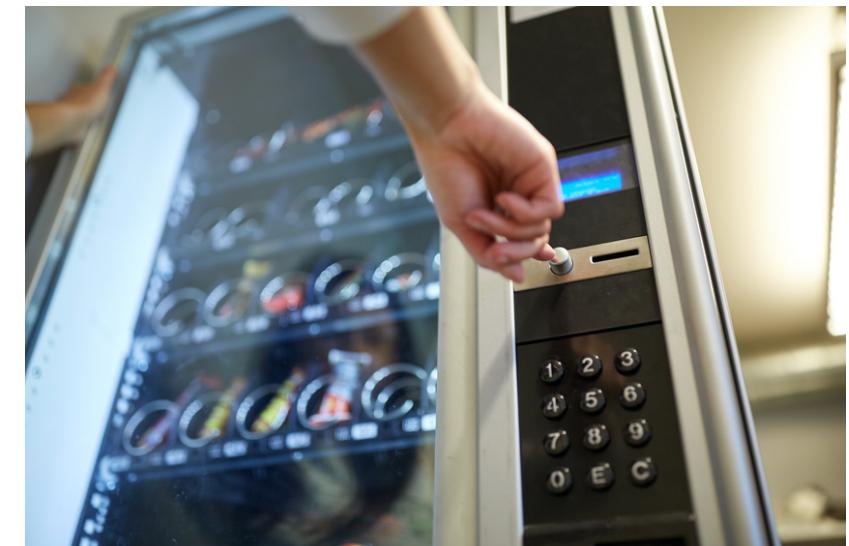
Estado



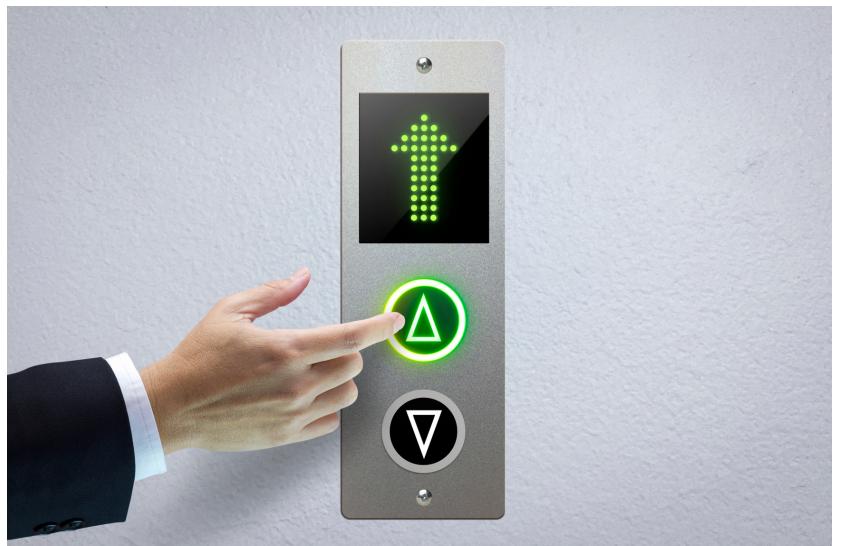
Situación

Tenemos 3 estados en un semáforo y cada uno tiene sus respectivas reglas.

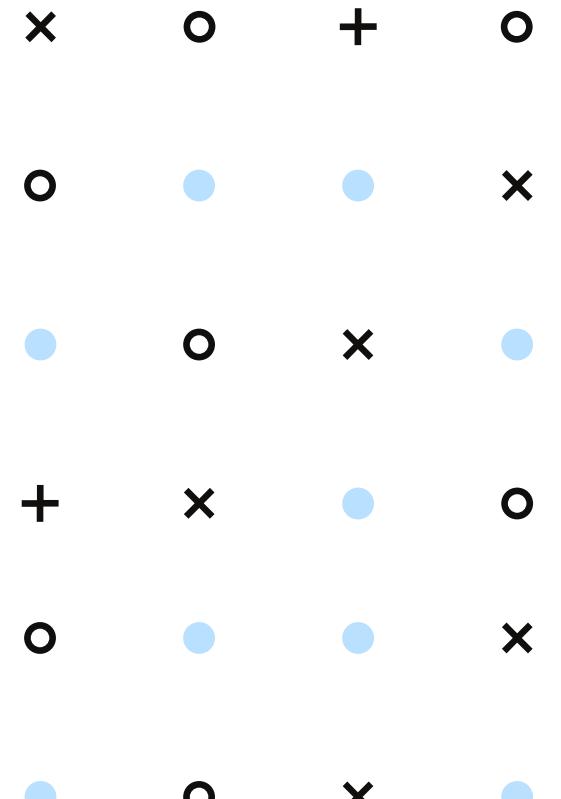
Otros ejemplos



Máquina expendedora



Elevador



Ejemplo

Estado

Diagrama

SOLICITUD ()

Contexto

Estado 1

Resolver ()

Estado 2

Resolver ()

Estado 3

Resolver ()

Objetos participantes:

Contexto: Permite que cambien su estado a uno diferente

Estado: Encapsula el comportamiento asociado al estado

```
let TrafficLight = function () {
    let count = 0;
    let currentState = new Red(this);

    this.change = function (state) {
        // Cambios (limite)
        if (count++ >= 10) return;
        currentState = state;
        currentState.go();
    };

    let Red = function (light) {
        this.light = light;

        this.go = function () {
            log.add("ROJO      -> UN MINUTO");
            light.change(new Green(light));
        };
    };

    let Yellow = function (light) {
        this.light = light;

        this.go = function () {
            log.add("AMARILLO -> 10 Segundos");
            light.change(new Red(light));
        };
    };

    let Green = function (light) {
        this.light = light;

        this.go = function () {
            log.add("VERDE     -> Un minuto");
            light.change(new Yellow(light));
        };
    };
}
```

```
};

this.start = function () {
    currentState.go();
};

function run() {
    let light = new TrafficLight();
    light.start();

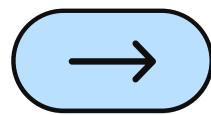
    log.show();
}

run();
```

