

Labels

Atanas Semerdzhiev, Kalin Georgiev, Petar Armyanov, Trifon Trifonov

Edition 0.1, 2024-11-04: Still a work in progress

Table of Contents

Introduction	1
Description	2
Tasks	4
Step 1: Label hierarchy	4
Step 2: Text transformations	4
Step 3: Decorators	4
Step 4: Option to remove decorators	5
Step 5: Applying multiple transformations	5
Step 6: Custom label	5
Step 7: Label with help text	5
Step 8: Dependency injection	6
Step 9: Frequently censored labels	6
Hints	7
Step 2	7
Step 3	7
Skills and Competencies	11

Introduction

In this exercise you will practice with multiple structural design patterns. They will be applied in very similar scenarios. This will help you better understand how each of them can be applied to the same task and will get a better understanding of their specifics, strengths and weaknesses.

!

All examples in this text are in Java, but can easily be translated to other languages. We are leaving this task to the reader.

Description

Suppose that some application uses the class below to represent labels:

```
interface Label {
    public String getText();
}

class SimpleLabel implements Label {
    public String value;

    public SimpleLabel(String value) {
        this.value = value;
    }

    public String getText() {
        return value;
    }
}
```

Labels will be assigned to different entities in the application to denote their names, store a brief comment, description, etc.

Assume that there is already some code in place, which relies on the `Label` interface. For example, the following simple class:

```
class LabelPrinter {
    public static print(Label label) {
        System.out.println("Here is a label: " +
            label.getText());
    }
}
```

We want to add the necessary functionality, so that for each label we can specify how it should appear to the end-user. The programmer should be able to request that zero, one or more of the following styles apply (the `!` symbol denotes a space character in the string):

Capitalize

If the first character of the label is a letter, then it is capitalized, e.g. `"some! text"` becomes `"Some! text"`.

LeftTrim

Whitespace is trimmed at the beginning of the label, e.g.

"! some! text" becomes "some! text".

RightTrim

Whitespace is trimmed at the end of the label, e.g.

"some! text! " becomes "some! text".

NormalizeSpace

Labels never contain more than one consecutive space, e.g.

"some! ! ! text" becomes "some! text".

Decorate

Labels are bracketed with "-={" and "}-=", e.g.

"abc" becomes "-={! abc! }=-"

Censor(W)

Censors a specific string W containing L characters by replacing all of its occurrences in the label with L asterisks, e.g.

if W="abc", then "! abc! def! abcdef" becomes "! ***! def! ***def"

Replace(A,B)

Replaces (case-sensitive) all occurrences of the string A with the string B, e.g.

if A="abc" and B="d", then "! abc! abcdef" becomes "! d_ddef"

Your solution must not break the existing code, which relies on the `Label` interface.

Tasks

Proceed through the tasks in the order they are specified below. Do not proceed to a later task, before you have successfully completed the previous one and have covered it with unit tests.

Step 1: Label hierarchy

Implement at least one more label type! `RichLabel`, which stores not only text, but also the color of the label and the size and name of the font to be used for that label.

Step 2: Text transformations

Start with an interface like this:

```
class TextTransformation {  
    public String transform(String text);  
}
```

Derive a class for each text transformation (e.g. `CapitalizeTransformation`, `TrimLeftTransformation`, etc.).

Cover them with unit tests to ensure they all work correctly.

Step 3: Decorators

Use the `Decorator` pattern and implement a solution which allows the user to apply a random combination of zero, one or more of the text transformations to a label.

On this step we assume that this will be done exactly once in the lifetime of a label object! when it is being created. After that the chosen styles remain in effect for the label and cannot be altered.

Instead of creating a separate decorator for each text transformation, make use of the `Strategy` pattern. Create a simple decorator class that receives a transformation that should be applied to the underlying label. It should be possible to stack multiple decorators over the same label object.

Create a second decorator, which receives a list of transformations and applies a random one of them each time when invoked.

Optionally, create a third decorator, which receives a list of transformations and applies a different transformation each time when invoked, starting from the first one, then the second one, and so on, reverting back to the first one after reaching the last one.

Step 4: Option to remove decorators

Modify your solution so that decorators can be added or removed from a label at any point during its lifetime.

Step 5: Applying multiple transformations

Use the [Composite](#) and [Strategy](#) patterns to achieve the same functionality as applying multiple simple decorators.

First, implement a `CompositeTransformation` class, which allows the user to specify a sequence of zero, one or more text transformations. The composite applies all transformations in the sequence, one by one, exactly in the order they were specified. This class will allow for a sequence of transformations to be treated as a single transformation.

!

The order of operations matters.

If the sequence of styles is `Capitalize`, `Decorate` and `Replace(abc, def)`, then `"abc! def"` becomes `"--{! Abc! def! }=-"`.

However, if we rearrange them to `Replace(abc, def)`, `Capitalize` and `Decorate`, then `"abc! def"` becomes `"--{! Def! def! }=-"`.

Step 6: Custom label

Use the [Proxy](#) pattern to implement a new type of label, whose text is read from the standard input when its text is requested for the first time. After the text is read, it is stored and automatically returned on each subsequent request for the label.

Add a "timeout" option, where after `timeout` requests for the label's text, the user is given the option to change the label text or keep using the current one.

Step 7: Label with help text

In this step we will implement a new kind of label, which extends the ordinary functionality with support for additional help text. The label will store a second text field, which may be invoked upon request, for example:

```
class LabelPrinter {
    public static print(Label label) {
        System.out.println("Here is a label: " +
            label.getText());
    }
    public static printWithHelpText(HelperLabel label) {
        print(label);
    }
}
```

```
        System.out.println("Some help information about this  
Label : " + Label.getHelpText());  
    }  
}
```

Use the [Bridge](#) pattern to enable the help text functionality for the other types of labels you have implemented in Step 2 (e.g. [RichTextLabel](#)) and Step 6 (custom label).

