MEMENTO PATTERN

Software Design – Mandatory Assignment 3

Group 1
Jacob Kurtzhals [Au537301]
Tobias Kjær Henriksen [Au479319]
Mikkel Overgaard [Au549290]
Flemming Lundahl [Au536900]

Aarhus Universitet - IHA

Indholdsfortegnelse

1 Memento Pattern

We have chosen to explore the Memento pattern. Through out this rapport, we will explain what the pattern is, how it is used, how to implement it and show an example of how to implement it. The example is written in C#.

1.1 Purpose

The purpose of the memento pattern is to give an object the ability to go back to a former state. In our implementation, we are making a wish list which consists of multiple wishes, of which you can add dynamically. The use of the memento pattern comes to use, if you add a wish, you can undo the wish list to a previous state, if you are not satisfied with the wish you just added.

Other real world application of this pattern could be, used in database transactions, so you could undo a transaction. But it was hard to find a specific use for the pattern.

1.2 Type

Memento is a behavioral pattern. It doesn’t make new objects only changes them, in this instance it is the objects state. The pattern allows the user to go back to a previous state, and thereby undo the actions that have been made on a object since that state.

1.3 Structure

The base structure of the memento pattern consists of three different classes, the memento class, the originator class and the caretaker class. The pattern is shown in figure 1 and is marked with a red box.

1.3.1 Memento

The memento object store the internal state of the originator object. It has two interfaces, an interface to the caretaker and an interface to the originator. The interface to the caretaker must not allow any access or any operation on the internal state. The interface to the originator allows it to access any state variables if necessary, so it can restore it. In our implementation it is realized as WishlistMemento.

1.3.2 Originator

The originator class shall make the Memento object, which contains the originators internal state and uses the memento to restore it is the previous state. In our implementation is it realized as Wishlist.

1.3.3 Caretaker

The caretaker is responsible for keeping the memento. The caretaker must not operate on the memento. In our implementation is it realized as IWishlistCaretaker.

1.4 Consequenses

The memento pattern protects encapsulation and won't expose the originators implementation and internal state. The pattern won't affect your classes, but if the class is too big, the program can use too many resources, especially if you want to keep multiple states.

2 Related pattern

The memento pattern is similar to the command pattern. The command pattern doesn't necessarily have anything to do with the undo nature of the memento pattern. But the use of the command pattern often include the make of a undo command, but they work a bit differently. The undo in the memento pattern works by restoring a saved state of an object. Meanwhile the undo with command pattern is by executing compensating actions.

3 Implementation

In our implementation, the memento pattern is realized as a wish list. The user will have the option to make his/her own wish list, and add wishes to it. The pattern is used to undo a recently added wish to the wish list or undo all the wishes that have been added during the session. The list will update dynamically. In the next part there will be shown our class diagram and a sequence diagram that shows how the memento handles the AddWish and UndoWish functions.

3.1 Class diagram

On figure 1 below is our class diagram. The memento pattern is marked with red. If we were to remove the pattern the program would still run, but without the ability to undo recent actions. As it is seen the caretaker stores our wish list memento, and our wish list menu makes sure to set or get the state from the caretaker, if the originator (Wishlist) calls restoreToCheckpoint it will undo the last wish added.