ROJO -> UN MINUTO
VERDE -> Un minuto
AMARILLO -> 10 Segundos
ROJO -> UN MINUTO
VERDE -> Un minuto
AMARILLO -> 10 Segundos
ROJO -> UN MINUTO
VERDE -> Un minuto
AMARILLO -> 10 Segundos
ROJO -> UN MINUTO
VERDE -> Un minuto



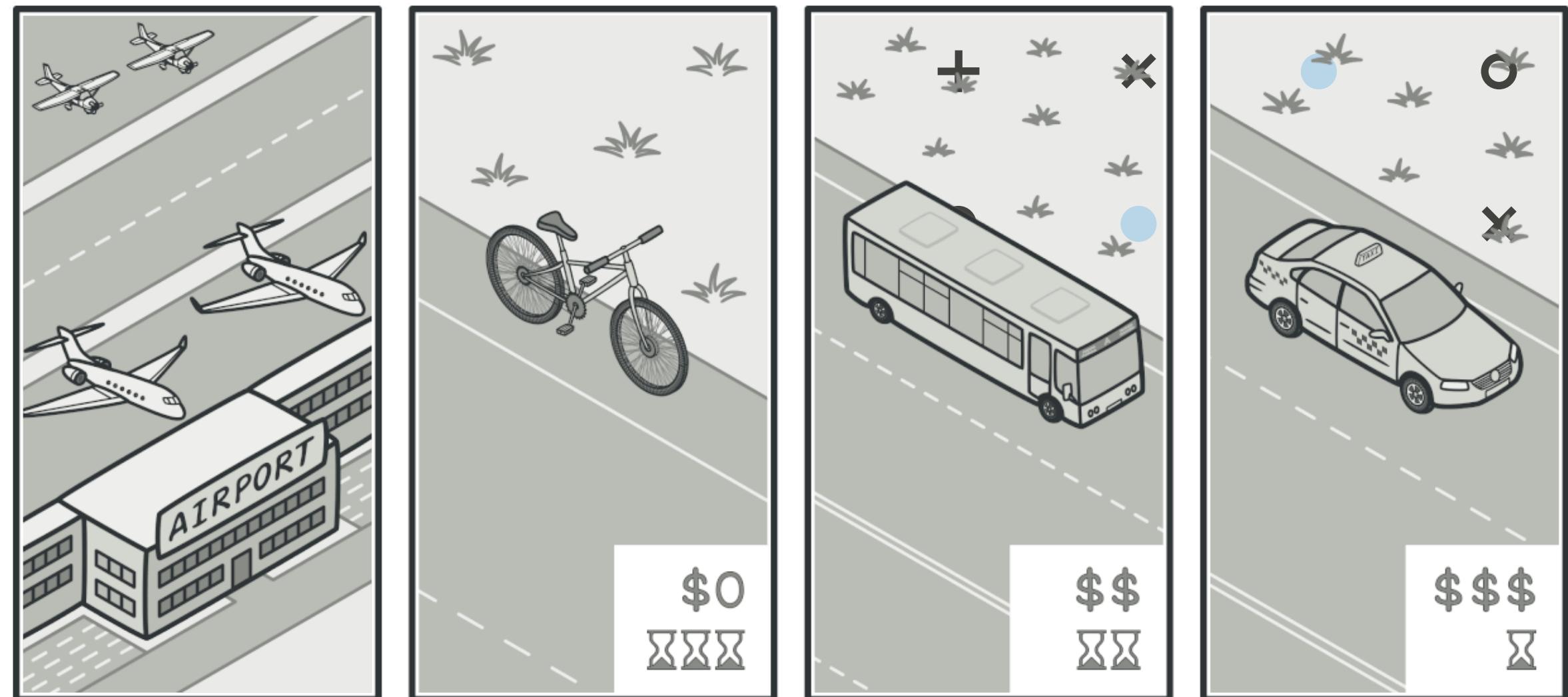
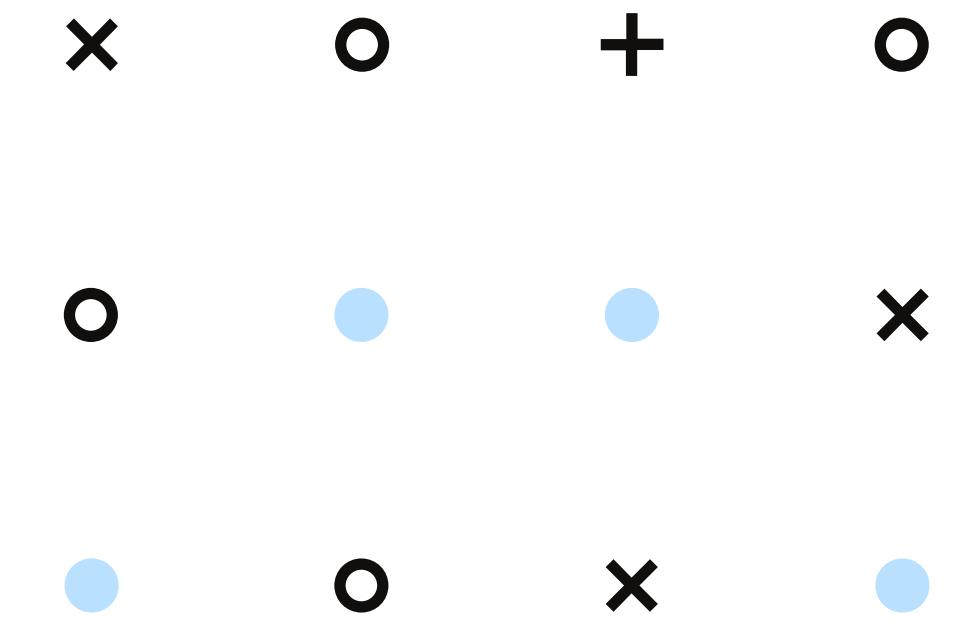
Aceptar



Strategy

Permite definir una familia de algoritmos, poner cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.

