SWD Assignment 1 – Design Patterns
Gruppe #15

Tue Alexander Jørgensen - 201810594

Asger Holm Brændgaard - 201810596

Jacob Nyvang Hansen – 201811364

Indhold

1. Chosen pattern - Decorator:

For this assignment the groups have chosen to work with the **Decorator Pattern**. Through carefully reviewing all the patterns, this was chosen on the base if its usefulness and applicability.

The main functionality of the Decorator Pattern is being able to add functionality to objects of a specific class. The purpose of this is to make objects of the same class able to handle different scenarios because of the added functionality. If the pattern is implemented correct, the program should be able to add the extended functionality at runtime. Therefore, responsibilities can be added to an object of a class without binding the whole class with a subclass at compile-time.

This pattern is only usable when multiple objects should have different functionality. If all objects of the same class should have the same functionality, a lot of unnecessary functionality addition would have to be done at runtime.

2. Structure

The structure of the pattern will be explained in relation to the actual implementation.

Figure 1. is the UML class diagram of the decorator pattern in relation to our implementation. IMenu is an interface with its concrete implementation in SimpleMenu. MenuDecorator, also referred to as the Base Decorator, references “wrapped” or extended objects. It uses FriesMenu, BurgerMenu etc., also referred to as Concrete Decorator, to add functionality to the beforementioned objects.

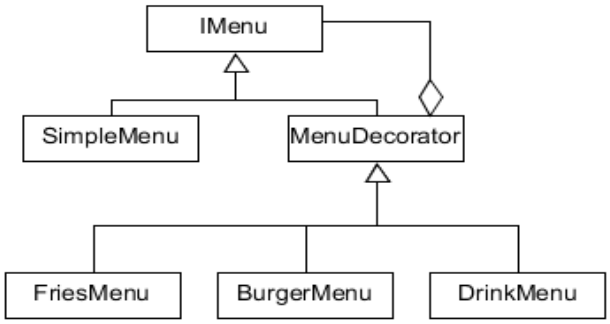

Figure 1. UML class diagram of Menu system

To extend objects a multiple of methods has to be called by a client interacting with the interface IMenu. As shown below in the code snippet, the sequence of creating a SimpleMenu object and adding both Fries and a burger is displayed.

```
var a = new SimpleMenu();
var b = new MenuFriesDecorator(a);
var c = new MenuBurgerDecorator(b);
```

Figure 2. Simple use of Decorator Pattern

Therefor would the concrete, or base, component SimpleMenu have the extended functionality of fries and a burger. In our concrete implementation we have used the pattern “multiple” times. By multiple times it is meant that Fries, Burger and Drinks is also build according to the Decorator Pattern. Therefore, it is possible to extend the burger e.g. with bacon or cheese.

3. Implementation

In this section a quick run-through of the implementation will be done. This will be based on the UML class diagram from figure 1.

```
public interface IMenu
{
    8 references
    double GetPrice();

    8 references
    string GetReceipt();
}
```

Figure 3. Codesnippet of IMenu interface

This is the “top” class of the pattern. This is what the client will be interacting with.

```
public class SimpleMenu : IMenu
{
    8 references
    public double GetPrice()
    {
        return 0;

    }
    8 references
    public string GetReceipt()
    {
        return "Menu: ";
    }
}
```

Figure 4. Codesnippet of SimpleMenu

As it is seen in the codesnippet on figure 4, SimpleMenu is the concrete implementation of the interface IMenu as it inherits from it. The class has two functions: GetPrice() and GetReceipt().

```
public abstract class MenuDecorator : IMenu
{
    private readonly IMenu _menu;

    2 references
    public MenuDecorator(IMenu menu)
    {
        _menu = menu;

    }
    8 references
    public  virtual double GetPrice()
    {
        return _menu.GetPrice();

    }
    8 references
    public virtual string GetReceipt()
    {
        return _menu.GetReceipt();
    }
}
```

Figure 5. Codesnippet of MenuDecorator

The above codesnippet is of the MenuDecorator. The constructor of it takes an IMenu object as a parameter. This way MenuDecorater both inherits and is a composition to IMenu. Therefore it is possible to the change the implementation of this at runtime. The methods GetPrice and GetReceipt are virtual as the actual implementation of these are made in the concrete Decorator.

```
public class MenuBurgerDecorator : MenuDecorator
{
    3 references
    public IBurger Burger { get; set; }
    2 references
    public MenuBurgerDecorator(IMenu menu) : base(menu)
    {

    }

    8 references
    public override double GetPrice()
    {
        return base.GetPrice() + Burger.BurgerPrice();
    }

    8 references
    public override string GetReceipt()
    {
        return base.GetReceipt() + Burger.BurgerDetails();

    }
}
```

Figure 6. Codesnippet of MenuBurgerDecorator

This is the “final” step of the Decorator Pattern. The keyword base specifies which constructor should be called when creating an instance of this class. In this case the base class is MenuDecorator.

