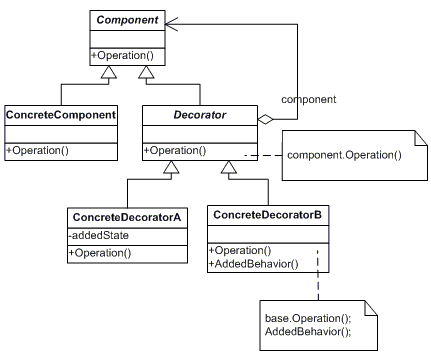
# GoF Design Patterns – Decorator

The purpose of this exercise is to implement a decorator design pattern, and to find out what this pattern is useful for.

## UML



## Definition

*Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.*

When looking at this definition, it consists out of two parts.

1. *It adds responsibilities dynamically*. Note that dynamically means that you can also use this to change the behavior of an interface at runtime.
2. *Decorators provide a flexible alternative to subclassing for extending functionality*. Although subclassing is one of the core concepts of OO, when subclassing very deep (more than, for instance 6 levels), you may create code that is difficult to maintain. With the quality tools integrated in Visual Studio, you can measure the depth of inheritance, and if it is higher than 6, your maintainability index will be lower.

## SOLID

Most GoF design patterns follow the SOLID principles, and although not mentioned in the definition, the decorator pattern also helps support the following principles:

1. Single responsibility principle. The SRP states that a class should only be responsible for one thing only. With the help of the decorator pattern we can separate various responsibilities in various classes while representing all of them in the same interface.
2. The open/closed principle. The OCP states that a class should be closed for modification but open for extension. In practice we strive to never touch an existing and fully tested class, even when new features are required. With the decorator pattern, we can add new functionality to an interface, without actually changing the existing and tested class.
3. The dependency inversion principle. The DIP states that we should depend on abstractions (or interfaces) rather than concrete implementations. Because we follow these principles, we can completely change the behavior of our application, or just add a new task or responsibility without altering existing and tested code.

## How to implement

All this theory may seem a bit confusing, however in laymen’s terms; this is how a decorator is implemented

1. Create a new class that implements the interface of the class you are decorating
2. In the new class, the constructor will take the old class (interface) as an argument.
3. In the implementation of the interface, you can now add extra responsibilities to the method(s), while also calling the old implementation.

## Basic example

In this example, we already have an existing *BookRepository*, the client requested for a new feature: The book repository needs to log its actions. Because it is bad practice to actually change the existing class and add more responsibilities (because it breaks the SOLID principles). We will implement it using a decorator.

/// The logging book repository implements the same interface

    public class LoggingBookRepository : IBookRepository

    {

        private IBookRepository repo;

        private ILogger log;

        // Note that we take the interface of the class as an argument

        public LoggingBookRepository(IBookRepository repo, ILogger log)

        {

            this.repo = repo;

            this.log = log;

        }

        // Decorated method: First we call the original implementation, after that  // we add the new behavior

        public void Save(Book item)

        {

            this.repo.Save(item);

            this.log.Write("Saved book: " + item.ToString());

        }

        // Decorated method: First we call the original implementation, after that  // we add the new behavior

        public Book Find(int id)

        {

            var result = this.repo.Find(id);

            if (result != null)

            {

                this.log.Write("Found book: " + result);

            }

            else

            {

                this.log.Write("Failed to find book: " + id);

            }

            return result;

        }

        // Decorated method: First we call the original implementation, after that  // we add the new behavior

        public void Update(Book item)

        {

            this.repo.Update(item);

            this.log.Write("Succesfully updated book: " + item);

        }

    }

In the example above, please note the following:

* We did not have to alter the original class (BookRepository) in order to add the behavior for logging. This also means that we did not have to alter the test, and therefore we can just expect the test to work.
* The only responsibility of this class is to add the logging behavior to the interface.
* Also not that the actual practice of logging is extended to another class (interface) in this example. This is also to follow the Single Responsibility Principle.
* Note that this pattern can also be useful – for instance if you want to implement before and after behavior. It would for instance be easy to time the execution period of the method, by adding code around (before and after) the original method call.
* Notice that we take the interface of the book repository in the constructor, not the actual call. This is following the Dependency Inversion Principle (rely on abstractions rather than concretions).
* The same goes for the logger. We don’t know how it is implemented, and frankly we don’t care. We just know that we have the method “Write”. If this goes to a database, a text file, or maybe even a web service that runs on a computer on Venus – we just rely on it to work. Dependency Inversion.

## Training Exercise Decorator 1

There are many applications for a decorator, however the general purpose is to add before- and after-behavior for a class. In this exercise we will add before- and after behavior to an existing implementation without actually touching the code of the original implementation. The value of this is, that you don’t need to change the original unit tests either, so you can expect this to work.

First create a C# console application that reads lines from the console, and displays this in reverse. You will not reverse the string directly, but delegate this behavior this to another class. This class implements the interface as displayed below



First create the following logic

* ReverseBehavior (implements IModifyBehavior). When apply is called, it returns the reversed string
* In your main procedure, ask the user to input something, and call the instance of IModifyBehavior (stored as instance variable with name behavior) and display that back to the user.
* All is unit tested as all the code we write is (you can skip the main procedure)

Now that that’s complete the client has a new feature request: The client wants to add logging behavior before the actual reverse code is called. Because we don’t want to alter the original class, as it is already tested, we can now decorate the interface and take the old original implementation in the constructor:

* Create a new class called LoggingModifyBehaviorDecorater that implements IModifyBehavior
* Take a parameter in the constructor with the name behavior of the type IModifyBehavior, and stores it as a private instance variable
* When implementing the Apply method, you first add the following
  + Debug.WriteLine(“Before”);
  + this.behavior.Apply(input);
  + Debug.WriteLine(“After”);

Congrats, you have now implemented your first decorator. Now make sure that you use it in the main procedure. Note that usually we would delegate the behavior of creating the actual classes to a factory, or rely on a dependency injection framework, so we would only have to change the behavior at a single place to add this to all instances of the behavior.

To complete the exercise try to add other behavior by creating the following decorators:

* Time how long the method takes to execute using a class called TimeBehaviorDecorator
* Decorate the original code with exception handling (try/catch) that will log something when it fails (try inputting null), call the class ExceptionHandlingBehaviorDecorator
* Decorate the original code with a class called ThreadingBehaviorDecorator that will ensure the reverse executes in a separate thread.
* Decorate the original code with a class called NullBehaviorDecorator, to ensure that the actual production code does not get executed. This can be useful for test environments that need access to a database or will alter the database, which you may not want in a test environment – using a decorator with a NullBehavior can ensure that the actual database does not get called
* And think of one more useful application for the decorator, and implement this.
* Note that all will be unit tested following the arrange-act-assert pattern

Good other applications for a decorator are the following

* Implement caching
* Logging and auditing – as per example
* Transaction management – ensure the opening and closing of a transaction before the actual code is called
* Interception – only execute code on certain conditions are inject other behavior
* Authorization – make sure that a user is authorized