Understand the "Decorator Pattern" with a real world example [closed]

Asked 14 years, 8 months ago Modified 10 months ago

Viewed 104k times

184

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 3 years ago.

Improve this question

I was studying the *Decorator Pattern* as documented in GOF.

Please, help me understand the *Decorator Pattern*. Could someone give a use-case example of where this is useful in the real world?

decorator design-patterns

Share

Improve this question

edited Aug 8, 2017 at 22:27

ROMANIA_engineer

56.5k ● 30 ● 208 ● 205

Follow

asked Apr 25, 2010 at 6:36

odiseh

26.5k ● 33 ● 113 ● 154

8 You can find here some realworld examples in Java API: stackoverflow.com/questions/1673841/… – BalusC Apr 29, 2010 at 15:27

An article that shows the benefits of the decorator pattern with simple examples: dzone.com/articles/is-inheritance-dead – nbilal Dec 8, 2016 at 9:28

14 Answers

Sorted by: Highest score (default)

239

> **Decorator pattern achieves a single objective of dynamically adding responsibilities to any object.**

Consider a case of a pizza shop. In the pizza shop they will sell few pizza varieties and they will also provide toppings in the menu. Now imagine a situation wherein if the pizza shop has to provide prices for each combination of pizza and topping. Even if there are four basic pizzas and 8 different toppings, the application would go crazy maintaining all these concrete combination of pizzas and toppings.

Here comes the decorator pattern.

As per the decorator pattern, you will implement toppings as decorators and pizzas will be decorated by those toppings' decorators. Practically each customer would want toppings of his desire and final bill-amount will be composed of the base pizzas and additionally ordered toppings. Each topping decorator would know about the pizzas that it is decorating and it's price. `GetPrice()` method of Topping object would return cumulative price of both pizza and the topping.

EDIT

Here's a code-example of explanation above.

```
public abstract class BasePizza
{
    protected double myPrice;

    public virtual double GetPrice()
    {
        return this.myPrice;
    }
}

public abstract class ToppingsDecorator : BasePizza
{
    protected BasePizza pizza;
    public ToppingsDecorator(BasePizza pizzaToDecorate
    {
        this.pizza = pizzaToDecorate;
    }

    public override double GetPrice()
    {
        return (this.pizza.GetPrice() + this.myPrice);
    }
```

```
}

class Program
{
    [STAThread]
    static void Main()
    {
        //Client-code
        Margherita pizza = new Margherita();
        Console.WriteLine("Plain Margherita: " + pizza

        ExtraCheeseTopping moreCheese = new ExtraChees
        ExtraCheeseTopping someMoreCheese = new ExtraC
        Console.WriteLine("Plain Margherita with doubl
someMoreCheese.GetPrice().ToString());

        MushroomTopping moreMushroom = new MushroomTop
        Console.WriteLine("Plain Margherita with doubl
mushroom: " + moreMushroom.GetPrice().ToString());

        JalapenoTopping moreJalapeno = new JalapenoTop
        Console.WriteLine("Plain Margherita with doubl
mushroom with Jalapeno: " + moreJalapeno.GetPrice().To

        Console.ReadLine();
    }
}

public class Margherita : BasePizza
{
    public Margherita()
    {
        this.myPrice = 6.99;
    }
}

public class Gourmet : BasePizza
{
    public Gourmet()
    {
        this.myPrice = 7.49;
    }
}
```

```
public class ExtraCheeseTopping : ToppingsDecorator
{
    public ExtraCheeseTopping(BasePizza pizzaToDecorat
        : base(pizzaToDecorate)
    {
        this.myPrice = 0.99;
    }
}

public class MushroomTopping : ToppingsDecorator
{
    public MushroomTopping(BasePizza pizzaToDecorate)
        : base(pizzaToDecorate)
    {
        this.myPrice = 1.49;
    }
}

public class JalapenoTopping : ToppingsDecorator
{
    public JalapenoTopping(BasePizza pizzaToDecorate)
        : base(pizzaToDecorate)
    {
        this.myPrice = 1.49;
    }
}
```

Share Improve this answer Follow

edited Oct 9, 2021 at 14:40

community wiki
15 revs, 6 users 88%
this. __curious_geek

121 Do not like this pattern one bit. Maybe it's the example though. The main issue I have with it in terms of OOD is that a topping *is not a pizza*. Asking the topping for the price of the pizza it's applied to just doesn't sit right with me. It's a very thoughtful and detailed example though, so I don't

mean to knock you for that. – Tom W Dec 8, 2010 at 8:27

47 @TomW I think part of the issue is the naming. All of the "Topping" classes should be called "PizzaWith<Topping>". For example, "PizzaWithMushrooms". – Josh Noe Aug 19, 2013 at 18:58