Encapsula un algoritmo dentro de una clase que separa la selección de la forma de implementar.



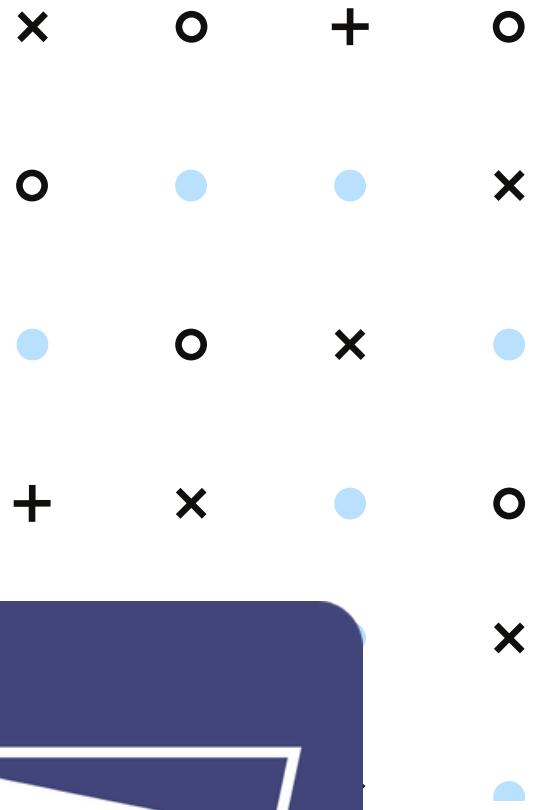
Ejemplo

Estrategia



Situación

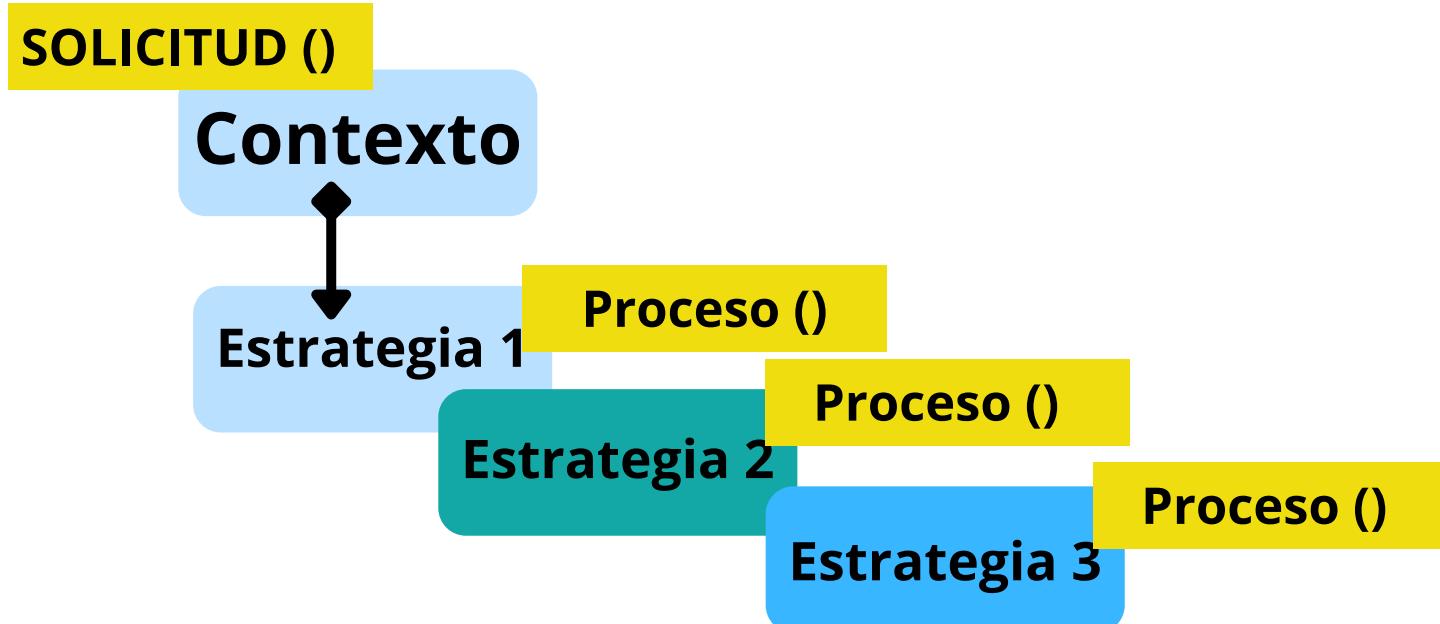
Tenemos 3 estrategias (UPS, USPS Y FEDEX).
¿Cuál sera la mejor alternativa para enviar mi paquete?



Ejemplo

Estrategia

Diagrama



Participantes

Contexto: Permite a los clientes cambiar de estrategia

Estrategia: Implementa el algoritmo dependiendo la estrategia

```
let Shipping = function() {
    this.company = "";
};

Shipping.prototype = {
    setStrategy: function(company) {
        this.company = company;
    },

    calculate: function(package) {
        return this.company.calculate(package);
    }
};
```

```
let UPS = function() {
    this.calculate = function(package) {
        // calculando
        return "$45.95";
    }
};

let USPS = function() {
    this.calculate = function(package) {
        // calculando
        return "$39.40";
    }
};

let Fedex = function() {
    this.calculate = function(package) {
        // calculando
        return "$43.20";
    }
};
```

```
function run() {
    let package = { from: "76712", to: "10012", weight: "lkg" };

    // 3 disponibles
    let ups = new UPS();
    let usps = new USPS();
    let fedex = new Fedex();

    let shipping = new Shipping();

    shipping.setStrategy(ups);
    log.add(`Estrega UPS: ${shipping.calculate(package)}`);
    shipping.setStrategy(usps);
    log.add(`Estrega USPS: ${shipping.calculate(package)}`);
    shipping.setStrategy(fedex);
    log.add(`Estrega FEDEX: ${shipping.calculate(package)}`);

    log.show();
}

run();
```

