**ABSTRACT**

As modern software grows more and more complex, various techniques are invented or introduced to this field to increase productivity and reliability of software development. The Visitor pattern performs an extensible set of operations on an expression tree without requiring any changes to the tree itself. In other words, it creates an external class to act on the data in other classes. This is reasonable when considering polymorphic operations that cannot stay in the class hierarchy for many reasons. For example, because the operation was not taken into consideration during the design of the hierarchy or the interface of the classes will be clogged unnecessarily. This report tries to highlight the benefits of visitors design pattern and its application. We believe this helps on both understanding the design pattern and using or developing the pattern.

**INTRODUCTION**

A familiar expression amongst software designers, “laziness is a virtue: it is better to reuse than to redo” (Meyer & Arnout, 2006). The practice of software reuse has been in existence since the origin of programming. It’s intended purpose is to improve software productivity and quality by making use of existing and proven software parts when developing a new software. It gets more interesting because people want to always build reliable systems that are massive and more complex than the old, but in a littler time frame. This brings us into the topic of Design Patterns.

In software engineering, a *Design Pattern* is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. (Design Patterns, n.d.).

Design patterns improve development process speed significantly by providing tested and proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns (Design Patterns, n.d.). People only often comprehend how to use certain software design techniques to particular problems. Design patterns provide general solutions which are documented in a unique format that does not need specifics tied to a particular problem. In addition, patterns allow different developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved upon over time, allowing them be more robust than other ad-hoc designs.

Design patterns can be broken down into three main categories being;

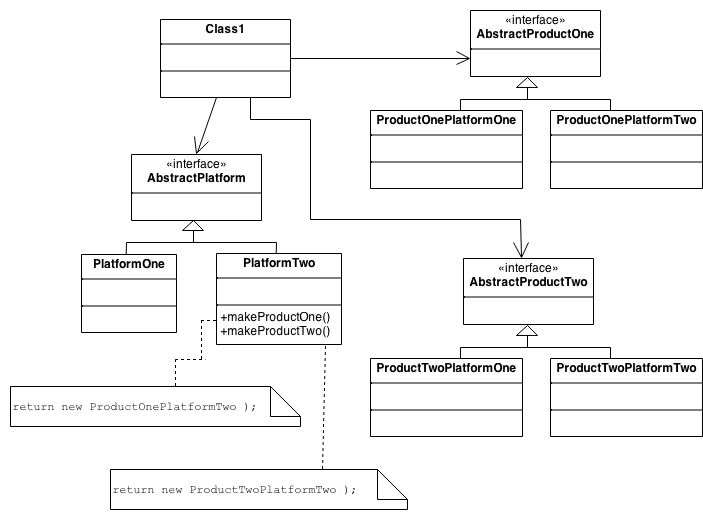
* Creational Design Pattern
* Structural Design Pattern
* Behavioral Design Pattern

1. Creational Design Patterns:

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done. Examples of Creational design patterns are Abstract factory, Builder, Factory method, Object Pool, Prototype and Singleton.

Abstract Factory, for example, creates an instance of several families of classes. That is, it creates families of related or dependent objects by providing interfaces without specifying their concrete classes. It also provides a hierarchy that encapsulates many possible platforms and the construction of a suite of products. For an application to be portable, it needs encapsulated platform dependencies which include windowing systems, operating systems, databases and so on. Many a times, this encapsulation is not engineered in advance. This allows a lot of #ifdef case statements with options for all currently supported platforms to begin to procreate like rabbits throughout the code. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them. This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory. Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton (Design Patterns, n.d.).

The Abstract Factory defines a Factory Method per product which encapsulates the new operator, concrete, platform-specific and product classes. The factory derived class is now modelled with each platform.

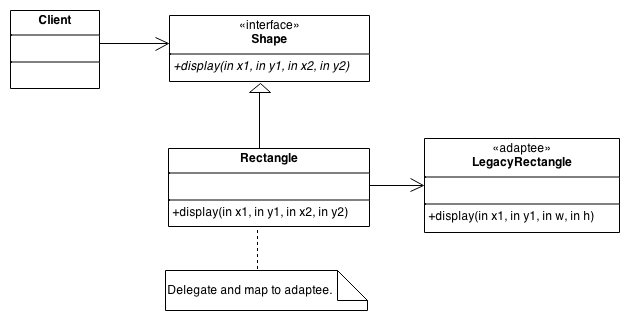


*Fig1.1: Diagram of the structure of Abstract factory pattern*

1. Structural Patterns:

Structural patterns are design patterns that simplify the design by recognizing an easier method to realize relationships among entities (Structural Patterns, 2018). In simpler terms, they are bothered with the organization of classes and objects to create a larger structure. The structural design patterns make the structures easier by identifying their relationships. These patterns focus on how classes inherit from each other and how they are composed from other classes (Structural Design Patterns, n.d.). There are 7 types of design patterns – Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Façade Pattern, Flyweight Pattern and Proxy Pattern. We shall briefly discuss the adapter pattern for additional understanding.