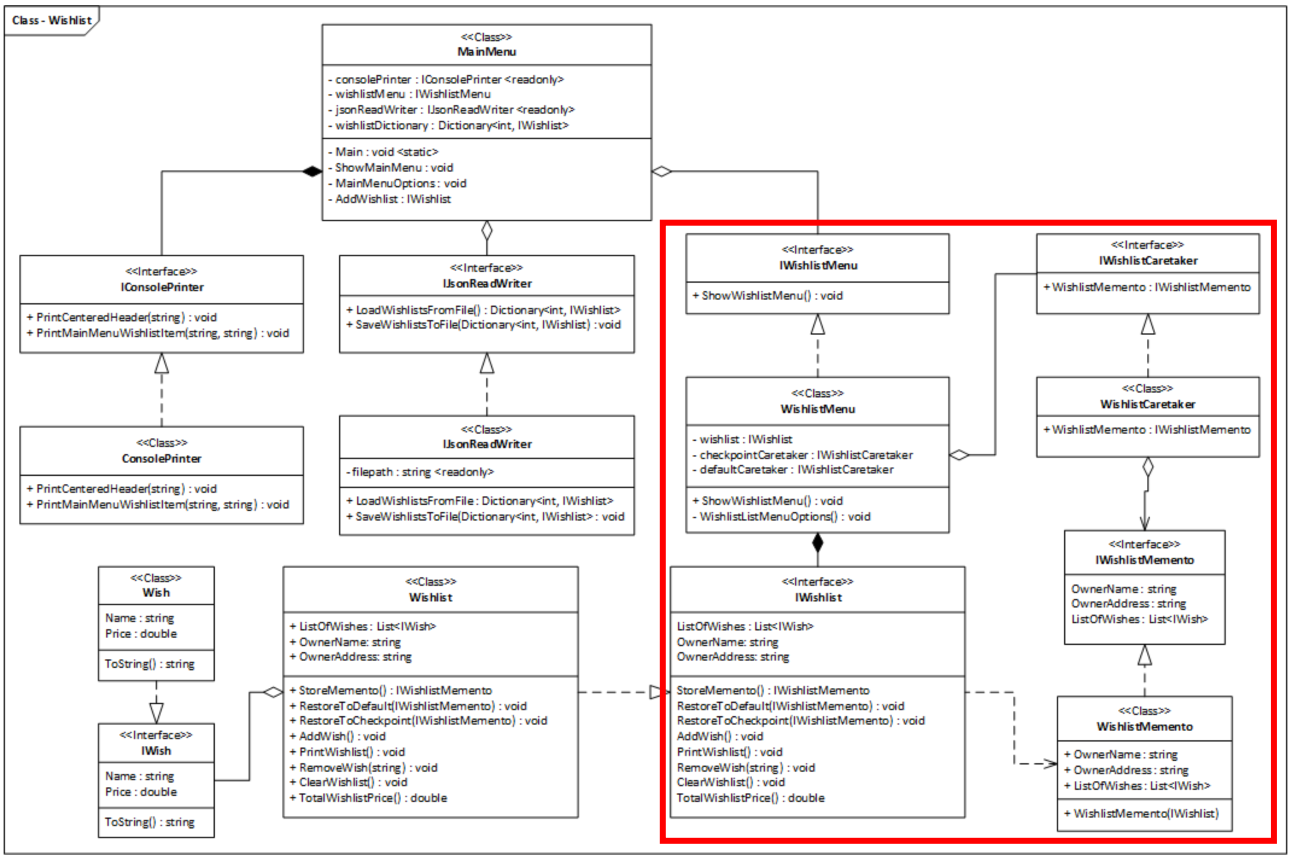
Class - Wishlist
<<Class>> MainMenu
- consolePrinter : IConsolePrinter <readonly>
- wishlistMenu : IWishlistMenu
- jsonReadWriter : IJsonReadWriter <readonly>
- wishlistDictionary : Dictionary<int, IWishlist>
- Main : void <static>
- ShowMainMenu : void
- MainMenuOptions : void
- AddWishlist : IWishlist
<<Interface>> IConsolePrinter
+ PrintCenteredHeader(string) : void
+ PrintMainMenuWishlistItem(string, string) : void
<<Interface>> IJsonReadWriter
+ LoadWishlistsFromFile() : Dictionary<int, IWishlist>
+ SaveWishlistsToFile(Dictionary<int, IWishlist) : void
<<Class>> ConsolePrinter
+ PrintCenteredHeader(string) : void
+ PrintMainMenuWishlistItem(string, string) : void
<<Class>> IJsonReadWriter
- filepath : string <readonly>
+ LoadWishlistsFromFile : Dictionary<int, IWishlist>
+ SaveWishlistsToFile(Dictionary<int, IWishlist> : void
<<Class>> Wish
Name : string
Price : double
ToString() : string
<<Interface>> IWish
Name : string
Price : double
ToString() : string
<<Class>> Wishlist
+ ListOfWishes : List<IWish>
+ OwnerName: string
+ OwnerAddress: string
+ StoreMemento() : IWishlistMemento
+ RestoreToDefault(IWishlistMemento) : void
+ RestoreToCheckpoint(IWishlistMemento) : void
+ AddWish() : void
+ PrintWishlist() : void
+ RemoveWish(string) : void
+ ClearWishlist() : void
+ TotalWishlistPrice() : double
<<Interface>> IWishlistMenu
+ ShowWishlistMenu() : void
<<Class>> WishlistMenu
- wishlist : IWishlist
- checkpointCaretaker : IWishlistCaretaker
- defaultCaretaker : IWishlistCaretaker
+ ShowWishlistMenu() : void
- WishlistListMenuOptions() : void
<<Interface>> IWishlist
ListOfWishes : List<IWish>
OwnerName: string
OwnerAddress: string
StoreMemento() : IWishlistMemento
RestoreToDefault(IWishlistMemento) : void
RestoreToCheckpoint(IWishlistMemento) : void
AddWish() : void
PrintWishlist() : void
RemoveWish(string) : void
ClearWishlist() : void
TotalWishlistPrice() : double
<<Interface>> IWishlistCaretaker
+ WishlistMemento : IWishlistMemento
<<Class>> WishlistCaretaker
+ WishlistMemento : IWishlistMemento
<<Interface>> IWishlistMemento
OwnerName : string
OwnerAddress : string
ListOfWishes : List<IWish>
<<Class>> WishlistMemento
+ OwnerName : string
+ OwnerAddress : string
+ ListOfWishes : List<IWish>
+ WishlistMemento(IWishlist)

Figure 1, Shows the class diagram of our implementation of the memento pattern.

