Software Architecture & Design CSC3077

Lecture No. 37

Muhammad Shahid
Department of Computer Science
National Textile University

shahid.abdullah@hotmail.com

Last Lecture Review

- Structural Design Patterns
- Façade Design Pattern
- Client Access without & without Facade
- The Principle of Least Knowledge (PLK)
- Applying PLK in Facade
- Façade Pattern – Class Diagram
- Façade Pattern – Implementation
- Façade DP Example - Building a Car

Agenda – What will you Learn Today?

Decorator Design Pattern

Decorator Design Pattern

Decorator - Definition

- The decorator pattern is a design pattern that extends the functionality of individual objects by wrapping them with one or more decorator classes
- These decorators can modify existing members and add new methods and properties at run-time

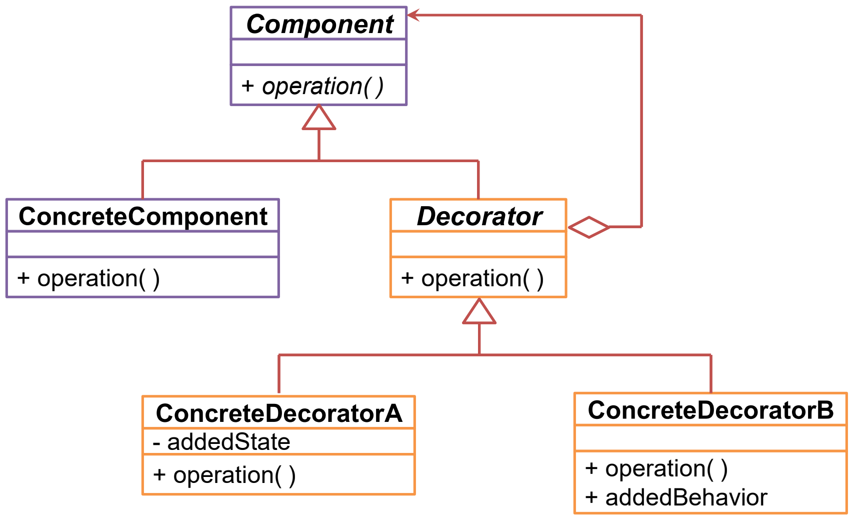
Decorator - Definition

“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality”

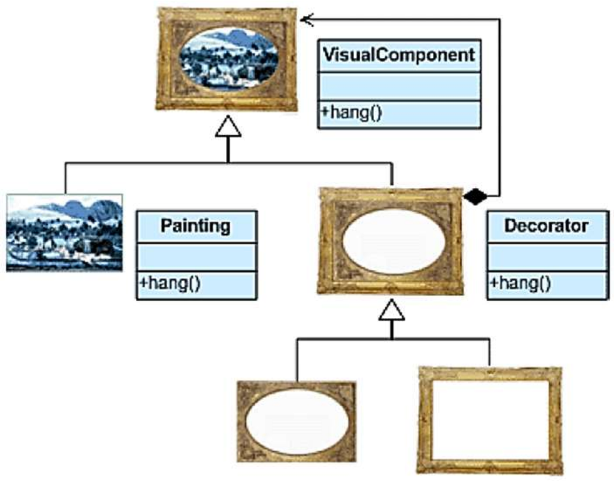
Decorator - Intent

- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Decorator – Class Diagram



Decorator – Class Diagram



Decorator – Code Implementation

```
public abstract class Component
{
        public abstract void Operation();
} // End of Component
```

```
public class ConcreteComponent : Component
{
        public override void Operation()
        {
                Console.Write("Concrete Component Called");
        }
}
```

Decorator – Code Implementation

```
public abstract class Decorator : Component
{
        protected Component component;

        public Decorator(Component component)
        {
                this.component = component;
        }

        public override void Operation()
        {
                component.Operation();
        }
} // End of Decorator class
```