Estrega UPS: \$45.95
Estrega USPS: \$39.40
Estrega FEDEX: \$43.20

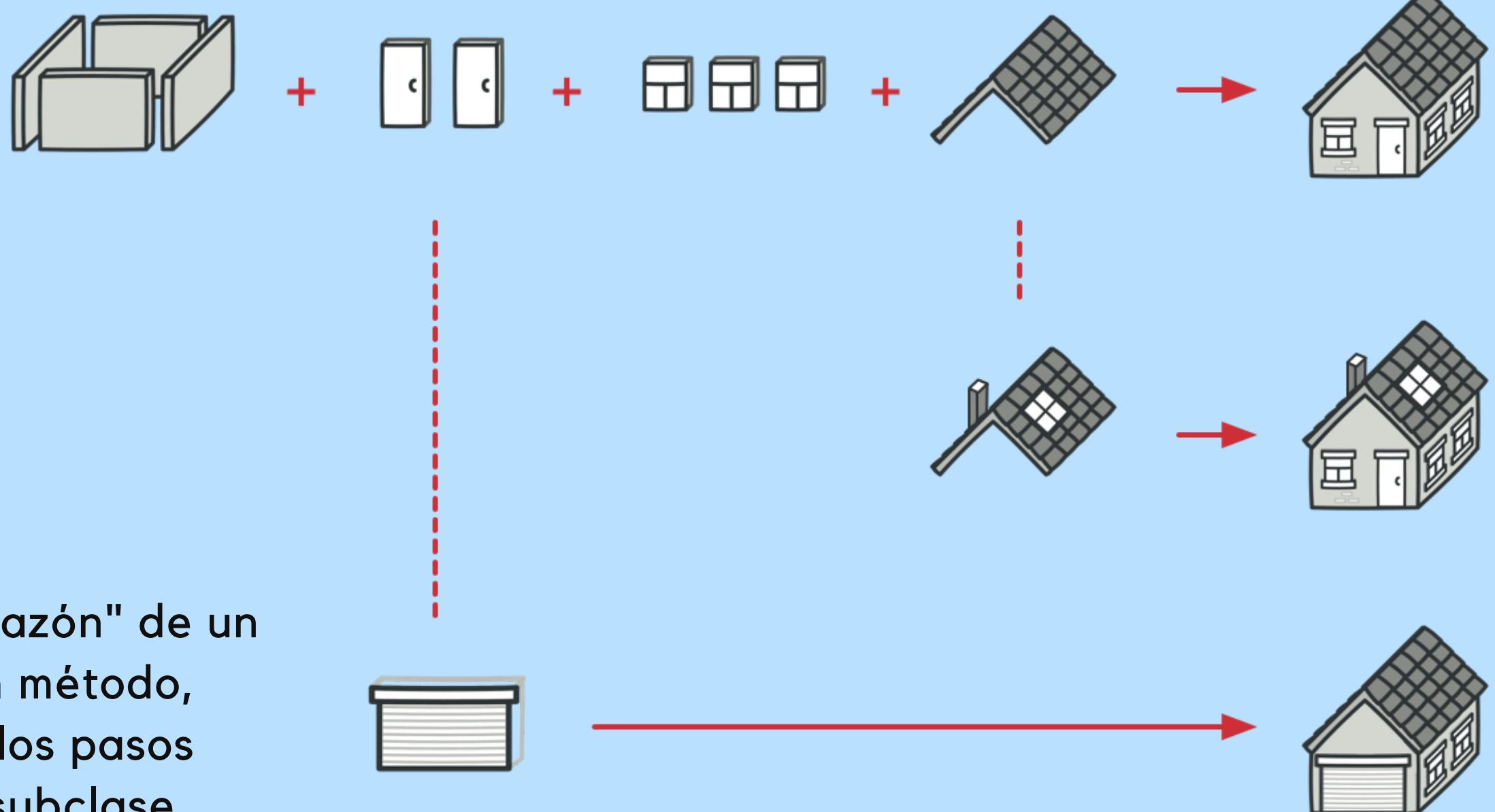
Aceptar



Template Method

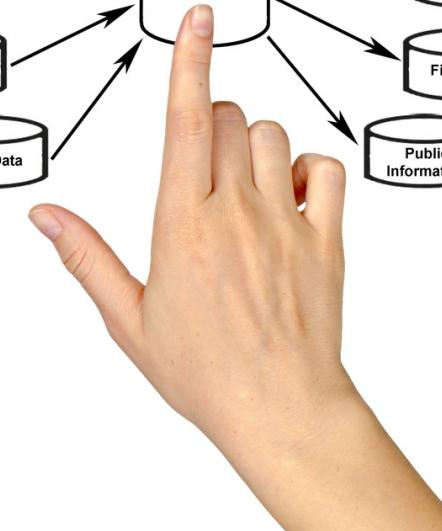
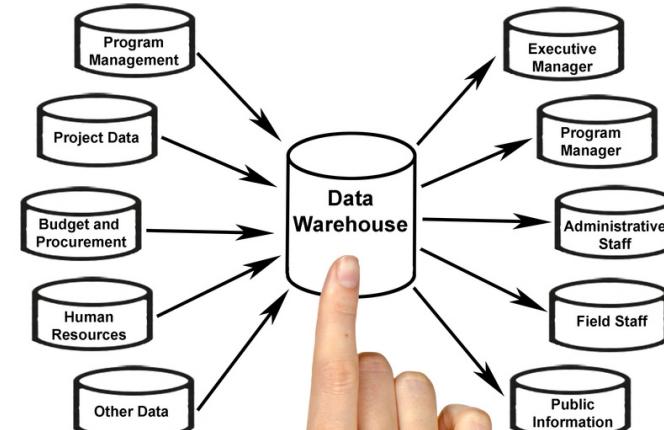


Define el "esqueleto" de un algoritmo en una superclase, pero permite que las subclases anulen pasos específicos del algoritmo sin cambiar su estructura.