3.2 Sequence diagrams

On figure 2 it shows one run through of our program. The user will be presented with a menu, where the user can edit a existing wish list or add a new one. In this diagram the user adds a new list and it shows who and when the memento and caretakers are instantiated. After the call ShowWishlistMenu the user is presented with options to Add a new wish, Undo last added wish, save and exit, and exit without saving. In the exit without saving it uses the defaultCaretaker and will undo all the wishes added during your session. The next bits are a bit chopped up, but the full sequence diagram are attached in the folder.

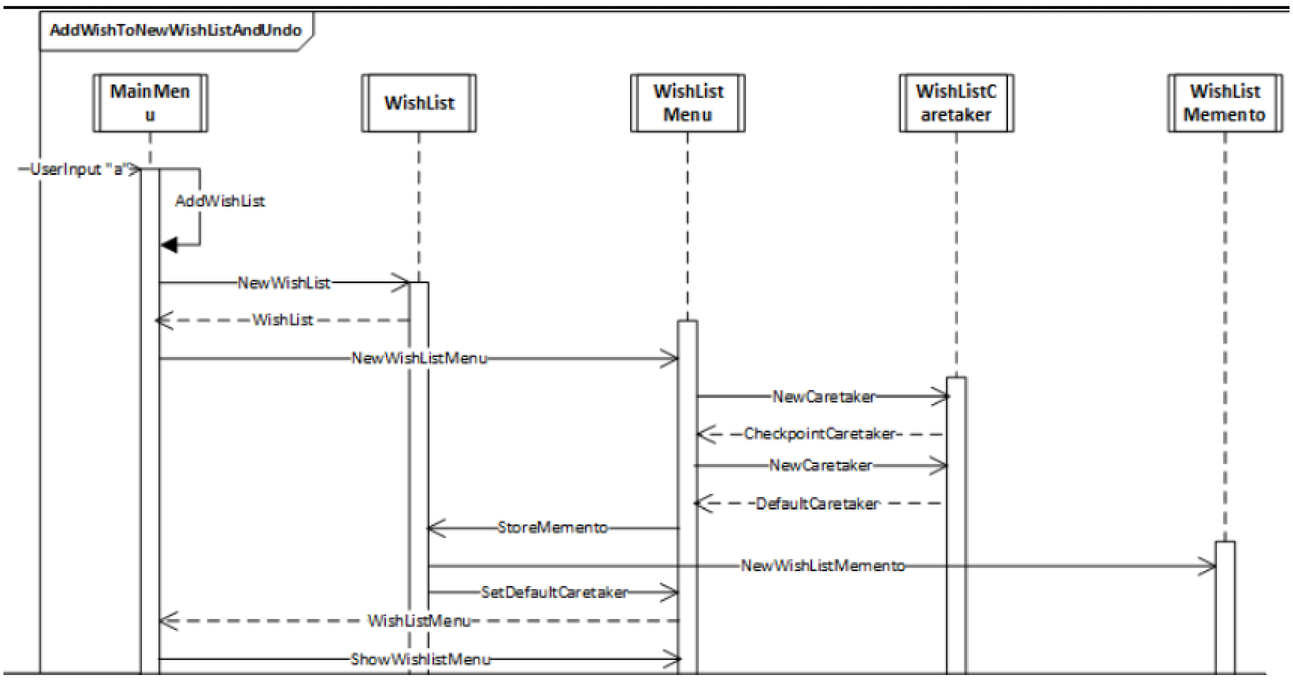

Figure 2, shows the sequence diagram for one run through of the program

3.2.1 Add command

This is after user are presented with the wish list menu. It shows how the add works and who calls the storeMemento and how it Sets the caretaker.

