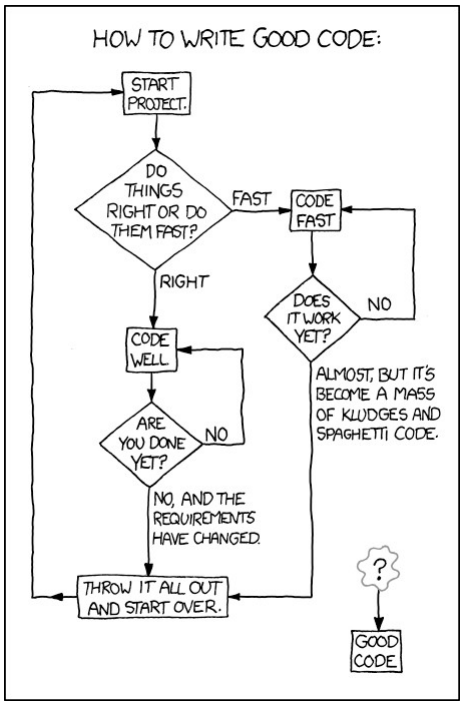


OBJECT-ORIENTED PROGRAMMING

Behavioral Design Patterns

Lecture #12

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Behavioral design patterns

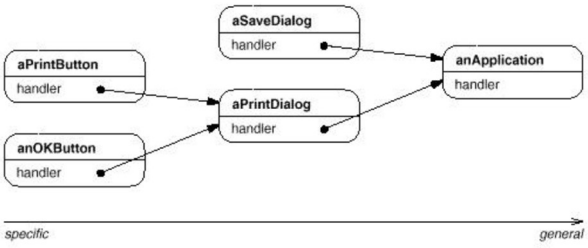
- Deal with dynamic interactions between classes and objects
- Plan for today
  - Chain of Responsibility
  - Iterator
  - Observer
  - Strategy
  - Visitor

Chain of Responsibility

- Purpose
  - Prevent a direct connection between the request sender and the recipient by creating a chain of objects processing the request
- Motivation
  - Contextual help functionality in GUI
  - The object that implements the help for a specific object (e.g. a button) is not directly known
  - We let the request pass through the entire chain of objects, from which the correct one is determined and performs the corresponding function
  - Each object in the chain has a uniform interface for receiving the request and a reference to the next object in the chain

Chain of Responsibility

- Motivation

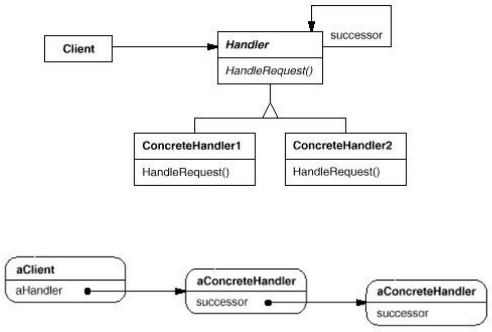


Chain of Responsibility

- Usage
  - When more than one object can service a request, and it is not known which specific object it is
  - When requests are based on a "serve or forward" model, where the request is either served directly in the object or needs to be sent to another object
- Implications
  - By preventing a direct link between the sender and receiver of the request, it allows for independent solutions
  - Response is not guaranteed - a request can reach the end of the chain without being served
  - The chain of service objects can be created dynamically

Chain of Responsibility

- Structure



Chain of Responsibility (example 1)

- Task
  - Software system for security monitoring. The system has multiple sensors (smoke, fire, motion, etc.) that transmit their status to a central computer
  - For each physical sensor, we create one instance of a sensor object
  - Each object knows that a certain value generated by a sensor represents some action, but the action itself also depends on data other than the sensor (e.g., sensor location, data from other devices located at the sensor location, etc.)
  - We want a scalable solution that can be deployed in any environment

Chain of Responsibility (example 1)

- Solution
  - We will use the Chain of Responsibility pattern
  - We will create a hierarchical composition of sensor objects that monitors a specific secured environment
  - We will define environment objects (wall, room, floor, building) as part of this hierarchical composition
  - We will send the alarm generated by the sensor at the top of this hierarchy and the corresponding composed object will perform the action

Chain of Responsibility (example 2)

- Main component of the pattern is `Handler` interface with one method `handleRequest()`

```
public interface Handler {
  void handleRequest();
}
```

- What if we want to handle more request types?

Chain of Responsibility (example 2)

- Solution #1
  - Extend `Handler` interface with more methods supporting more request types

```
public interface Handler {
  void handleHelp();
  void handlePrint();
  void handleFormat();
}
```

  - In this solution each concrete handler implementation must handle all request types

Chain of Responsibility (example 2)

- Example of concrete handler class

```
public class ConcreteHandler implements Handler {
  private Handler successor;
  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }
  @Override
  public void handleHelp() {
    // Request for help is handled here
  }
  @Override
  public void handlePrint() {
    this.successor.handlePrint();
  }
  @Override
  public void handleFormat() {
    this.successor.handleFormat();
  }
}
```

- When we want to add more request type handling later, we need to rewrite existing classes!

Chain of Responsibility (example 2)

- Solution #2
  - Each request type defines own handler interface

```
public interface HelpHandler {
  void handleHelp();
}
public interface PrintHandler {
  void handlePrint();
}
public interface FormatHandler {
  void handleFormat();
}
```

  - Now a concrete handler class can implement one or more handler interfaces as needed and must have an object of the next object in the chain for each type of request

Chain of Responsibility (example 2)

```
public class ConcreteHandler implements HelpHandler, PrintHandler, FormatHandler {
  private HelpHandler helpSuccessor;
  private PrintHandler printSuccessor;
  private FormatHandler formatSuccessor;

  public ConcreteHandler(HelpHandler helpSuccessor, PrintHandler printSuccessor,
   FormatHandler formatSuccessor) {
    this.helpSuccessor = helpSuccessor;
    this.printSuccessor = printSuccessor;
    this.formatSuccessor = formatSuccessor;
  }

  @Override
  public void handleHelp() {
    // Request for Help is handled here
  }
  // Forward other requests
  @Override
  public void handlePrint() { this.printSuccessor.handlePrint(); }
  @Override
  public void handleFormat() { this.formatSuccessor.handleFormat(); }
}
```

Chain of Responsibility (example 2)

- Solution #3
  - Another possible solution is to have only one service interface method that accepts the type of request being serviced as a parameter (string, enumeration, reflection type, etc.)

```
public interface Handler {
  void handleRequest(String request);
}
```


Chain of Responsibility (example 2)

- Concrete handler class is following

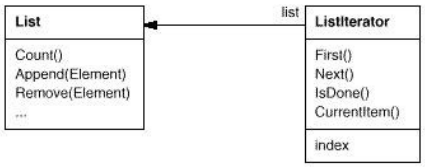
```
public class ConcreteHandler implements Handler {

  private Handler successor;

  public ConcreteHandler(Handler successor) {
    this.successor = successor;
  }

  @Override
  public void handleRequest(String request) {
    if (request.equals("Help")) {
      // Request for help is handled here
    }
    else {
      // Forward other requests
      this.successor.handle(request);
    }
  }
}
```

Iterator

- Purpose
  - Provide sequential access (cursor) to aggregated objects (collection, list) without knowing their representation
  - Also called Cursor
- Motivation
  - An aggregated object e.g. a list should provide a way to access individual contained elements without exposing its structure
  - It should provide different access methods
  - It should provide multiple access for parallel processing
  - We do not want to define these functions directly in the aggregated object

Iterator

- Motivation example #1
  - List with its cursor

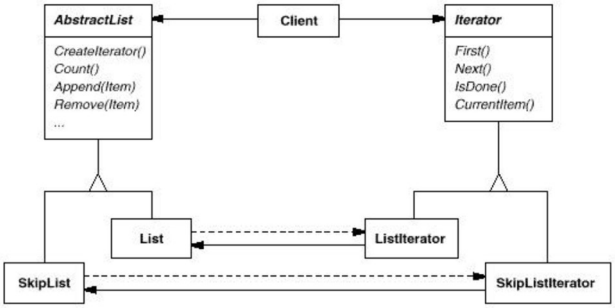


  - Typical client code

```
...
List list = new List();
...
ListIterator iterator = new ListIterator(list);
iterator.First();
while (!iterator.IsDone()) {
  Object item = iterator.CurrentItem();
  // Here we use selected item
  iterator.Next();
}
...
```

Iterator

- Motivation example #2
  - Polymorfic cursor



Iterator

- Typical client code

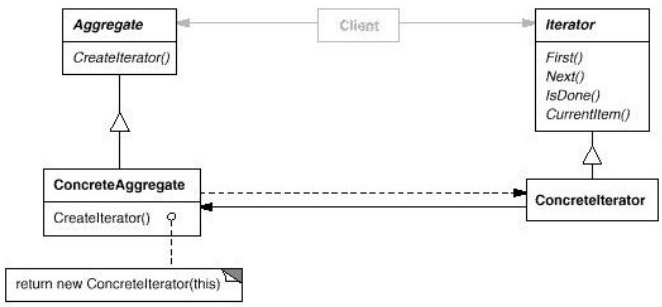
```
List list = new List();
SkipList skipList = new SkipList();
Iterator listIterator = list.CreateIterator();
Iterator skipListIterator = skipList.CreateIterator();
handleList(listIterator);
handleList(skipListIterator);
...
public void handleList(Iterator iterator) {
  iterator.First();
  while (!iterator.IsDone()) {
    Object item = iterator.CurrentItem();
    // Here we use selected item
    iterator.Next();
  }
}
```

Iterator

- Usage
  - Accessing elements of an aggregate object without using its actual structure
  - Need for multiple (parallel) access to elements of an aggregate object
  - Provide a uniform interface for accessing elements (i.e. polymorphic cursor)

Iterator

- Structure



Iterator

- Elements involved
  - `Iterator` – defines an interface for sequential access to elements
  - `ConcreteIterator` – implements the `Iterator` interface, maintains state of the aggregated object
  - `Aggregate` – defines an interface for creating a cursor object (using Factory Method)
  - `ConcreteAggregate` – implements the `Aggregate` interface and creates a concrete cursor object

Iterator

- Advantages
  - Simplifies the interface of an aggregate object by not including methods for accessing its elements
  - Supports multiple access
  - Supports various cursor variants

Iterator

- Implementation
  - Who controls the cursor traversal?
    - Client – more flexible, also called "external cursor"
    - Cursor itself – called "internal cursor"
  - Who defines the traversal algorithm?
    - Cursor – more general, easier if we want to have different cursor variants
    - Aggregate object – cursor only keeps the state of the traversal
  - Can the aggregate object be changed while the cursor is in use?
    - If so, we also call it "robust cursor"
  - Implementation of other operations such as `Previous()`?

Iterator (examples)

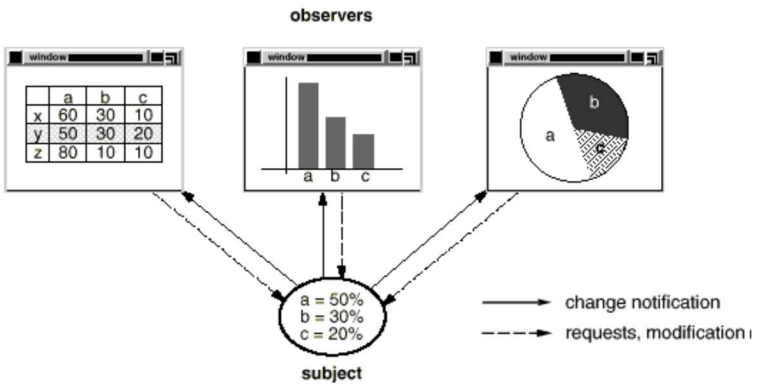
- Examples – see Java
  - `java.util.Enumeration`
  - `java.util.Hashtable`
  - `java.util.Collection`
  - `java.util.Iterator`
  - `java.util.LinkedList`
  - `java.util.List`
  - `java.util.ListIterator`
  - `java.util.Vector`
  - ...

