Patron: Decorator

http://www-public.it-sudparis.eu/~gibson/Teaching/DesignPatterns/DesignPatterns-Decorator.ppt
http://www-public.it-sudparis.eu/~gibson/Teaching/DesignPatterns/DesignPatterns-Decorator.pdf

Patron: Decorator

Ajout dynamique de responsabilités à un objet.

L'ajout de responsabilités à un objet sonne évidemment comme un héritage. C'est en effet, le moyen le plus évident de lui fournir des responsabilités supplémentaires.

Le manque de souplesse de cette méthode (l'héritage) est bien entendu, son inconvénient majeur.

> Premièrement, l'héritage n'est pas toujours possible.
> Deuxièmement, l'héritage peut donner lieu à la prolifération de classes.

Enfin, les responsabilités supplémentaires ne sont pas dynamiques.

Patron: Decorator

L'héritage n'est pas toujours possible/desirable

Vous n'avez en effet pas toujours moyen d'accéder à la classe mère, vous ne l'avez pas nécessairement développée vous-même et elle peut être définie comme étant finale.

Il peut aussi arriver que la classe ne soit pas finale mais qu'en hériter vous rende nerveux à l'idée de devoir maintenir dans toutes vos classes héritières toute modification apportée par l'éditeur de cette classe.

Bref que vous ne puissiez réellement pas ou que vous ne désiriez pas, ne change rien à votre problème : il faut éviter d'hériter

Patron: Decorator

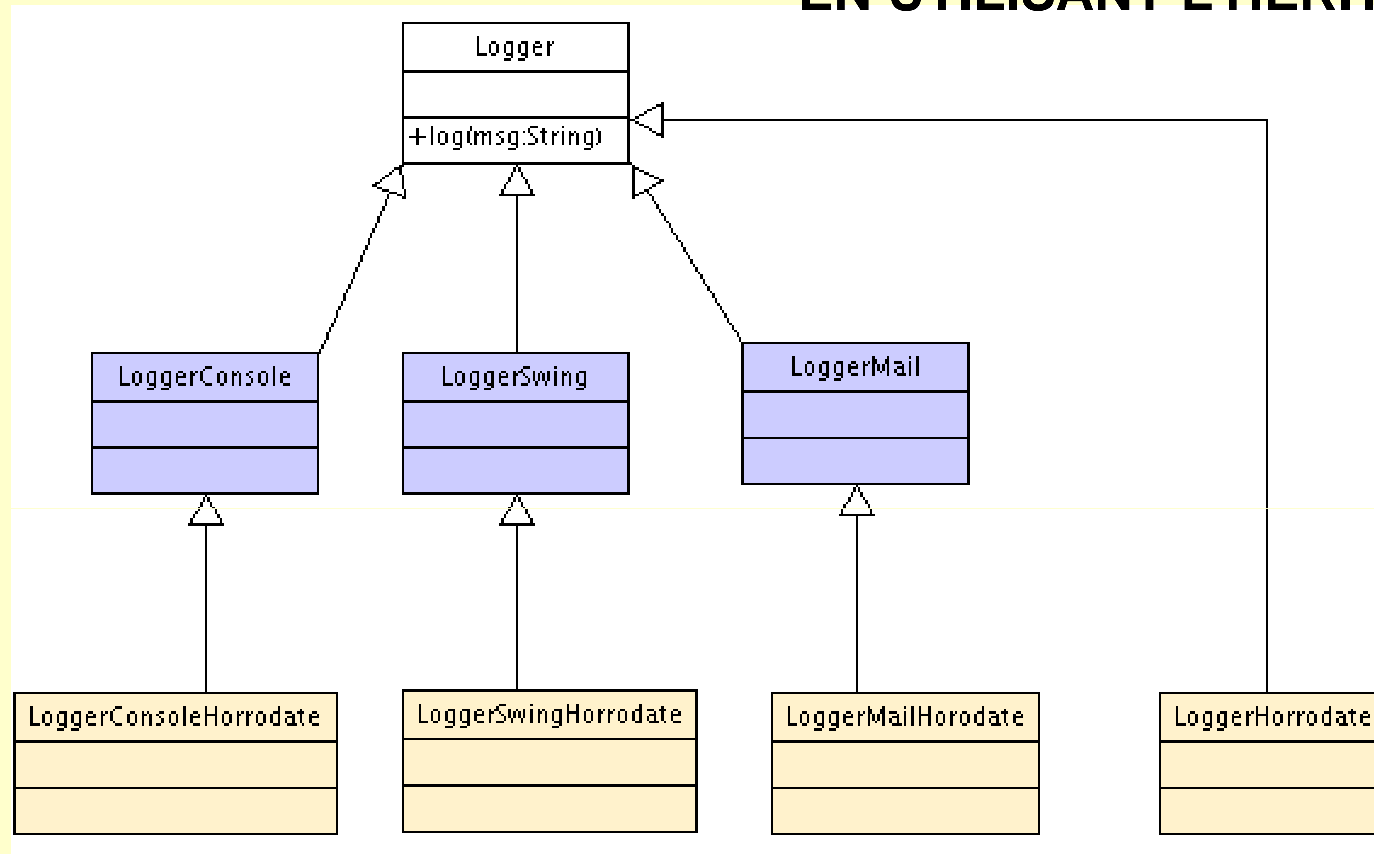
Evite la prolifération de classes :

