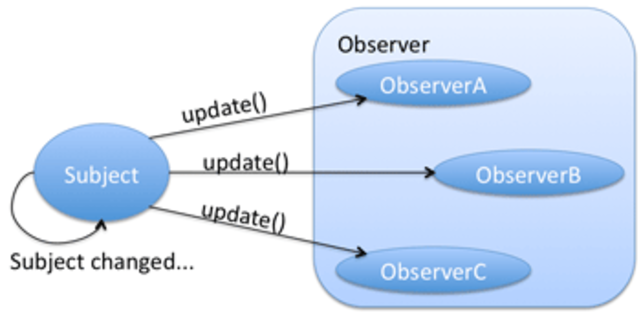
Top of Form

**observer**

این الگو زیرمجموعه ی الگوهای رفتاری (Behavioral) هست



طبق تعریف ویکی پدیا:

الگوی ناظر یا همان Observer یک الگوی طراحی نرم‌افزار است که در آن یک شی به نام موضوع (subject)، فهرست وابستگی‌هایش را با نام ناظران (observers) نگه می‌دارد و هرگونه تغییر در وضعیتش را به‌طور خودکار و معمولاً با صدا کردن یکی از روش‌های آن به اطلاع آن اشیا می‌رساند.

در حقیقت سعی داریم یه الگوی اصطلاحاً *رویدادمحور* رو پیاده سازی بکنیم، یعنی هربار موردی (شئ، پارامتر یا ...) تغییر میکنه بتونیم بدون صدا زدن تابع یا روش خاصی، و بصورت خودکار از این موضوع مطلع بشیم. توی این الگو، اشیائی که منتظر اعلام تغییر وضعیت هستن رو Observer و یک شئ که به بقیه اشیا تغییر رو اعلام می‌کنه Subject می‌گیم.

مثال :

اول یه کلاس abstract به اسم Observer بصورت زیر تعریف میکنیم:

abstract class Observer {

protected Subject subject;

public abstract void update();

}

همونطور که میبینیم این کلاس تابعی داره به اسم update میشه که abstractه و طبیعتاً هرکلاسی از این کلاس ارثبری کنه باید این تابع رو پیاده سازی بکنه. و یه شئ هم از کلاس Subject نگه میداره، این کلاس رو بصورت زیر تعریف میکنیم:

Subject {

private List<Observer> observers = new ArrayList<>();

private int state;

public void add(Observer o) {

observers.add(o);

}

public int getState() {

return state;

}

public void setState(int value) {

this.state = value;

execute();

}

private void execute() {

for (Observer observer : observers) {

observer.update();

}

}

}

این کلاس لیستی از Observer ها داره. تابعی با اسم add که یه Observer به لیست اضافه میکنه، تابع getState و setState که توابع setter و getter پارامتر state هستن. و در نهایت هم تابع execute که تابع *update* تمامی observerهای موجود توی لیست رو صدا میزنه.

خب حالا سه تا کلاس تعریف میکنیم، کلاس *HexObserver* (تبدیل state به hex)، کلاس *OctObserver* (تبدیل عدد state به oct) و در نهایت کلاس *BinObserver* (تبدیل عدد state به باینری):

HexObserver extends Observer {

public HexObserver(Subject subject) {

this.subject = subject;

this.subject.add(this);

}

public void update() {

System.out.print(" " + Integer.toHexString(subject.getState()));

}

}

123456789class OctObserver extends Observer {

public OctObserver(Subject subject) {

this.subject = subject;

this.subject.add( this );

}

public void update() {

System.out.print(" " + Integer.toOctalString(subject.getState()));

}

}

class BinObserver extends Observer {

public BinObserver(Subject subject) {

this.subject = subject;

this.subject.add(this);

}

public void update() {

System.out.print(" " + Integer.toBinaryString(subject.getState()));

}

}

هر سه تای این کلاس ها از کلاس Observer ارث بردن و تابع update رو با توجه به عملکرد خودشون پیاده سازی کردن. حالا برای تست قطعه کد زیر رو اجرا میکنیم (یه شئ از کلاس subject به اسم sub تعریف میکنیم، و اون رو به اشیا جدیدی که از کلاسای HexObserver و OctObserver و BinObserver میسازیم میدیم، و 5 عدد توی کنسول از کاربر میگیریم و اون رو به hex و oct و binary تبدیل میکنیم):

Subject sub = new Subject();

// Client configures the number and type of Observers

new HexObserver(sub);

new OctObserver(sub);

new BinObserver(sub);

Scanner scan = new Scanner(System.in);

for (int i = 0; i < 5; i++) {

System.out.print("\nEnter a number: ");

sub.setState(scan.nextInt());

}

خروجی این قطعه کد بصورت زیره:

Enter a number: 55

37 67 110111

Enter a number: 12

c 14 1100

Enter a number: -10

fffffff6 37777777766 11111111111111111111111111110110

Enter a number: 112

70 160 1110000

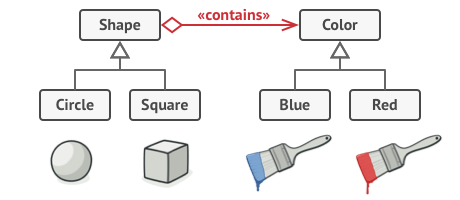
Enter a number: 5

5 5 101

همونطور که متوجه شدیم تابع setState علاوه بر اینکه پارامتر state رو ست میکنه، تابع execute رو هم صدا میزنه و اون تابع کارش صدا زدن تابع update مربوط به همه ی observerهاست! به همین راحتی.

**Bridge**

این الگو زیرمجموعه ی الگوهای ساختاری (Structural) هست



این الگو انتزاع را از پیاده سازی جدا می کند، بنابراین می توانند مستقل از هم باشند.

جمله ی بالا گفته ی گروه GOF هست که این الگو رو معرفی کردن. به عبارت دیگه این نوع الگو، از اون جایی که کلاسِ پیاده سازی شده رو جداسازی و کلاسی را با ارئه ی ساختارِ Bridge بین اونا abstract میکنه، به این نام معروف شده. در حقیقت وقتی که نیاز داریم انتزاع (Abstraction) رو از پیاده سازی (Implementation) مجزا کنیم، از Bridge استفاده میکنیم. بصورت کلی زمانی که کلاس ها کاری که انجام میدن نسبت به همدیگه تفاوتای زیادی داشته باشن، کلاس ها مدام در حال تغییر باشن، یا مواردی مشابه، از الگوی پل استفاده میکنیم. در این الگو خودِ کلاس به عنوان انتزاع در نظر گرفته و کاری که انجام میده در مرحله پیاده سازی مشخص میشه. همچنین الگوی پل میتونه به عنوان دو لایه از انتزاع در نظر گرفته بشه. این الگو از چهار قسمت اصلی تشکیل میشه:

* کلاسی Abstract که با client سرو کار داره و ارجاع یک شی از نوع Implementor را نگهداری می کند.
* کلاس RefinedAbstraction که کلاس Abstraction را توسعه میده تا چندین نسخه از متدهای abstract داشته باشه.
* اینترفیس Implementor یا Bridge که مثل یه پل یا رابط بین کلاس های abstract و کلاس های معمولی عمل میکنه.
* کلاس ConcreteImplementor که اینترفیس Implementor رو پیاده میکنه.