Jailbreak Pong

Group KAZE

Karan Bhandari	Adil Mian	Zeyad Abdulghani	Eric Luo
k3bhanda	am2mian	zaabdulg	e2luo

1.0 Motivation

It is easy to make mistakes when adding a new operation (i.e. taxes calculation) to many different element classes, especially if some of the element classes implement the operation slightly differently (i.e. necessities, tobacco, alcohol).

When the operation needs to be modified (i.e. holiday has different tax rates), developers need to modify every single element class. This makes it more difficult to maintain as the number of affected classes increases.

1.1 Solutions

The visitor pattern resolves the aforementioned issue by housing the new operation in a separate visitor class (i.e. Taxes class). For each different element class (i.e. necessities, tobacco, alcohol), the visitor class will have a separate function that only accepts their data type.

The element classes virtually add the relevant function (i.e. taxes calculation function) from the visitor class by accepting a visitor, and calling the function in the visitor that corresponds to their type.

1.2 Intended Use Cases

- When many unrelated operations on an object structure are required
- When new operations are added frequently
- When existing operations may be changed frequently
- When the developers would like to have functions related to a single type of operation in a single class for better maintainability

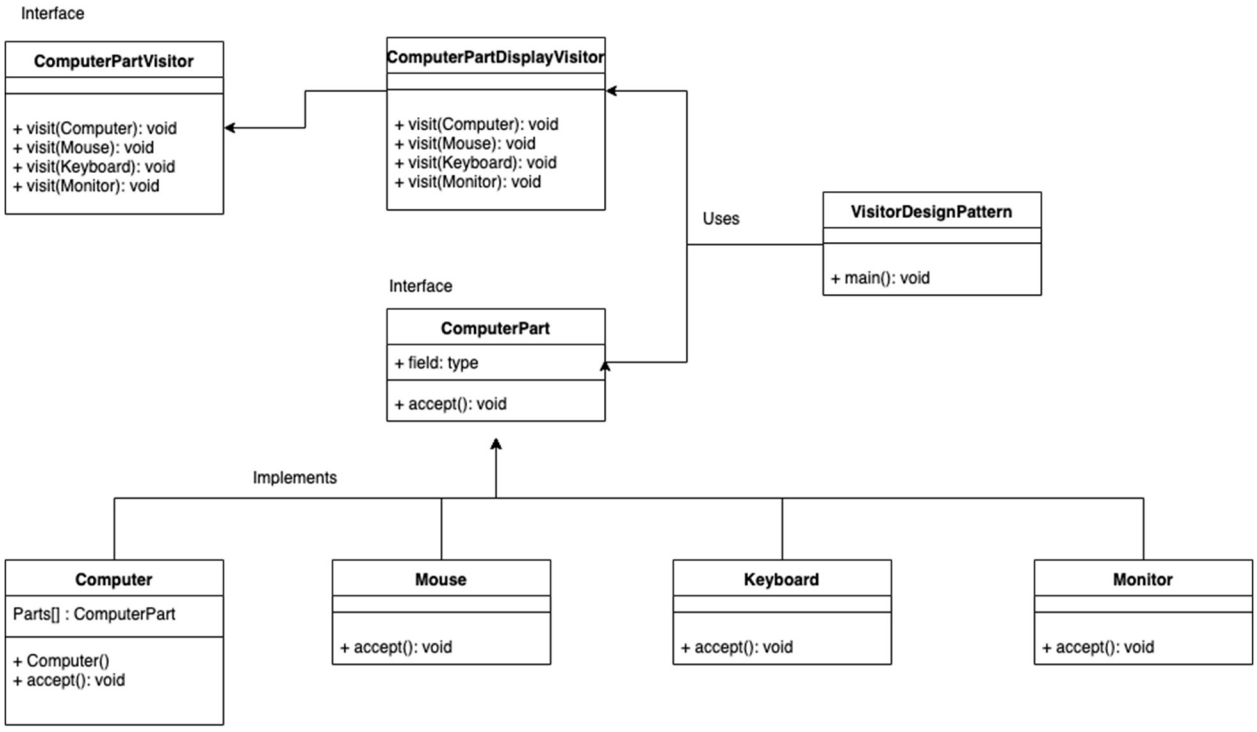
2.0 Topology

2.1 Definition

In a Visitor design pattern, a Visitor class is used, which changes the execution of an element class's algorithm. This pattern is used when we have to perform different operations on a group of similar objects. It consists of two parts:

- A *visit()* method which is implemented by a Visitor and is called for every element of the program
- Element classes that implement *accept()* methods which accept Visitors

2.2 Sample Structure



3.0 Consequences

Advantages:

- If logic of operations in Elements has to change i.e. we want to handle the Elements differently, we only need to modify code implemented in the *visit()* methods of the concrete Visitor classes
- Visitors can have and maintain state relating to the different Elements
- Adding new Elements does not affect current functionality implemented for the other Elements. This ensures that the Visitor pattern does not violate the Open/Closed principle

Disadvantages:

- The return type of the *visit()* methods must be known prior to implementation, in the designing phase, otherwise the signatures in the interface and all of the implemented *visit()* methods will have to be changed to adhere to the new return type
- As we increase the number of Visitor implementations, it makes the system difficult to maintain and extend. If a new Element is added, all the Visitors need to implement its *visit()* method, even if that particular implementation is just an empty function

- A Visitor has access to the Element object and can therefore modify its properties, which may result in unwanted side effects

4.0 Non-Functional Properties

- **Scalability** - Improves scalability in the case when new operations on Element classes are added frequently, and the program's class hierarchy consists of many unrelated classes.

- **Readability** - Improves readability as the code for the new operation is packed up in a Visitor class and not cluttered throughout the many Element classes

- **Maintainability** - Degrades maintainability in the case when we need to add new Elements to the program's class hierarchy. When a new Element is added, all existing Visitors must be updated with the new corresponding *visit()* method for processing this Element

- **Testability** - Degrades testability due to the extensive use of polymorphism and the fact that its implementation is based on double dispatch