Observer

- Purpose
  - Definition of a 1:N dependency between objects, if the state of an object changes, all dependent objects are informed of it
- Motivation
  - Need for consistency between dependent objects, without the need for a tight coupling between them

Observer

- Motivation

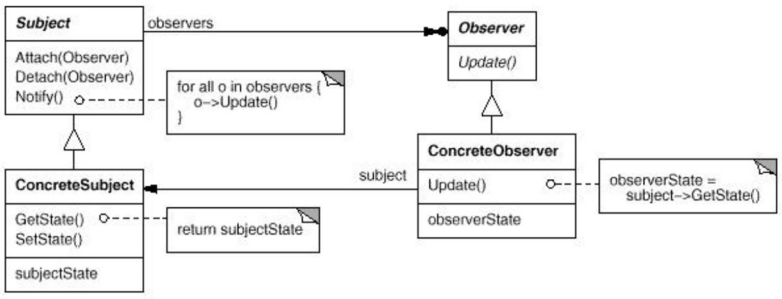


Observer

- Usage
  - Flexible implementation of dependencies between objects on their state
  - When changing one object should cause another object to change
  - When an object should inform other objects about its state without knowing these objects

Observer

- Structure

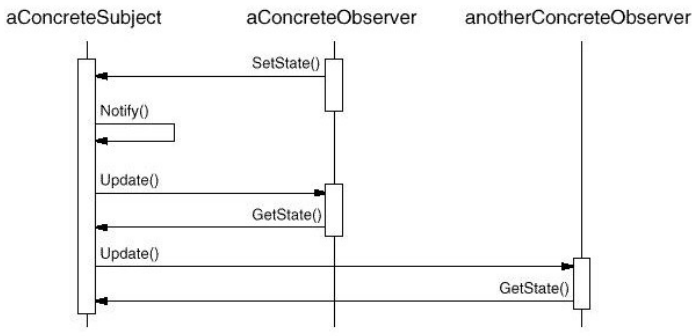


Observer

- Elements involved
  - `Subject` – maintains information about dependent objects, provides an interface for inserting and removing dependent objects
  - `Observer` – interface for implementing state change notifications
  - `ConcreteSubject` – a concrete class of the observed object, implements `Subject` and maintains references to `ConcreteObservers`, which notify about the change via the `Observer` interface
  - `ConcreteObserver` – a concrete class of the observing (dependent) object, its state is consistent with the state of the observed object, implements `Observer` and performs a change of its state after the notification is executed by the `ConcreteSubject`

Observer

- Cooperation



Observer

- Advantages
  - Minimal coupling between the monitored and dependent objects
    - Possibility of extending the monitored object without having to change the dependent objects
    - Adding dependent objects without having to change the monitored object
    - The monitored object only knows the list of dependent objects (not specific objects) and their interface for executing the notification
    - The monitored object and dependent objects can belong to different abstractions
  - Support for event distribution
    - The monitored object distributes a notification (event) to registered dependent objects
    - Dependent objects can freely register and unregister in the monitored object

Observer

- Disadvantages
  - Possibility of cascading object notifications
    - Dependent objects are usually unaware of each other and their actions may not be consistent with other dependent objects
  - Simple notification interface can make it difficult for dependent objects to distinguish which monitored object has actually changed

Observer

- Implementation
  - How does a monitored object maintain a set of dependent objects?
    - Array, link list, etc.
  - What if a dependent object wants to monitor multiple entities?
    - The monitored object must identify itself when notifying a dependent object
  - Who performs a state change on a dependent object?
    - The monitored object at each notification
    - The dependent object itself, e.g. when changing multiple states
    - Another object outside the given structure
  - Before the monitored object performs a notification, it **must** perform a state change
  - How much information should the monitored object send to the dependent objects when notified?
    - **Push architecture** – send all necessary information
    - **Pull architecture** – only identification, the monitored object requests (pulls) what it needs later

Observer

- Implementation
  - Can a dependent object only register for specific events?
    - If so, is this a publish-register model?
  - Can a dependent object also be a monitored object?
    - In principle, yes
  - What if a dependent object only wants to be notified when the state of multiple monitored objects changes?
    - Using a mediator object (Mediator pattern)
    - A monitored object only notifies the mediator object, which performs the necessary operations before notifying the dependent objects themselves

Observer (example 1)

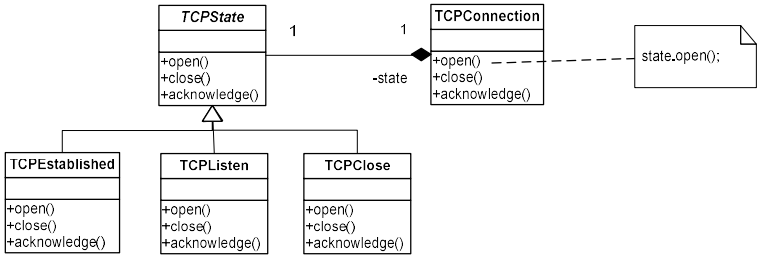
- Typical usage
  - Model-View-Controller (MVC) software architecture
    - Model – data processed by the application (observed objects)
    - View – presentation logic of the application that monitors data changes and presents results to the user
    - Controller – business logic of the application that causes the state of objects to change
  - MVC is the basis for OO frameworks for both web and desktop applications (Swing, Django, ASP.NET, etc.)

