THE OBSERVER PATTERN (Behavioral pattern)

« Observatör är ett designmönster där ett objekt (‘subject’) håller i en lista över beroende, som kallas ‘observatörer’, och meddelar dem automatisk när någonting ändras, vanligtvis genom att anropa en av deras metoder. » - Wikipedia

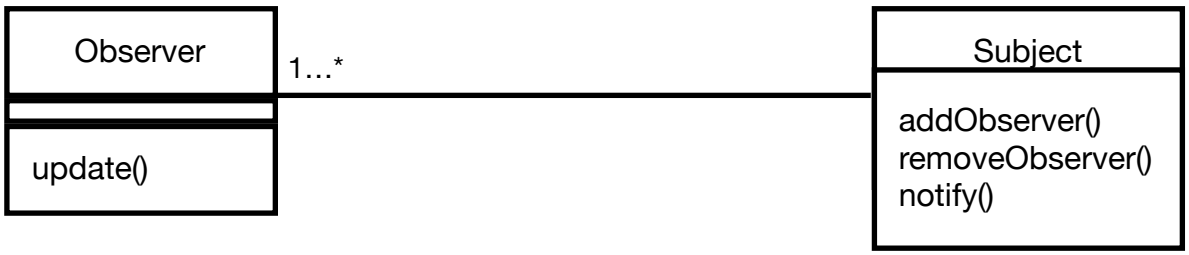
Observatörmönstret definierar en en-till-många relation. När ett objekt uppdateras meddelar det många andra objekt att det har uppdaterats.

‘The subject’ är objektet som skickar ut ett meddelande till alla observatörer som vill veta att ‘the subject’ har uppdaterats.

Observatörerna är de objekten som vill veta när ‘the subject’ har förändrats.

Implementering:

För att implementera detta designmönster i Javascript kan man skapa föräldrar klasser som kan ‘extend’. ‘The subject’ och observatörsföräldraklasserna har de egenskaper och metoder som krävs för att implementera observatörmönstret.



Subject klass

Syftet är att hålla en lista över observatörer som den ska meddela när den uppdateras.
Then bör ha metoder för att lägga till och ta bort observatörer.

Egenskaper och metoder:
- *observers* - den här egenskapen rymmer en array av observatörer.
- *addObserver*() - den här metoden kommer att pusha en observatör till observatörens array.
- *removeObserver*() - den här metoden tar bort en observatör från observatörens array.
- *notify*() - den här metoden skickar ett meddelande till alla observatörer om att en förändring har hänt.

Observer klass

Syftet är att implementera en uppdateringsmetod() som kommer att anropas av ‘the subjects’ metoden.

När kan man använda den?
- För att förbättra kodhanteringen genom att dela upp stora applikationer i ett system med löst kopplage objekt.
- För att ge större flexibilitet genom att möjliggöra en dynamisk relation mellan observatörer och ‘subjects’.
- För att förbättra kommunikationen mellan olika delar av applikationen.
- Skapa ett en-till-många förhållande mellan objekt som är löst kopplade.

THE CONSTRUCTOR PATTERN (Creational pattern)

« I klassbaserad objektorienterad programmering är en konstruktör en speciell typ av ‘subrutin’ som klass för att skapa ett objekt. Det förbereder det nya objektet för användning och accepterar ofta argument som konstruktören använder för att ställa in önskade egenskaper och methoder » - Wikipedia (översätt från engelska till svenska…)

‘Subrutin: är en sektens av programinstruktioner som utför en specifik uppgift, packad som en enhet.

I Javascript är användningen av konstruktörmönstret mycket populär för att skapa ett objekt eftersom det gör det möjligt att skapa flera objekt av ett visst slag.

I ES5-versionen av Javascript används funktioner som konstruktörer. Dessa funktioner används för att instantiera objekt.