29 From another perspective this is not even close to "real world". In the real world you should not recompile each time you need to add a new topping in menu (or change the price). Toppings are (usually) stored in database and thus render the above example useless. – Stelios Adamantidis Oct 8, 2015 at 13:06

6 ^ This. I think this is what's been bothering me all along while studying this pattern. If I was a software company and wrote pizza shop software, I wouldn't want to have to recompile and reship every time. I'd want to add a row in a table in the backend or something that would easily take care of their requirement. Well said, @Stelios Adamantidis. I guess the patterns biggest strength though would be modifying 3rd-party classes then. – Canucklesandwich Dec 9, 2015 at 1:16

5 The reason this is a bad example is that you're not solving a real problem here by using the Decorator Pattern. A "ham and mushroom pizza" isn't a "some mushrooms with a (ham with a pizza under it) under it". No, it's a pizza with the following ingredients: [ham, mushroom]. If you were writing a real world application, you'd just be making the whole thing more complicated than it needs to be. I'd love to see an example where a genuine problem is solved with this pattern. – Rocketmagnet Jun 23, 2017 at 9:48

This is a simple example of adding new behavior to an existing object dynamically, or the Decorator pattern. Due

40

to the nature of dynamic languages such as Javascript, this pattern becomes part of the language itself.

```
// Person object that we will be decorating with log
var person = {
  name: "Foo",
  city: "Bar"
};

// Function that serves as a decorator and dynamical
given object
function MakeLoggable(object) {
  object.log = function(property) {
    console.log(this[property]);
  }
}

// Person is given the dynamic responsibility here
MakeLoggable(person);

// Using the newly added functionality
person.log('name');
```

Run code snippet Expand snippet

Share Improve this answer edited Aug 18, 2019 at 17:00

Follow

answered Apr 25, 2010 at 9:22

Anurag

142k ● 37 ● 222 ● 261

Simple and precise! Great example! – nagendra547 Aug 6, 2019 at 3:04

3 I don’t think that the concept of Decorator Pattern is applicable here. Actually it is not a pattern at all!. Yes, You are adding a new method at runtime. And probably inside a `switch` or a simple `if` ,you would be able to claim that this is a great example of dynamically adding behavior to a class.BUT, we need at least two classes to define a decorator and decorated objects in this pattern. – Iman Rosstin Aug 17, 2019 at 22:11

5 @Zich I understand there's no decorator in my example but that's easily fixed by adding a function that serves as a decorator. But there is a decorated object in my example. Does the pattern say anywhere that you need two **classes** specifically? – Anurag Aug 18, 2019 at 16:55

23

It's worth noting that the Java i/o model is based on the decorator pattern. The layering of this reader on top of that reader on top of...is a really real world example of decorator.

Share Improve this answer

Follow

answered Apr 26, 2010 at 14:39

frankc

11.5k ● 4 ● 33 ● 49

Are there any other examples in real public APIs? This is the only one I know. – Josiah Yoder Jan 9, 2018 at 22:29

It seems that all wrapper functions in nature has some sort of decorator pattern build in, is that what I think it is? – Harvey Lin Apr 23, 2018 at 3:42

1 Good example !! – nagendra547 Aug 6, 2019 at 3:05

Example - Scenario- Let's say you are writing an encryption module. This encryption can encrypt the clear file using DES - Data encryption standard. Similarly, in a system you can have the encryption as AES - Advance encryption standard. Also, you can have the combination of encryption - First DES, then AES. Or you can have first AES, then DES.

Discussion- How will you cater this situation? You cannot keep creating the object of such combinations - for example - AES and DES - total of 4 combinations. Thus, you need to have 4 individual objects This will become complex as the encryption type will increase.

Solution - Keep building up the stack - combinations depending on the need - at run time. Another advantage of this stack approach is that you can unwind it easily.

Here is the solution - in C++.

Firstly, you need a base class - a fundamental unit of the stack. You can think as the base of the stack. In this example, it is clear file. Let's follow always polymorphism. Make first an interface class of this fundamental unit. This way,you can implement it as you wish. Also, you don't need to have think of dependency while including this fundamental unit.

Here is the interface class -

```
class IclearData
{
```

```
public:

    virtual std::string getData() = 0;
    virtual ~IclearData() = 0;
};

IclearData::~IclearData()
{
    std::cout<<"Destructor called of IclearData"
<<std::endl;
}
```

Now, implement this interface class -

```
class clearData:public IclearData
{
private:

    std::string m_data;

    clearData();

    void setData(std::string data)
        {
            m_data = data;
        }

public:

    std::string getData()
    {
        return m_data;
    }

    clearData(std::string data)
    {
        setData(data);
    }

    ~clearData()
    {
        std::cout<<"Destructor of clear Data
```

```
Invoked"<<std::endl;
    }

};
```