Observer (example 2)

- Common implementation in Jave
  - `java.util.Observer`
  - `java.util.Observable`

Strategy

- Purpose
  - Defines a group of similar algorithms (functions) that are encapsulated together and can be interchanged
  - This pattern allows for varying usage of these algorithms by clients
- Motivation

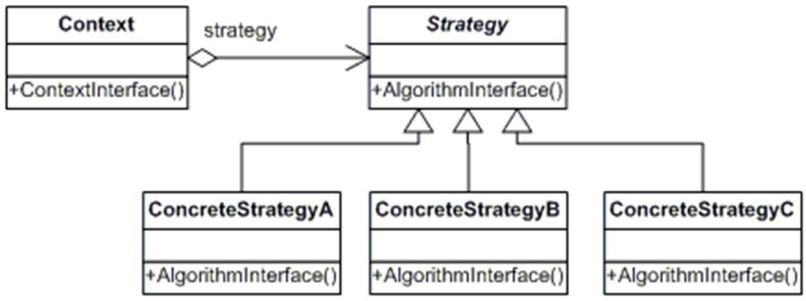


Strategy

- Usage
  - Related classes differ only in their behavior
  - We need different variants of some algorithm
  - The algorithm uses data that the client should not know about (Strategy can be used if we want to hide complex specific data structures of the algorithm)
  - The class defines many functions, and their execution is conditional
  - Instead of defining multiple conditional operations, it is better to define each such function in a separate class

Strategy

- Structure

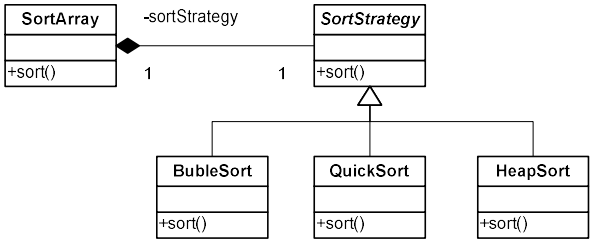


Strategy

- Advantages
  - Provides an alternative to inheritance, where a Context can contain different variations of algorithms or behavior
  - Eliminates large conditional functions
  - Provides the ability to choose an implementation for the same function

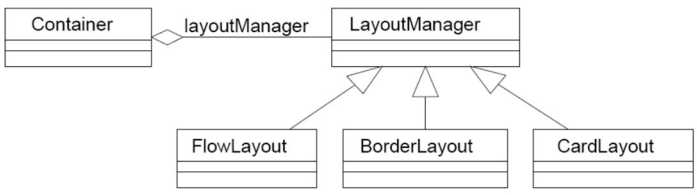
Strategy (example 1)

- The class wants to decide during execution which array sorting algorithm to use
- The solution is to encapsulate each algorithm in a separate class using the Strategy pattern



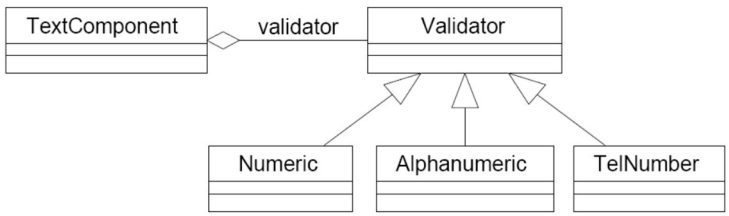
Strategy (example 2)

- A GUI object composed of other GUI components wants to decide which strategy to use for layout on the screen
- The solution is to encapsulate the different layout strategies into a separate class using the Strategy pattern



Strategy (example 3)

- A GUI component wants to decide how to validate the correctness of the input data entered by the user (numeric data, text, phone number)
- The solution is to encapsulate different data validation strategies into a single object using the Strategy pattern