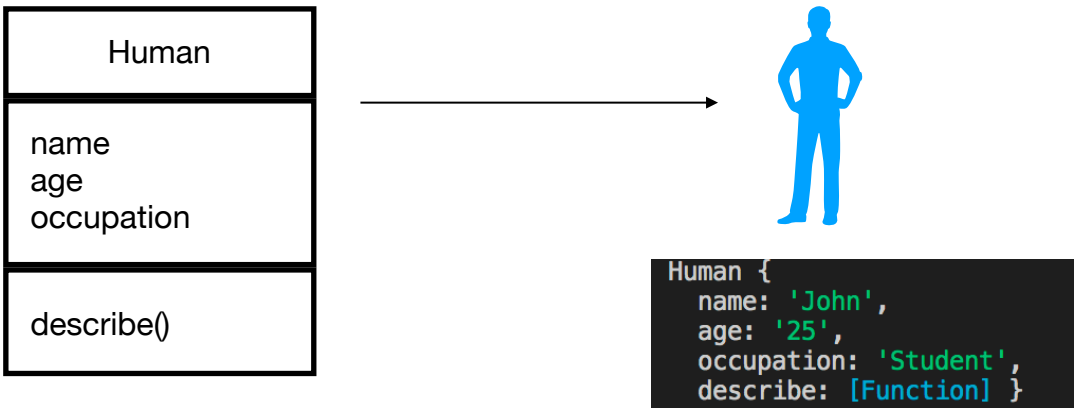
JS constructor.js > Human > describe

```
1   function Human(name, age, occupation) {
2     // defining properties
3     this.name = name;
4     this.age = age;
5     this.occupation = occupation;
6
7     // defining methods
8     this.describe = function () {
9       console.log(`${this.name} is a ${this.age} year old ${this.occupation}`);
10    };
11  }
12
13  // creating a 'person' object by sending arguments in the constructor
14  var person = new Human("John", "25", "Student");
15
16  // calling the describe method for the person object
17  person.describe();
18
```

I exemplet ovan definieras konstruktörfunktionen Human. Den har följande egenskaper och metod:

- age
- name
- occupation
- describe()

På rad 14 skapas ett objekt från Human med det ‘new’ nyckelordet. Det skapade objektet innehåller alla egenskaper och metoder som definieras i Human.



När ‘person’ -objektet skapas kommer dess ‘name’, ‘age’, ‘occupation’ egenskaper att vara inställda på argumenten som skickas till konsruktörfunktionen. På samma sätt kommer den också att innehålla metoden ‘describe’.

På rad 17 anropar vi det för objektet ‘person’ och får följande utdata på konsolen:

```
John is a 25 year old Student
```

För att uppnå denna utgång använder vi nyckelordet ‘this’ i konstuktörfunktionen. ‘This’ refererar det nya objektet som skapas, ‘person’ i vårt fall.

