Decorator pattern

A decorator allows us to dynamically extend the behavior of a particular object at runtime, without needing to resort to unnecessary inheritance.

class BasicInspection

{

    public function getCost()

    {

        return 25;

    }

}

echo (new BasicInspection())->getCost();//echoes 25

if we need to add a 2nd service, we can just create 1 more class:

class BasicInspection

{

    public function getCost()

    {

        return 25;

    }

}

class BasicInspectionAndOilChange

{

    public function getCost()

    {

        return 25 + 20;

    }

}

echo (new BasicInspectionAndOilChange())->getCost();//echoes 25

\*this approach is bad as for each new service combination we have to create a new class;

class BasicInspection

{

    public function getCost()

    {

        return 25;

    }

}

class BasicInspectionAndOilChange

{

    public function getCost()

    {

        return 25 + 20;

    }

}

class BasicInspectionAndOilChangeAndTyreRotation

{

    public function getCost()

    {

        return 25 + 20 + 15;

    }

}

echo (new BasicInspectionAndOilChangeAndTyreRotation())->getCost();//echoes 60

\*this is also bad because the values are hard-coded in multiple places;

\*we cannot decorate classes

The Decorator pattern allows us to modify objects at run-time (by adding a wrapper to them):

interface CarService

{

    public function getCost();

}

class BasicInspection implements CarService

{

    public function getCost()

    {

        return 25;

    }

}

//decorators for additional services

class OilChange implements CarService{

//declare CarService as protected property

    protected $CarService;

    //inject the CarService instance in the OilChange at runtime;

   function \_\_construct(CarService $carService)

    {

        $this->carService = $carService;

    }

    public function getCost()

    {

        return 29 + $this->carService->getCost();

    }

}

$service = new OilChange(new BasicInspection());

echo $service->getCost();  //echoes 54

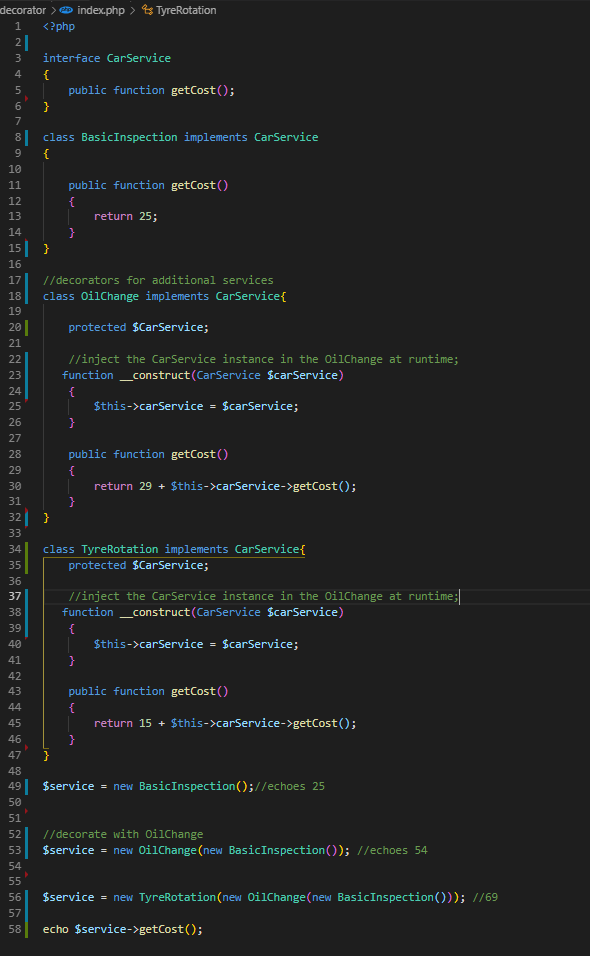
\*we have the BasicInspection class which returns 25 as a cost;

\*the OilChange decorates it and adds 29 to the initial cost;

\*note that each decorator must have a CarService instance injected in the constructor at runtime;

\*above the BasicInspection is the protected $carService in the OilChange;

\*if we add a 3rd class:



\*we can decorate it too (note that each time we add a new wrapper, the class inside of it ‘becomes’ the CarService instance that is being injected in the constructor of the new decorator:

$service = new TyreRotation(new OilChange(new BasicInspection())); //69

\*above the OilChange is the protected $carService in the TyreRotation;

\*we can combine the decorators however we want (as long as the base class from which we start to ‘decorate’ is the basic inspection;

$service = new TyreRotation(new BasicInspection()); //40

\*if I feel pressured to inherit from other classes but I feel that I do not need to inherit all the methods of these classes, and instead I only need to adjust the behavior of some of them, then the decorator patterns seems a good choice;