Decorator

Decorate objects in runtime!
Cool?

What is the problem?

If we need a hierarchy of classes where differences can be combined, we'll have a cartesian mess of classes:

```
main type:
Collection
specialized types:
CheckedCollection SynchronizedCollection UnmodifiableCollection
CheckedSynchronizedCollection CheckedUnmodifiableCollection
SynchronizedUnmodifiableCollection
```

How many more classes will there be if we add one more specialization?
Hint: n!

The problem seems to be combining treats

Think about combining the various I/O-streams classes in Java's API!

A "Buffered" InputStream Provides buffering. So we need:

BufferedAudioInputStream, BufferedByteArrayInputStream, BufferedFileInputStream, BufferedFilterInputStream, BufferedObjectInputStream, BufferedPipedInputStream, BufferedSequenceInputStream, BufferedStringBufferInputStream....?

Instead, we decorate an object with treats

Decorating is done using “wrapping” the object inside another object.

```
List<Character> myList = getList(); //get the list from some method
List<Character> unmodifiableList = Collections
                                        .unmodifiableList(myList);
```

The unmodifiable list has a list “inside it”

You want an inputstream to be buffered

You want a file input stream, and you want it to be buffered!

```
InputStream input =  new BufferedInputStream
                ( new FileInputStream("/home/file.txt") );
```

The FileInputStream now is decorated with buffer-capabilities.

The BufferedInputStream “has a FileInputStream”.

All method calls on the BufferedInputStreams are forwarded to this internal FileInputStream.

Intent of the Decorator pattern

You have an interface for a component (for instance WeaponBehavior).

You want to extend (decorate) the component in runtime but not use inheritance (because of the many classes needed for all combinations).

The decorators wrap the component. We'll see how soon!

How to do the wrapping

- Implement the component in a decorator class
- The decorator class should have an instance variable of type component (for instance a WeaponBehavior)
- Add a constructor which takes a WeaponBehavior and set the instance variable
- Override all methods in the product (WeaponBehavior) but forward all calls to the instance variable

Pause...

WeaponDecorator example

We decide that in our game, there shall be some magic. One spell makes a weapon usable twice as fast! So one call to useWeapon() uses it twice (and doubles the damage).

Another spell makes the weapon double its damage (but it will be used only once as usual).

The spells can of course be combined. So a weapon can get both spells (so it will be used twice and then double its damage).

A sword with both spells would cut twice (15+15) and then double that - and get damage 60 in total. Not bad for a sword.

What we want to avoid

Sword

DoubleInjurySword DoubleSpeedSword

DoubleInjuryDoubleSpeedSword

(and the same for all other weapons)

Solution: Use composition instead of inheritance or class explosion!

Solution - we'll make a WeaponDecorator

```
public abstract class WeaponDecorator implements WeaponBehavior{
  private final WeaponBehavior decoratedWeapon;
  public WeaponDecorator(WeaponBehavior weapon){
    this.decoratedWeapon = weapon;
  }
  @Override
  public int useWeapon(){
    // Forward the call to the decorated weapon
    return decoratedWeapon.useWeapon();
  }
  @Override
  public String toString(){   /// We want to replicate also toString()
    return decoratedWeapon.toString();
  }
}
```

Double Injury decorator

```
public class WithDoubleInjury extends WeaponDecorator{
  public WithDoubleInjury(WeaponBehavior weapon){
    super(weapon);
  }
  @Override
  public int useWeapon(){
    System.out.println("Using " + this + " with double injury");
    return 2 * super.useWeapon();
  }
  @Override
  public String toString(){
    return super.toString();
  }
}
```

Double Speed decorator

```
public class WithDoubleSpeed extends WeaponDecorator{
  public WithDoubleSpeed(WeaponBehavior weapon){
    super(weapon);
  }
  @Override
  public int useWeapon(){
    System.out.println("Using " + this +
                       " with double speed - two times!");
    return super.useWeapon() +  super.useWeapon();
  }
  @Override
  public String toString(){
    return super.toString();
  }
}
```

Client code

```
WeaponBehavior sword = new Sword();
System.out.println("Player gets a double injury spell!");
sword = new WithDoubleInjury(sword);
damage=sword.useWeapon();

// result (normal damage is 15):
Player gets a double injury spell!
Using Excalibur with double injury
The sword shines and cuts through the air before it hits its target
Damage from Excalibur 30
```

Client code - Double Speed spell

```
System.out.println("Player gets a double speed spell!");
sword = new WithDoubleSpeed(sword);
damage=sword.useWeapon();

// Result: (normal damage is 15 - only strikes once)
Player gets a double speed spell!
Using Excalibur with double speed - two times!
The sword shines and cuts through the air before it hits its target
The sword shines and cuts through the air before it hits its target
Damage from Excalibur 30
```

Client code - double spell

```
System.out.println("Whoa! Double spell! Speed and injury!");
sword = new WithDoubleSpeed(new WithDoubleInjury(sword));
damage=sword.useWeapon();

// Result - normal injury is 15, only strikes once
Whoa! Double spell! Speed and injury!
Using Excalibur with double speed - two times!
Using Excalibur with double injury
The sword shines and cuts through the air before it hits its target
Using Excalibur with double injury
The sword shines and cuts through the air before it hits its target
Damage from Excalibur 60
```

Lessons learned

Rather than creating a lot of combined treats classes, we made a decorator which takes a WeaponBehavior as the only argument to the construcor.

It forwards all method calls to the wrapped WeaponBehavior

This allows us to combine two decorators:

```
// sword is a normal Sword
sword = new WithDoubleSpeed(new WithDoubleInjury(sword));

// The sword is decorated in runtime!
```