Lab 4: Tìm hiểu và cài đặt nhóm mẫu Behavioral (5 tiết)

Yêu cầu:

- Sinh viên đọc hiểu rõ mục đích, ý nghĩa và áp dụng ứng dụng của nhóm mẫu cấu trúc.
- Sử dụng Visual Studio cài đặc nhóm mẫu trên.
- Nộp bài báo cáo: Mỗi parttern hãy lấy 2 ví dụ thể hiện bằng sơ đồ lớp (Class diagram)

Structural Patterns:

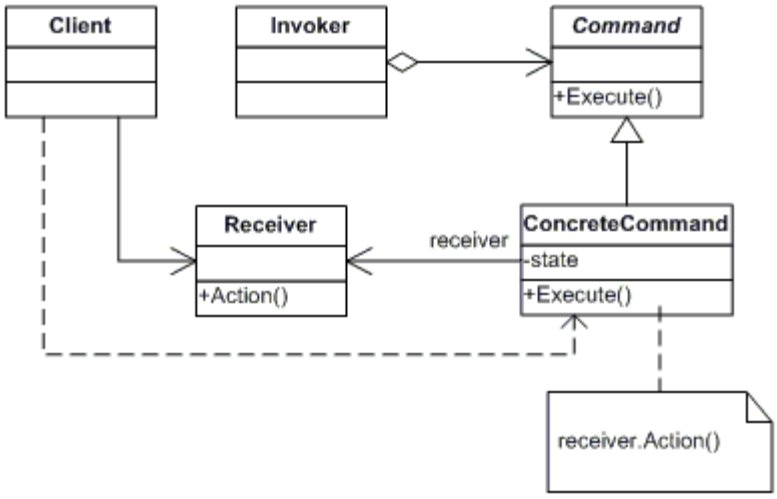
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

1. Command

Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Command (Command)**
  - declares an interface for executing an operation
- **ConcreteCommand (CalculatorCommand)**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (CommandApp)**
  - creates a ConcreteCommand object and sets its receiver
- **Invoker (User)**
  - asks the command to carry out the request
- **Receiver (Calculator)**
  - knows how to perform the operations associated with carrying out the request.

Sample code in C#

This structural code demonstrates the Command pattern which stores requests as objects allowing clients to execute or playback the requests.

```
// Command pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Command.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Command Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
```

```
  static void Main()
  {
    // Create receiver, command, and invoker
    Receiver receiver = new Receiver();
    Command command = new ConcreteCommand(receiver);
    Invoker invoker = new Invoker();

    // Set and execute command
    invoker.SetCommand(command);
    invoker.ExecuteCommand();

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Command' abstract class
/// </summary>
abstract class Command
{
  protected Receiver receiver;

  // Constructor
  public Command(Receiver receiver)
  {
    this.receiver = receiver;
  }

  public abstract void Execute();
}

/// <summary>
/// The 'ConcreteCommand' class
/// </summary>
class ConcreteCommand : Command
{
  // Constructor
  public ConcreteCommand(Receiver receiver) :
    base(receiver)
  {
  }

  public override void Execute()
  {
    receiver.Action();
  }
}

/// <summary>
/// The 'Receiver' class
```

```
/// </summary>
class Receiver
{
  public void Action()
  {
    Console.WriteLine("Called Receiver.Action()");
  }
}

/// <summary>
/// The 'Invoker' class
/// </summary>
class Invoker
{
  private Command _command;

  public void SetCommand(Command command)
  {
    this._command = command;
  }

  public void ExecuteCommand()
  {
    _command.Execute();
  }
 }
}
```