Step 8: Dependency injection

Use [dependency injection](#) to allow the user to interactively create labels with different combinations of features: with or without help text, simple, rich text, custom label, decorators, and transformations.

Step 9: Frequently censored labels

Use the [Flyweight](#) pattern to allow reuse of frequently used Censor transformations.

Create a [CensorTransformationFactory](#), which allows for creation of Censor transformations. If the length of the censored word is four characters or less, a flyweight object is stored by the factory and reused each time it is requested. If the censored word is longer, then a new object is created each time when requested from the factory.

Hints

Step 2

For step 2, implement the Decorator pattern in its classic form, as described in the GoF book.

Create a class `LabelDecoratorBase`, which implements the `Label` interface. This class will serve as a parent for all decorators.

Create a `TextTransformationDecorator` class that inherits from `LabelDecoratorBase`. Use that to implement the actual transformation of text applied to a label.

Create a `RandomTransformationDecorator` class that inherits from `LabelDecoratorBase` and stores a list of transformations. Think about when to select a random transformation, so that it is different each time when the decorator is invoked.

Create a `CyclingTransformationsDecorator` class that inherits from `LabelDecoratorBase` to implement the behavior of applying a different transformation when invoked. Think about what state the decorator will need to have, and when and how it should change.

When writing unit tests, do not test only for a single decorator, but also check what happens if multiple decorators are chained. For example, check if all of them are applied properly.

Step 3

For step 3, if you have already solved step 2, you will already have the necessary functionality to be able to add styles at a later point in time. You just need to add another decorator on top of the existing ones.

To add an option to remove an already existing decorator, consider modifying `LabelDecoratorBase`.

A simple approach would be to add two functions (example is for Java, but can be extended to other languages):

```
public static Label removeDecoratorFrom(Label label, Class decoratorType);  
public Label removeDecorator(Class decoratorType);
```

`removeDecorator` receives the type of decorator to remove and then proceeds to remove it from a series of chained decorators.

`removeDecoratorFrom` can be applied to an arbitrary `Label` variable and if it refers to

an actual decorator, proceeds to remove `decoratorType`. It may look something like

```
public static Label removeDecoratorFrom(Label label, Class <?
extends LabelDecoratorBase> decoratorType) {
    if(label == null) {
        // Nothing to do
    }

    else if(LabelDecoratorBase.class.isAssignableFrom(label))
    {
        // label refers to a decorator. Proceed to remove.
        LabelDecoratorBase ldb = (LabelDecoratorBase)label;
        return ldb.removeDecorator(decoratorType);
    }

    else {
        // Label is not a decorator, but an actual label.
        // Nothing to do in this case
    }
}
```

`removeDecorator` recursively iterates the linked list of decorators. If it finds the requested type in the list, it removes its first occurrence. Otherwise, the list remains unaltered. The implementation may look like this:

```
class LabelDecoratorBase {
    Label subject; // Reference to the decorated object.
    Cannot be null.

    LabelDecoratorBase(Label subject) {
        this.subject = subject;
    }

    public static Label removeDecorator(Class <? extends
LabelDecoratorBase> decoratorType) {

        if(getClass() == decoratorType) {
            // This is the decorator to remove
            return subject;
        }
        else if( LabelDecoratorBase.class.
isAssignableFrom(subject.getClass()) ) {
            // The subject is itself a decorator.
            // Try to remove decoratorType from it.
```

```

    // Note that we may need to reassign subject,
    because the
    // requested decorator may be on top of the list.
    subject =
    ((LabelDecoratorBase)subject).removeDecorator(decoratorType);

    // Return the current object to keep it on top of
    the list
    return this;
}
else {
    // The subject is not a decorator.
    // We have reached the end of the list and have
    not found decoratorType.
    // We can either throw an exception here,
    // or do nothing and simply return the current
    object.
    return this;
}

}

// ... Other stuff goes here
}

```

Note that both functions return `Label` instead of `LabelDecoratorBase`. This is, because, there may be just a single decorator applied to a label. When we remove it, all that remains will be the label itself. This also allows us to call `removeDecoratorFrom` on any object that implements `Label`, without concerning ourselves whether it is a decorator or not.

!

The example above assumes that we are looking for a decorator solely based on its type. However, it may be that multiple instances of the same decorator type are present in the list. For example, we may have two instances of the `Replace` decorator, for different words.

In this case we need to distinguish between the properties of the objects themselves. Thus, it may be better to define an `equals()` (or similar) function for the decorators and use it to identify the one to remove. Correspondingly, instead of passing the type's information as a `Class` object, you may pass a specific decorator object, set up like the one you want to remove.

For example, assume we have a decorator, which censors a specific word in the label. It replaces all of its characters with dots. We may have applied different instances of that decorator to censor multiple

words. A call to `removeDecorator` may look something like this:

```
Label l;  
l = new Label("abcd efgh ijkl mnop");  
l = new CensorWordDecorator(l, "abcd");  
l = new CensorWordDecorator(l, "mnop");  
l.getText(); // returns "***** efgh ijkl *****"  
LabelDecoratorBase whatToRemove = new  
CensorWordDecorator(null, "abcd");  
l = LabelDecoratorBase.removeDecoratorFrom(l,  
whatToRemove);  
l.getText(); // returns "***** efgh ijkl *****";
```

Skills and Competencies

Design Patterns

- ¥ [Bridge](#),
- ¥ [Composite](#),
- ¥ [Decorator](#),
- ¥ [Dependency injection](#),
- ¥ [Proxy](#),
- ¥ [Strategy](#),
- ¥ [Template Method](#)