Lab 5: 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:

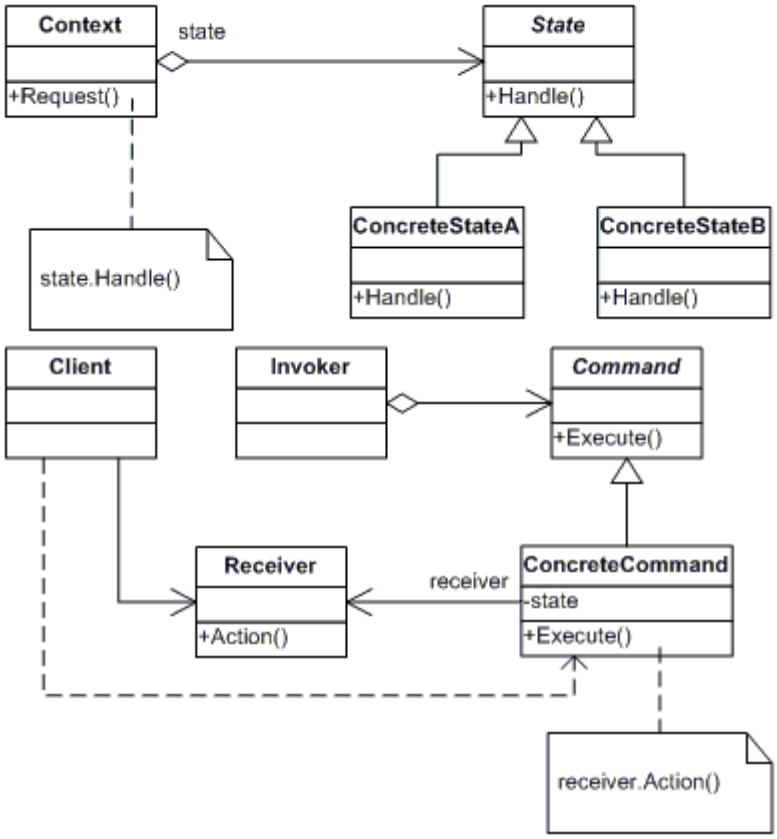
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	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. State

Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

UML class diagram



Participants

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

- **Context (Account)**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State (State)**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State (RedState, SilverState, GoldState)**
  - each subclass implements a behavior associated with a state of Context

Sample code in C#

This structural code demonstrates the State pattern which allows an object to behave differently depending on its internal state. The difference in behavior is delegated to objects that represent this state.

```
// State pattern -- Structural example

using System;


namespace DoFactory.GangOfFour.State.Structural
```

```
{
  /// <summary>
  /// MainApp startup class for Structural
  /// State Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Setup context in a state
      Context c = new Context(new ConcreteStateA());

      // Issue requests, which toggles state
      c.Request();
      c.Request();
      c.Request();
      c.Request();

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

  /// <summary>
  /// The 'State' abstract class
  /// </summary>
  abstract class State
  {
    public abstract void Handle(Context context);
  }

  /// <summary>
  /// A 'ConcreteState' class
  /// </summary>
```

```
class ConcreteStateA : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateB();
  }
}

/// <summary>
/// A 'ConcreteState' class
/// </summary>
class ConcreteStateB : State
{
  public override void Handle(Context context)
  {
    context.State = new ConcreteStateA();
  }
}

/// <summary>
/// The 'Context' class
/// </summary>
class Context
{
  private State _state;

  // Constructor
  public Context(State state)
  {
    this.State = state;
  }

  // Gets or sets the state
  public State State
  {
    get { return _state; }
    set
```

```
      {
        _state = value;
        Console.WriteLine("State: " +
          _state.GetType().Name);
      }
    }

    public void Request()
    {
      _state.Handle(this);
    }
  }
}
```

Output

```
State: ConcreteStateA
State: ConcreteStateB
State: ConcreteStateA
State: ConcreteStateB
State: ConcreteStateA
```