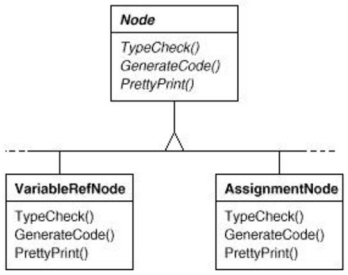


Visitor

- Purpose
  - Representation of an operation to be performed on each element of a structure, allows to extend operations without interfering with the structure of the elements
- Motivation
  - The compiler, which reads the source text of the program, creates an abstract syntax tree from it, this tree has various nodes (assignment, variable, expression)
  - Operations that we want to perform on the nodes
    - Check whether all used variables are defined
    - Check initialization of variables before their first use
    - Check type of variable (expression)
    - Generate (machine) code

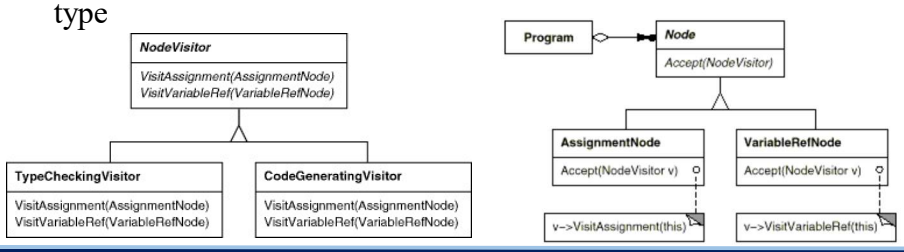
Visitor

- Motivation
  - Each of these operations is performed differently on different nodes
  - One way to do this is to create a dependent node structure (through inheritance)



Visitor

- Motivation
  - Problems
    - Adding a new operation requires changing the entire node structure
    - It is not clear to define operations directly in each node
  - Another approach is to define all operations in a special object called a "visitor", which will visit each node of the structure and the node will use it to perform the given operations according to its type

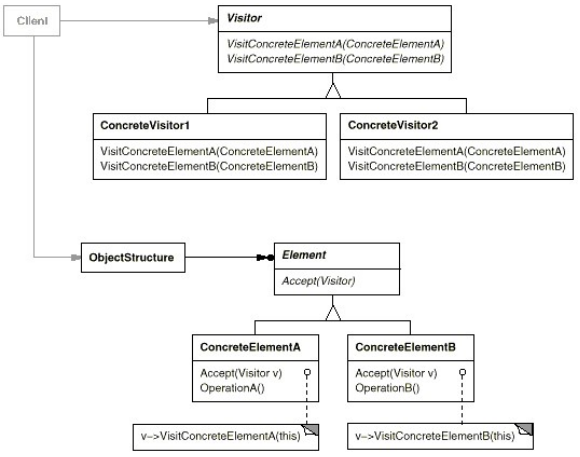


Visitor

- Usage
  - If we need to perform many different and independent operations on each element of some object structure
  - If the object structure does not change, but the operations performed on the elements of the object structure do change (otherwise it is better to define operations directly with objects)
  - If the object structure contains many classes with different interfaces and the operation execution depends on the type of each element of the object structure

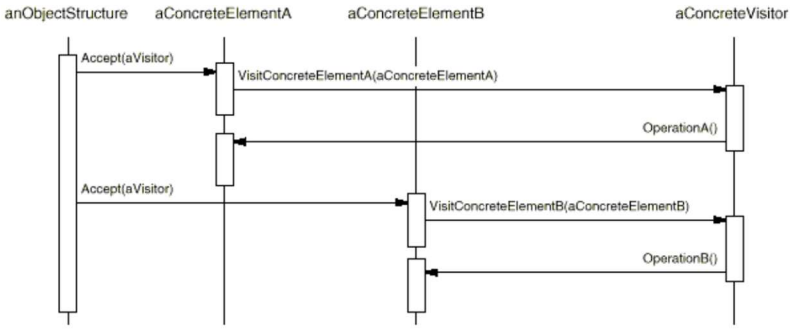
Visitor

- Structure



Visitor

- Components cooperation



Visitor

- Advantages
  - Adding a new operation is easy
  - Related operations are not spread across different classes, they are localized in one object, unrelated ones are spread across derived classes
  - Visitor object can maintain state of operations execution while traversing the entire object structure
- Disadvantages
  - Adding a new `ConcreteElement` is difficult, requires a new abstract operation in `Visitor` and implementation of `ConcreteVisitor`
  - `ConcreteElement` must have a very strong interface, sometimes it is necessary to break encapsulation to get the internal state of the object

