

Decorator Pattern and Extending Interfaces

CPSC 2150

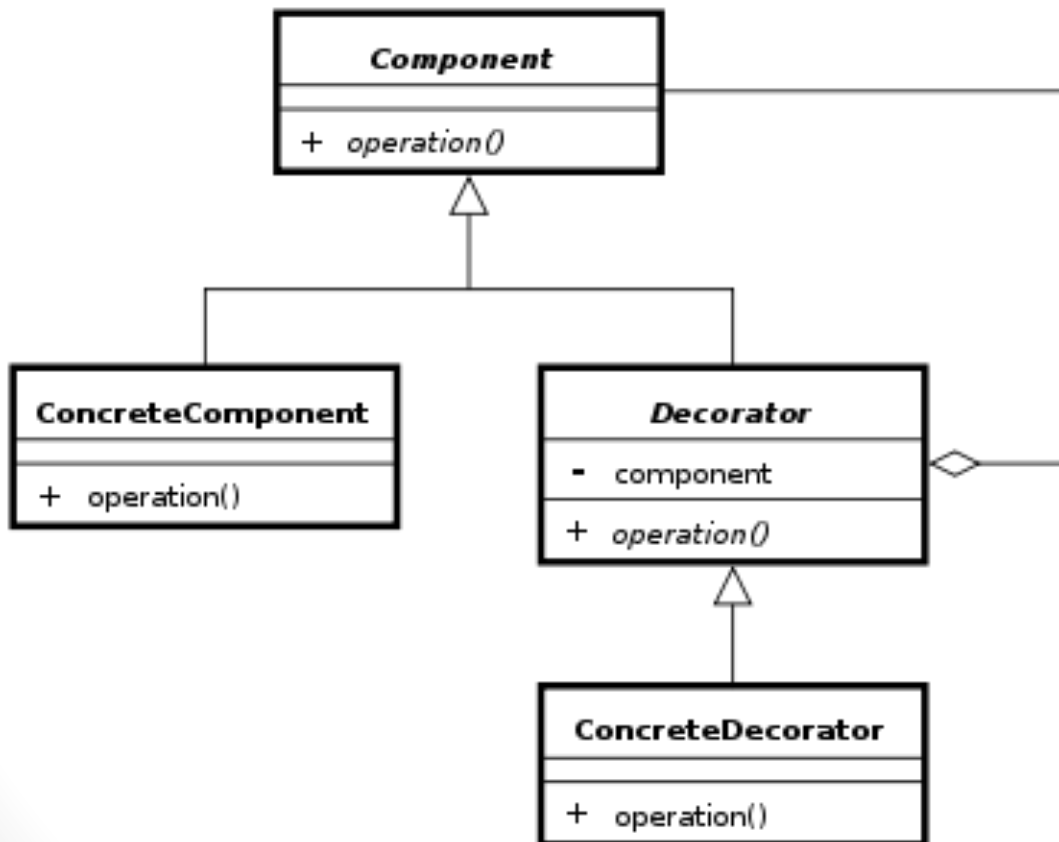
Decorator Pattern

- A structural design pattern
- Allows us to add some functionality to a class that we can't edit by adding additional steps to the process
 - Like decorating the method
- We delegate most functionality to the existing implementation
 - Now we can add in the extra steps to our implementation

Decorator Pattern

- Create a class `C` like you typically do
 - `C` may already exist
 - Better if you provide interface and abstract class as well
- Wrap it with a decorator `D`
 - `D` has only one private data member of type `C` (called `cObj`)
 - An object of type `C` is passed in through the constructor
- Decorate your methods
 - For all methods `m()` that exist in `C`
 - Add `m()` to `D` with the same arguments and return types
 - Method `D.m()` will call `cObj.m()`
 - Optionally add extra steps in that occur before or after the call to `cObj.m()`