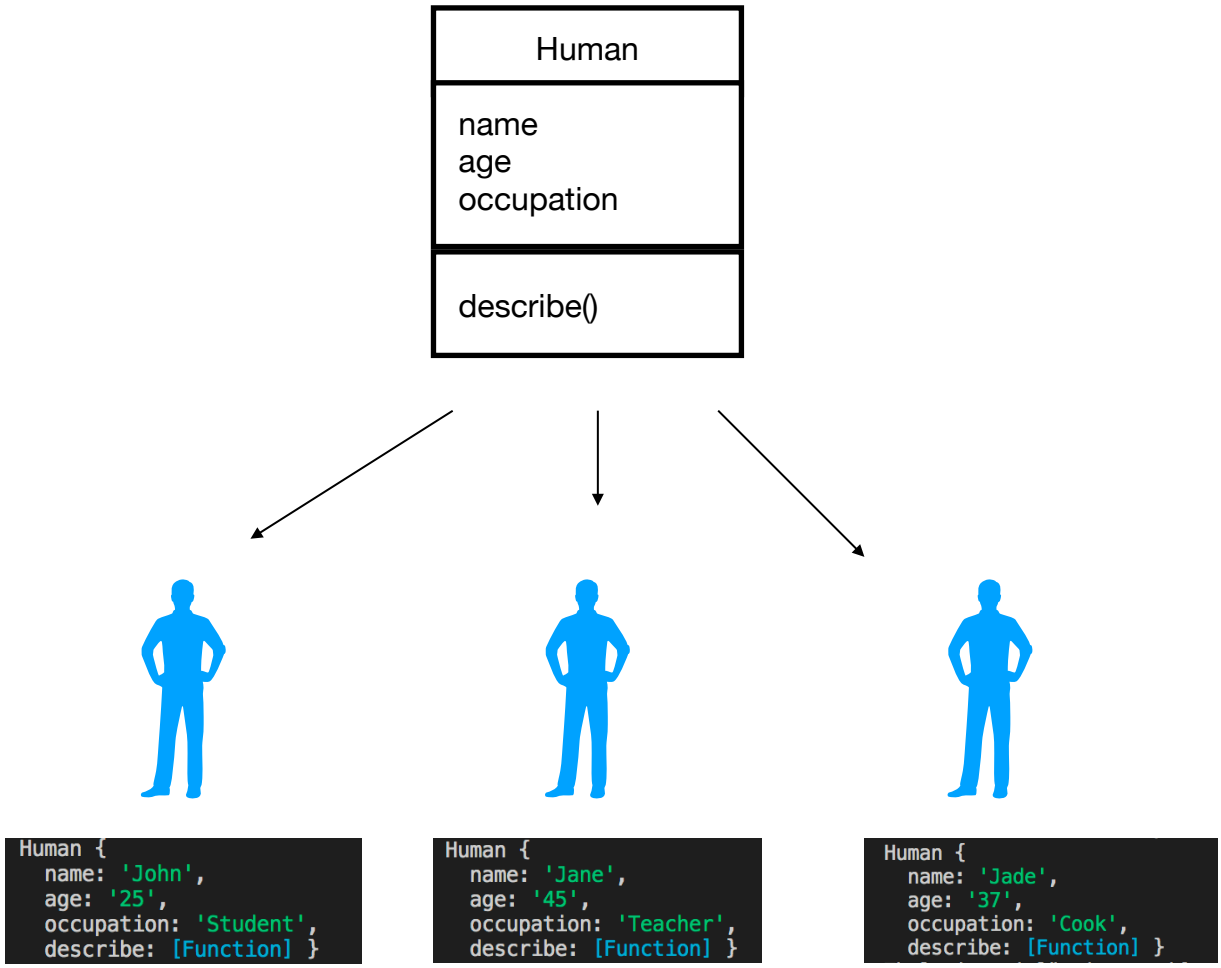
```
// defining properties
this.name = name;
this.age = age;
this.occupation = occupation;

// defining methods
this.describe = function () {
  console.log(`${this.name} is a ${this.age} year old ${this.occupation}`);
};
```

Vi kan använda vår konstruktör för att skapa nya skapa objekt:

```
var person = new Human("John", "25", "Student");
var person2 = new Human("Jane", "45", "Teacher");
var person3 = new Human("Jade", "37", "Cook");
```

Vi har nu skapat tre instanser av konstruktörfunktionen som delar samma egenskaper och metoder.