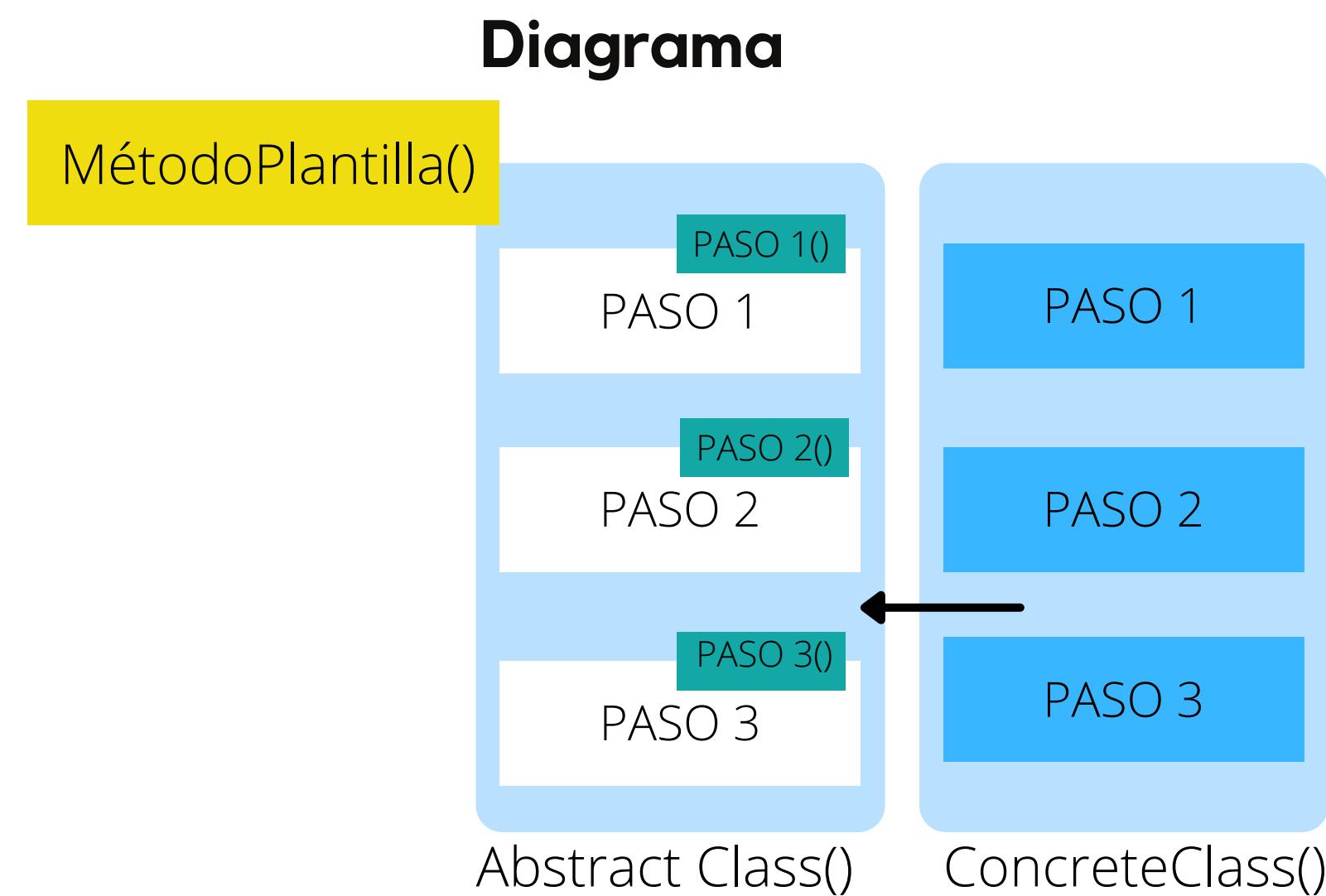
Ejemplo

Método de plantilla



Situación

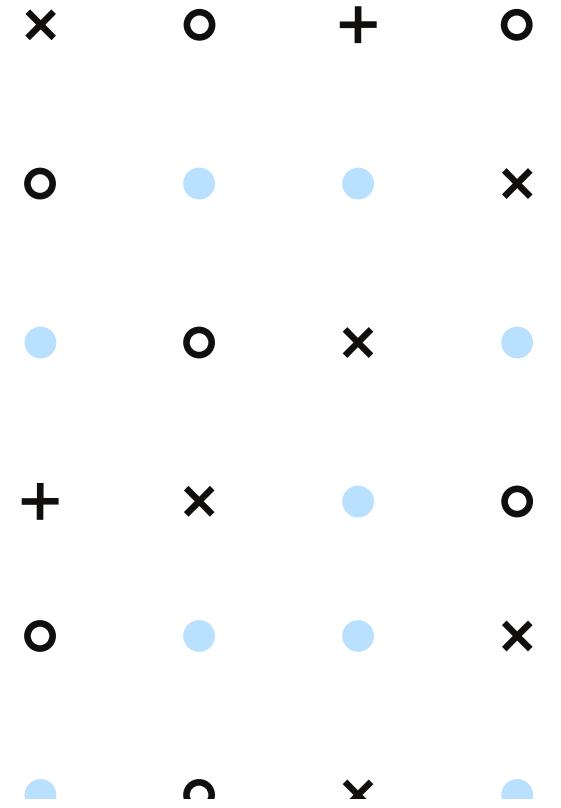
Haremos la vinculación de métodos a un almacén de datos

