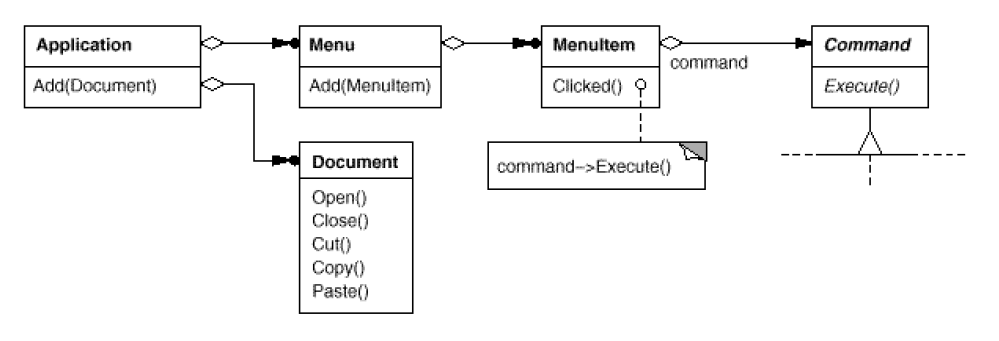
# Command

## **Intent**

* Encapsulate a request as an object to make easier customizations and decouple the invoker of the request from the entity that solves the request.

## **Motivation**

Ex: A UI application with documents where MenuItems execute commands defined by user. The UI library doesn’t know how to perform the commands so they will be executed by another entity (application, specified receiver etc).





* Let user customize the actions performed by a command inside an object. Commands are an object-oriented replacement for callbacks.
* Queue commands and execute them at at different time (see also the ActiveObject pattern as an example). Commands can be carried out in different address spaces / processes (e.g. CPU – GPU).
* Helps to support Undo/Redo operations in an application. Implementation example:
  + Each **Command** interface has two functions: Execute and Unexecute. The role of Unexecute is to revert the operations done previously by Execute
  + You keep a list of last N operations.
  + When user adds something, add at the back of the list.
  + When UNDO, move the head pointer one unit back without deleting the last element (and so on) and call the Unexecute function.
  + When REDO, move the head pointer on unit forward and call Execute function.
* Supports system recovery. For example, if a server crashes/gets an exception and you store the commands on a persistent storage (such as an HDD) you can go back and load / revert operations. Using the same Undo/Redo operations as above with Load / Unload function names instead of Execute / Unexecute.
* Useful for transaction systems (i.e. distributed systems).

## **Structure**

* Client (Application) : creates the receivers and several concrete commands by specifying the receiver they need to execute the operation.
* Invoker (Menu Item) : asks the command to carry out the request.
* Receiver (Document, Application): performs the actual operation of the ConcreteCommand



* Command decouples the object that invokes the operation from the one that knows how to perform it.
* Easy to add and compose commands (e.g. you can aggregate composite commands inside one like: open a document command would involve asking the user the path then performing the actual opening).

## **Implementation**

**Observations**:

* Commands can implement operations on their own without a receiver (e.g. spawn a new process).
* For Undo/Redo implementation user must be carefully about maintaining the internal state. Memento patter described below can help with this without exposing the internal state of the objects.
* See the code attached. Both Java and templated C++ implementation provided (the last one takes advantage in applications where commands don’t require arguments; we can use then C++ templates to avoid creating a Command subclass for every kind of action and receiver).

# Memento

## **Intent**

* Capture and restore the state of an object without violating encapsulation.

## **Motivation**

Use this pattern when:

* A snapshot of an object’s state must be saved so that it can be restored to that state later (
* A direct interface to obtaining the state would expose implementation details and break the object’s encapsulation.

E.g. of usage: Undo/Redo implementation in combination with Command pattern. Memento can store the state of the objects being modified.

## **Structure**



* Originator is the object that needs his state to be stored in a Memento. He is the only one who has access to memento object (E.g. in C++ / Java we ca use friend). No one else can modify its internal data structure (private interfaces).
* Caretaker is the object that keeps the memento objects (e.g. In a Undo/Redo impl, this is the owner of the history of lists).

## **Implementation**

<https://www.tutorialspoint.com/design_pattern/memento_pattern.htm>

# Visitor

## **Intent**

* Lets you define a new operation without changing the hierarchy / classes.