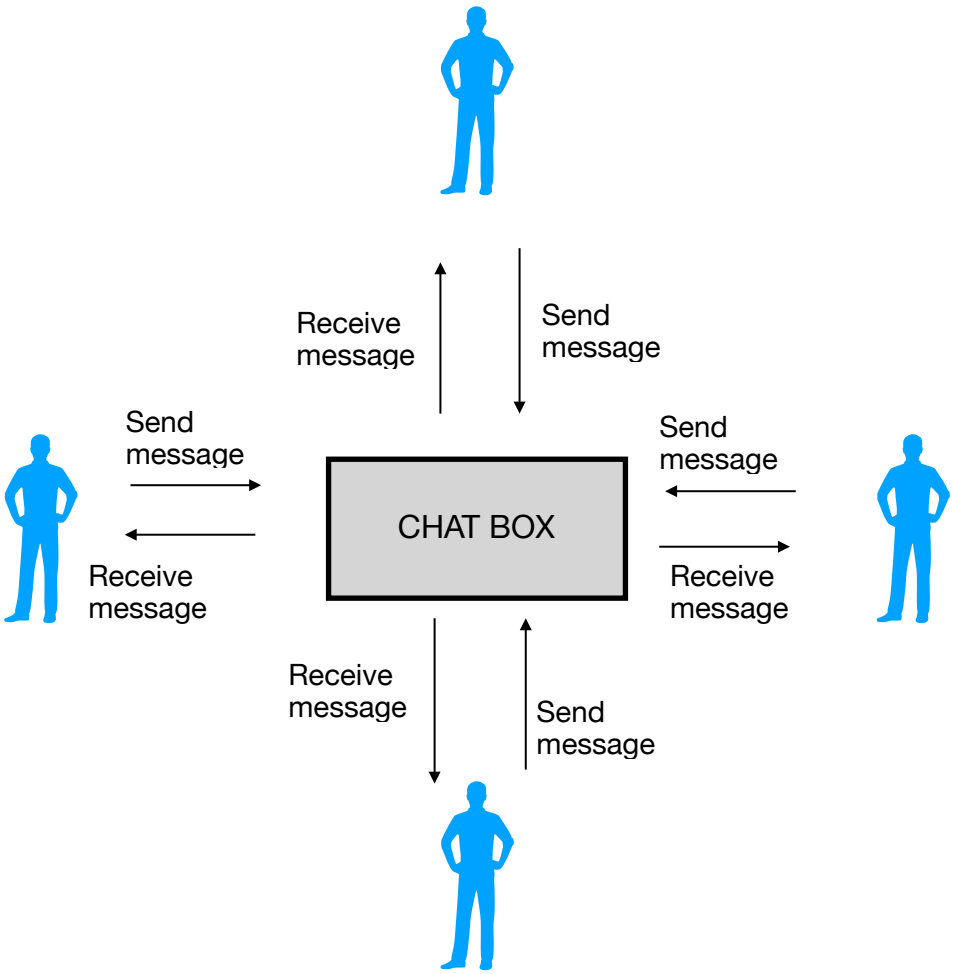

Human
name
age
occupation
describe()
Human {
name: 'John',
age: '25',
occupation: 'Student',
describe: [Function] }
Human {
name: 'Jane',
age: '45',
occupation: 'Teacher',
describe: [Function] }
Human {
name: 'Jade',
age: '37',
occupation: 'Cook',
describe: [Function] }

THE MEDIATOR PATTERN (Behavioral pattern)