Decorator – Code Implementation

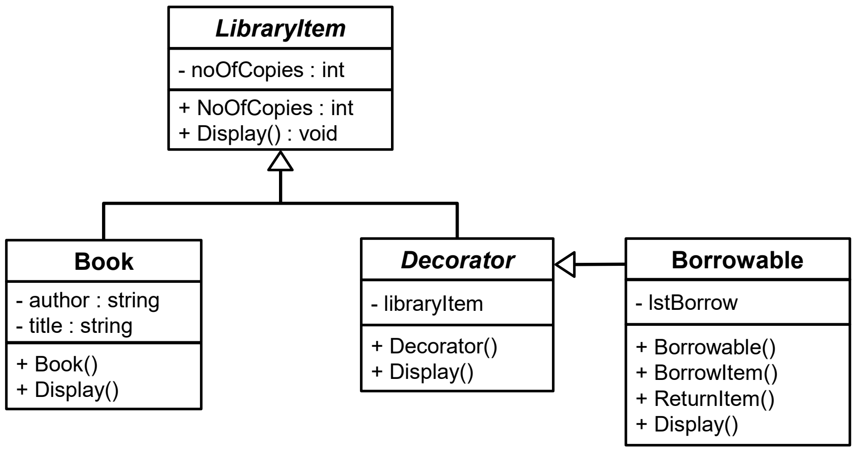
```
public class ConcreteDecorator : Decorator
{
        public ConcreteDecorator(Component component) :
        base(component) { }

        public override void Operation ( )
        {
                base.Operation();
                Console.Write("Concrete Decorator Called");
        }

} // End ConcreteDecorator class
```

Decorator – Code Implementation

```
static void Main(string[] args)
{
        ConcreteComponent comp = new ConcreteComponent();
        ConcreteDecorator decor = new
        ConcreteDecorator(comp);
        decor.Operation();
} // End of Client
```

```
Concrete Component Called

Concrete Decorator Called
```

Decorator – Class Diagram



Code Implementation

```
public abstract class LibraryItem
{
        public int noOfCopies;

        public int NoOfCopies
        {
                get { return noOfCopies; }
                set { noOfCopies = value; }
        }

        public abstract void Display();
} // End of LibraryItem
```

Code Implementation

```
public class Book : LibraryItem
{
        private string title;
        private string author;
        public Book(string title, string author, int noOfCopies)
        {
                this.title = title;
                this.author = author;
                this.noOfCopies = noOfCopies;
        }
        public override void Display()
        {
                Console.WriteLine("----- Book Information -----");
                Console.WriteLine("Title:{0}", title);
                Console.WriteLine("Author:{0}", author);
                Console.WriteLine("No of Copies:{0}", noOfCopies);
        }
} // End of Book class
```

Code Implementation

```
public abstract class Decorator : LibraryItem
{
        protected LibraryItem libraryItem;

        public Decorator(LibraryItem libraryItem)
        {
                this.libraryItem = libraryItem;
        }

        public override void Display()
        {
                libraryItem.Display();
        }
}
```

Code Implementation

```
public class Borrowable : Decorator
{
    protected List<string> lstBorrow = new List<string>();

    public Borrowable(LibraryItem libItem) : base(libItem)
    { }

    public void BorrowItem(string name)
    {
            lstBorrow.Add(name);
            libraryItem.NoOfCopies--;
    }
```

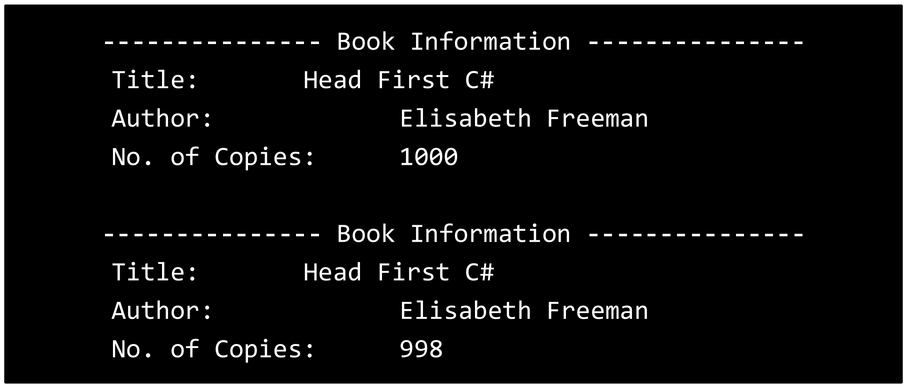
Code Implementation

```
    public void ReturnItem(string name)
    {
        libraryItem.NoOfCopies++;
    }

    public override void Display()
    {
        base.Display();
        foreach(string borrower in lstBorrow)
        {
            Console.Write("Borrower: " + borrower);
        }
    }
} // End of Borrowable class
```