This real-world code demonstrates the State pattern which allows an Account to behave differently depending on its balance. The difference in behavior is delegated to State objects called RedState, SilverState and GoldState. These states represent overdrawn accounts, starter accounts, and accounts in good standing.

```
//State pattern -- Real World example

using System;


namespace DoFactory.GangOfFour.State.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// State Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
```

```
  static void Main()
  {
    // Open a new account
    Account account = new Account("Jim Johnson");

    // Apply financial transactions
    account.Deposit(500.0);
    account.Deposit(300.0);
    account.Deposit(550.0);
    account.PayInterest();
    account.Withdraw(2000.00);
    account.Withdraw(1100.00);

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

/// <summary>
/// The 'State' abstract class
/// </summary>
abstract class State
{
  protected Account account;
  protected double balance;

  protected double interest;
  protected double lowerLimit;
  protected double upperLimit;

  // Properties
  public Account Account
  {
    get { return account; }
    set { account = value; }
  }
```

```
  public double Balance
  {
    get { return balance; }
    set { balance = value; }
  }

  public abstract void Deposit(double amount);
  public abstract void Withdraw(double amount);
  public abstract void PayInterest();
}



/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Red indicates that account is overdrawn
/// </remarks>
/// </summary>
class RedState : State
{
  private double _serviceFee;

  // Constructor
  public RedState(State state)
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  private void Initialize()
  {
    // Should come from a datasource
    interest = 0.0;
    lowerLimit = -100.0;
    upperLimit = 0.0;
    _serviceFee = 15.00;
```

```
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    amount = amount - _serviceFee;
    Console.WriteLine("No funds available for withdrawal!");
  }

  public override void PayInterest()
  {
    // No interest is paid
  }

  private void StateChangeCheck()
  {
    if (balance > upperLimit)
    {
      account.State = new SilverState(this);
    }
  }
}

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Silver indicates a non-interest bearing state
/// </remarks>
/// </summary>
class SilverState : State
{
  // Overloaded constructors
```

```
public SilverState(State state) :
  this(state.Balance, state.Account)
{
}

public SilverState(double balance, Account account)
{
  this.balance = balance;
  this.account = account;
  Initialize();
}

private void Initialize()
{
  // Should come from a datasource
  interest = 0.0;
  lowerLimit = 0.0;
  upperLimit = 1000.0;
}

public override void Deposit(double amount)
{
  balance += amount;
  StateChangeCheck();
}

public override void Withdraw(double amount)
{
  balance -= amount;
  StateChangeCheck();
}

public override void PayInterest()
{
  balance += interest * balance;
  StateChangeCheck();
```

```
  }

  private void StateChangeCheck()
  {
    if (balance < lowerLimit)
    {
      account.State = new RedState(this);
    }
    else if (balance > upperLimit)
    {
      account.State = new GoldState(this);
    }
  }
}

/// <summary>
/// A 'ConcreteState' class
/// <remarks>
/// Gold indicates an interest bearing state
/// </remarks>
/// </summary>
class GoldState : State
{
  // Overloaded constructors
  public GoldState(State state)
    : this(state.Balance, state.Account)
  {
  }

  public GoldState(double balance, Account account)
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  private void Initialize()
```

```
  {
    // Should come from a database
    interest = 0.05;
    lowerLimit = 1000.0;
    upperLimit = 10000000.0;
  }

  public override void Deposit(double amount)
  {
    balance += amount;
    StateChangeCheck();
  }

  public override void Withdraw(double amount)
  {
    balance -= amount;
    StateChangeCheck();
  }

  public override void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  private void StateChangeCheck()
  {
    if (balance < 0.0)
    {
      account.State = new RedState(this);
    }
    else if (balance < lowerLimit)
    {
      account.State = new SilverState(this);
    }
  }
}
```

```
/// <summary>
/// The 'Context' class
/// </summary>
class Account
{
  private State _state;
  private string _owner;

  // Constructor
  public Account(string owner)
  {
    // New accounts are 'Silver' by default
    this._owner = owner;
    this._state = new SilverState(0.0, this);
  }

  // Properties
  public double Balance
  {
    get { return _state.Balance; }
  }

  public State State
  {
    get { return _state; }
    set { _state = value; }
  }

  public void Deposit(double amount)
  {
    _state.Deposit(amount);
    Console.WriteLine("Deposited {0:C} --- ", amount);
    Console.WriteLine(" Balance = {0:C}", this.Balance);
    Console.WriteLine(" Status = {0}",
      this.State.GetType().Name);
    Console.WriteLine("");
```

```
    }

    public void Withdraw(double amount)
    {
      _state.Withdraw(amount);
      Console.WriteLine("Withdrew {0:C} --- ", amount);
      Console.WriteLine(" Balance = {0:C}", this.Balance);
      Console.WriteLine(" Status = {0}\n",
        this.State.GetType().Name);
    }

    public void PayInterest()
    {
      _state.PayInterest();
      Console.WriteLine("Interest Paid --- ");
      Console.WriteLine(" Balance = {0:C}", this.Balance);
      Console.WriteLine(" Status = {0}\n",
        this.State.GetType().Name);
    }
  }
}
```

