Behavioral Design Patterns
Behavioral Design Patterns

Creational Design Patterns
- Initialize objects or create new classes

Structural Design Patterns
- Compose objects to get new functions

Behavioral Design Patterns
- Communication between objects
Behavioral Design Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template
11. Visitor
1. Chain of Responsibility

• Pass requests to the chain of handlers
Transform Behavior into “handlers”

• Example: node.js
Example: node.js

- Callback function: next()
Chain of Responsibility Structure

- **Handler** declares the interface, common for all concrete handlers
- **Base Handler** is an optional class where you can put the boilerplate code
- **Concrete Handlers** contain the actual code for processing requests
Working with Composite Pattern

• Find the right class to do `showHelp()`
2. COMMAND

- Button
- «interface» Command
  - + execute()
- Shortcut

- Save Command
  - Code
- Open Command
  - Code
- Print Command
  - Code
Command Pattern

• Turn a request into a stand-alone object that contains all information
Command Structure

- **ConcreteCommand**
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
Collaboration

new Command(aReceiver)

StoreCommand(aCommand)

Action()

Execute()
3. Interpreter

- Given a language, define a representation for its grammar with an interpreter that uses the representation to interpret sentences in the language.
4. Iterator

- A pattern that traverses elements of a collection
/ std::iterator example
#include <iostream>   // std::cout
#include <iterator>   // std::iterator, std::input_iterator_tag

class MyIterator : public std::iterator<
    std::input_iterator_tag, int>
{
    int* p;

public:
    MyIterator(int* x) : p(x) {}  
    MyIterator(const MyIterator& mit) : p(mit.p) {}  
    MyIterator& operator++() { ++p; return *this; }  
    MyIterator operator++(int) { 
        MyIterator tmp(*this); 
        operator++(); 
        return tmp; }  
    bool operator==(const MyIterator& rhs) const { return p == rhs.p; }  
    bool operator!=(const MyIterator& rhs) const { return p != rhs.p; }  
    int& operator*() { return *p; }  
};

int main() {
    int numbers[] = { 10,20,30,40,50 };  
    MyIterator from(numbers);  
    MyIterator until(numbers + 5);  
    for (MyIterator it = from; it != until; it++)  
    std::cout << *it << ' ';
    std::cout << '\n';
    return 0;
}
5. Mediator (a.k.a. Intermediary, Controller)

- Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Example: Font Dialog

Diagram showing aClient, aFontDialogDirector, aListBox, and anEntryField, with interactions marked by arrows (e.g., ShowDialog(), WidgetChanged(), GetSelection(), SetText()).
6. Memento

- Save and restore the previous state of an object without revealing the details of its implementation
Memento Structure

- Memento: stores the internal state of the Originator
- Originator: creates a memento with a snapshot of its current state
- Caretaker: for memento’s safekeeping
Memento Collaborations
7. Observer

- Define a subscription mechanism to notify multiple objects
```java
for (Subscriber s : subscribers)
  s.update(this);

mainState = newState;
notifySubscribers();
mainBusinessLogic();

ConcreteSubscriber s = new ConcreteSubscriber();
publisher.subscribe(s);
```

Example:

The Internet of Things
Decoupling Producers & Consumers of M2M Device Data
8. STATE

- Let an object alter its behavior when its internal state changes
Example: TCP Connection

TCPConnection
- Open()
- Close()
- Acknowledge()

state → Open()

TCPState
- Open()
- Close()
- Acknowledge()

TCPEstablished
- Open()
- Close()
- Acknowledge()

TCPLlisten
- Open()
- Close()
- Acknowledge()

TCPclosed
- Open()
- Close()
- Acknowledge()
State Structure
9. Strategy

- Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable
Example: Text Editor
Strategy Structure

- Context
  - ContextInterface()

- Strategy
  - AlgorithmInterface()

- ConcreteStrategyA
  - AlgorithmInterface()

- ConcreteStrategyB
  - AlgorithmInterface()

- ConcreteStrategyC
  - AlgorithmInterface()
10. Template

- Defines the skeleton of an algorithm and let subclasses override specific steps
AbstractClass

+ templateMethod()
+ step1()
+ step2()
+ step3()
+ step4()

step1()
if (step2()) {
    step3()
} else {
    step4()
}

ConcreteClass1

+ step3()
+ step4()

ConcreteClass2

+ step1()
+ step2()
+ step3()
+ step4()
11. Visitor

• Separate algorithms from the objects on which they operate
Visitor vs. Iterator

- Visitor Pattern is used to perform an action on a structure of elements

```csharp
public void VisitorExample()
{
    MyVisitorImplementation visitor = new MyVisitorImplementation();
    List<object> myListToHide = GetList();

    //Here you hide that the aggregate is a List<object>
    ConcreteIterator i = new ConcreteIterator(myListToHide);

    IAcceptor item = i.First();
    while (item != null)
    {
        item.Accept(visitor);
        item = i.Next();
    }
    //... do something with the result
}
```
The diagram illustrates the Visitor design pattern. It consists of the following components:

1. **Visitor Interface**
   - `Visitor` interface with methods `visit(c: ComponentA)` and `visit(c: ComponentB)`.

2. **Concrete Visitor**
   - `ConcreteVisitors` class with methods `visit(c: ComponentA)` and `visit(c: ComponentB)`.
   - Note: Visitor methods know the concrete type of the component it works with. Example: `c.featureB()`.

3. **Component Interface**
   - `Component` interface with methods `accept(v: Visitor)`.

4. **Concrete Components**
   - `ComponentA` with methods `featureA()` and `accept(v: Visitor)`.
   - `ComponentB` with methods `featureB()` and `accept(v: Visitor)`.

5. **Client**
   - The client calls `component.accept(new ConcreteVisitor())` to apply the visitor pattern.
References

• Alexander Shvets, “Dive into Design Patterns,” 2018
• https://www.tutorialspoint.com/design_pattern/index.htm
• Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns,” 1994