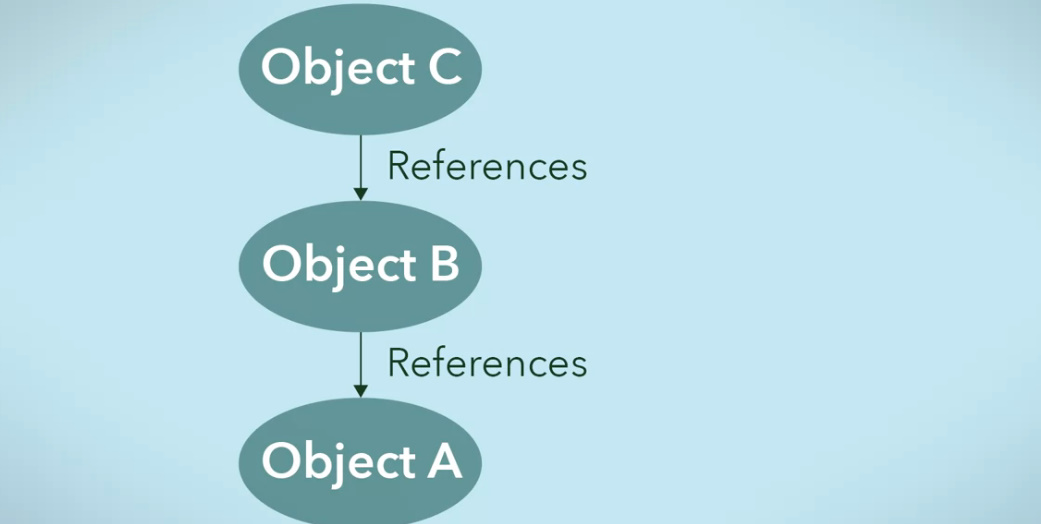
* In lumea reala adesea combinam lucrurile. De ex, bem cafea. Putem pune lapte, apa, oricum cafeua mereu e cafea, doar ca pusa in altceva. Mai putem pune spuma, zahar, si oricum e cafea, doar ca mai contine chestii inca.
* Indiferent de ce nu am pune, cafeaua e mereu cafea, si are aceleasi proprietati stimulante, doar ca e compusa din mai multe elemente.
* Combinarea ingredientelor cu cafea o face mai buna, mai gustoasa, si tot e mai bine sa le combinam decat sa le consumam aparte.
* Cand cream un obiect, comportamentul lui e strict legat de clasa sa. Pentru a-i modifica comportamentul,gen sa-i mai adaugam functiuni, trebuie sa modificam clasa, si logic ca asta are loc doar la compile time, nici de cum la runtime. Nu putem modifica structura clasei la runtime
* Unica solutie pare ca ar fi sa facem iar o clasa si sa mai rulam programul, desi nu e unica. Daca vrem sa mai adaugam functiuni si apoi iar si iar, am putea ajunge sa avem foarte multe clase pentru fiecare functiune.
* Deci, am putea avea o gramada de combinatii de comportament posibile, care ar duce la crearea unui numar enorm de clase
* **Decorator** – se bazeaza pe a adauga la runtime reposnabilitati sau comportamente noi unui obiect. El foloseste aggregation(has a) pentru a combina behaviours la runtime
* **Stack Aggregation** – un stack format prin obiecte ce au referinte la obiecte ce le ofera noi behaviours. De ex:

****

A are doar comportamentele si responsabilitatile sale,insa acum o sa le extindem si o sa-i mai oferim si comportamentele lui B si pe ale lui C. Desi C are o referinta la B si B la A, A e cel careuia ii extindem comportamentele. Prin C, in mod recursiv, vom incepe propriu zis executia lui A, apoi a lui B si abea a lui C propriu zis. Deci, A va avea inca functionalitatile lui B si C, ce vor fi executate dupa ele.