## **Motivation**

Consider an AST (abstract syntax tree) of a program. You have an hierarchy of elements each one with operations:

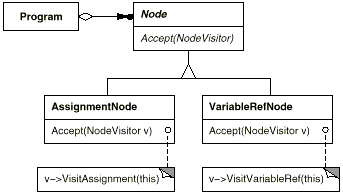


But there are two problems:

1. How do you add a new operation (e.g. a code generation operation for the assignment node )? You’ll have to modify the entire hierarchy.
2. Operations inside each element are unrelated (i.e. bad code decoupling).

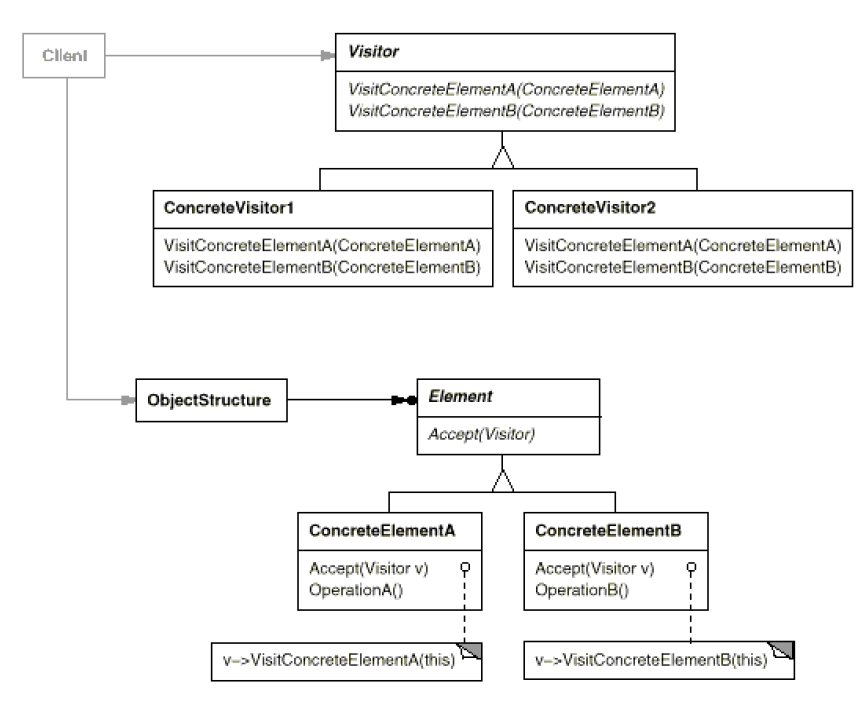
Solution:

* Separate the operations inside different classes (visitors). Have a base visitor class and multiple concrete visitors representing each class (type) of operation.
* 
* The nodes have to accept the visitor and dispatch further to visitor. The technique used is called **double dispatching (because operation depend both on the Visitor and Element !).** Explain how it works and difference between double and single dispatching.The implementation of concrete visitor operations generally call back nodes operations.



## **Structure**

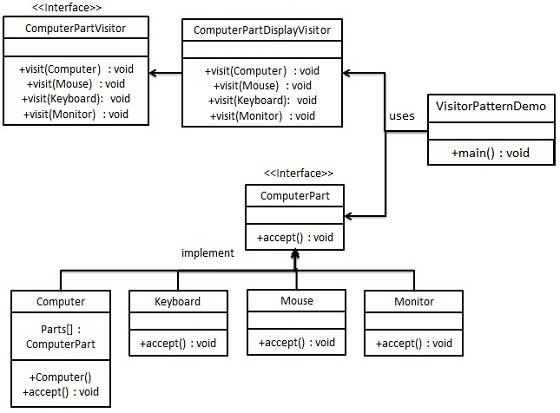
An easier way to view the decoupling of operations and elements:



* You can decouple the operations from the hierarchy of elements. This way you can easily add new operations.
* A visitor gathers related operations together in classes. Unrelated operations would be implemented in different classes. This increase decoupling and promotes easier maintenance. Also you can use the classes of visitor where you need in different applications.
* A nice thing is that visitor can store accumulation state (this would be messier without visitor since you’ll have to store the state somewhere and send it as arguments to the operations).
* Notice that if the structure of elements / hierarchy change often it’s better to not use visitor and let operation defined in the elements hierarchy.
* Notice the encapsulation breaking too...

## **Implementation**

* Computer parts and 2 visitors: A display visitor + Energy consumption accumulator visitor (only in code).



TODO many others: