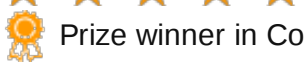
Design Patterns 3 of 3 - Behavioral Design Patterns

By **Kanasz Robert**, 10 Jan 2013



Prize winner in Competition "Best overall article of September 2012"

Download Behavioral Patterns structural code examples - 222.2 KB

Download Behavioral Patterns real world examples - 320.8 KB

Introduction

This is the third (and the last) article about Design Patterns. In the first article of this series I discussed about Creational design patters. The second article was about Structural design patterns, and now I will describe another set of patterns called Behavioral design patterns.

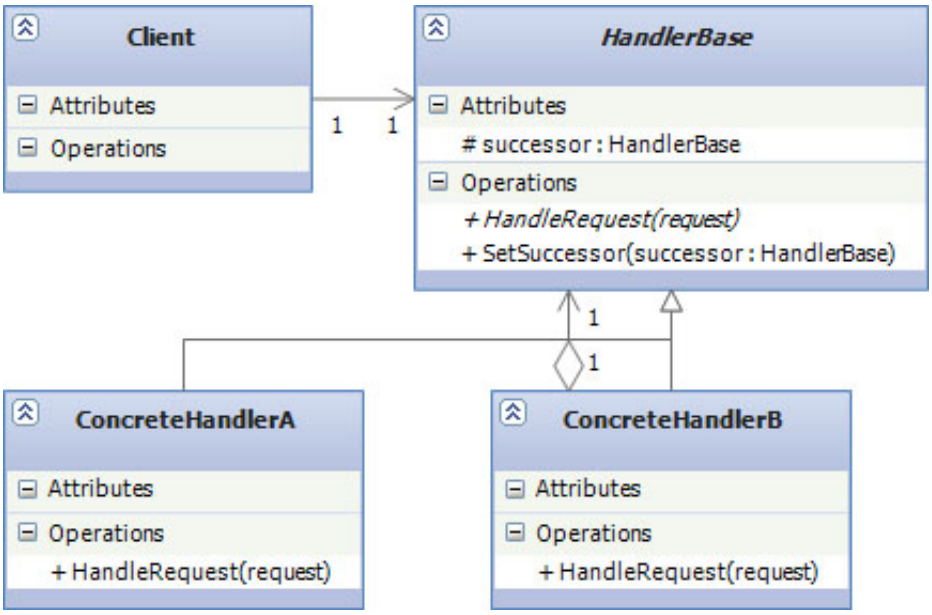
A behavioral pattern explains how objects interact. It describes how different objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects. Where Creational patterns mostly describe a moment of time (the instant of creation), and Structural patterns describe a more or less static structure, Behavioral patterns describe a process or a flow.

Chain of Responsibility pattern

The chain of responsibility pattern is a design pattern that defines a linked list of handlers, each of which is able to process requests. When a request is submitted to the chain, it is passed to the first handler in the list that is able to process it.

As a programmer, I'm sure that you have found a problem when an event generated by an object needed to be handled by another object. The Chain of Responsibility pattern can solve this problem. In this pattern we have a source of command objects and a series of processing objects. The command is passed to the first processing object which can handle this command or send to its successor. This continues until the command is processed or the end of the chain is reached. In this pattern, the object which sends a command doesn't know which object will process the command.

Structural code example



The UML diagram below describes an implementation of the Chain of Responsibility design pattern.

This diagram consists of three parts:

- **Client:** this class passes commands (or requests) to the first object of the chain of processing objects.
- **HandlerBase:** represents an interface or base class for all concrete handlers. It contains a member variable which points to the next processing object.
- **ConcreteHandlers:** this is a concrete implementation of the `HandlerBase` class.

```
static class Program
{
    static void Main()
    {
        HandlerBase handlerA = new ConcreteHandlerA();
        HandlerBase handlerB = new ConcreteHandlerB();
        handlerA.SetSuccessor(handlerB);
        handlerA.HandleRequest(1);
        handlerB.HandleRequest(11);
    }
}

public abstract class HandlerBase
{
    protected HandlerBase _successor;

    public abstract void HandleRequest(int request);

    public void SetSuccessor(HandlerBase successor)
    {
        _successor = successor;
    }
}

public class ConcreteHandlerA : HandlerBase
{
    public override void HandleRequest(int request)
    {
        if (request == 1)
            Console.WriteLine("Handled by ConcreteHandlerA");
```

```
        else if (_successor != null)
            _successor.HandleRequest(request);
    }
}


public class ConcreteHandlerB : HandlerBase
{
    public override void HandleRequest(int request)
    {
        if (request > 10)
            Console.WriteLine("Handled by ConcreteHandlerB");
        else if (_successor != null)
            _successor.HandleRequest(request);
    }
}
```