Now, let's make a decorator abstract class - that can be extended to create any kind of flavours - here the flavour is the encryption type. This decorator abstract class is related to the base class. Thus, the decorator "is a" kind of interface class. Thus, you need to use inheritance.

```
class encryptionDecorator: public IclearData
{

protected:
    IclearData *p_mclearData;

    encryptionDecorator()
    {
      std::cout<<"Encryption Decorator Abstract
class called"<<std::endl;
    }

public:

    std::string getData()
    {
        return p_mclearData->getData();
    }

    encryptionDecorator(IclearData *clearData)
    {
        p_mclearData = clearData;
    }

    virtual std::string showDecryptedData() = 0;

    virtual ~encryptionDecorator() = 0;

};
```

```
encryptionDecorator::~encryptionDecorator()
{
    std::cout<<"Encryption Decorator Destructor
called"<<std::endl;
}
```

Now, let's make a concrete decorator class - Encryption type - AES -

```
const std::string aesEncrypt = "AES Encrypted ";

class aes: public encryptionDecorator
{

private:

    std::string m_aesData;

    aes();

public:

    aes(IclearData *pClearData):
m_aesData(aesEncrypt)
    {
        p_mclearData = pClearData;
        m_aesData.append(p_mclearData->getData());
    }

    std::string getData()
        {
            return m_aesData;
        }

    std::string showDecryptedData(void)
    {
        m_aesData.erase(0,m_aesData.length());
        return m_aesData;
    }
```

```
};
```

Now, let's say the decorator type is DES -

```
const std::string desEncrypt = "DES Encrypted ";

class des: public encryptionDecorator
{

private:

    std::string m_desData;

    des();

public:

    des(IclearData *pClearData):
m_desData(desEncrypt)
    {
        p_mclearData = pClearData;
        m_desData.append(p_mclearData->getData());
    }

    std::string getData(void)
        {
            return m_desData;
        }

    std::string showDecryptedData(void)
    {
        m_desData.erase(0,desEncrypt.length());
        return m_desData;
    }

};
```

Let's make a client code to use this decorator class -

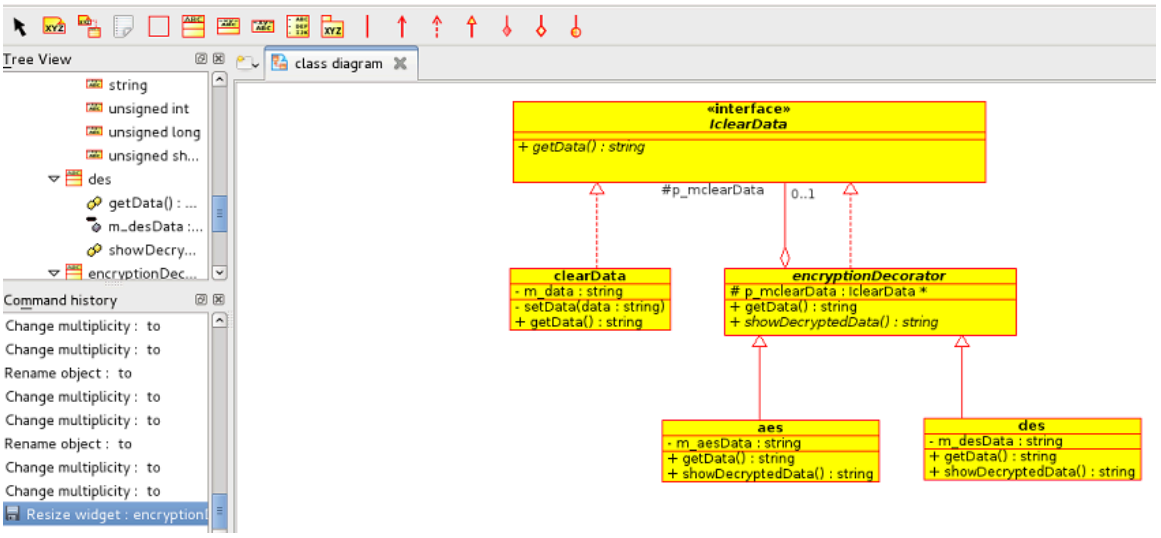
```
int main()
{
    IclearData *pData = new
clearData("HELLO_CLEAR_DATA");

    std::cout<<pData->getData()<<std::endl;


    encryptionDecorator *pAesData = new
aes(pData);

    std::cout<<pAesData->getData()<<std::endl;

    encryptionDecorator *pDesData = new
des(pAesData);

    std::cout<<pDesData->getData()<<std::endl;

    /** unwind the decorator stack ***/
    std::cout<<pDesData->showDecryptedData()
<<std::endl;

    delete pDesData;
    delete pAesData;
    delete pData;

    return 0;
}
```

You will see the following results -

```
HELLO_CLEAR_DATA
Encryption Decorator Abstract class called
AES Encrypted HELLO_CLEAR_DATA
Encryption Decorator Abstract class called
DES Encrypted AES Encrypted HELLO_CLEAR_DATA
AES Encrypted HELLO_CLEAR_DATA
Encryption Decorator Destructor called
Destructor called of IclearData
Encryption Decorator Destructor called
Destructor called of IclearData
```

```
Destructor of clear Data Invoked
Destructor called of IclearData
```

Here is the UML diagram - Class representation of it. In case, you want to skip the code and focus on the design aspect.