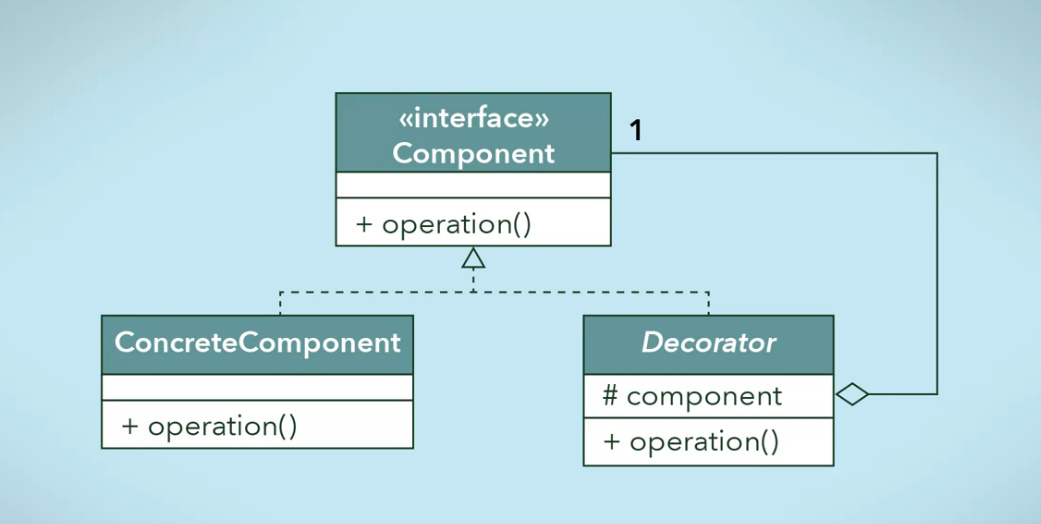
* **In Decorator, Aggregation relationship e mereu 1 TO 1, pentru a face acest stack**
* **Decorator** foloseste **Interfaces** si **Inheritance**

Asta e necesar ca sa putem face acest stack, acest apel de obiecte in linie.

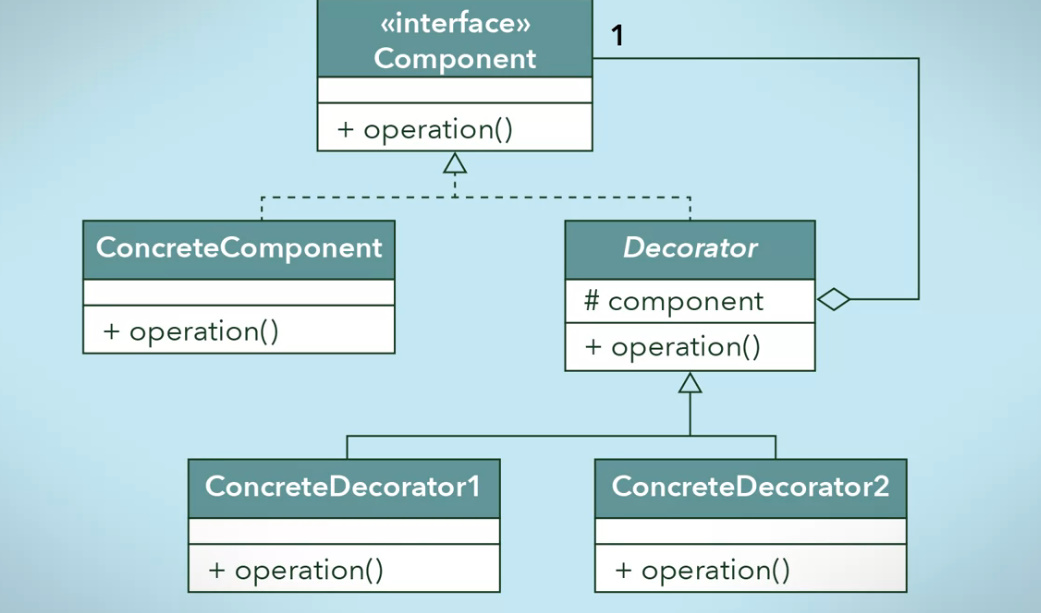
* **Decorator este un asbtract class**
* **Roluri:**

1. Agregheaza alte tipuri de componente, ceea ce va permite sa punem componentele in Decorator unul peste altul, in Stack deci
2. Este clasa abstracta ce va fi implementata de Decoratori concreti care vor oferi o incrementare la behaviour

**UML**



* Atat componentele, cat si Decoratorul, implementeaza o interfata,ca sa aiba o metoda comuna.
* Decoratorul are in ea un Component ca field.
* Decorator e asbtract class