The adapter pattern converts the interface of an older class into another interface a client expects. In other words, it allows other classes work together which couldn’t have worked otherwise because of conflicting interfaces. This pattern is also known as Wrapper. It is used when an object needs to make use of an existing class with an incompatible interface and when one wants to make a reusable class that works well with classes that don't have like-minded interfaces. It is like the challenge of plugging a new three-prong electrical cord in an old two-prong wall outlet – a special adapter or intermediary or connection is needed (Structural Design Patterns, n.d.). The adapter structure can be described as thus;



*Fig1.2: Diagram of the structure of Adapter pattern*

1. Behavioral pattern:

In Software engineering, behavioral patterns are concerned with the interaction and responsibilities of objects and realize these patterns. From doing this, the inter-relatability of patterns increase in carrying out this communication. In simpler terms, it means that the implementation and client should be loosely put together in order to prevent hardcoding and dependencies. There many types of behavioral patterns – Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method and Visitor pattern.

The Visitor pattern is a design pattern to represent an operation to be carried out on the elements of an object structure. The Visitor lets you create a new operation without altering the classes of the elements from which it operates. It is a classic technique for missing-type information. We shall be discussing the Visitor pattern on a broader spectrum in the chapter below.

**THE VISITOR PATTERN**

The Visitor pattern alters the tables on the Object-Oriented model and develops an outside class which acts on the data in other classes. It is especially important for polymorphic operations which cannot be inside the class hierarchy for peculiar reasons. A reason could be because the operation as a whole was not considered during the hierarchy design or that it will clog the interface of classes unnecessarily.

It might appear to be unreasonable to put operations inside a class that should be existing in another. Referencing James W. Cooper’s example, assuming the object classes of a number of drawing objects have the same code for drawing itself, the drawing methods may be dissimilar but they will most likely make use of underlying utility functions that might have to be reproduced in each class. Alternately, a Visitor class is written which encompasses of all the related draw methods and have to visit each of the objects one after the other. (Cooper, 2002)



*Fig2.1: A Visitor class (Drawer) that visits each of three triangle classes*

Observing this, one can see that the only option whereby an external class can gain access to another class is by calling it’s public method using a function “visit()”. The method, “accept” shows that you are calling the method already installed for the purpose of visiting each class. The accept method has only one argument, which is the instance of the Visitor. In exchange, it passes itself as an argument by calling the visit method of the Visitor.

Visited(VO)

Visitor(v)

*Visited.accept(v)*

*v.visit(this)*

*Fig2.2: The way the visit and accept methods interact*

Now taking a look at a practical example, where a program calculates the price and quantity of food. It may so happen you have to do a functional change in price and quantity to a collection of classes but you no control over the class hierarchy. This is a perfect situation the Visitor pattern comes in. The purpose of the Visitor design is to create a new operation for a collection of classes (visited classes) without having any change on the hierarchy itself.

In this situation, the requirement is to make functional changes in price and quantity where the food price will increase by 10% and quantity by 3.

Pizza

Price = $100

Quantity: 10

Pasta

Price = $50

Quantity: : 5

Burger

Price = $20

Quantity: 25

PriceVisitor

QuantityVisitor

Pizza

Price : $110

Quantity: 13

Pasta

Price = $55

Quantity: : 8

Burger

Price = $22

Quantity: 28

*Fig2.3: Visitor Pattern example*

Now we have the original code written to calculate the price and quantity of the food, we have to add the new price and quantity functions without changing the original code base, where we are incrementing the price by 10% and the quantity by 3 of each food. To begin this we create a new visitor for price and another visitor for quantity to show the changes in the prices and quantity. In this example, the new price of pizza will be 110 and the quantity will be 13 respectively.

In the implementation part, first we need to implement the visitor abstract class. Next is the Visitable interface, where we have defined an accept method, passing the visitor as a parameter. The next thing was where I have defined the structure of the food, where we have passed the name price and quantity of the food as a parameter.

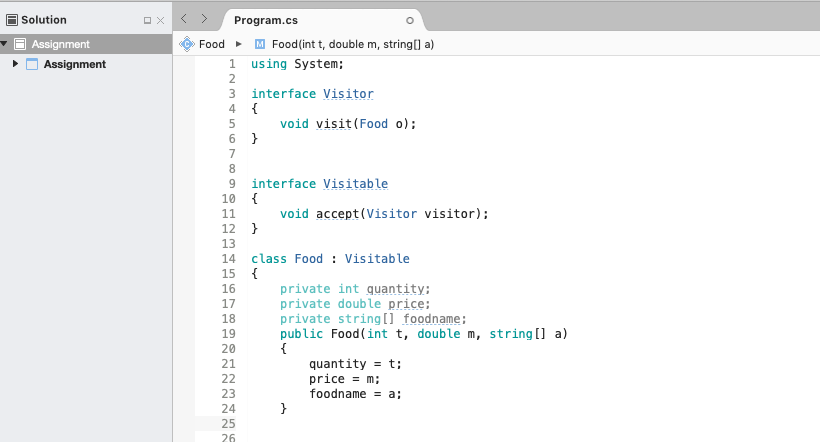


Figure 2.4: Code sample of the interface: visitor

The next section is where we have pass three methods; food name, old quantity amount and old price amount. Also, added the accept method, which calls the Accept, creating and accepting a visitor one by one.