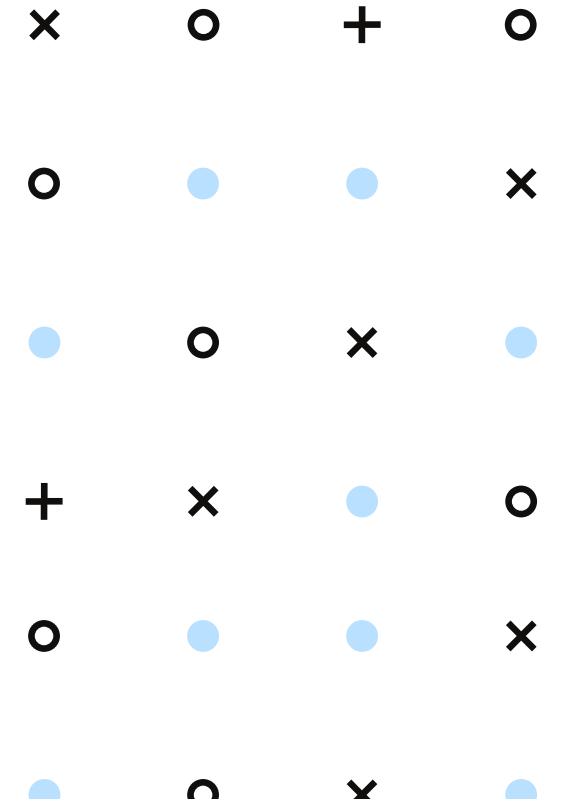


Participantes

AbstractClass: Clientes invocan el método templateMethod

ConcretClass: Implementa los pasos conforme a los definidos en AbstracClass





Ejemplo

Método de plantilla

```
let datastore = {
  process: function() {
    this.connect();
    this.select();
    this.disconnect();
    return true;
  }
};
```

```
function inherit(proto) {
  var F = function() { };
  F.prototype = proto;
  return new F();
}
```

```
function run() {
  let mySql = inherit(datastore);

  // implement template steps

  mySql.connect = function() {
    log.add("MySQL: connect step");
  };

  mySql.select = function() {
    log.add("MySQL: select step");
  };

  mySql.disconnect = function() {
    log.add("MySQL: disconnect step");
  };

  mySql.process();
  log.show();
}

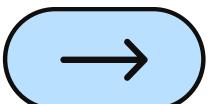
run();
```

MySQL: connect step
MySQL: select step
MySQL: disconnect step

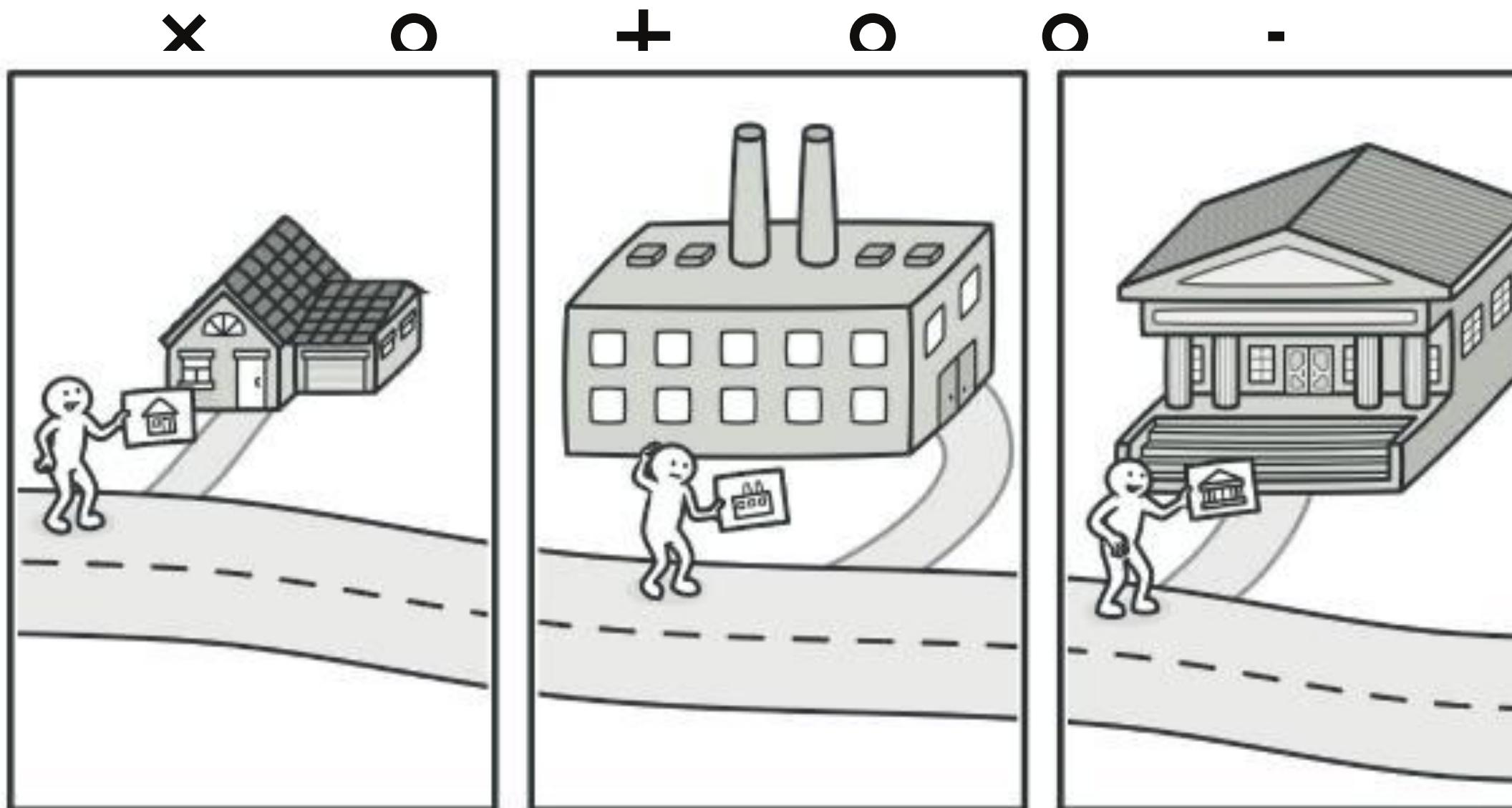
Aceptar



Visitor



Permite separar algoritmos de los objetos en los que operan.