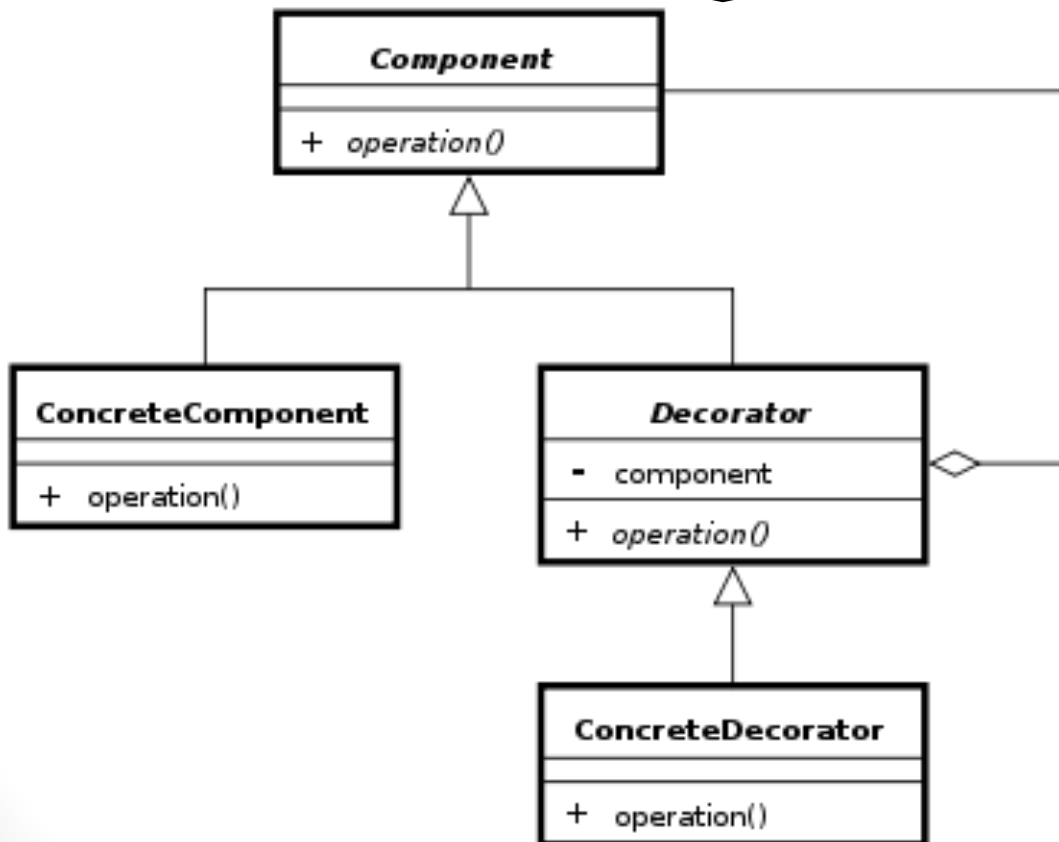
Decorator Pattern



1. Subclass the original Component into the Decorator class
2. In the Decorator class, add a pointer to a Component object as an attribute.
3. In the Decorator class, the constructor takes a Component object to set the pointer
4. In Decorator class, check input and forward call the method of the pointer
5. Use a ConcreteDecorator to override methods