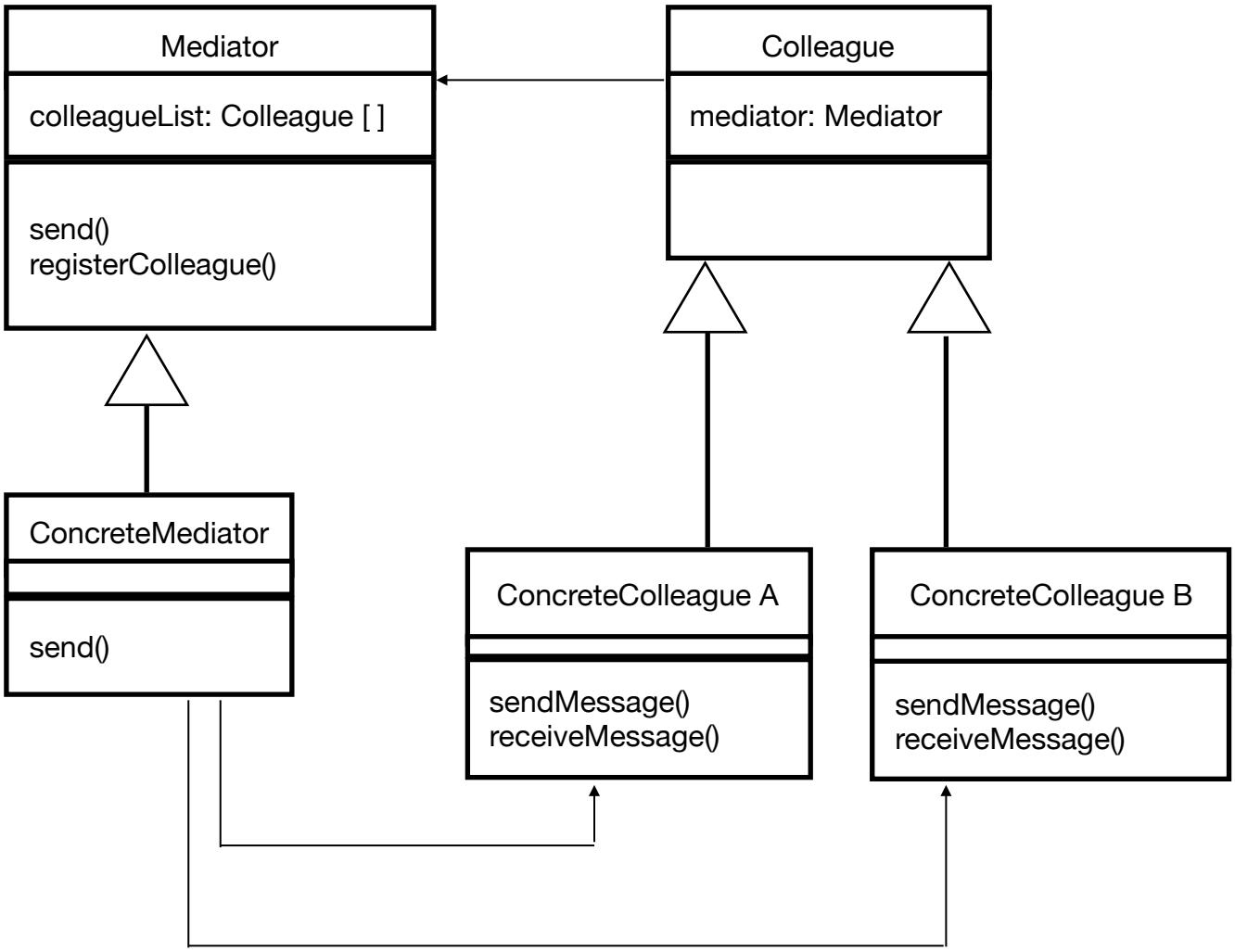
« Mediatormönstret definierar ett objekt som inkapslar hur en en uppsättning objekt interagerar» - Wikipedia

Det är ett beteendemönster som gör det möjligt för en central auktoritet att fungera som en koordinator mellan olika objekt, istället för att objekten hänvisar till varandra direkt. ‘The Mediator’ är den centrala auktoritet som olika komponenter kan kommunicera genom.

Om vi tar exemplet med en chattapplikation fungera ‘Chat Box’ som medlare ‘mediator’ genom vilken olika användare interagerar med varandra



För att implementera detta mönster skapar vi en ‘Mediator’ and ‘Colleagues’ (medlare och kollegor)



När ConcreteMediator registrerar en ConcreteColleague sätter den ConcreteColleagues ‘mediator’ egenskap till ‘this’ och pushar den till ‘colleagueLists’ array. ConcreteColleague blir en del ac Concrete Mediator.

Syftet med ConcreteMediator är att ta emot ett meddelande från en ConcreteColleague och skicka det till det andra.

ConcreteColleagues sendMessage() och receiveMessage() metoder can se ut så här:

```
sendMessage(message,sendTo){
   this.chatbox.send(message,this,sendTo)
}
```

```
receiveMessage(message,recieveFrom){
   console.log(`${recieveFrom.name} sent the message: ${message}`)
}
```

Och ConcreteMediators send() metod :

```
send(message,recieveFrom,sendTo){
   sendTo.receiveMessage(message, recieveFrom);
}
```

ConcreteMediator tar ‘message’ mottaget från en ConcreteColleague (‘receiveFrom’) och skickar det vidare till mottagaren (‘sendTo’) genom att anropa funktionen ‘receiveMessage’ för de ConcreteColleague son ska ta emot ‘message’.

‘The mediator’ mönster kan användas:
- När systemet har flera delar som behöver kommunicera.
- För att undvika tät koppling av föremål it ett system med många föremål.
- För att förbättra kodläsbarheten.
- För att göra koden enklare att underhålla.
- Om kommunikationen mellan objekt blir komplex och hindrar återvändbarheten av koden.

Källor:

- wikipedia.com
- jsmanifest.com
- webdevstudios.com
- educative.io