Software Design - Mandatory Design Patterns Assignment
Group 11

Chris Olesen, AU453533
Mads Jørgensen, AU541836
Kasper Kronborg, AU550900

December 2018

Purpose:

The purpose of this exercise is to familiarize with a single design pattern not studied in class. In this report the chosen pattern is the Visitor pattern, both in concept and in implementation.

Contents

1 Introduction

We have chosen to examine the Visitor Design Pattern. This rapport will explain the design pattern and show and explain a simple example program implementing the Visitor Pattern.
The example program shows a stock with an inventory of different food objects with attributes that can be changed using visitors.

2 The Visitor Design Pattern

The Visitor Pattern is a Behavioral design pattern. The primary purpose of the Visitor Pattern is to abstract functionality from the element classes. This encourage lightweight element classes because processing functionality is removed from their list of responsibilities. General functionality can easily be added because it can be performed to the elements as a visitor instead of changing the element classes.
The Visitor Pattern allows an operation to be performed on the elements of an object structure without the need of changing the base code of those elements classes.
Imagine that you have to add functionality to manipulate an object without making changes to the existing code. You could do so using the visitor design pattern.
The visitor pattern makes it possible to perform an operation on a collection of objects. The operation performed can be easily changed without changing the classes of the elements. The objects in the collection does not have to be of the same type.

2.1 Structure

Figure 1 shows the pattern structure.

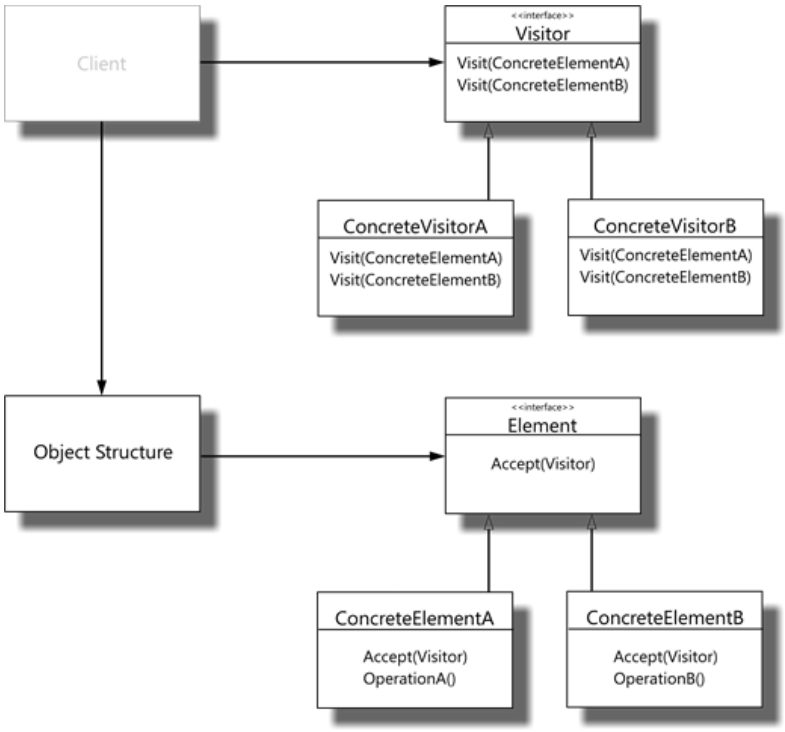

Figure 1: Visitor Pattern Class Diagram
(from: https://lostechies.com/derekgreer/2010/04/23/the-agile-visitor/)

Client is a consumer of the classes of the Visitor pattern. It could be main(). The Client has access to the objects and can instruct them to accept() a visitor.
Visitor encapsulates the common operations that needs to be performed to the different types of elements. Visitor is a interface or an abstract class that is implemented by the concrete visitors. The interface declares a visitor for all the types of elements. The Visitor could be implemented as an abstract class instead.

Concrete Visitor implements the Visitor Interface. There shall be one Concrete Visitor for each different operation that needs to be performed on the elements. All the methods of the Visitor Interface needs to be implemented by each Concrete Visitor.
Object Structure is the element container structure of all elements added to the visitor.
Element is an interface that declares the accept operation. This interface enables an object to be visited by the visitor object.
Concrete Element is objects that implement the Element Interface (defines the accept operation). The accept operation defined is the same for all Concrete Elements but it cannot be moved to the Element Interface (or abstract class) because of the reference to "this" . The visitor object is passed to the concrete object using the accept operation.

2.2 Related Patterns

Single-Serving Visitor
The Single-Serving Visitor pattern is a version of the Visitor Pattern where elements need to be served only once. When a visitor has performed its task it is removed from memory. The advantage of using the Single-Serving Visitor is that no "zombie" visitor objects will exist. They are allocated when needed and destroyed once useless. The interface of the Single-Serving Visitor is also very simple. For operations on multiple elements the repeated allocation and deallocation will be time-consuming.

3 Implementation

Our implementation of the visitor patterns is showed on fig. 2. It's not a perfectly correct build UML diagram, but it's meant to give a quick overview of our code without showing every single detail.