Share Improve this answer

Follow

edited Jan 13, 2021 at 18:59

Yogesh Umesh Vaity

47.7k • 23 • 157 • 114

answered Dec 1, 2014 at 13:36

dexterous

6,526 • 12 • 56 • 103

2 isn't the example more suited for `strategy pattern` ? – exexzian Aug 8, 2017 at 17:04

@exexzian Yes, my students consistently suggest a list of strategies to me for this type of problem, and it feels like the cleanest solution to me, too. – Josiah Yoder Jan 9, 2018 at 22:30

4 No, with the strategy pattern you cannot combine the encryption methods. Therofore you would have to create an

strategy class for every possible combination. – deetz Oct 7, 2019 at 15:49

@deetz not necessarily true. I have a model where an object can have a ordered list of strategies. each one acts on the 'Pizza price'. I think it has helped me clarify the difference in my mind. In a Decorator pattern we use an -IS- chain to combine functionality, where a strategy patter, the parent object -HAS- an object w/ the functionality. Im sure i could wordsmith that better.... – greg Mar 11, 2022 at 14:55

@greg not sure if I understand you correct, but you cannot have a list of strategies, you have exactly one strategy. – deetz Nov 15, 2022 at 10:13

6

Decorator pattern helps you to change or configure a functionality of your object by chaining with other similar subclasses of this object.

Best example would be InputStream and OutputStream classes in java.io package

```
    File file=new File("target","test.txt");
    FileOutputStream fos=new
FileOutputStream(file);
    BufferedOutputStream bos=new
BufferedOutputStream(fos);
    ObjectOutputStream oos=new
ObjectOutputStream(bos);


    oos.write(5);
    oos.writeBoolean(true);
    oos.writeBytes("decorator pattern was here.");


//... then close the streams of course.
```

Share Improve this answer

Follow

answered Aug 8, 2017 at 19:45

huseyin

1,427 ● 17 ● 19

> In this case, the calling chain starts at ObjectOutputStream, then goes all the way up to the File class, then File class returns the value, then the other three subclasses adds them all up and finally, the value of ObjectOutputStream's method returns it, is that correct? – Harvey Lin Apr 23, 2018 at 3:46

4

> The decorator pattern lets you dynamically add behavior to objects.

Let's take an example where you need to build an app that calculates the price of different kinds of burgers. You need to handle different variations of burgers, such as "large" or "with cheese", each of which has a price relative to the basic burger. E.g. add $10 for burger with cheese, add extra $15 for large burger, etc.

In this case you might be tempted to create subclasses to handle these. We might express this in Ruby as:

```
class Burger
  def price
    50
  end
end

class BurgerWithCheese < Burger
  def price
```

```
    super + 15
  end
end
```

In the above example, the BurgerWithCheese class inherits from Burger, and overrides the price method to add $15 to the price defined in the super class. You would also create a LargeBurger class and define the price relative to Burger. But you also need to define a new class for the combination of "large" and "with cheese".

Now what happens if we need to serve "burger with fries"? We already have 4 classes to handle those combinations, and we will need to add 4 more to handle all combination of the 3 properties - "large", "with cheese" and "with fries". We need 8 classes now. Add another property and we'll need 16. This will grow as 2^n.

Instead, let's try defining a BurgerDecorator that takes in a Burger object:

```
class BurgerDecorator
  def initialize(burger)
    self.burger = burger
  end
end

class BurgerWithCheese < BurgerDecorator
  def price
    self.burger.price + 15
  end
end

burger = Burger.new
```

```
cheese_burger = BurgerWithCheese.new(burger)
cheese_burger.price    # => 65
```

In the above example, we've created a BurgerDecorator class, from which BurgerWithCheese class inherits. We can also represent the "large" variation by creating LargeBurger class. Now we could define a large burger with cheese at runtime as:

```
b = LargeBurger.new(cheese_burger)
b.price  # => 50 + 15 + 20 = 85
```

Remember how using inheritance to add the "with fries" variation would involve adding 4 more subclasses? With decorators, we would just create one new class, BurgerWithFries, to handle the new variation and handle this at runtime. Each new property would need just more decorator to cover all the permutations.

PS. This is the short version of an article I wrote about using the Decorator Pattern in Ruby, which you can read if you wish to find out more detailed examples.