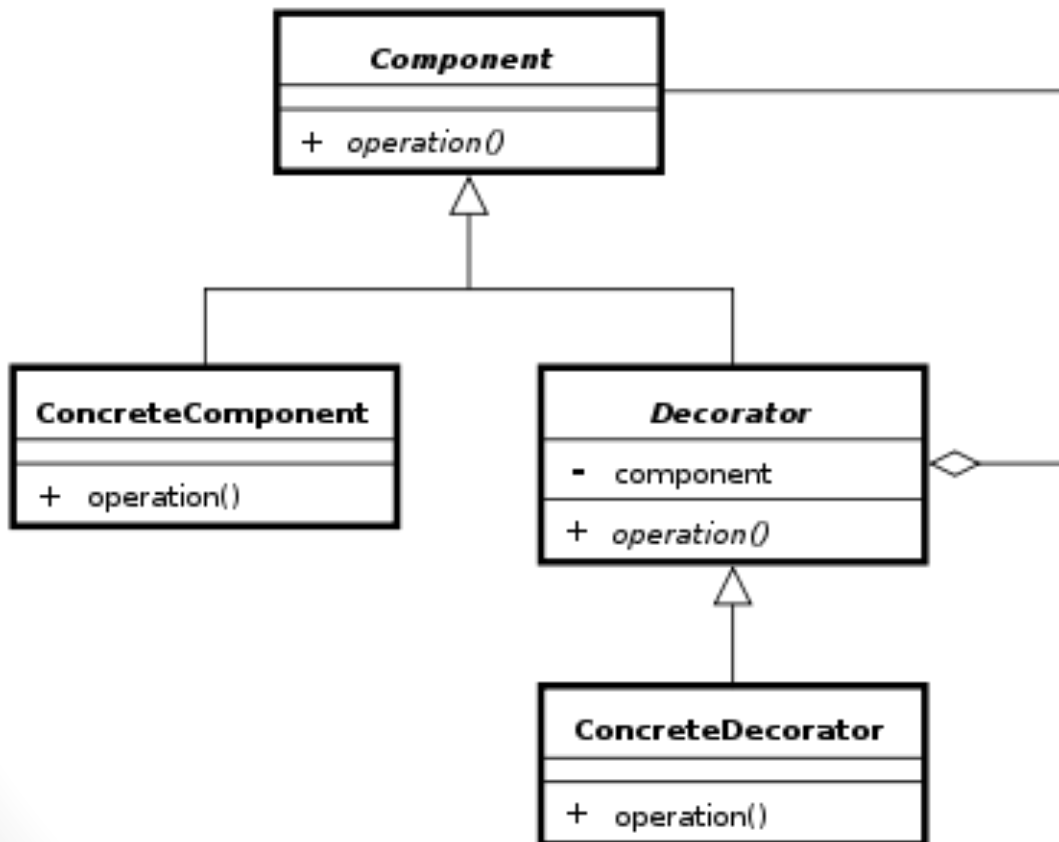
Decorator Pattern

Italics are used to show that something is *abstract*



1. Subclass the original Component into the Decorator class
2. In the Decorator class, add a pointer to a Component object as an attribute.
3. In the Decorator class, the constructor takes a Component object to set the pointer
4. In Decorator class, check input and forward call the method of the pointer
5. Use a ConcreteDecorator to override methods

Decorator Pattern



1. *Extend* the original *interface* into the *Decorator interface*
2. In the decorator class, add *a component interface object* as an attribute.
3. In the *Decorator* class, the constructor takes a *Component* object to set the attribute to
4. In *Decorator* class, check input and forward call to object

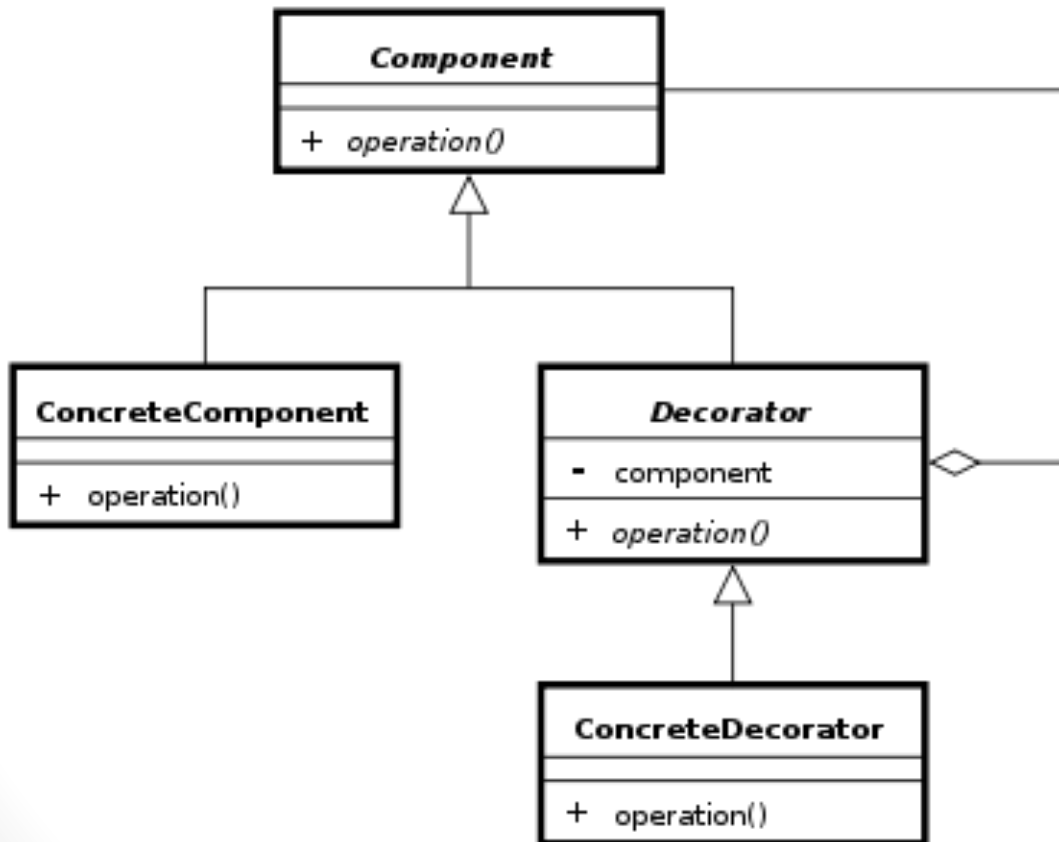
Abstract Class AbsC and C

```
public abstract class AbsC {  
    //some data?  
    public abstract int method1(int a);  
    public abstract bool method2();  
    public abstract void method3(int a, int b);  
}  
  
//another file  
public class C extends AbsC {  
    //some data  
    public C() {...};  
    public int method1(int a) {...};  
    public bool method2() {...};  
    public void method3(int a, int b) {...};  
}
```