Visitor (example)

- Combining Composite and Visitor patterns
- Definition of abstract component of a composite

```
public abstract class Component {
  private String name;
  public Component(String name) { this.name = name; }
  public String getName() { return this.name; }
  public void setName(String name) { this.name = name; }
  public abstract double getPrice();
  public abstract void accept(ComponentVisitor v);
}
```


Visitor (example)

- Definition of concrete component

```
public class Widget extends Component {
  private double price;
  public Widget(String name, double price) {
    super(name);
    this.price = price;
  }
  public void setPrice(double price) { this.price = price; }
  @Override
  public double getPrice() { return this.price; }
  @Override
  public void accept(ComponentVisitor v) { v.visit(this); }
}
```

Visitor (example)

- Definition of concrete composite

```
public class WidgetAssembly extends Component {
  private List<Component> components;
  public WidgetAssembly(String name) {
    super(name);
    this.components = new LinkedList<>();
  }
  public void addComponent(Component c) {
    this.components.add(c);
  }
  public void removeComponent(Component c) {
    this.components.remove(c);
  }
  @Override
  public double getPrice() {
    double totalPrice = 0.0;
    for (Component c : this.components)
      totalPrice += c.getPrice();
    return totalPrice;
  }
  @Override
  public void accept(ComponentVisitor v) { v.visit(this); }
}
```


Visitor (example)

- Interface of visitor object

```
public interface ComponentVisitor {
  void visit(Widget w);
  void visit(WidgetAssembly wa);
}
```

- If the structure of the composite changes, we also need to add a new `visit()` operation

Visitor (example)

- Implementing a simple visiting object operation (just lists which element was visited)

```
public class SimpleVisitor implements ComponentVisitor {
  @Override
  public void visit(Widget w) {
    System.out.println("Visiting Widget");
  }
  @Override
  public void visit(WidgetAssembly wa) {
    System.out.println("Visiting WidgetAssembly");
  }
}
```


Visitor (example)

- Another visitor object implementation for comparing prices of elements

```
public class PriceVisitor implements ComponentVisitor {
  private double maxPrice;
  public PriceVisitor(double maxPrice) { this.maxPrice = maxPrice; }
  @Override
  public void visit(Widget w) {
    double price = w.getPrice();
    if (price > this.maxPrice)
      System.out.println("Do not buy! Widget price " + price + " exceeds max price (" + this.maxPrice
 + ").");
    else
      System.out.println("Buy! Widget price " + price + " is lower than max price (" + this.maxPrice
 + ").");
  }
  @Override
  public void visit(WidgetAssembly wa) {
    double price = wa.getPrice();
    if (price > this.maxPrice)
      System.out.println("Do not buy! WidgetAssembly price " + price + " exceeds pax price (" +
 this.maxPrice + ").");
    else
      System.out.println("Buy! WidgetAssembly price " + price + " is lowed than max price (" +
 this.maxPrice + ").");
  }
}
```

Visitor (example)

- Test program

```
public class VisitorTest {
  public static void main(String[] args) {
    Widget w1 = new Widget("Widget1", 10.0);
    Widget w2 = new Widget("Widget2", 20.0);
    WidgetAssembly wa = new WidgetAssembly("Assembly");
    wa.addComponent(w1);
    wa.addComponent(w2);
    SimpleVisitor sv = new SimpleVisitor();
    w1.accept(sv);
    w2.accept(sv);
    wa.accept(sv);
    PriceVisitor pv = new PriceVisitor(25.0);
    w1.accept(pv);
    w2.accept(pv);
    wa.accept(pv);
  }
}
```


Visitor (example)

- Output of the program

```
Visiting Widget
Visiting Widget
Visiting WidgetAssembly
Buy! Widget price 10.0 is lower than max price (25.0).
Buy! Widget price 20.0 is lower than max price (25.0).
Do not buy! WidgetAssembly price 30.0 exceeds max price (25.0).
```