Output

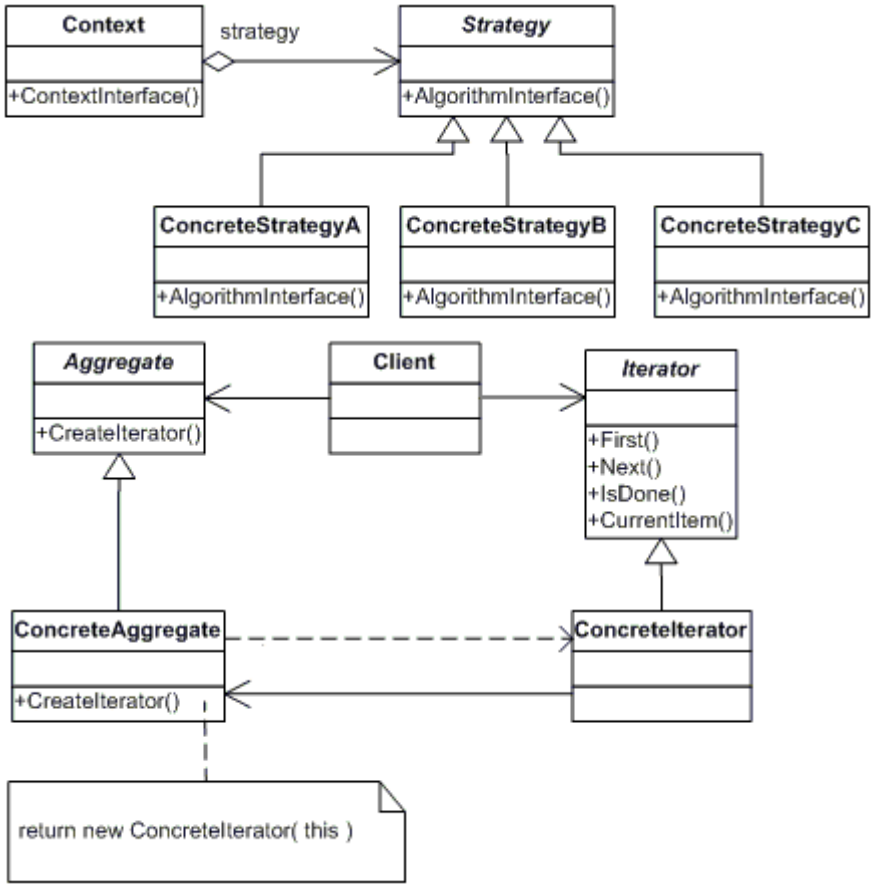
```
Deposited $500.00 ---
 Balance = $500.00
 Status = SilverState


Deposited $300.00 ---
 Balance = $800.00
 Status = SilverState


Deposited $550.00 ---
 Balance = $1,350.00
 Status = GoldState


Interest Paid ---
 Balance = $1,417.50
 Status = GoldState

Withdrew $2,000.00 ---
 Balance = ($582.50)
 Status = RedState

No funds available for withdrawal!
Withdrew $1,100.00 ---
 Balance = ($582.50)
 Status = RedState
```