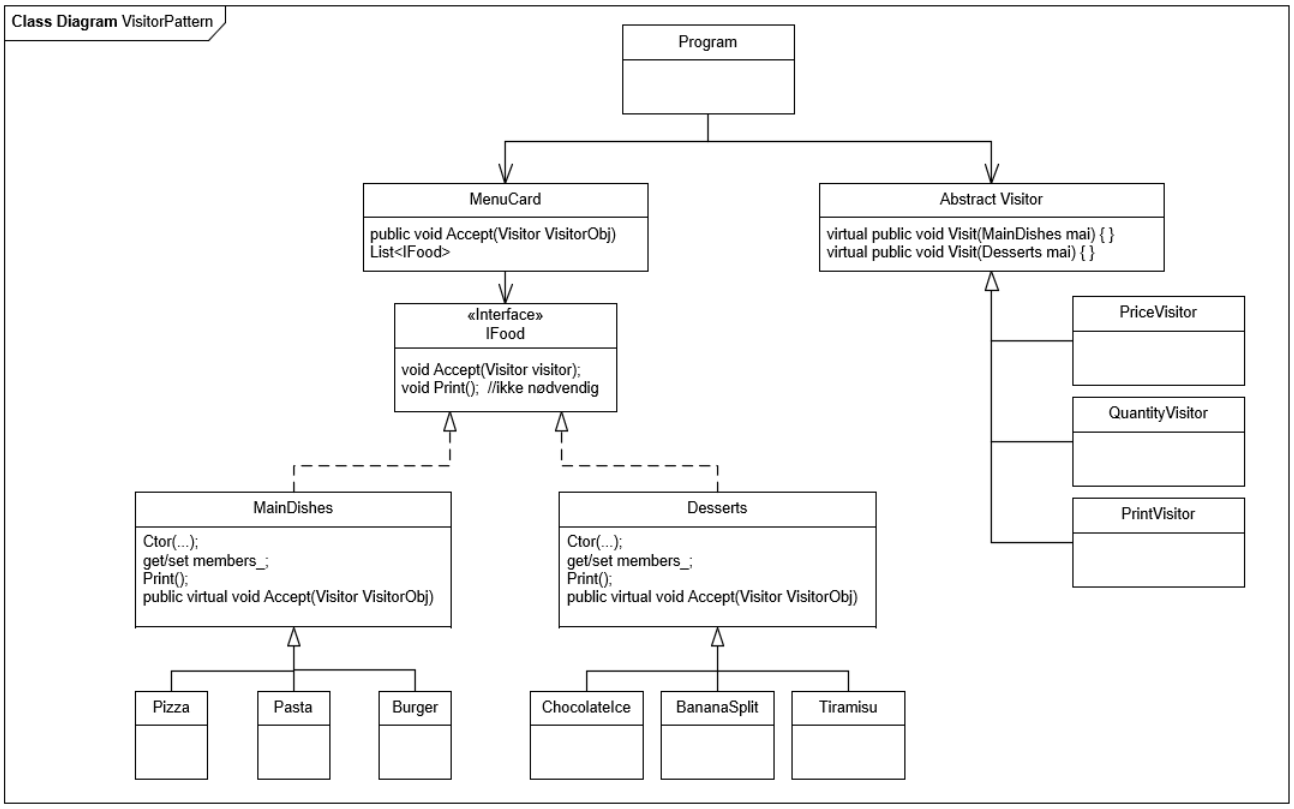

Figure 2: Class Diagram for Visitor Pattern

MenuCard
MenuCard is a <list> called Menu containing IFood elements, which means it contains instances of MainDishes and Desserts.

- It has an Accept() function, for forwarding the accept() call to the instances of mainDishes and Desserts via IFood.
- It has an Print() function, for forwarding the Print() call to the instances of mainDishes and Desserts via IFood.

IFood
The IFood interface makes the connection to mainDishes and Desserts = passes the Acceps() and Print() functions.

- It has an Accept() function, for forwarding the accept() call to the instances of mainDishes and Desserts via IFood.
- It has an Print() function, for forwarding the Print() call to the instances of mainDishes and Desserts via IFood.

MainDishes Desserts
Instances of mainDishes and Desserts are instantiated underneath their implementation. The objects are inserted into the Menu list by program.

- These classes' member variables are set in the constructor, but can also be accessed via get/set functions.

- A Print() function has been implemented here, since we meant it would make most sense that the class decided how it would print itself instead of putting that responsibility in MenuCard or Program.
- The Accept() function takes a visitor object, and makes that object point to a specific instance of either mainDishes or Desserts, which can then be modified by the visitor object.
- Pizza, Pasta, Burger, ChocolateIce, BananaSplit and Tiramisu are just instances of mainDishes and Desserts with set name_, price_ and quantity_ members.

Visitor
Visitor is an abstract class for prototyping the Visit() function to each classes which we want to change.

PriceVisitor
PriceVisitor contains the implementation of the Visit() function to each classes which we want to change the price_ member on, and how we would like to change it.
The same goes for **QuantityVisitor**

PrintVisitor
If IFood had not had the Print() function in it, this would be the way to access the Print() function of mainDishes and Desserts.
In main you'll see we use both ways of printing.

Program
Program contains main, where all the instances of mainDishes and Desserts are inserted into the Menu, then printed from menu.
Then we create a the three described visitors to make a black friday sale, setting all prices 50% lower, and stocking ten times more goods. Followed by a call to the PrintVisitor() function.

4 Conclusion

Through the example code and explanation it is shown how the Visitor Pattern can be used to add functionality to groups of objects without changing the code of these classes. The Visitor Pattern is a great way to add functionality or perform operations on groups of elements but if functionality has to be added to only a few classes, it might be more simple to just add the functionality directly to those classes without using the visitor pattern.