Imaginons que l'on ai définie une classe dont on veut étendre les capacités.

Par exemple nous avons la classe *Logger* et on veut ajouter à chaque message envoyer vers le logger, l'heure et la date d'émission. Imaginons de plus que nous voulions aussi définir une classe pour un affichage console, un log dans un fichier, un log par email à un administrateur un affichage SWING ! Dans chaque cas, il faut pouvoir afficher ou non l'heure et la date. Nous voilà donc obligés de surcharger la classe logger (pour un fichier) 3 fois pour les divers types de logger et 2 fois pour que chacun affiche l'heure et la date, ou non.

Patron: Decorator (UML)

PROBLEME: CONCEPTION EN UTILISANT L'HERITAGE



Patron: Decorator (UML)

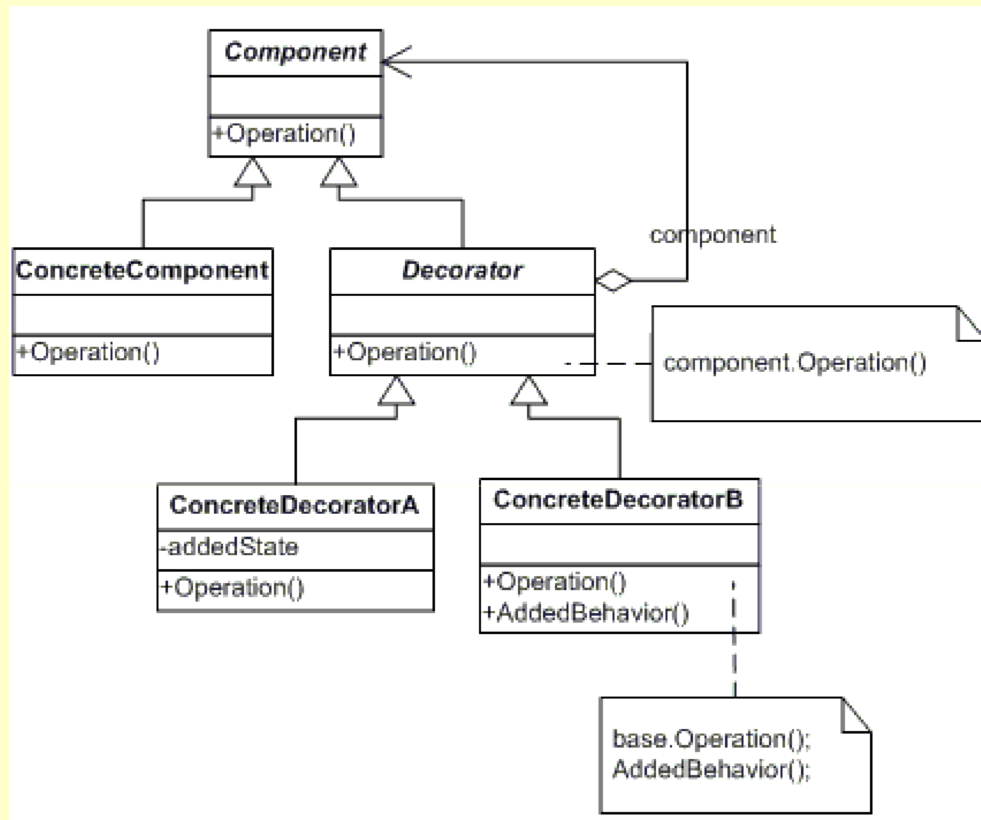
Les responsabilités supplémentaires – avec l'heritage - ne sont pas dynamiques