Figure 2.5: code sample of the accept method

The following section is where I created a new class MoreFunctions, which calls from the visitor. This is where the new price and quantity for the food menu is added to the old. Here the new quantity should have an increased value of 3 from the old; also the new price should have an increased value of 10% from the old.

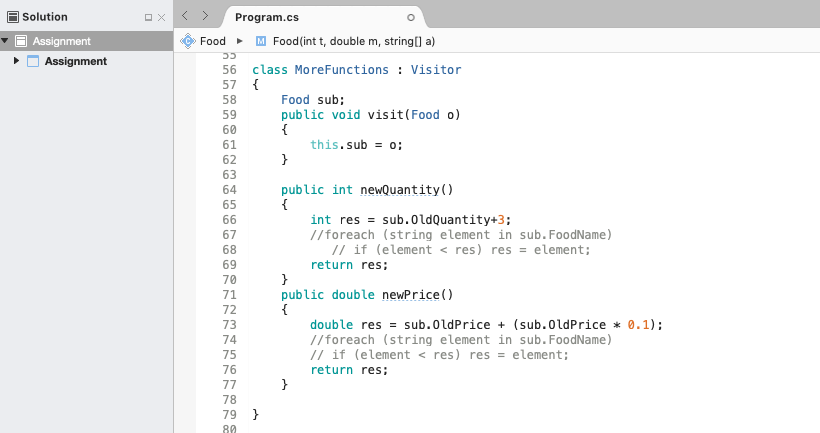


Figure 2.6: code sample of the morefunction

Now finally, we call these classes in the main method. The user inputs the old price of food and the program converts this string value to double so it can display the former price and the new price. Also, the user inputs the old quantity amount and it’s converted from string to Int before display. The new food price and quantity is called and displayed to the user.

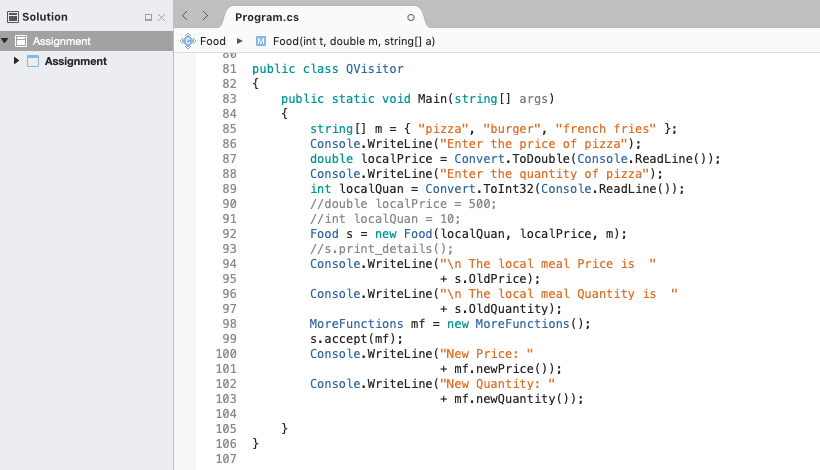


Figure 2.7: calling the main method

The working program is shown below;

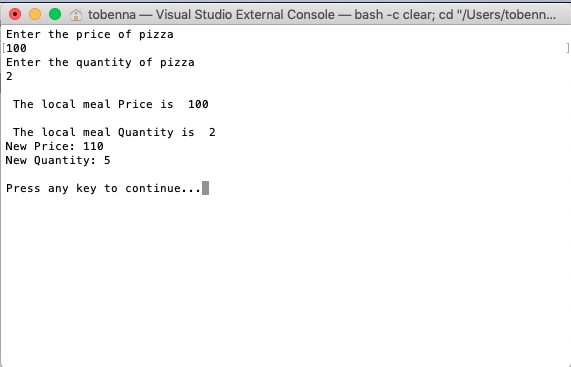


Figure 2.8: Terminal view of working program

**CONCLUSION**

The visitor pattern is an easy and very convenient method of applying new functionalities to an already existing code base. It is easy to use and very understandable. While this is so, the visitor is not the best for a scenario where the visited class is not fixed. For any time a new derived class is added, every visitor derived class must be adjusted to suit this. This can prove to be cumbersome and complex on the eventuality.

# Bibliography

Cooper, J. W. (2002). *Introduction to Design Patterns in C#.* IBM T J Watson Research Center.

*Design Patterns*. (n.d.). Retrieved from Source Making: https://sourcemaking.com/design\_patterns

Meyer, B., & Arnout, K. (2006). *Componentization: the Visitor example.* California.

*Structural Design Patterns*. (n.d.). Retrieved from Javatpoint: https://www.javatpoint.com/structural-design-patterns

*Structural Patterns*. (2018, October 19). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Structural\_pattern