2. Strategy

Definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

UML class diagram



Participants

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

- **Strategy (SortStrategy)**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - implements the algorithm using the Strategy interface
- **Context (SortedList)**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Sample code in C#

This structural code demonstrates the Strategy pattern which encapsulates functionality in the form of an object. This allows clients to dynamically change algorithmic strategies.

```
// Strategy pattern -- Structural example

using System;
```

```
namespace DoFactory.GangOfFour.Strategy.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Strategy Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      Context context;

      // Three contexts following different strategies
      context = new Context(new ConcreteStrategyA());
      context.ContextInterface();

      context = new Context(new ConcreteStrategyB());
      context.ContextInterface();

      context = new Context(new ConcreteStrategyC());
      context.ContextInterface();

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

  /// <summary>
  /// The 'Strategy' abstract class
  /// </summary>
  abstract class Strategy
  {
    public abstract void AlgorithmInterface();
  }
```

```
/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ConcreteStrategyA : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyA.AlgorithmInterface()");
  }
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ConcreteStrategyB : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyB.AlgorithmInterface()");
  }
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ConcreteStrategyC : Strategy
{
  public override void AlgorithmInterface()
  {
    Console.WriteLine(
      "Called ConcreteStrategyC.AlgorithmInterface()");
  }
}
```

```
  /// <summary>
  /// The 'Context' class
  /// </summary>
  class Context
  {
    private Strategy _strategy;

    // Constructor
    public Context(Strategy strategy)
    {
      this._strategy = strategy;
    }

    public void ContextInterface()
    {
      _strategy.AlgorithmInterface();
    }
  }
}
```

Output

```
Called ConcreteStrategyA.AlgorithmInterface()
Called ConcreteStrategyB.AlgorithmInterface()
Called ConcreteStrategyC.AlgorithmInterface()
```

This real-world code demonstrates the Strategy pattern which encapsulates sorting algorithms in the form of sorting objects. This allows clients to dynamically change sorting strategies including Quicksort, Shellsort, and Mergesort.

```
// Strategy pattern -- Real World example

using System;
using System.Collections.Generic;

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

```
/// Strategy Design Pattern.
/// </summary>
class MainApp
{
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    // Two contexts following different strategies
    SortedList studentRecords = new SortedList();

    studentRecords.Add("Samual");
    studentRecords.Add("Jimmy");
    studentRecords.Add("Sandra");
    studentRecords.Add("Vivek");
    studentRecords.Add("Anna");

    studentRecords.SetSortStrategy(new QuickSort());
    studentRecords.Sort();

    studentRecords.SetSortStrategy(new ShellSort());
    studentRecords.Sort();

    studentRecords.SetSortStrategy(new MergeSort());
    studentRecords.Sort();

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

/// <summary>
/// The 'Strategy' abstract class
/// </summary>
abstract class SortStrategy
{
```

```
  public abstract void Sort(List<string> list);
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class QuickSort : SortStrategy
{
  public override void Sort(List<string> list)
  {
    list.Sort(); // Default is Quicksort
    Console.WriteLine("QuickSorted list ");
  }
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class ShellSort : SortStrategy
{
  public override void Sort(List<string> list)
  {
    //list.ShellSort(); not-implemented
    Console.WriteLine("ShellSorted list ");
  }
}