Output

```
Called Receiver.Action()
```

This real-world code demonstrates the Command pattern used in a simple calculator with unlimited number of undo's and redo's. Note that in C# the word 'operator' is a keyword. Prefixing it with '@' allows using it as an identifier.

```
// Command pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Command.RealWorld
{
 /// <summary>
 /// MainApp startup class for Real-World
 /// Command Design Pattern.
 /// </summary>
 class MainApp
 {
  /// <summary>
```

```
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    // Create user and let her compute
    User user = new User();

    // User presses calculator buttons
    user.Compute('+', 100);
    user.Compute('-', 50);
    user.Compute('*', 10);
    user.Compute('/', 2);

    // Undo 4 commands
    user.Undo(4);

    // Redo 3 commands
    user.Redo(3);

    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Command' abstract class
/// </summary>
abstract class Command
{
  public abstract void Execute();
  public abstract void UnExecute();
}

/// <summary>
/// The 'ConcreteCommand' class
/// </summary>
class CalculatorCommand : Command
{
  private char _operator;
  private int _operand;
  private Calculator _calculator;

  // Constructor
  public CalculatorCommand(Calculator calculator,
    char @operator, int operand)
  {
    this._calculator = calculator;
    this._operator = @operator;
    this._operand = operand;
  }
```

```
  // Gets operator
  public char Operator
  {
    set { _operator = value; }
  }

  // Get operand
  public int Operand
  {
    set { _operand = value; }
  }

  // Execute new command
  public override void Execute()
  {
    _calculator.Operation(_operator, _operand);
  }

  // Unexecute last command
  public override void UnExecute()
  {
    _calculator.Operation(Undo(_operator), _operand);
  }

  // Returns opposite operator for given operator
  private char Undo(char @operator)
  {
    switch (@operator)
    {
      case '+': return '-';
      case '-': return '+';
      case '*': return '/';
      case '/': return '*';
      default: throw new
        ArgumentException("@operator");
    }
  }
}

/// <summary>
/// The 'Receiver' class
/// </summary>
class Calculator
{
  private int _curr = 0;

  public void Operation(char @operator, int operand)
  {
    switch (@operator)
    {
      case '+': _curr += operand; break;
```

```
      case '-': _curr -= operand; break;
      case '*': _curr *= operand; break;
      case '/': _curr /= operand; break;
    }
    Console.WriteLine(
      "Current value = {0,3} (following {1} {2})",
      _curr, @operator, operand);
  }
}

/// <summary>
/// The 'Invoker' class
/// </summary>
class User
{
  // Initializers
  private Calculator _calculator = new Calculator();
  private List<Command> _commands = new List<Command>();
  private int _current = 0;

  public void Redo(int levels)
  {
    Console.WriteLine("\n---- Redo {0} levels ", levels);
    // Perform redo operations
    for (int i = 0; i < levels; i++)
    {
      if (_current < _commands.Count - 1)
      {
        Command command = _commands[_current++];
        command.Execute();
      }
    }
  }

  public void Undo(int levels)
  {
    Console.WriteLine("\n---- Undo {0} levels ", levels);
    // Perform undo operations
    for (int i = 0; i < levels; i++)
    {
      if (_current > 0)
      {
        Command command = _commands[--_current] as Command;
        command.UnExecute();
      }
    }
  }

  public void Compute(char @operator, int operand)
  {
    // Create command operation and execute it
```

```
    Command command = new CalculatorCommand(
      _calculator, @operator, operand);
    command.Execute();

    // Add command to undo list
    _commands.Add(command);
    _current++;
   }
  }
}
```

Output

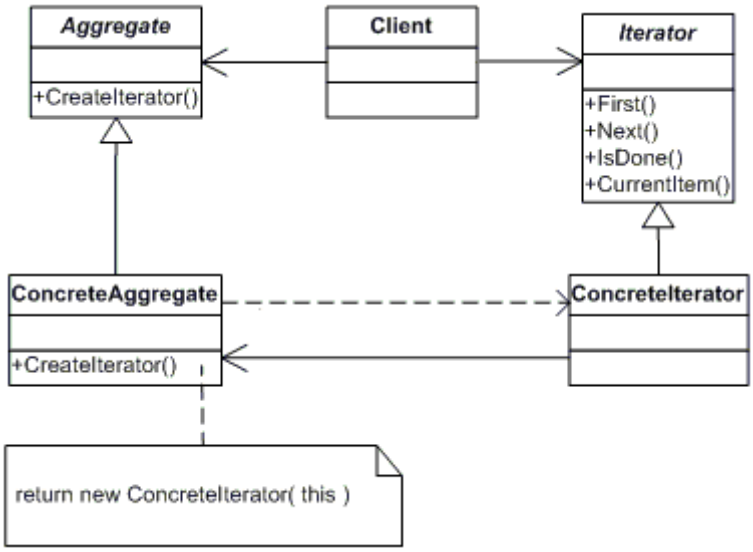
```
Current value = 100 (following + 100)
Current value =  50 (following - 50)
Current value = 500 (following * 10)
Current value = 250 (following / 2)

---- Undo 4 levels
Current value = 500 (following * 2)
Current value =  50 (following / 10)
Current value = 100 (following + 50)
Current value =   0 (following - 100)

---- Redo 3 levels
Current value = 100 (following + 100)
Current value =  50 (following - 50)
Current value = 500 (following * 10)
```