Agrega una nueva operación a una clase sin cambiar la clase.

$+$ \times \bullet \circ
 \circ \bullet \bullet \times

Ejemplo

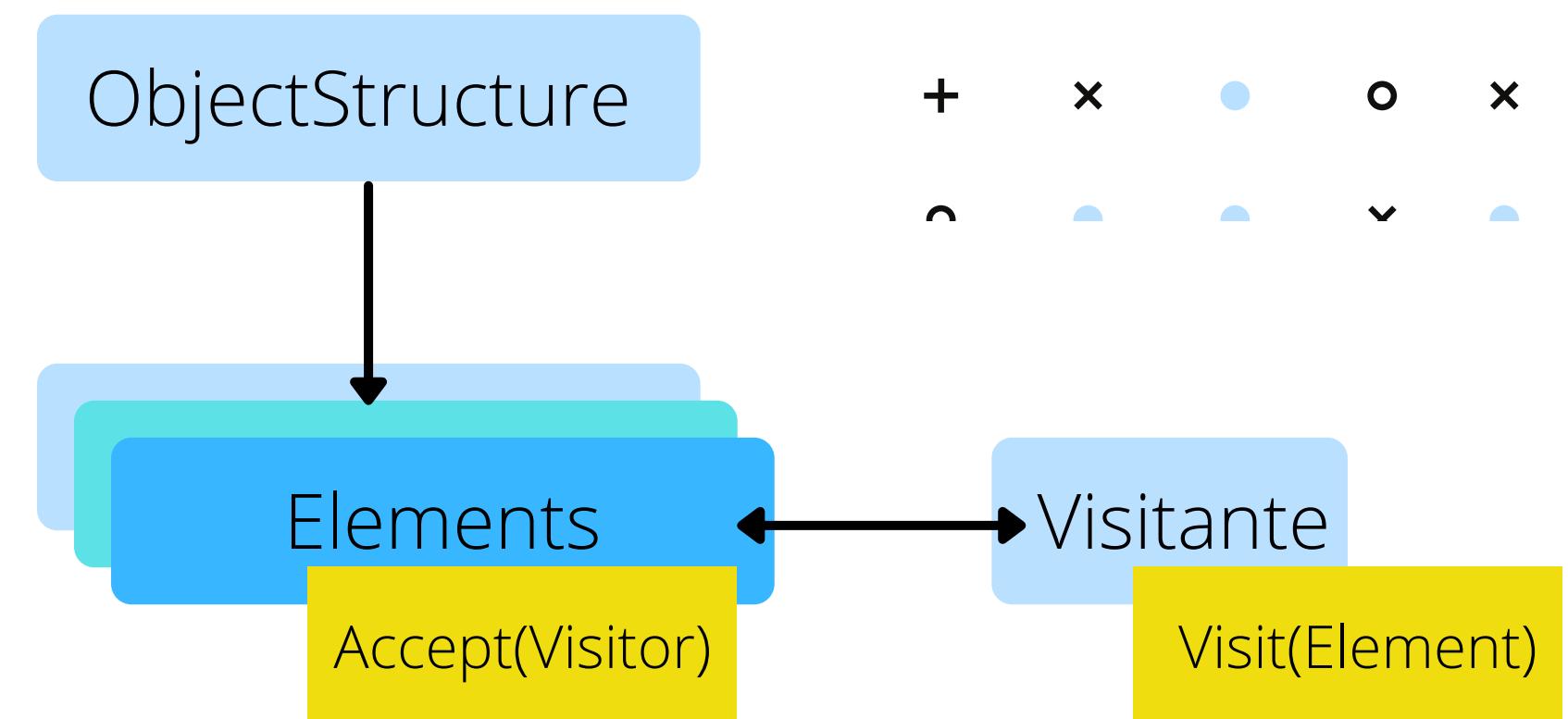
Visitante



Situación

Tres empleados donde cada uno recibirá su aumento salarial del 10% Y 2 Días más de vacaciones, veamos como realizar cambios necesarios en estos empleados

Diagrama



Participantes

ObjectStructure: Mantiene la colección para iterar

Elements: Define un método de aceptación

Visitante: Implementa un método de visita

Ejemplo

Visitante

```
function run() {
    //ObjectStructure
    let employees = [
        new Employee("Justin", 10000, 10),
        new Employee("Harry", 20000, 21),
        new Employee("tanner", 250000, 51)
    ];
    let visitorSalary = new ExtraSalary();
    let visitorVacation = new ExtraVacation();

    for (let i = 0, len = employees.length; i < len; i++) {
        let emp = employees[i];

        emp.accept(visitorSalary);
        emp.accept(visitorVacation);
        log.add(`${emp.getName()}: ${emp.getSalary()} y ${emp.getVacation()} días de vacaciones`);

    }
    log.show();
}
run();
```

```
let Employee = function (name, salary, vacation) {
    let self = this;

    this.accept = function (visitor) {
        visitor.visit(self);
    };

    this.getName = function () {
        return name;
    };

    this.getSalary = function () {
        return salary;
    };

    this.setSalary = function (sal) {
        salary = sal;
    };

    this.getVacation = function () {
        return vacation;
    };

    this.setVacation = function (vac) {
        vacation = vac;
    };
};
```