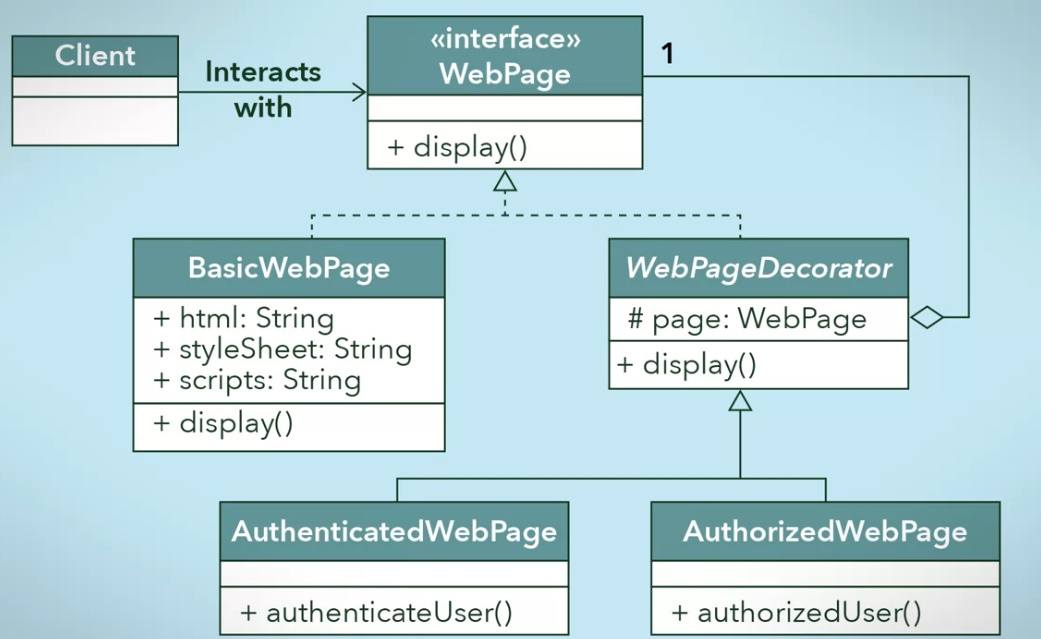
Fiecare Decorator va avea ca fields tot alti Decoratori, si cel final, la sfarsit de Stack, va avea un Component concret, cu decorator

Daca luam exemplul ca Cafeaua, Cafeaua va fi un ConcreteComponent, Decorators vor fi Milk,Spuma, Sugar

* Totusi, exemplul cu cofee nu e prea bun daca sa il luam ca exemplu, caci nu se aplica in software

**Exemplu**

* Luam ca exemplu o WebPage
* In webpage putem folosi pe larg acest design pattern
* WebPage poate avea multe behaviours. WebPage poate avea nevoie de a adauga foarte multe chestii, ca verificarea daca persoana e autorizata, sau daca un search ofera mai multe rezultate, sa le paginam
* Nu putem crea o webpage diferita pentru fiecare behaviour posibil.In schimb, putem crea o webpage cu mai multe behaviours, fara a crea foarte multe clase pentru orice combinatie. Gen, daca vrem o pagina ce face si autentificarea, ne va trebui o clasa pentru pagina, autentificare si apoi una pentru ambele. Daca mai vrem autorizare, mai trebuie o clasa pentru autorizare, una pentru pagina si autorizare, una pentru autorizare si autentificare si una pentru toate 3. E prea mult.
* In scimb, putem folosi decorator pentru a crea o clasa comuna pentru orice behaviour





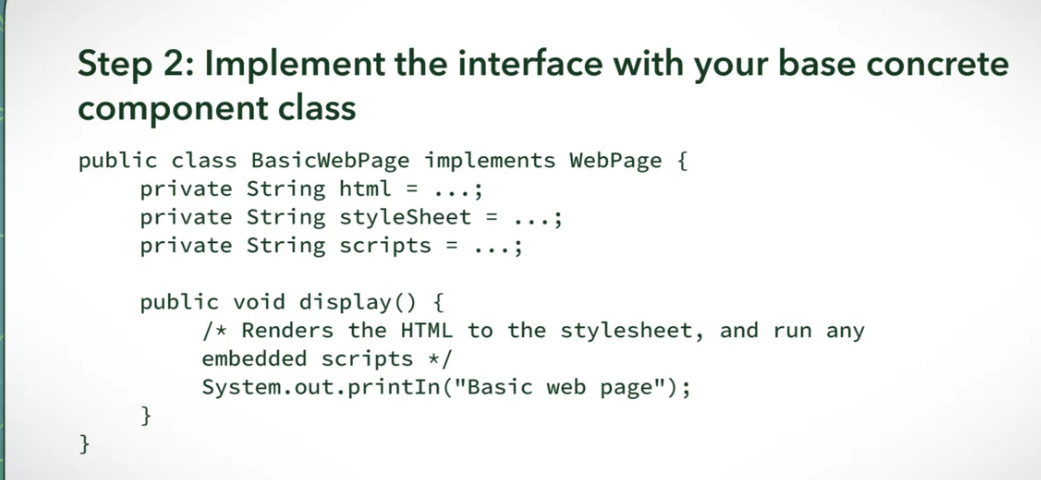
WebPage – defineste comportamentul pentru fiecare viitoare webpage