Et le dynamisme peut se représenter par plusieurs choses aussi utiles qu'esthétiques (si tant est que le développement vous paraisse esthétique). Plusieurs points donc :

Un dynamisme en runtime : à tout moment vous pouvez basculer du mode horodaté au mode sans horodatage, du mode swing au mode console, etc.

Enfin, et on découvre alors là une grande puissance du pattern Décorateur, vous pouvez composer les différentes parties entre elles. Un exemple : Logger classique (donc fichier) associé à LoggerHorodaté (donc horodatage des messages) associés à LoggerSwing (donc affichage écran graphique) donne des résultats dans un fichier et dans un écran, le tout horodaté. Pour faire cela avec l'héritage pur, il m'aurait fallu hériter par exemple de LoggerSwingHorodaté et ajouter les responsabilités pourtant déjà implémentées de log dans un fichier ! Soit encore une classe.

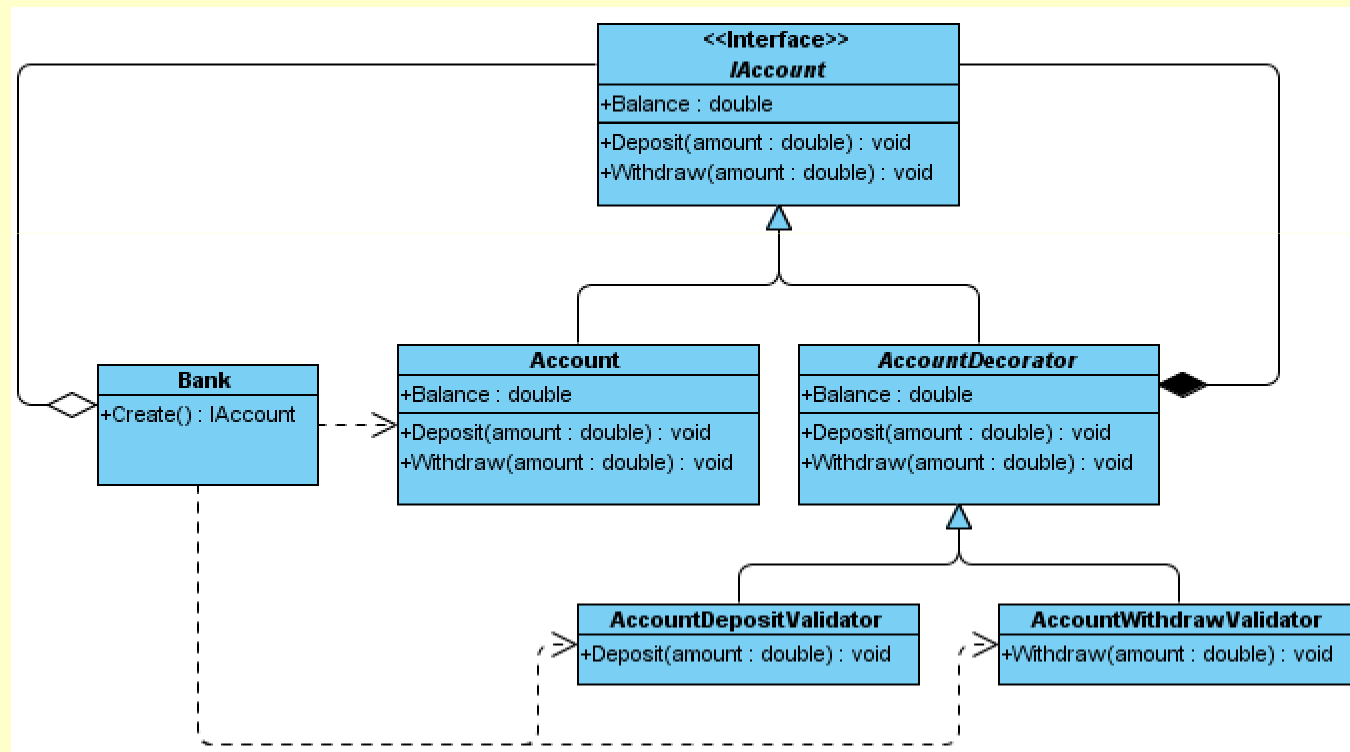
Patron: Decorator (UML generique)