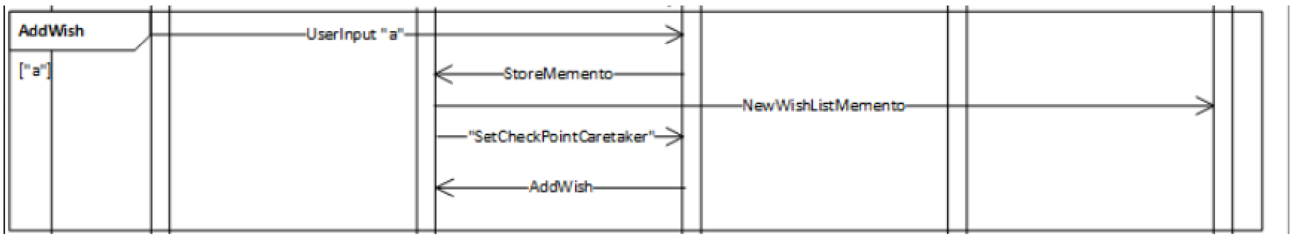

Figure 3, shows the sequence diagram for AddWish

3.2.2 Undo command

This is after user are presented with the wish list menu. It shows how the undo works who calls the restore and how it gets the caretaker.

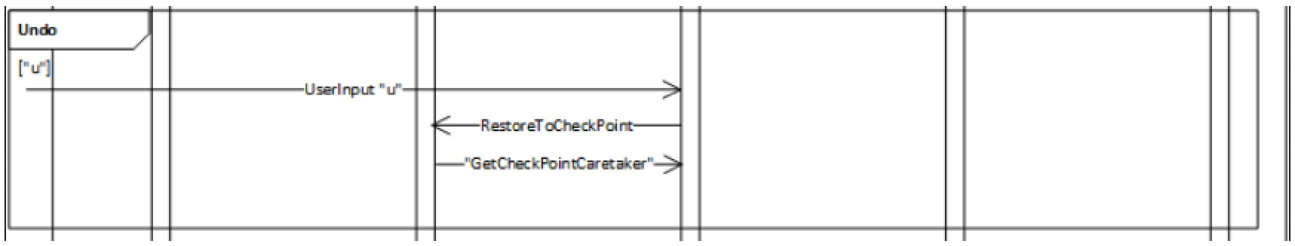

Figure 4, shows the sequence diagram for Undo

3.3 Originator klasse

In our implementation is the originator class, wishlist. It has responsibility for all its functions aswell as the memento functions. In the code snippet below it shows how the method RestoreToCheckpoint works. It changes the state of the wish list and restores last saved and it saves to checkpoint.

```
        public void RestoreToCheckpoint(IWishlistMemento checkpointWishlistMemento)
        {
            if (checkpointWishlistMemento != null)
            {
                OwnerName = checkpointWishlistMemento.OwnerName;
                OwnerAddress = checkpointWishlistMemento.OwnerAddress;
                ListOfWishes = checkpointWishlistMemento.ListOfWishes;
            }
        }
```

3.4 Memento class

The memento class is similar to the originator, but it only contains its properties. Because it has to save all the information of the originator.

```
        public WishlistMemento(IWishlist wishlist)
        {
            OwnerAddress = wishlist.OwnerAddress;
            OwnerName = wishlist.OwnerName;
            if(wishlist.ListOfWishes != null)
            ListOfWishes = new List<IWish>(wishlist.ListOfWishes);
        }
```

3.5 Caretaker klasse

The caretaker class holds the saved mementos and that is the only responsibility for the caretaker.

```
    public class WishlistCaretaker : IWishlistCaretaker
    {
        public IWishlistMemento WishlistMemento { get; set; }
    }
```

4 Usefulness

The pattern isn't used much in real world application. The only use we could is that is used with database transactions. Where if the transactions fail or didn’t have the right impact, when you will have the ability undo it. It was also hard to a use for it when we should come up with our example. But it is easy to implement on almost all classes, that aren’t too complex. The next section will list some pros and cons.

4.1 Pros

Reusable if implemented with a list of mementos, so you can undo, undo and undo. Like you would use Ctrl+Z in almost any other program. It does not violate capsulation of the class. It is very easy to implement on already existing classes, only need the caretaker and memento class.

4.2 Cons

Hard to find a specific use, very few real world applications. Another pattern can often replace this, like command patter, and do it better and faster. If classes too complex or a device with very little memory, it might give memory problems.

5 Conclusion

The Memento pattern might be of value to some special types of applications, where some classes often changes state. This would, without violating the classes encapsulation, allow rerolling to some previous states. The pattern might not be viable for more advanced programs, where having many saved states would potentially use up too much memory. As stated previously, this pattern isn't used much in real world applications anymore, since these often are either too complex, or because their data isn't in need of the ability to reroll.

6 References

https://en.wikipedia.org/wiki/Memento_pattern

https://sourcemaking.com/design_patterns/memento

http://www.dofactory.com/net/memento-design-pattern