2. Iterator

Definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Iterator (AbstractIterator)**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator (Iterator)**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate (AbstractCollection)**
  - defines an interface for creating an Iterator object
- **ConcreteAggregate (Collection)**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Sample code in C#

This structural code demonstrates the Iterator pattern which provides for a way to traverse (iterate) over a collection of items without detailing the underlying structure of the collection.

```
// Iterator pattern -- Structural example

using System;
using System.Collections;

namespace DoFactory.GangOfFour.Iterator.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Iterator Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
```

```
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ConcreteAggregate a = new ConcreteAggregate();
      a[0] = "Item A";
      a[1] = "Item B";
      a[2] = "Item C";
      a[3] = "Item D";

      // Create Iterator and provide aggregate
      ConcreteIterator i = new ConcreteIterator(a);

      Console.WriteLine("Iterating over collection:");

      object item = i.First();
      while (item != null)
      {
        Console.WriteLine(item);
        item = i.Next();
      }

      // Wait for user
      Console.ReadKey();
    }
}

/// <summary>
/// The 'Aggregate' abstract class
/// </summary>
abstract class Aggregate
{
  public abstract Iterator CreateIterator();
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class ConcreteAggregate : Aggregate
{
  private ArrayList _items = new ArrayList();

  public override Iterator CreateIterator()
  {
    return new ConcreteIterator(this);
  }

  // Gets item count
  public int Count
  {
    get { return _items.Count; }
```

```
    }

    // Indexer
    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Insert(index, value); }
    }
}

/// <summary>
/// The 'Iterator' abstract class
/// </summary>
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

/// <summary>
/// The 'ConcreteIterator' class
/// </summary>
class ConcreteIterator : Iterator
{
    private ConcreteAggregate _aggregate;
    private int _current = 0;

    // Constructor
    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this._aggregate = aggregate;
    }

    // Gets first iteration item
    public override object First()
    {
        return _aggregate[0];
    }

    // Gets next iteration item
    public override object Next()
    {
        object ret = null;
        if (_current < _aggregate.Count - 1)
        {
            ret = _aggregate[++_current];
        }

        return ret;
```

```
        }

        // Gets current iteration item
        public override object CurrentItem()
        {
            return _aggregate[_current];
        }

        // Gets whether iterations are complete
        public override bool IsDone()
        {
            return _current >= _aggregate.Count;
        }
    }
}
```