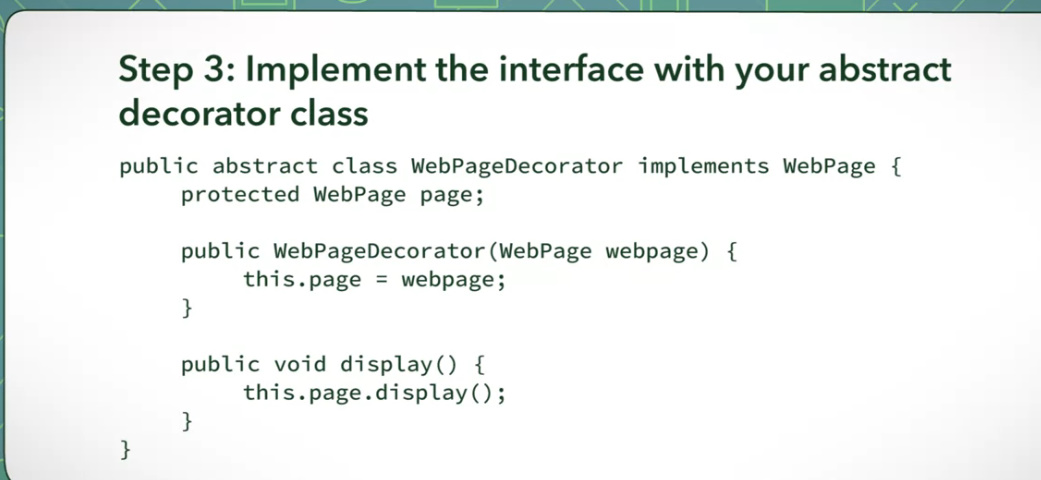
BasicWebPage – o pagina concreta

WebPageDecorator – subclasele sale adauga mai multe functionalitati la BasicWebPage, si foloseset Aggregation, in loc de a crea noi pages si apoi clase pentru a le combina comportamentele cate unul. In loc de a crea clase ce implementeaza doar WebPage, le cream de la WebPageDecorator ca sa le folosim behaviours.

* Daca nu am folosi Decorator, ar trebui sa cream clase pentru fiecare combinatie posibila la aceste behaviours.
* Code:







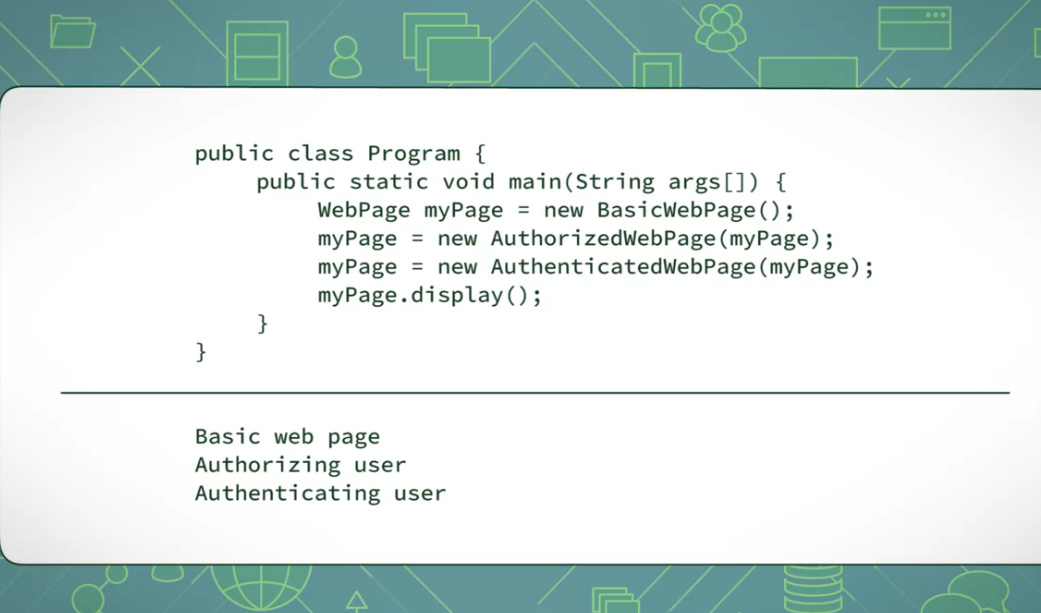
Avem doar un field ca sa cream stack. Observam ca metoda display() apeleaza metoda display() la field, apoi field apeleaza metoda display() la field din el si tot asa. Fiecare WebPage va apela recursiv metoda la field al sau.