Share Improve this answer

Follow

answered Mar 23, 2015 at 18:14

Nithin

314 ● 2 ● 5

tried thanking you on Linkedin for this, but some oddities prevented me from adding you (security things your end I think). – Luke Hill Oct 25, 2022 at 10:26

What is Decorator Design Pattern in Java.

The formal definition of the Decorator pattern from the GoF book (Design Patterns: Elements of Reusable Object-Oriented Software, 1995, Pearson Education, Inc. Publishing as Pearson Addison Wesley) says you can,

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Let's say we have a Pizza and we want to decorate it with toppings such as Chicken Masala, Onion and Mozzarella Cheese. Let's see how to implement it in Java ...

Program to demonstrate how to implement Decorator Design Pattern in Java.

- See more at: http://www.hubberspot.com/2013/06/decorator-design-pattern-in-java.html#sthash.zKj0xLrR.dpuf

Pizza.java:

```
<!-- language-all: lang-html -->

package
com.hubberspot.designpattern.structural.decorator;

public class Pizza {

public Pizza() {

}
```

```
public String description(){
    return "Pizza";
}

}



package
com.hubberspot.designpattern.structural.decorator;

public abstract class PizzaToppings extends Pizza
{

public abstract String description();

}

package
com.hubberspot.designpattern.structural.decorator;

public class ChickenMasala extends PizzaToppings {

private Pizza pizza;

public ChickenMasala(Pizza pizza) {
    this.pizza = pizza;
```

Share Improve this answer

Follow

edited Jun 22, 2013 at 16:49

user155407

answered Jun 22, 2013 at 16:27

Jonty

41 • 3

3

Decorators are just a compositional alternative to subclassing. You're augmenting an existing object with new functionality by wrapping it in another object which



contains the new implementation. The wrapper maintains the existing interface, and outside code should interact with the wrapped object in the same way as before. The common example from the original book on this topic, that everyone mentions, is with a text processing application.

Lets say you write a paragraph. You highlight it yellow. You italicize one sentence. You bolden half of the italicized sentence, and half of the next sentence too. You increase the font size of one of the italicized & bold letters. You change the font style of half the highlighted portion, some of it running over into the italicized portion, some not...

So I'm going to ask you how you'd implement that functionality. You start out with a class for a simple, undecorated letter. What do you do next?

I'm going to assume you would not use subclassing. You would need such a complex, convoluted hierarchy of multiple inheritance to achieve all the combinations I described and more, that subclassing and multiple inheritance just would be absurd. And I think that needs no explanation.

What you'd probably suggest is just packing all these properties into your letter object. Properties to define the font style, the size, the highlighting, bold, italicized, the list goes on. Every kind of property you could add to a letter object, you've got a property in your letter class for it.

So what are the problems with this property based approach?

1. now your class is bloated, it takes up a giant amount of memory. It has all these unnecessary properties associated with it, most that it will never use. Most letters are just... letters. Not decorated.
2. The data of your letter class is being used in a way that is completely exposed, your class is just a glorified struct. You have a bunch of getters and setters for all these properties. Outside code accesses those setters and modifies the graphical appearance of your object. There's tight coupling between your object and the outside code.
3. Everything is packed into one place, it's not modular. It's just going to be a bloated, interconnected bundle of code. That's going to be true in the outside code that handles your letter object as well, too.

Fundamentally it's a question of object oriented design, proper encapsulation and separation of concerns.

Now, let's just take it for granted we wanted to use better OO design principles. We want to use encapsulation, we want to maintain loose coupling between the outside code and our letter class. We wanted to minimize our letter objects memory footprint. How...? We can't use subclassing...

So we use decorators, which are a compositional approach to object oriented design - it's sort of the

opposite of a top-down, subclassing approach. You wrap these letter objects with more functionality at runtime, building on top of them.

So that's what the decorator pattern is - it's a compositional alternative to subclassing. In our example you add a decorator to the letter object that needs highlighting. You can combine any number of decorators in an arbitrary number of ways, and wrap them all around a given letter. The decorator interface is always transparent, so you still treat these letters the same from the outside.

Any time you need to augment functionality in a way that's arbitrary and recombinable, consider this approach. Multiple inheritance runs into all kinds of problems, it just isn't scalable.

Share Improve this answer Follow

edited Feb 18 at 13:53

answered Jan 16, 2021 at 6:44

ibrust

180 ● 1 ● 8

2

Some time back I had refactored a codebase into using Decorator pattern, so I will try to explain the use case.

Lets assume we have a set of services and based on whether the user has acquired licence of particular service, we need to start the service.