Patron: Decorator (UML – exemple Account)

Class Diagram UML

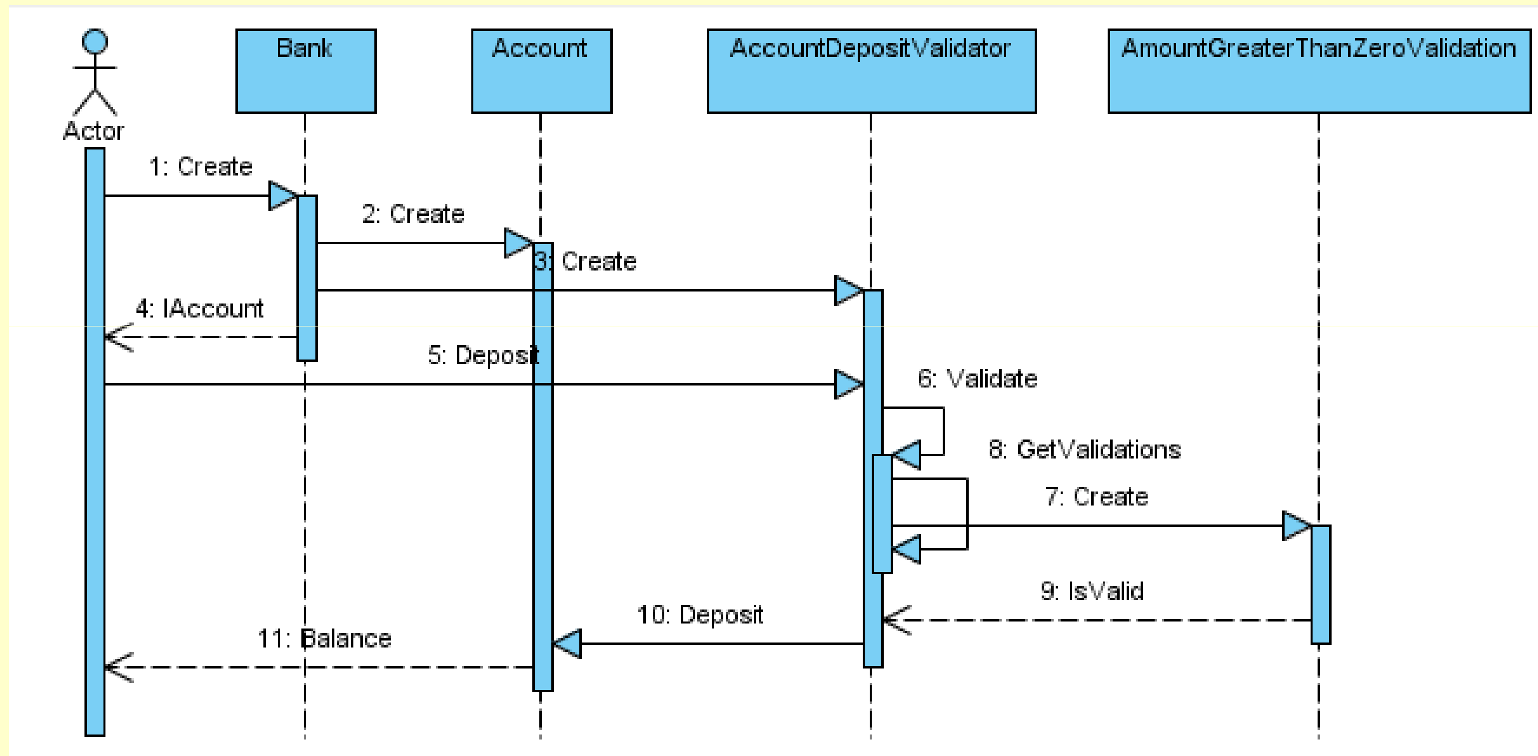
The following diagram shows a concrete example of an Account (from http://blog.decarufel.net/2009/09/using-decorator-or-wrapper-design.html)