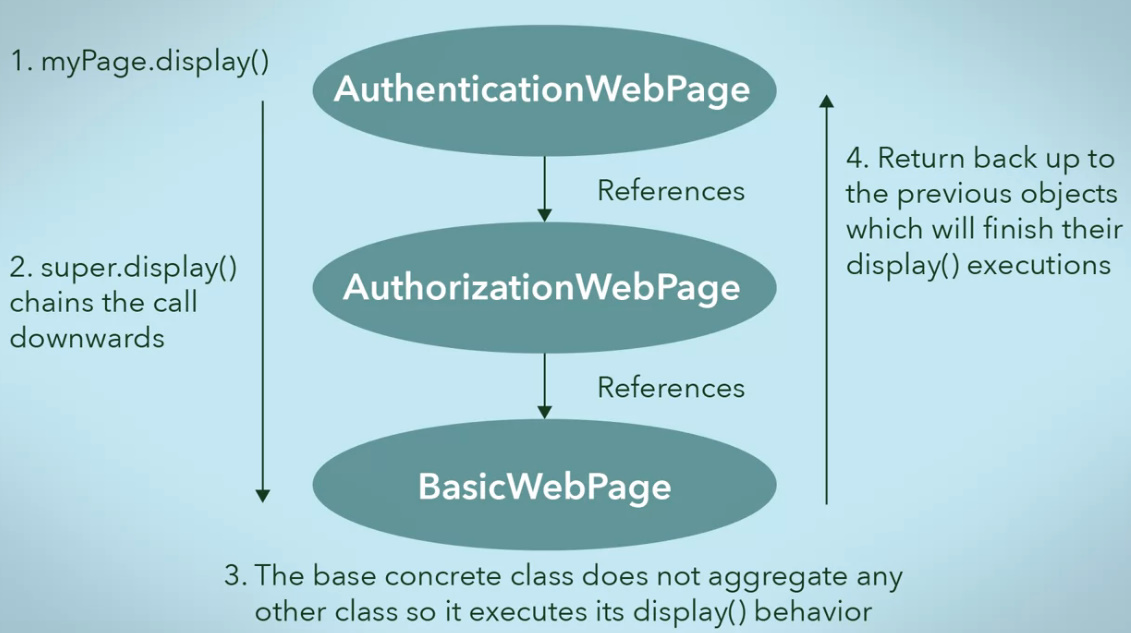
* Avem mare grija, ca ultimul element din stack sa fie un BasicWebPage, ca sa nu fie recursie infinita sau sa ajungem la un null field.