Justin: \$11000 y 12 días de vacaciones
Harry: \$22000 y 23 días de vacaciones
tanner: \$275000 y 53 días de vacaciones

```
//Visitante
let ExtraSalary = function () {
    this.visit = function (emp) {
        emp.setSalary(emp.getSalary() * 1.1);
    };
};

let ExtraVacation = function () {
    this.visit = function (emp) {
        emp.setVacation(emp.getVacation() + 2);
    };
};
```

Aceptar



Resumen

4

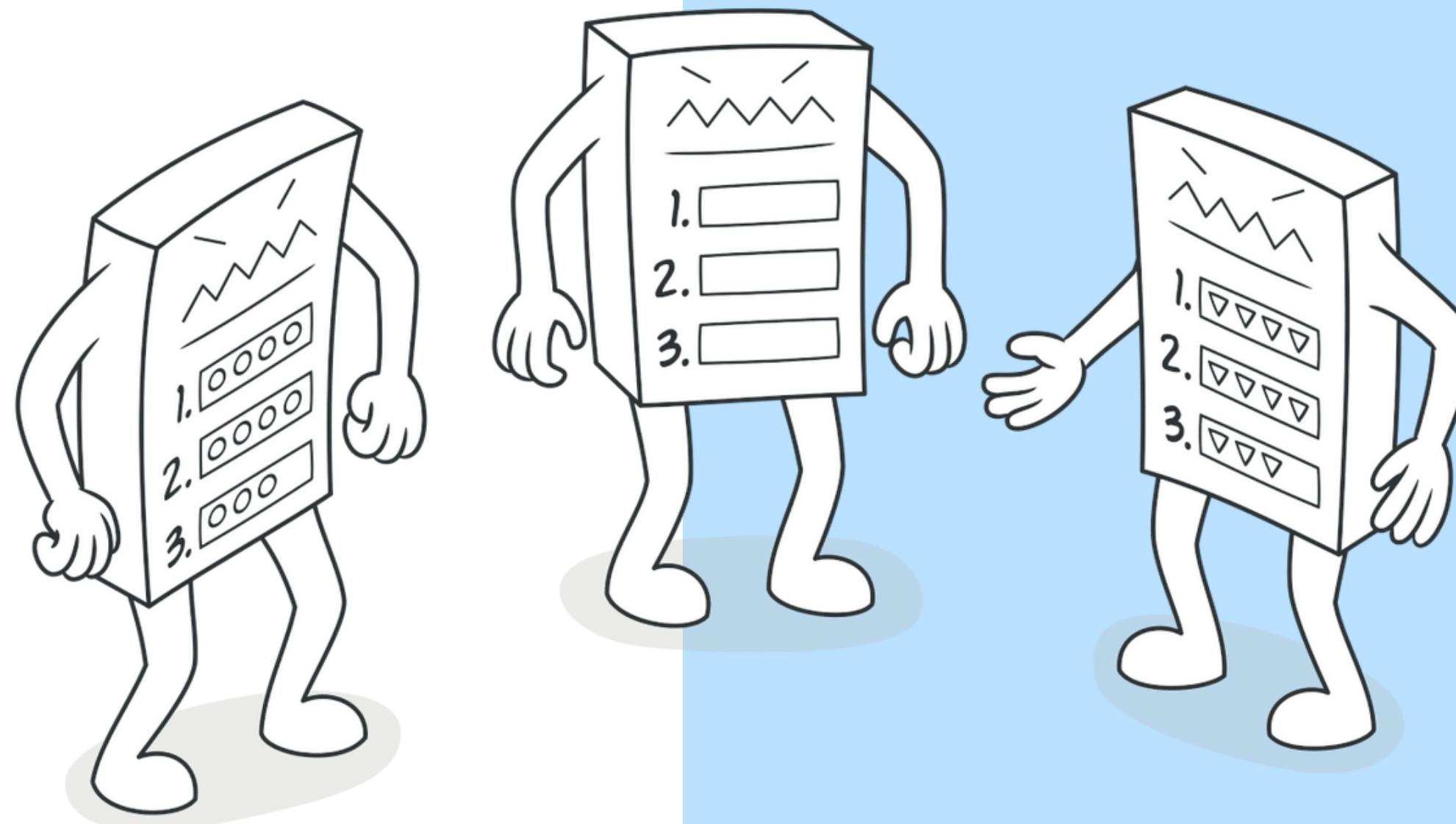
Behavioral	Basado en la forma en que los objetos juegan y trabajan juntos.	
	Clase	
	Template Method	
	Crea el caparazón de un algoritmo en un método, luego pospone los pasos exactos a una subclase.	
	Objeto	
	Chain of Responsibility	
	Una forma de pasar una solicitud entre una cadena de objetos para encontrar el objeto que puede manejar la solicitud.	
	Command	
	Encapsule una solicitud de comando como un objeto para habilitar, registrar y / o poner en cola las solicitudes, y proporciona manejo de errores para las solicitudes no manejadas.	
	Iterator	
	Acceda secuencialmente a los elementos de una colección sin conocer el funcionamiento interno de la colección.	
	Mediator	
	Define la comunicación simplificada entre clases para evitar que un grupo de clases se refieran explícitamente entre sí.	
	Memento	
	Capture el estado interno de un objeto para poder restaurarlo más tarde.	
	Observer	
	Una forma de notificar cambios en una serie de clases para garantizar la coherencia entre las clases.	
	State	
	Alterar el comportamiento de un objeto cuando cambia su estado.	
	Estrategy	
	Encapsula un algoritmo dentro de una clase que separa la selección de la implementación.	
	Visitor	
	Agrega una nueva operación a una clase sin cambiar la clase.	

BIBLIOGRAFÍA

<https://refactoring.guru/design-patterns/behavioral-patterns>

<https://stackabuse.com/behavioral-design-patterns-in-java/>

**Gracias por
su atención**



Ejercicio

Simula un PULL REQUEST DE
GITHUB PARA UN PROYECTO