Decorator Pattern

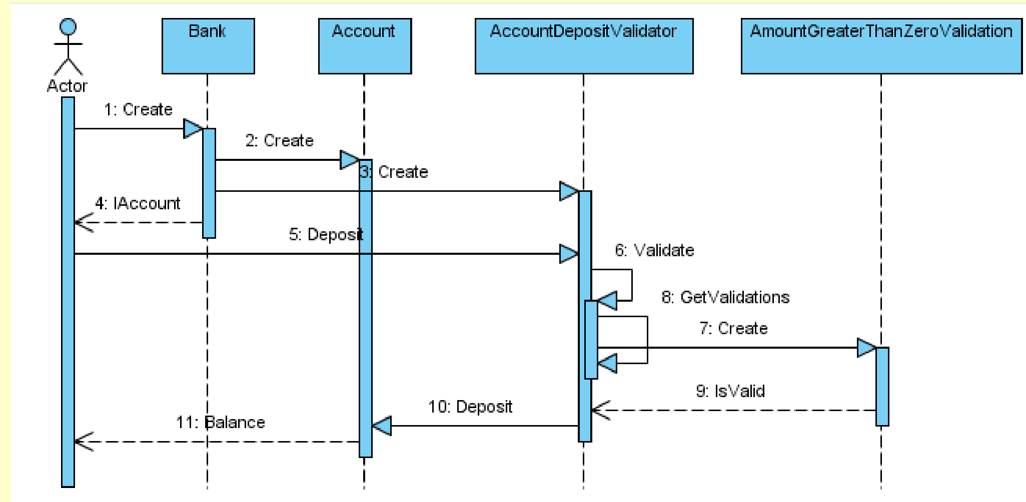
Sequence Diagram UML

The following diagram shows a concrete example of Account dynamic behaviour



Decorator Pattern

Sequence Diagram UML: explanation



1. Call Bank.CreateAccount.
2. The Bank instantiates an Account class.
3. The Bank creates an AccountDepositValidator and wrap Account with it
4. The Bank return an instance of IAccount.
5. Deposit is called on IAccount which is an instance of AccountDepositValidator
6. AccountDepositValidator calls Validate
7. AccountDepositValidator call GetValidations to retrieve the list of validation to evaluate
8. An AmountGreaterThaZeroValidation is created
9. IsValid returns true
10. AccountDepositValidator call base Deposit method
11. Balance value is updated