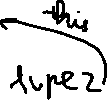
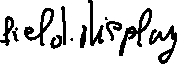
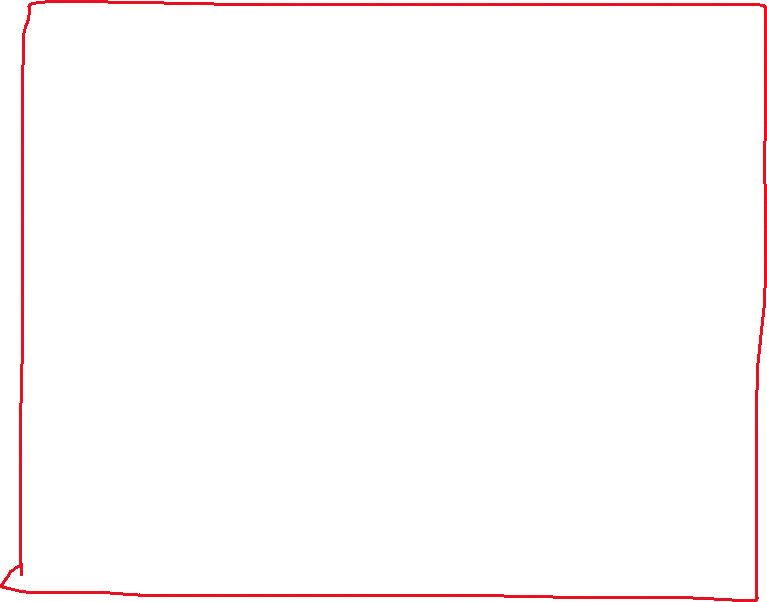
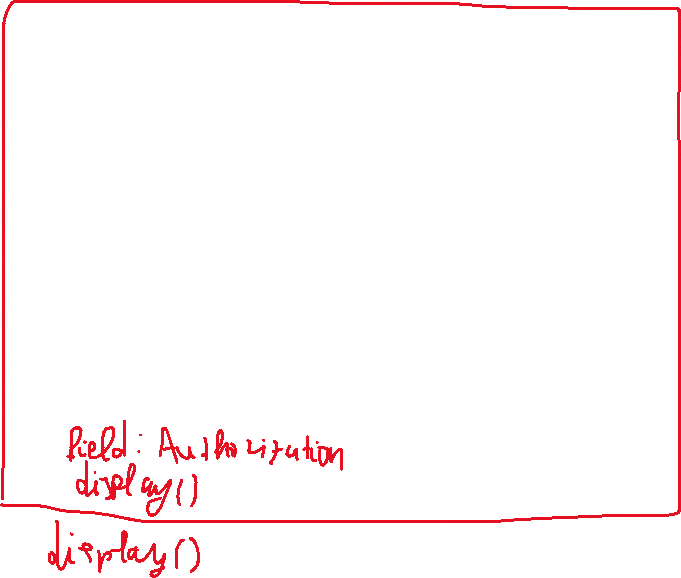
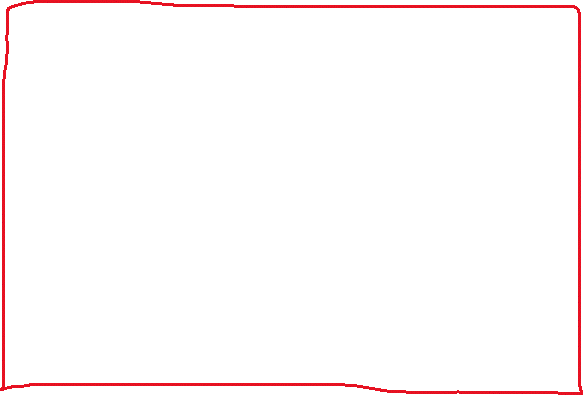
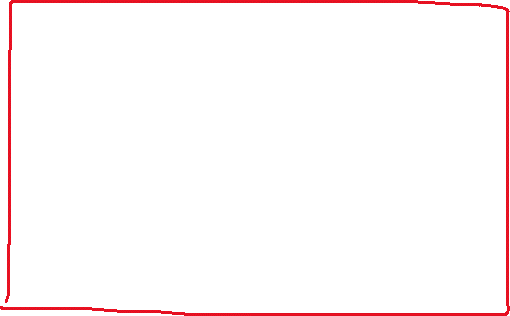
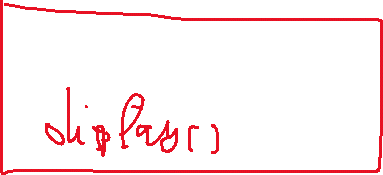
BasicPage, datorita recursiei, va fi prima ce e executata, ca se va ajunge la ea si la ea recursia si se va opri, si restul vor incepe sa se execute dupa ea.

Deci, avem o pagina ce e si custompage, face si ceea ce ar trebui sa faca authentificationpage si authorizationpage.