Output

```
Iterating over collection:
Item A
Item B
Item C
Item D
```

This real-world code demonstrates the Iterator pattern which is used to iterate over a collection of items and skip a specific number of items each iteration.

```
// Iterator pattern -- Real World example

using System;
using System.Collections;

namespace DoFactory.GangOfFour.Iterator.RealWorld
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Iterator Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Build a collection
            Collection collection = new Collection();
            collection[0] = new Item("Item 0");
            collection[1] = new Item("Item 1");
            collection[2] = new Item("Item 2");
            collection[3] = new Item("Item 3");
```

```
        collection[4] = new Item("Item 4");
        collection[5] = new Item("Item 5");
        collection[6] = new Item("Item 6");
        collection[7] = new Item("Item 7");
        collection[8] = new Item("Item 8");

        // Create iterator
        Iterator iterator = new Iterator(collection);

        // Skip every other item
        iterator.Step = 2;

        Console.WriteLine("Iterating over collection:");

        for (Item item = iterator.First();
            !iterator.IsDone; item = iterator.Next())
        {
            Console.WriteLine(item.Name);
        }

        // Wait for user
        Console.ReadKey();
    }
}

/// <summary>
/// A collection item
/// </summary>
class Item
{
    private string _name;

    // Constructor
    public Item(string name)
    {
        this._name = name;
    }

    // Gets name
    public string Name
    {
        get { return _name; }
    }
}

/// <summary>
/// The 'Aggregate' interface
/// </summary>
interface IAbstractCollection
{
    Iterator CreateIterator();
```

```
}

/// <summary>
/// The 'ConcreteAggregate' class
/// </summary>
class Collection : IAbstractCollection
{
   private ArrayList _items = new ArrayList();

   public Iterator CreateIterator()
   {
      return new Iterator(this);
   }

   // Gets item count
   public int Count
   {
      get { return _items.Count; }
   }

   // Indexer
   public object this[int index]
   {
      get { return _items[index]; }
      set { _items.Add(value); }
   }
}

/// <summary>
/// The 'Iterator' interface
/// </summary>
interface IAbstractIterator
{
   Item First();
   Item Next();
   bool IsDone { get; }
   Item CurrentItem { get; }
}

/// <summary>
/// The 'ConcreteIterator' class
/// </summary>
class Iterator : IAbstractIterator
{
   private Collection _collection;
   private int _current = 0;
   private int _step = 1;

   // Constructor
   public Iterator(Collection collection)
   {
```

```
            this._collection = collection;
        }

        // Gets first item
        public Item First()
        {
            _current = 0;
            return _collection[_current] as Item;
        }

        // Gets next item
        public Item Next()
        {
            _current += _step;
            if (!IsDone)
                return _collection[_current] as Item;
            else
                return null;
        }

        // Gets or sets stepsize
        public int Step
        {
            get { return _step; }
            set { _step = value; }
        }

        // Gets current iterator item
        public Item CurrentItem
        {
            get { return _collection[_current] as Item; }
        }

        // Gets whether iteration is complete
        public bool IsDone
        {
            get { return _current >= _collection.Count; }
        }
    }
}
```

Output

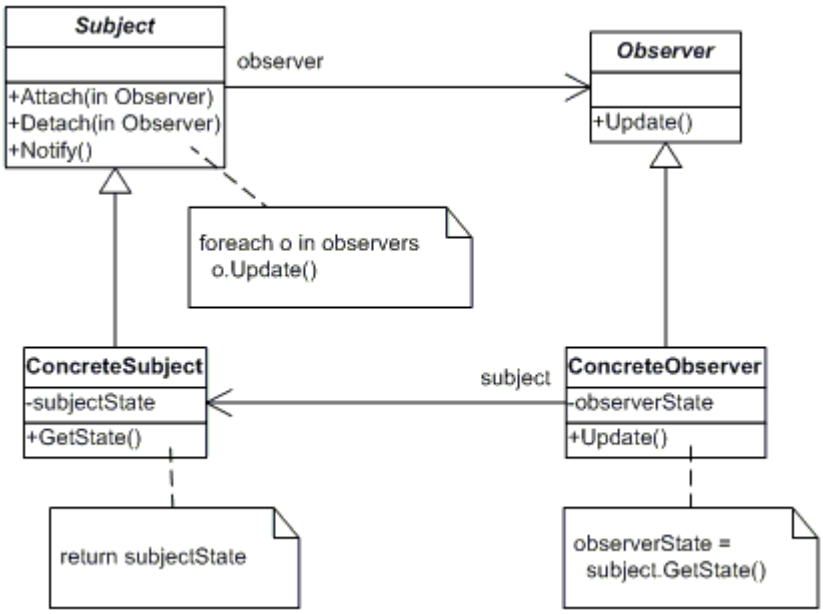
```
Iterating over collection:
Item 0
Item 2
Item 4
Item 6
Item 8
```

3. Observer

Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

UML class diagram



Participants

The classes and/or objects participating in this pattern are:

- **Subject (Stock)**
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- **ConcreteSubject (IBM)**
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- **Observer (IInvestor)**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteObserver (Investor)**
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - implements the Observer updating interface to keep its state consistent with the subject's