All the services have a common interface

```
interface Service {
  String serviceId();
  void init() throws Exception;
  void start() throws Exception;
  void stop() throws Exception;
}
```

Pre Refactoring

```
abstract class ServiceSupport implements Service {
  public ServiceSupport(String serviceId, LicenseManag
    // assign instance variables
  }

  @Override
  public void init() throws Exception {
    if (!licenseManager.isLicenseValid(serviceId)) {
       throw new Exception("License not valid for serv
    }
    // Service initialization logic
  }
}
```

If you observe carefully, `ServiceSupport` is dependent on `LicenseManager`. But why should it be dependent on `LicenseManager`? What if we needed background service which doesn't need to check license information. In the current situation we will have to somehow train `LicenseManager` to return `true` for background services. This approach didn't seem well to me. According to me license check and other logic were orthogonal to each other.

So *Decorator Pattern* comes to the rescue and here starts refactoring with TDD.

Post Refactoring

```
class LicensedService implements Service {
  private Service service;
  public LicensedService(LicenseManager licenseManager
    this.service = service;
  }

  @Override
  public void init() {
    if (!licenseManager.isLicenseValid(service.service
      throw new Exception("License is invalid for serv
service.serviceId());
    }
    // Delegate init to decorated service
    service.init();
  }

  // override other methods according to requirement
}

// Not concerned with licensing any more :)
abstract class ServiceSupport implements Service {
  public ServiceSupport(String serviceId) {
    // assign variables
  }

  @Override
  public void init() {
    // Service initialization logic
  }
}

// The services which need license protection can be d
service
Service aLicensedService = new LicensedService(new Ser
licenseManager);
// Services which don't need license can be created wi
```

```
need to pass license related information
Service aBackgroundService = new BackgroundService1("B
```

Takeaways

- Cohesion of code got better
- Unit testing became easier, as don't have to mock licensing when testing ServiceSupport
- Don't need to bypass licensing by any special checks for background services
- Proper division of responsibilities

Share Improve this answer

Follow

answered Mar 13, 2019 at 23:46

Narendra Pathai

41.9k ● 18 ● 86 ● 122

2

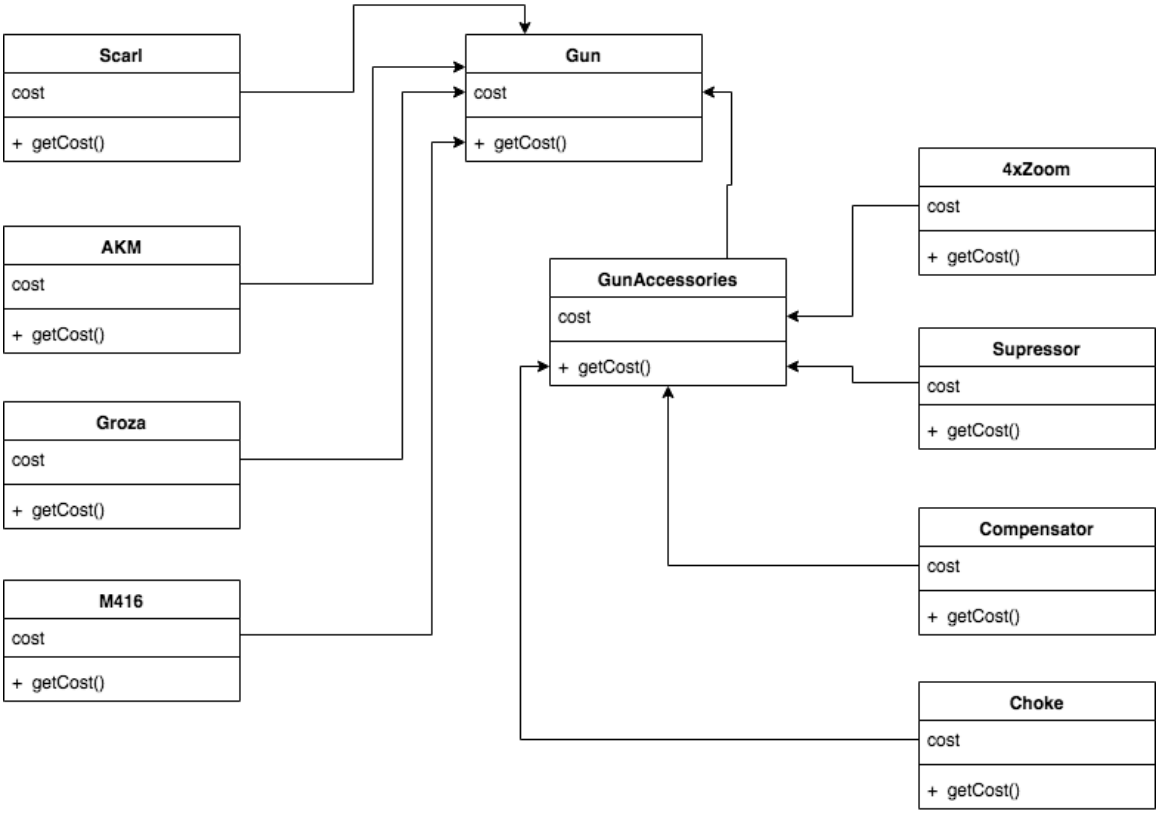
Let's take example of PubG. Assault rifles works best with 4x zoom and while we are on it, we would also need compensator and suppressor. It will reduce recoil and reduce firing sound as well as echo. We will need to implement this feature where we will allow players to buy their favourite gun and their accessories. Players can buy the gun or some of the accessory or all of the accessory and they would be charged accordingly.