The group has further “extended” the pattern. We have decorated the decorator. With this implementation we are able the add an item to the Menu (burger, fries, (drink)) and furthermore decorate these individual food items. The UML class for this is as followed:

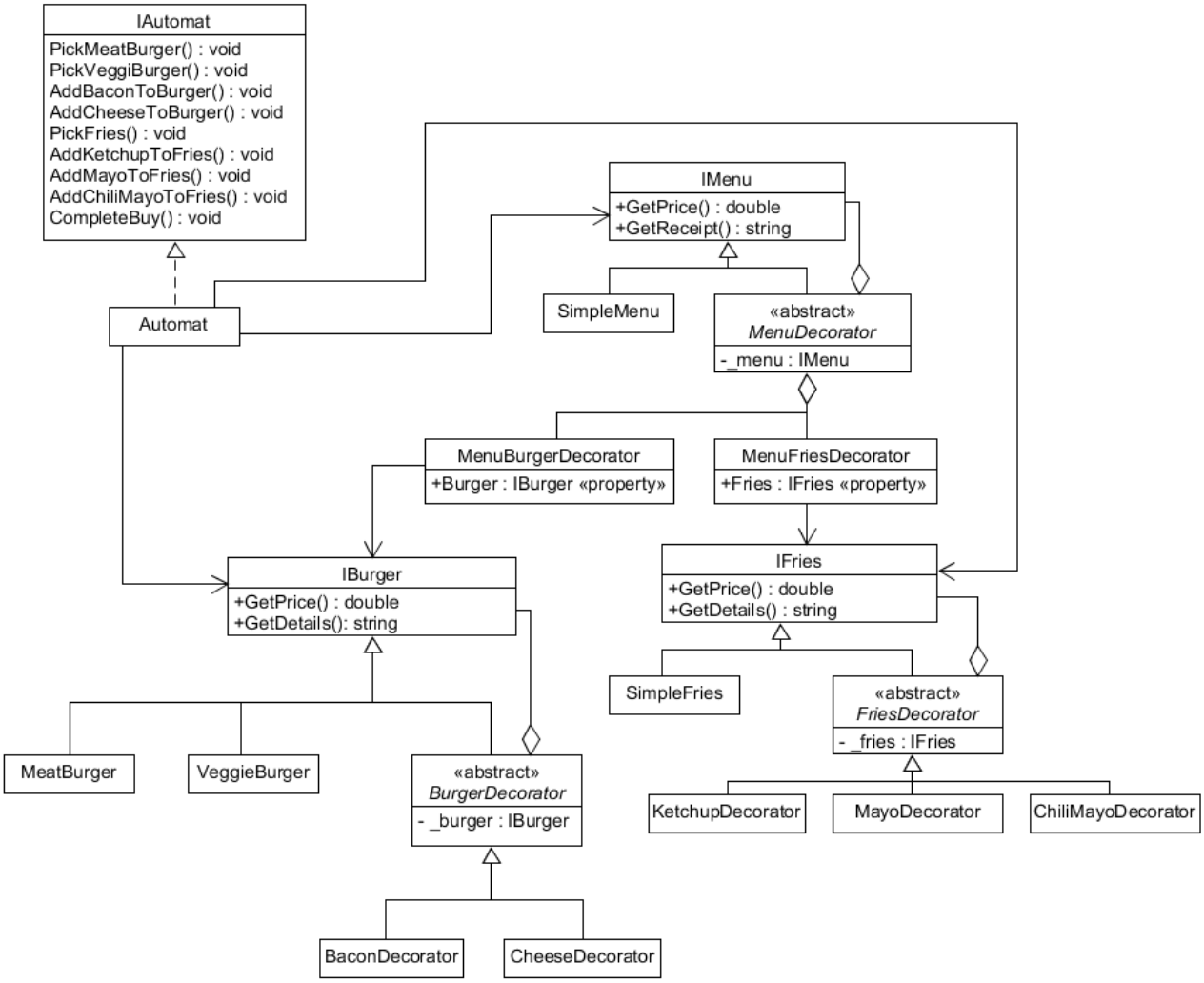

Figure 7. UML class diagram of the whole system

As shown on the above class diagram, each concrete MenuDecorator has an inherence. Therefore, the client (Automat) can decide how each individual Menu should be put together. The pattern allows the user to add burger and fries to their menu and decorate the fries and burger as they please. By designing the system this way, each object of the class SimpleMenu can have different functionality and attributes.

The sequence of class/object calls in figure 2 has been “hidden” away in the class Automat. Therefore, a user of the system doesn’t have to know the correct sequencing of method calls. The user can simply add different menu items, and the decorator pattern will insure correct object creation.