Sample code in C#

This structural code demonstrates the Observer pattern in which registered objects are notified of and updated with a state change.

```
// Observer pattern -- Structural example
```

```
using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Observer.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Observer Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Configure Observer pattern
      ConcreteSubject s = new ConcreteSubject();

      s.Attach(new ConcreteObserver(s, "X"));
      s.Attach(new ConcreteObserver(s, "Y"));
      s.Attach(new ConcreteObserver(s, "Z"));

      // Change subject and notify observers
      s.SubjectState = "ABC";
      s.Notify();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Subject' abstract class
  /// </summary>
  abstract class Subject
  {
    private List<Observer> _observers = new List<Observer>();

    public void Attach(Observer observer)
    {
      _observers.Add(observer);
    }

    public void Detach(Observer observer)
    {
      _observers.Remove(observer);
    }

    public void Notify()
    {
```

```
    foreach (Observer o in _observers)
    {
      o.Update();
    }
  }
}

/// <summary>
/// The 'ConcreteSubject' class
/// </summary>
class ConcreteSubject : Subject
{
  private string _subjectState;

  // Gets or sets subject state
  public string SubjectState
  {
    get { return _subjectState; }
    set { _subjectState = value; }
  }
}

/// <summary>
/// The 'Observer' abstract class
/// </summary>
abstract class Observer
{
  public abstract void Update();
}

/// <summary>
/// The 'ConcreteObserver' class
/// </summary>
class ConcreteObserver : Observer
{
  private string _name;
  private string _observerState;
  private ConcreteSubject _subject;

  // Constructor
  public ConcreteObserver(
    ConcreteSubject subject, string name)
  {
    this._subject = subject;
    this._name = name;
  }

  public override void Update()
  {
    _observerState = _subject.SubjectState;
    Console.WriteLine("Observer {0}'s new state is {1}",
```

```
      _name, _observerState);
    }

    // Gets or sets subject
    public ConcreteSubject Subject
    {
      get { return _subject; }
      set { _subject = value; }
    }
  }
}
```

Output

```
Observer X's new state is ABC
Observer Y's new state is ABC
Observer Z's new state is ABC
```

This real-world code demonstrates the Observer pattern in which registered investors are notified every time a stock changes value.

```
// Observer pattern -- Real World example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Observer.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Observer Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Create IBM stock and attach investors
      IBM ibm = new IBM("IBM", 120.00);
      ibm.Attach(new Investor("Sorros"));
      ibm.Attach(new Investor("Berkshire"));

      // Fluctuating prices will notify investors
      ibm.Price = 120.10;
      ibm.Price = 121.00;
      ibm.Price = 120.50;
      ibm.Price = 120.75;
```

```
    // Wait for user
    Console.ReadKey();
  }
}

/// <summary>
/// The 'Subject' abstract class
/// </summary>
abstract class Stock
{
  private string _symbol;
  private double _price;
  private List<IInvestor> _investors = new List<IInvestor>();

  // Constructor
  public Stock(string symbol, double price)
  {
    this._symbol = symbol;
    this._price = price;
  }

  public void Attach(IInvestor investor)
  {
    _investors.Add(investor);
  }

  public void Detach(IInvestor investor)
  {
    _investors.Remove(investor);
  }

  public void Notify()
  {
    foreach (IInvestor investor in _investors)
    {
      investor.Update(this);
    }

    Console.WriteLine("");
  }

  // Gets or sets the price
  public double Price
  {
    get { return _price; }
    set
    {
      if (_price != value)
      {
        _price = value;
        Notify();
```

```
      }
    }
  }

  // Gets the symbol
  public string Symbol
  {
    get { return _symbol; }
  }
}

/// <summary>
/// The 'ConcreteSubject' class
/// </summary>
class IBM : Stock
{
  // Constructor
  public IBM(string symbol, double price)
    : base(symbol, price)
  {
  }
}

/// <summary>
/// The 'Observer' interface
/// </summary>
interface IInvestor
{
  void Update(Stock stock);
}

/// <summary>
/// The 'ConcreteObserver' class
/// </summary>
class Investor : IInvestor
{
  private string _name;
  private Stock _stock;

  // Constructor
  public Investor(string name)
  {
    this._name = name;
  }

  public void Update(Stock stock)
  {
    Console.WriteLine("Notified {0} of {1}'s " +
      "change to {2:C}", _name, stock.Symbol, stock.Price);
  }
```

```
  // Gets or sets the stock
  public Stock Stock
  {
    get { return _stock; }
    set { _stock = value; }
  }
 }
}
```

Output

```
Notified Sorros of IBM's change to $120.10
Notified Berkshire of IBM's change to $120.10

Notified Sorros of IBM's change to $121.00
Notified Berkshire of IBM's change to $121.00

Notified Sorros of IBM's change to $120.50
Notified Berkshire of IBM's change to $120.50

Notified Sorros of IBM's change to $120.75
Notified Berkshire of IBM's change to $120.75
```