Patron: Decorator TP – Christmas Tree

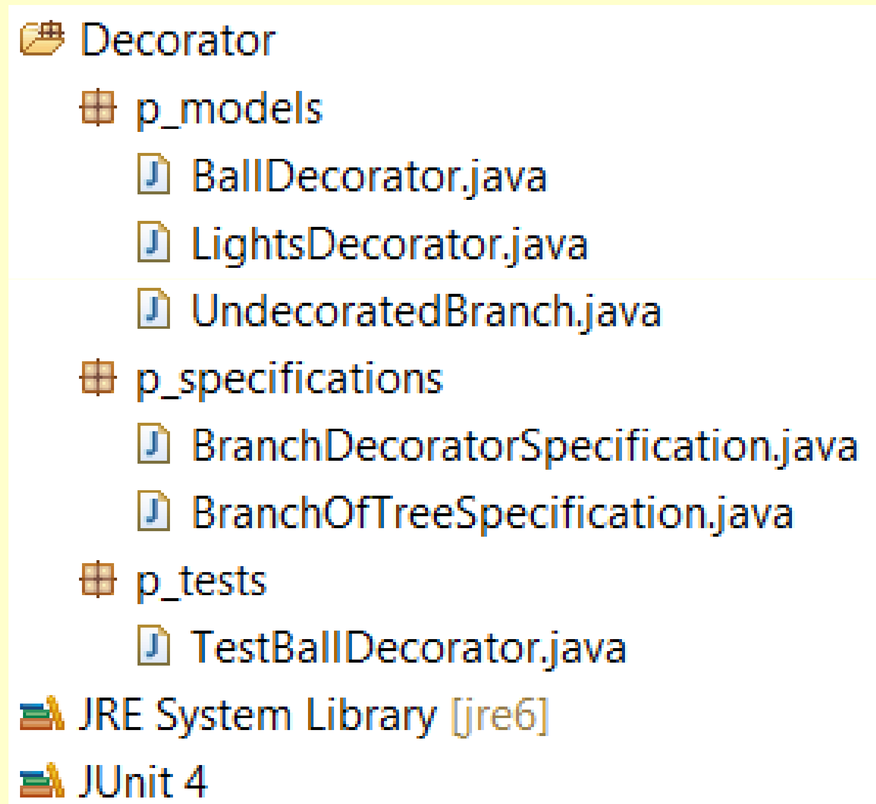
There is a need to decorate a Christmas tree.

Now we have many branches which need to be decorated in different ways.

Download source code from:
DecoratorSource.zip

Decorator Problem : The classic Christmas tree (revisited)

Problem: Examine the Java code in source package (http://www-public.int-evry.fr/~gibson/Teaching/DesignPatterns/DecoratorSource.zip)



Patron: Decorator TP – Christmas Tree

```
package p_specifications;

public interface BranchOfTreeSpecification
{

public void animate();
public String getDecorations();

}
```

Patron: Decorator TP – Christmas Tree

```
package p_specifications;


public abstract class BranchDecoratorSpecification implements
BranchOfTreeSpecification{

protected BranchOfTreeSpecification decoratedBranch;

public BranchDecoratorSpecification(BranchOfTreeSpecification
branchToDecorate){
decoratedBranch= branchToDecorate;
}

public void animate(){decoratedBranch.animate(); }

}
```

Patron: Decorator TP – Christmas Tree

```
package p_models;
import p_specifications.BranchDecoratorSpecification;
import p_specifications.BranchOfTreeSpecification;

public class BallDecorator extends BranchDecoratorSpecification{

boolean spinning;

       public BallDecorator(BranchOfTreeSpecification branchOfTree){
         super(branchOfTree);spinning = true;
        }

        public String getDecorations(){
        String str = decoratedBranch.getDecorations();
        if (spinning) str = str +" Spinning";
        str = str+ " Ball.";
        return str;
        }

        public void animate(){
        decoratedBranch.animate();
        spinning = !spinning;
        }
}
```