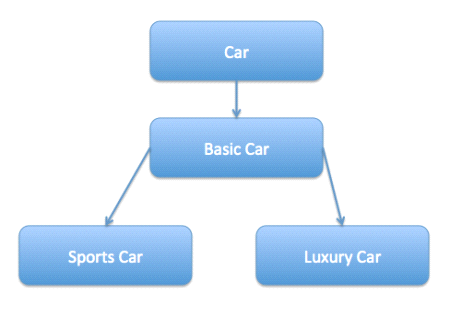


1. Authentification e ulimul. Deci, apelam display() din el. Display din el va apela super.display(). Asa, se va apela deja metoda display() din blocul secundar ce l-a mostenit de la WebPageDecorator, si acel display() va apela this.webpage.display()
2. Acum, Authentification a apelat deci field din el.
3. Field din el este de tip Authorization, ca la el are referinta, si acum se executa display() de la Authorization. Se apeleaza iar super.display(), se duce in blocul secundar din Authorization mostenit de la clasa abstracta, se apeleaza display() de acolo, si el executa this.webpage.display().
4. Field executat din Authorization deja e de tip CustomWebPage. Deci, se executa metoda display() a lui. El deja nu mai are field ca membru
5. Acum, Authorization continua executia la display() a sa din bloc secundar, dar nu mai e nimic si se executa a lui din bloc secundar
6. Tot asa si la Authentification



**Ex2**

* Decorator Permite sa adaugam noi functionalitati la un obiect existent fara a-i altera structura la runtime
* Acest pattern creaza o clasa decorator care impacheteaza clasa originala si ofera functionalitati aditionale pastrand metodele clasei intacte.
* De exemplu, fie ca avem un Car interface. Putem cream o clasa ce o implementeaza, gen BasicCar, ce creaza o car general, si apoi altele 2 mai concrete, gen SportsCar si LuxuryCar



* Dar, daca vrem sa cream un car ce este si sport si luxury, aici e problema cam. Daca mai cream vreo 10 tipuri de masini, si vrem ca sa fie creat un tip ce le contine pe toate,mostenirea nu ne poate ajuta, de aceea folosim decorator

1. Avem nevoie de o interfata:

interface Car {  
  
 void assemble();  
}

2. Avem nevoie de o implementre concreta:

class BasicCar implements Car {  
  
 @Override  
 public void assemble() {  
 System.*out*.print("Basic Car.");  
 }  
  
}

3. Acum, ne trebuie o clasa care implementeaza interfata si are o relatie HAS-A cu interfata. Ideea este ca clasele copii, create cu decorator, trebuie sa aiba acces la metoda assemble() a obiectului trimis. Obiectul trimis mereu va trimis in constructor cu un BasicCar. Rolul la la CarDecorator e pur si simplu de a rula metoda obiectului de tip Car trimis cu aceeasi metoda ca la interfata si gata

class CarDecorator implements Car {  
  
 private Car car;  
  
 public CarDecorator(Car c){  
 this.car=c;  
 }  
  
 @Override  
 public void assemble() {  
 this.car.assemble();  
 }  
  
}

4. Cream decoratori concreti, care vor apela metoda la CarDeecorator numita assemble(), ce apeleaza si ea metoda la Car(), si plus SportsCar va mai face ceva specific la sports car

class SportsCar extends CarDecorator {  
  
 public SportsCar(Car c) {  
 super(c);  
 }  
  
 @Override  
 public void assemble(){  
 super.assemble();  
 System.*out*.print(" Adding features of Sports Car.");  
 }  
}  
class LuxuryCar extends CarDecorator {  
  
 public LuxuryCar(Car c) {  
 super(c);  
 }  
  
 @Override  
 public void assemble(){  
 super.assemble();  
 System.*out*.print(" Adding features of Luxury Car.");  
 }  
}

Si avem main:

{  
 public static void main( String[] args ) {  
 Car sportsCar = new SportsCar(new BasicCar());  
 sportsCar.assemble();  
 System.*out*.println("\n\*\*\*\*\*");  
  
 Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));  
 sportsLuxuryCar.assemble();  
 }

Vedem ca am putut crea un sportsCar si apoi un car care este deja si luxury si sports

E important mereu sa oferim un obiect care implementeaza interfata, aici BasicCar, cand cream obiecte de tip copiii ai decorator, ca el sa creeze functiunile de baza la oricare Car, adica BasicCar, si deja apoi orice setam noi la ale tippuri de cars. Putem face macar si u car ce cuprinde specificatile la 1000 de cars, cu new(new(new(new....)))), la oate se vor executa metoda assemble()

