Decorator

By Alexander Tilkin

Motivation

- We want to augment an object with additional functionality
- We don't want to rewrite or alter code (OCP)
- We want to keep new functionality separate (SRP)
- We need to be able to interact with existing structures
- Two options:
    - Inherit from required object if possible; some classes are final
    - Build decorator, which simply references the decorated object(s)

Decorator

Facilitates the addition of behaviors to individual objects without inheriting from them.

units.unit_09_decorator.DecoratedStringExample.java

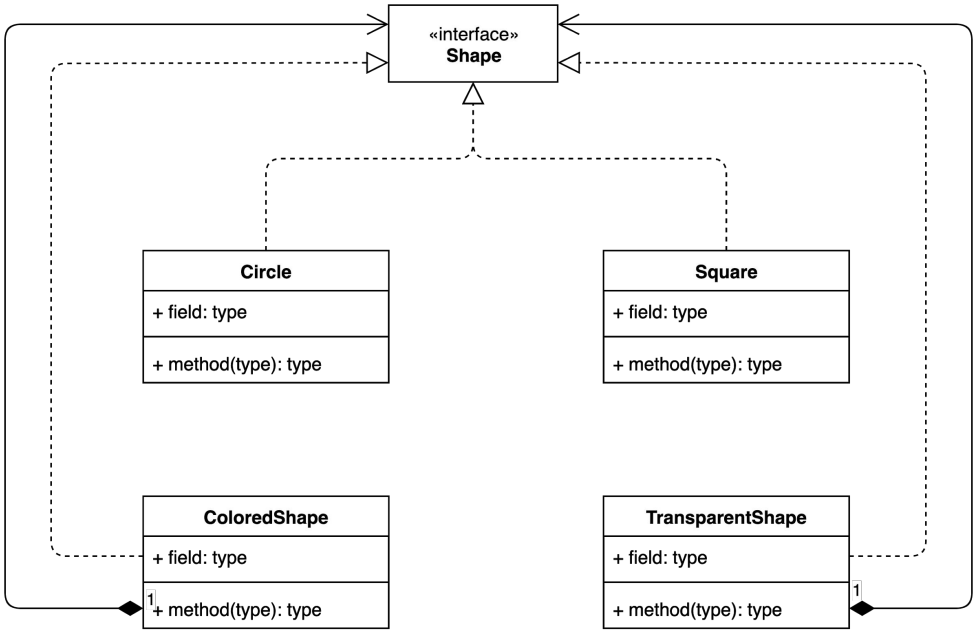
Decorated String

- String is a final class, meaning, we can't extend it
- The only way to manipulate this class is by containing it, delegating methods and adding additional methods which we desire

units.unit_08_composite.DynamicDecoratorComposition.java

Dynamic Decorator Composition

- The idea in dynamic decorator composition is allowing you to expand the characteristics and attributes of a certain class without modifying the original class. Recall the "Open-Closed Principle"



units.unit_08_composite.StaticDecoratorComposition.java

Static Decorator Composition

- In this example we demonstrate a decorator that uses a static definition
- We apply the static definition by using generics
- The motivation to use static decorator is to enforce the the object to be of a specific type without letting it change

Adapter-Decorator

Decorator Coding Exercise

The code in the path above shows a Dragon which is both a Bird and a Lizard.

Complete the Dragon's intraface (there's no need to extract any interfaces!). Take special care when implementing setAge()

Summary

- A decorator keeps the reference to the decorated object(s)
- May or may not forward calls; IDE can Generate Delegated members
- Exists in a static variation
  - X<T<Foo>>(/* unpleasant constructor args */)
  - Very limited due to type erasure; inability to inherit form type parameters