Decorators

```
public abstract class AbsDec extends AbsC{  
    public abstract int method1(int a);  
    public abstract bool method2();  
    public abstract void method3(int a, int b);  
}  
//another file
```

Decorator Pattern



1. Subclass the original Component into the Decorator class
2. In the Decorator class, add a pointer to a Component object as an attribute.
3. In the Decorator class, the constructor takes a Component object to set the pointer
4. In Decorator class, check input and forward call the method of the pointer
5. Use a ConcreteDecorator to override methods

Decorators

```
public class Dec extends AbsDec {  
    private AbsC cObj;  
    public Dec(AbsC c){cObj = c;}  
    public int method1(int a) {  
        //extra steps?  
        int ret = cObj.method1(a);  
        //extra steps?  
        return ret;  
    }  
    public boolean method2() {  
        //extra steps?  
        boolean ret = cObj.method2();  
        //extra steps?  
        return ret;  
    }  
    ...  
}
```

In code that uses C and Dec

- Swap out all constructor calls to `new C ()` with `new Dec (new C ()) ;`

```
AbsC myVar = new C ();
```

Is replaced with...

```
AbsC myVar = new Dec (new C ()) ;
```

- Dec `extends` AbsDec which `extends` AbsC, so Dec `also extends` AbsC
- Note the use of **programming to the interface** here
- C constructor still called, but as an argument to the Dec constructor

In Java...

- Remember in Java we have interfaces available
 - `AbsC` would be replaced with `IC` – an interface that `C` would implement
 - `AbsDec` would be replaced with `IDec`, an interface that extends `IC` and would be implemented by `Dec`
- Everything else would be the same
- Remember to **program to the interface** to allow multiple implementations of `C` to work with `IDec` and `Dec`

Decorator Pattern

- We can use the same design pattern to decorate more than just individual methods
- Decorate the object itself by adding in new functionality

Decorator Pattern Motivation

- What if we want to extend an interface to add functionality?
 - Normally we can do so by adding a default method in the interface
 - But what if we can't edit the interface itself?

Decorator Pattern Motivation

- We want to add a new method called `Shuffle` to the Java `List<T>` interface
 - Would arrange the list in a random order
 - Can be coded as a secondary method
 - `get(int pos)` gets a value from position `pos`
 - `set(int pos, T val)` changes the value in `pos` to `val`
 - We could randomly pick two positions in the list to swap
 - Repeat this multiple times to get a lot of swaps in

Decorator Pattern Motivation

- Since our method is secondary we can just add it to the Java `List` interface, right?
- NO! Oracle won't let us edit their code!
- However, the decorator pattern helps us
 - We have the original interface (`IC`), and many implementations of it (`C`)
 - We just need to add in our extended interface (`IDec`) and our implementation (`Dec`)

Extended interface

```
public interface IShuffleList<T> extends List<T> {  
    //don't need to repeat the methods in List  
    //just add our new method  
    public default randomize() {  
        //code  
        ...  
    }  
}
```

Implementing our interface

- Now we need to implement our `IShuffleList` interface
 - Don't need to provide code for the `randomize` method
 - Its in `IShuffleList` as a default method
 - Do need to provide the code for all the methods inherited from the `List` interface
 - However, we can delegate back to the original interface
 - Decorate the `List` interface, but don't add any new functionality besides `randomize`

Implementing our interface

```
public class ShuffleList<T> implements IShuffleList<T> {  
    private List<T> myList; //wrapped object  
    public RandomList<T>(List<T> l) {  
        //can accept any list type  
        myList = l;  
    }  
    //decorate all the list methods  
    public void add(T val) {  
        myList.add(val);  
    }  
    public T get(int pos) {  
        return myList.get(pos);  
    }  
    ...  
}
```

Shuffle List

- Now we have added the ability to randomize our List
 - All the hard code delegated to the `List` interface
 - Our `ShuffleList` constructor can take in any type the implements the `List` interface
 - `ArrayList`, `LinkedList`, `Vector`, etc.
 - We **programmed to the interface**