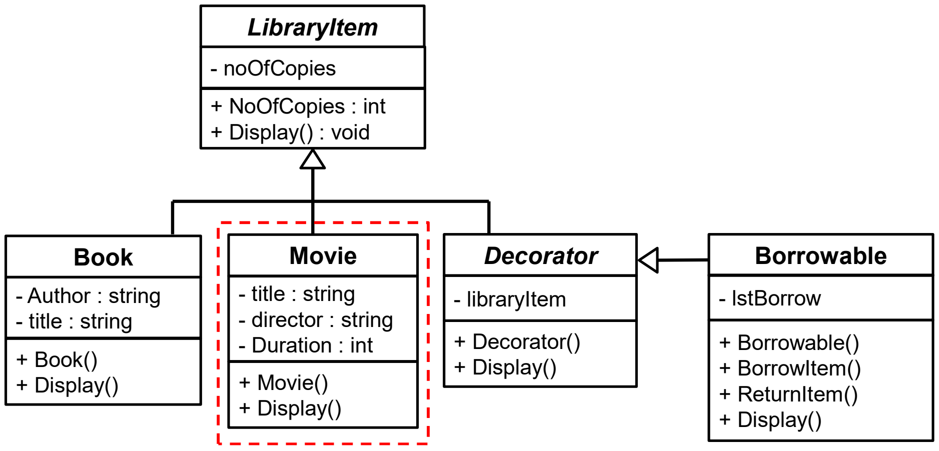
Code Implementation

```
static void Main(string[] args)
{
    Book book = new Book("Head First C#", "Elisabeth
    Freeman", 1000);
    book.Display();

    Borrowable borrowBook = new Borrowable(book);
    borrowBook.BorrowItem("Customer#1");
    borrowBook.BorrowItem("Customer#2");
    book.Display();
}
```

Code Implementation


```
--------------- Book Information ---------------
 Title:        Head First C#
 Author:             Elisabeth Freeman
 No. of Copies:      1000

--------------- Book Information ---------------
 Title:        Head First C#
 Author:             Elisabeth Freeman
 No. of Copies:      998
```

Decorator – Class Diagram



Code Implementation

```
public class Movie : LibraryItem
{
        private string title;
        private string director;
        private int duration;
        public Movie(string title, string director,
                        int duration, int noOfCopies)
        {
                this.title = title;
                this.director = director;
                this.duration = duration;
                this.noOfCopies = noOfCopies;
        }
```

Code Implementation

```
        public override void Display()
        {
            Console.Write("---- Movie Information ----");
            Console.Write("Title:{0}", title);
            Console.Write("Director:{0}", director);
            Console.Write("Duration:{0}", duration);
            Console.Write("No of Copies:{0}",noOfCopies);
        }
} // End of Movie class
```

Code Implementation

```
static void Main (string[] args)
{
       Movie movie = new Movie("Gladiator", "Ridley Scott",
                               120, 500);
       movie.Display();

       Borrowable borrowVideo = new Borrowable(movie);
       borrowVideo.BorrowItem("Customer#1");
       borrowVideo.BorrowItem("Customer#2");
       movie.Display();
}
```

Output

```
--------------- Movie Information ---------------
       Title:        Gladiator
       Director:            Ridley Scott
       Duration:            120
       No of Copies:500

--------------- Movie Information ---------------
       Title:        Gladiator
       Director:            Ridley Scott
       Duration:            120
       No of Copies:        498
```

Decorator – Pros & Cons

+ Provides a more flexible way to add responsibilities to a class by using inheritance, since it can add these responsibilities to selected instances of the class

+ Allows to customize a class without creating subclasses high in the inheritance hierarchy

- A Decorator and its enclosed component are not identical. Thus, tests for object types will fail

- Decorators can lead to a system with "lots of little objects" that all look alike to the programmer trying to maintain the code

Recap

- Structural Design Patterns
- Decorator Design Pattern
  - Intent
  - Definition
  - Class Diagram
  - Code Implementation
- Decorator – Example
- Decorator – Pros & Cons

Questions