Let's see how decorator pattern is applied here:

Suppose someone wants to buy SCAR-L with all three accessories mentioned above.

1. Take an object of SCAR-L
2. Decorate (or add) the SCAR-L with 4x zoom object
3. Decorate the SCAR-L with suppressor object
4. Decorate the SCAR-L with compressor object
5. Call the cost method and let each object delegate to add on the cost using cost method of accessories

This will lead to a class diagram like this:



Now, we can have classes like this:

```
public abstract class Gun {
    private Double cost;
    public Double getCost() {
        return cost;
       }
    }

public abstract class GunAccessories extends Gun {
```

```
}

public class Scarl extends Gun {
    public Scarl() {
        cost = 100;
        }
     }

public class Suppressor extends GunAccessories {
    Gun gun;
    public Suppressor(Gun gun) {
    cost = 5;
    this.gun = gun;
    }
    public double getCost(){
        return cost + gun.getCost();
    }
}

public class GunShop{
    public static void main(String args[]){
    Gun scarl = new Scarl();
    scarl = new Supressor(scarl);
    System.out.println("Price is
"+scarl.getCost());
    }
}
```

We can similarly add other accessories too and decorate our Gun.

Reference:

https://nulpointerexception.com/2019/05/05/a-beginner-guide-to-decorator-pattern/

Share Improve this answer

Follow

answered May 6, 2019 at 9:04

H-V

131 ● 1 ● 4

2 I feel like this example doesn't justify the complexity of a decorator. It would be much simpler to have each gun hold a list of attachments, and calculate the gun's cost by summing the costs of the attachments – Ted Brownlow Jan 13, 2021 at 19:05

2

Decorator:

1. *Add behaviour to object at run time*. Inheritance is the key to achieve this functionality, which is both advantage and disadvantage of this pattern.
2. It enhances the *behaviour* of interface.
3. Decorator add responsibilities to objects without sub-classing

JDK example:

```
BufferedInputStream bis = new
BufferedInputStream(new FileInputStream(new
File("a.txt")));
while(bis.available()>0)
{
        char c = (char)bis.read();
        System.out.println("Char: "+c);;
}
```

Have a look at below SE question for UML diagram and code examples.

Use Cases and Examples of GoF Decorator Pattern for IO

Real word example of Decorator pattern: *VendingMachineDecorator* has been explained @

When to Use the Decorator Pattern?

```
Beverage beverage = new SugarDecorator(new
LemonDecorator(new Tea("Assam Tea")));
beverage.decorateBeverage();

beverage = new SugarDecorator(new
LemonDecorator(new Coffee("Cappuccino")));
beverage.decorateBeverage();
```

In above example, Tea or Coffee (Beverage) has been decorated by Sugar and Lemon.

Share Improve this answer Follow

edited Apr 2, 2023 at 12:40

answered May 30, 2016 at 10:26

Ravindra babu

38.9k ● 11 ● 256 ● 219

1

There is a example on Wikipedia about decorating a window with scrollbar:

http://en.wikipedia.org/wiki/Decorator_pattern

Here is another very 'real world' example of "Team member, team lead and manager", which illustrates that decorator pattern is irreplaceable with simple inheritance:

https://zishanbilal.wordpress.com/2011/04/28/design-patterns-by-examples-decorator-pattern/