/// <summary>
/// A 'ConcreteStrategy' class
/// </summary>
class MergeSort : SortStrategy
{
  public override void Sort(List<string> list)
  {
    //list.MergeSort(); not-implemented
    Console.WriteLine("MergeSorted list ");
  }
```

```
  }

  /// <summary>
  /// The 'Context' class
  /// </summary>
  class SortedList
  {
    private List<string> _list = new List<string>();
    private SortStrategy _sortstrategy;

    public void SetSortStrategy(SortStrategy sortstrategy)
    {
      this._sortstrategy = sortstrategy;
    }

    public void Add(string name)
    {
      _list.Add(name);
    }

    public void Sort()
    {
      _sortstrategy.Sort(_list);

      // Iterate over list and display results
      foreach (string name in _list)
      {
        Console.WriteLine(" " + name);
      }
      Console.WriteLine();
    }
  }
}
```

Output

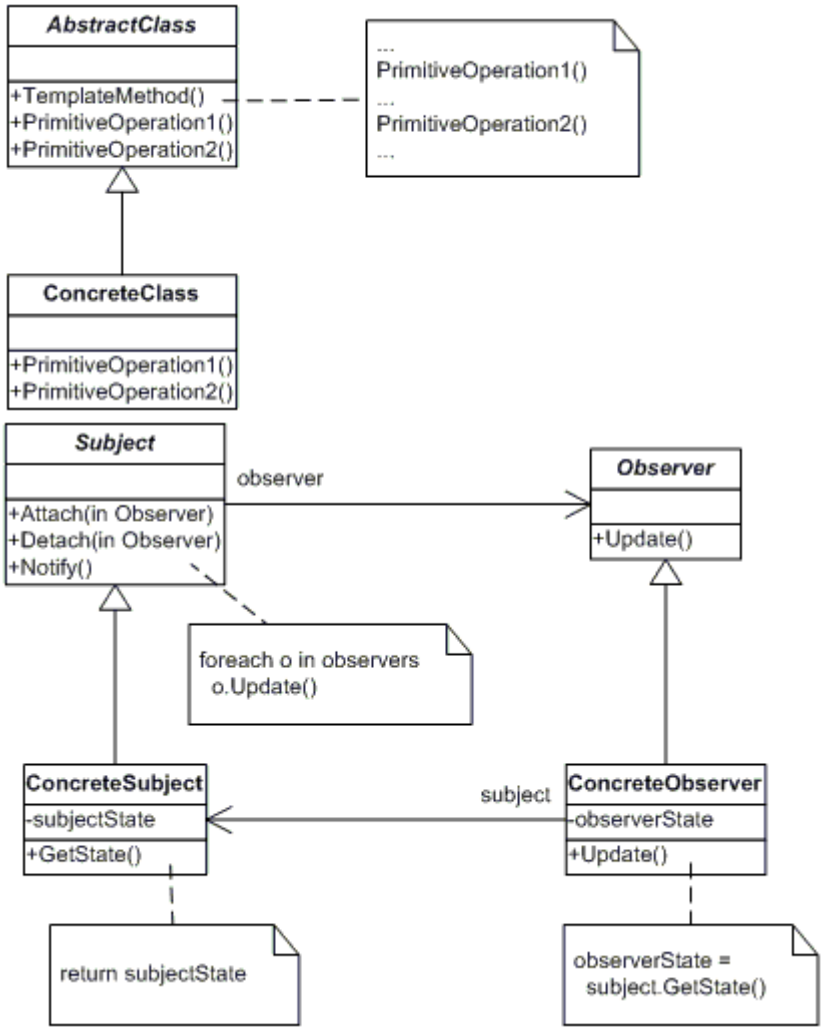
```
QuickSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek

ShellSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek

MergeSorted list
 Anna
 Jimmy
 Samual
 Sandra
 Vivek
```

3. Template Method

Definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

UML class diagram



Participants

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

- **AbstractClass (DataObject)**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass (CustomerDataObject)**
  - implements the primitive operations ot carry out subclass-specific steps of the algorithm

Sample code in C#

This structural code demonstrates the Template method which provides a skeleton calling sequence of methods. One or more steps can be deferred to subclasses which implement these steps without changing the overall calling sequence.