Our system is of simple functionality. The added functionality in the concrete Decorator are only an increase in price and a string addition to the GetReceipt method. An extension of this could be e.g. by a plate attribute which is added when burger and fries are added to the menu.

The actual implementation IBurger and BurgerDecorator can be found at the end of the rapport as an appendix.

Related patterns

Adapter pattern

Adapter pattern changes the interface of an object, decorator enhances an object. Adapter is used, as a middleman between e.g. a client and an object.

An example that could be used(taken from Wikipedia page), can be an adapter from a microUSB to a iPhone charger. Basically a device, that can be put on a microUSB so it fits an iPhone. Code wise there will be, made a class that, inherits the same as the microUSB phone does, this class takes an iPhone, and implements what's needed, for the adaption. So here we change the interface by inheritance, so that another implementation of the interface is made, and then used. (https://en.wikipedia.org/wiki/Adapter_pattern#Usage).

So compared to decorator, where we change the behavior of an object, we now change the implementation of an interface, that later can be used. In our usage, decorator is the better choice, when we wants to make, many different customizations, but when few more specific implementations are needed, adapter pattern would make more sense.

Strategy pattern

Strategy pattern makes it possible to select behavior of an object in runtime. This sound a lot like decorator pattern, but this is not an add on, but a complete change of behavior. For an example, the price of a ticket can change, depending on your age, what membership you have or a calculator need to work, with numbers different, if plus is chosen versus minus.

So depending on what strategy that is selected/giving or later changed to, the object behavior will change, corresponding to that.

(https://en.wikipedia.org/wiki/Strategy_pattern)

Chain-of-responsibility pattern:

This particular pattern is very similar to the Decorator pattern. The pattern contains a source of command objects and multiple processing objects. Each of these procession objects can handle a specific set of command objects. The command objects the processing object can't handle is passed along to the next processing object in the chain. This design is an object-oriented version of the if-else if- else-endif idiom. These processing objects can be rearranged and reconfigured at runtime, which in turn alters the idiom's (if-else if…..) internal structure.

The main difference between this pattern and decorator is the number of classes that handles a request. In the decorator pattern all classes are handles the request (SimpleMenu, MenuDecorator, MenuBurgerDecorator), where as in the chain-of-responsibility only one class handles the request. If you compare this to the if-idiom, only one "block" of an if-else statement is executed, as they are guarded by the condition of each block. As seen on figure 8, each concrete handle implementation is directly connected to the Handler. A request is compared to each Receiver in the specified order until a proper handler is found.

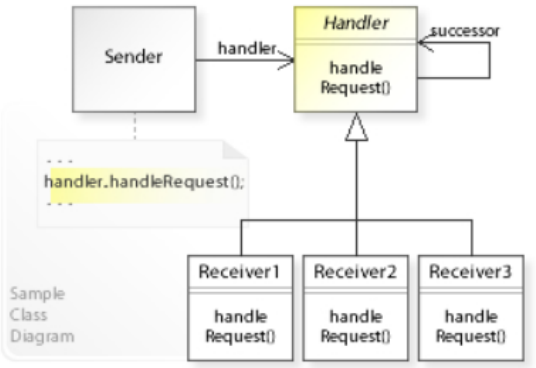

Figure 8. UML class diagram Chain-of-responsability pattern(diagram is taken from the Wikipedia page listed below.

https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

Appendix:

```
public interface IBurger
{
    13 references
    double BurgerPrice();
    13 references
    string BurgerDetails();
}
```

Figure 9. IBurger Interface

```
public class MeatBurger: IBurger
{

    13 references
    double IBurger.BurgerPrice()
    {
        return 49.95;
    }

    13 references
    public string BurgerDetails()
    {
        return "MeatBurger";
    }
}
```

Figure 10. MeatBurger. 1 of two implementations for the interface IBurger

```
public abstract class BurgerDecorator : IBurger
{
    private readonly IBurger _burger;

    2 references
    public BurgerDecorator(IBurger burger)
    {
        _burger = burger;
    }

    13 references
    public virtual double BurgerPrice()
    {
        //default value for Burger is 50
        return _burger.BurgerPrice();
    }

    13 references
    public virtual string BurgerDetails()
    {
        return _burger.BurgerDetails();
    }
}
```

Figure 11. Implementation of the BurgerDecorator

```
public class BaconDecorator: BurgerDecorator
{
    1 reference
    public BaconDecorator(IBurger burger) : base(burger)
    {
    }

    13 references
    public override string BurgerDetails()
    {
        return base.BurgerDetails() + ", Bacon ";
    }

    13 references
    public override double BurgerPrice()
    {
        return base.BurgerPrice() + 10;
    }
}
```

Figure 12. 1 of two concrete decorator implementation: BaconDecorator