Share Improve this answer

Follow

edited Dec 27, 2016 at 21:11

stonedauwg

1,406 ● 1 ● 16 ● 39

answered Aug 8, 2014 at 14:05

gm2008

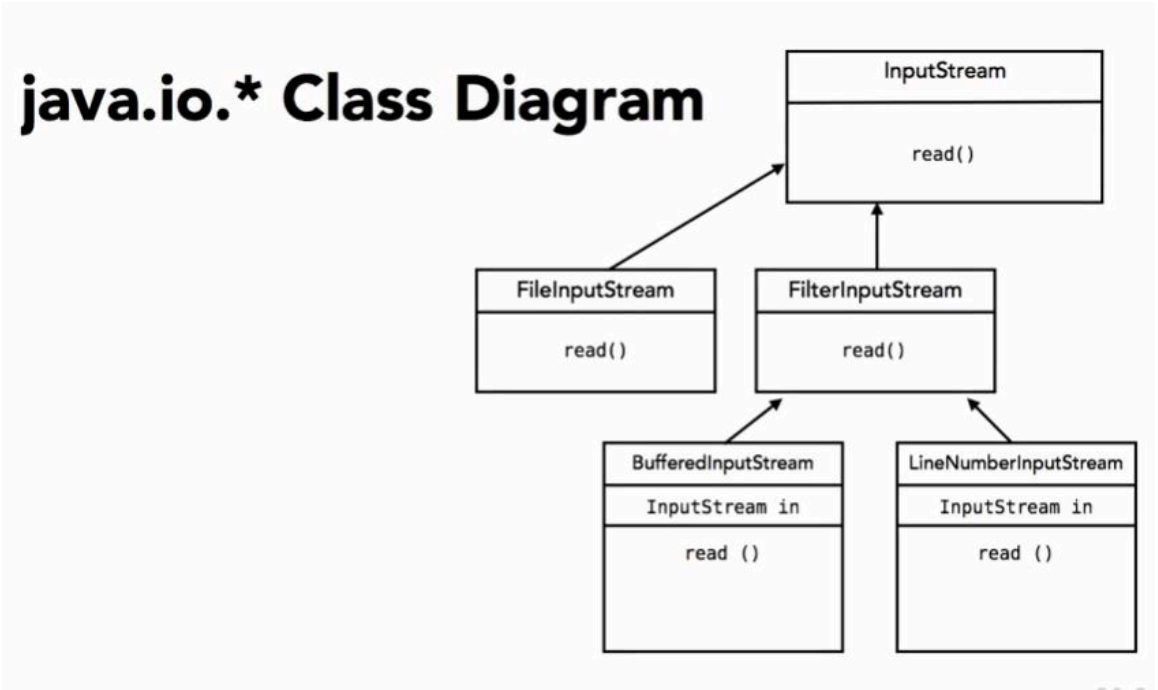
4,325 ● 1 ● 37 ● 38

That Zishan Bilal link is great - best example I've seen – stonedauwg Dec 27, 2016 at 21:09

1

Decorator pattern achieves a single objective of **dynamically adding responsibilities to any object**.

Java I/O Model is based on decorator pattern.



Share Improve this answer

answered Jul 29, 2018 at 6:28

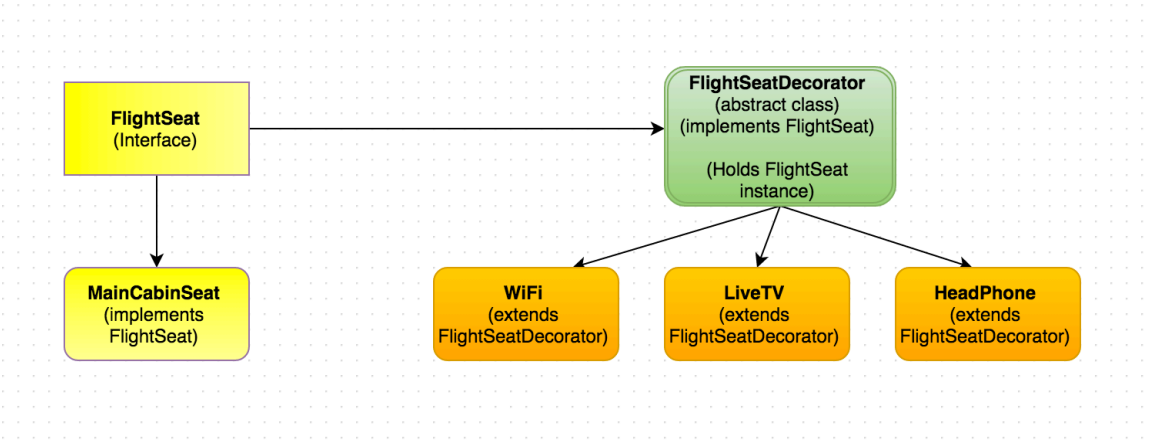
Follow

ChandraBhan Singh
2,971 ● 1 ● 25 ● 29

0

[Decorator Design Pattern](): This pattern helps to modify the characteristics of an object at runtime. It provides different flavours to an object and gives flexibility to choose what ingredients we want to use in that flavour.

Real Life Example: Lets say you have a main cabin seat in a flight. Now you are allowed to choose multiple amenities with the seat. Each amenity has its own cost associated with it. Now if a user chooses Wifi and premium food, he/she will be charged for seat + wifi + premium food.



In this case decorator design pattern can really help us. Visit the above link to understand more about decorator pattern and implementation of one real life example.

Share Improve this answer

Follow

answered Oct 30, 2015 at 14:40

Ajit Singh
2,490 ● 24 ● 27