```
// Template Method pattern -- Structural example
```

```
using System;

namespace DoFactory.GangOfFour.Template.Structural
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Template Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      AbstractClass aA = new ConcreteClassA();
      aA.TemplateMethod();

      AbstractClass aB = new ConcreteClassB();
      aB.TemplateMethod();

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

  /// <summary>
  /// The 'AbstractClass' abstract class
  /// </summary>
  abstract class AbstractClass
  {
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    // The "Template method"
    public void TemplateMethod()
```

```
    {
      PrimitiveOperation1();
      PrimitiveOperation2();
      Console.WriteLine("");
    }
  }

  /// <summary>
  /// A 'ConcreteClass' class
  /// </summary>
  class ConcreteClassA : AbstractClass
  {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
    }
  }

  /// <summary>
  /// A 'ConcreteClass' class
  /// </summary>
  class ConcreteClassB : AbstractClass
  {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
    }
  }
}
```

Output

```
ConcreteClassA.PrimitiveOperation1()
ConcreteClassA.PrimitiveOperation2()
```

This real-world code demonstrates a Template method named Run() which provides a skeleton calling sequence of methods. Implementation of these steps are deferred to the CustomerDataObject subclass which implements the Connect, Select, Process, and Disconnect methods.

```
// Template Method pattern -- Real World example

using System;
using System.Data;
using System.Data.OleDb;


namespace DoFactory.GangOfFour.Template.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Template Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      DataAccessObject daoCategories = new Categories();
      daoCategories.Run();


      DataAccessObject daoProducts = new Products();
      daoProducts.Run();


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

```
}

/// <summary>
/// The 'AbstractClass' abstract class
/// </summary>
abstract class DataAccessObject
{
  protected string connectionString;
  protected DataSet dataSet;

  public virtual void Connect()
  {
    // Make sure mdb is available to app
    connectionString =
      "provider=Microsoft.JET.OLEDB.4.0; " +
      "data source=..\\..\\..\\db1.mdb";
  }

  public abstract void Select();
  public abstract void Process();

  public virtual void Disconnect()
  {
    connectionString = "";
  }

  // The 'Template Method'
  public void Run()
  {
    Connect();
    Select();
    Process();
    Disconnect();
  }
}

/// <summary>
```

```
/// A 'ConcreteClass' class
/// </summary>
class Categories : DataAccessObject
{
  public override void Select()
  {
    string sql = "select CategoryName from Categories";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
      sql, connectionString);

    dataSet = new DataSet();
    dataAdapter.Fill(dataSet, "Categories");
  }

  public override void Process()
  {
    Console.WriteLine("Categories ---- ");

    DataTable dataTable = dataSet.Tables["Categories"];
    foreach (DataRow row in dataTable.Rows)
    {
      Console.WriteLine(row["CategoryName"]);
    }
    Console.WriteLine();
  }
}

/// <summary>
/// A 'ConcreteClass' class
/// </summary>
class Products : DataAccessObject
{
  public override void Select()
  {
    string sql = "select ProductName from Products";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
      sql, connectionString);
```

```
      dataSet = new DataSet();
      dataAdapter.Fill(dataSet, "Products");
    }

    public override void Process()
    {
      Console.WriteLine("Products ---- ");
      DataTable dataTable = dataSet.Tables["Products"];
      foreach (DataRow row in dataTable.Rows)
      {
        Console.WriteLine(row["ProductName"]);
      }
      Console.WriteLine();
    }
  }
}
```

Output

```
Categories ----
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Products ----
Chai
Chang
Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
Grandma's Boysenberry Spread
Uncle Bob's Organic Dried Pears
Northwoods Cranberry Sauce
Mishi Kobe Niku
```