Patron: Decorator TP – Christmas Tree

```
package p_tests;

import p_models.BallDecorator;
import p_models.UndecoratedBranch;
import p_specifications.BranchOfTreeSpecification;

public class TestBallDecorator {


public static void main (String [] args){  }

}
```

Patron: Decorator TP – Christmas Tree

```
public static void main (String [] args){

BranchOfTreeSpecification plainBranch = new UndecoratedBranch();
BranchOfTreeSpecification decoratedBranch = new BallDecorator( plainBranch);
BranchOfTreeSpecification reDecoratedBranch = new BallDecorator( new BallDecorator(
plainBranch));

System.out.println("plainBranch.getDecorations() ="+plainBranch.getDecorations());
System.out.println("decoratedBranch.getDecorations() ="+decoratedBranch.getDecorations());
System.out.println("reDecoratedBranch.getDecorations() ="
            +reDecoratedBranch.getDecorations());

plainBranch.animate();
System.out.println("plainBranch.animate.getDecorations() ="+plainBranch.getDecorations());

decoratedBranch.animate();
System.out.println("decoratedBranch.animate.getDecorations()
="+decoratedBranch.getDecorations());

reDecoratedBranch.animate();
System.out.println("reDecoratedBranch.animate.getDecorations() ="
            +reDecoratedBranch.getDecorations());
}
```

Patron: Decorator TP – Christmas Tree

Compile and Execute the TestBallDecorator class.

Verify the output:

```
plainBranch.getDecorations() =
decoratedBranch.getDecorations() = Spinning Ball.
reDecoratedBranch.getDecorations() = Spinning Ball. Spinning Ball.
plainBranch.animate.getDecorations() =
decoratedBranch.animate.getDecorations() = Ball.
reDecoratedBranch.animate.getDecorations() = Ball. Ball.
```

```
public class LightsDecorator extends BranchDecoratorSpecification{

String colour;

      public LightsDecorator(BranchOfTreeSpecification branchOfTree){

       super(branchOfTree); colour =  "red";
       }

        public String getDecorations(){

        String str = decoratedBranch.getDecorations();
        str =str+ " colour = "+colour;
        return str;
        }

        public void animate(){

        decoratedBranch.animate();
        if (colour.equals("red")) colour ="white";
        else if (colour.equals("white")) colour ="blue";
        else colour = "red";
        }
}
```

QUESTION: What is the behaviour of the light decorator?

TO DO: Write a test class for the light decorator that illustrates its complete behaviour

Patron: Decorator TP – Christmas Tree

TO DO:

- Add new Decoration (Stars, eg)
- Test 3 Decorations together

Create a new XMAS tree with 4 branches.

As a test, create a XMAS tree object which is decorated as follows:

1. 2 Lights and 2 Balls
2. 1 Light and 2 Stars
3. 1 Star and 3 Balls
4. 1 Lights and 2 Star and 2 Balls

PROBLEM: Ensure that the state of decorations on the same branch of tree is co-ordinated (but not necessarily on different branches)

OPTION: Maximum number of decorations on any branch is 4

Decorator– Christmas Tree

What to learn from the problem:

If you really need multiple instances of same decoration then this pattern is probably not correct.

The following 4 sentences seem equivalent but may lead to different designs/implementations:

The branch « is a » branch with lights and stars
The branch « is a » branch with lights and « is a » branch with stars

The branch « has » lights and stars
The branch « has » lights and the branch « has » stars

Knowing which design is best depends on knowing precisely what is the intended/required behaviour.

The decorator is not intended to allow statements like:
The branch « is a » branch with lights and « is a » branch with lights