Real world example

Let's take a loot at a real world example. Imagine you have a wending machine which accepts coins. Rather than having a slot for each type of coin, the machine has only one for all of them. The inserted coin is sent to the appropriate storage place that is determined by the receiver of the command. In this example we have a `Coin` class with two properties: `Diameter` and `Weight`. The `Coin` class is in this example a request (or a command). Next we have an abstract `CoinHandlerBase` class with a method `SetSuccessor()` which sets the next processing object and an abstract method `EvaluateCoin` which will be overridden by concrete handlers. In this example we will handle only 50 pence, 5 pence, 1 pound, 10 pence, and 20 pence coins. For each type of coin we must implement a handler. So now we have five handlers. Each handler will work with the diameter and weight of the coin respecting slight differences between fixed values defined by the handler and the properties of the coins. This helps us to fix the problem when a coin has a little bigger/smaller diameter or when a coin is a little heavier or lighter.

```
class FiftyPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 8) < 0.02 && Math.Abs(coin.Diameter - 27.3) < 0.15)
        {
            Console.WriteLine("Captured 50p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}

public class FivePenceHandler : CoinHandlerBase
{

    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 3.25) < 0.02 && Math.Abs(coin.Diameter - 18) < 0.1)
        {
            Console.WriteLine("Captured 5p");
            return CoinEvaluationResult.Accepted;
        }
```

```
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}

class OnePoundHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 9.5) < 0.02 && Math.Abs(coin.Diameter - 22.5) < 0.13)
        {
            Console.WriteLine("Captured £1");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}

public class TenPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 6.5) < 0.03 && Math.Abs(coin.Diameter - 24.5) < 0.15)
        {
            Console.WriteLine("Captured 10p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}

class TwentyPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 5) < 0.01 && Math.Abs(coin.Diameter - 21.4) < 0.1)
        {
            Console.WriteLine("Captured 20p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}

public abstract class CoinHandlerBase
{
    protected CoinHandlerBase _successor;
```

```
    public void SetSuccessor(CoinHandlerBase successor)
    {
        _successor = successor;
    }

    public abstract CoinEvaluationResult EvaluateCoin(Coin coin);
}

public class Coin
{
    public float Weight { get; set; }
    public float Diameter { get; set; }
}

class Program
{
    static void Main()
    {
        var h5 = new FivePenceHandler();
        var h10 = new TenPenceHandler();
        var h20 = new TwentyPenceHandler();
        var h50 = new FiftyPenceHandler();
        var h100 = new OnePoundHandler();
        h5.SetSuccessor(h10);
        h10.SetSuccessor(h20);
        h20.SetSuccessor(h50);
        h50.SetSuccessor(h100);

        var tenPence = new Coin { Diameter = 24.49F, Weight = 6.5F };
        var fiftyPence = new Coin { Diameter = 27.31F, Weight = 8.01F };
        var counterfeitPound = new Coin { Diameter = 22.5F, Weight = 9F };

        Console.WriteLine(h5.EvaluateCoin(tenPence));
        Console.WriteLine(h5.EvaluateCoin(fiftyPence));
        Console.WriteLine(h5.EvaluateCoin(counterfeitPound));
    }
}

public enum CoinEvaluationResult
{
    Accepted,
    Rejected
}
```

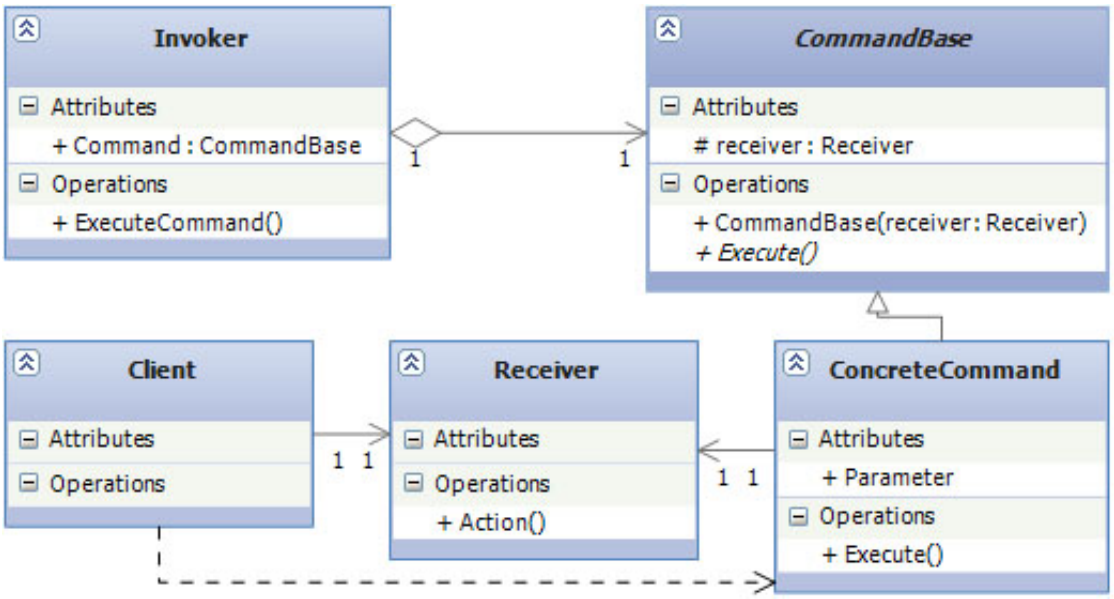
Command pattern

The command pattern is a design pattern that enables all of the information for a request to be contained within a single object. The command can then be invoked as required, often as part of a batch of queued commands with rollback capabilities.

Command pattern is a behavioral design pattern in which all information needed to execute a method is encapsulated within an object which could be used immediately or held for later use. This object doesn't execute anything, it only includes information.

There are three key terms which need to be explained: client, invoker, and receiver. The client creates the command object. The invoker decides when the method which needs information encapsulated within the command object should be called. Receiver is an instance of the class which contains the method's code.

Structural code example



The UML diagram below describes an implementation of the command design pattern. This diagram consists of five parts:

- **Client:** The class is a consumer of the classes of the Command design pattern. It creates the command objects and links them to receivers.
- **Receiver:** this is the class which knows how to perform the operations associated with carrying out the request.
- **CommandBase:** this is the abstract class (or interface) for all command objects. It holds information about the receiver which is responsible for executing an operation using information encapsulated within the command object.
- **ConcreteCommand:** concrete implementation of the `CommandBase` abstract class or interface.
- **Invoker:** is the object which decides when to execute the command.

```
class Program
{
    static void Main()
    {
        Client client = new Client();
        client.RunCommand();
    }
}

public class Client
{
    public void RunCommand()
    {
        var invoker = new Invoker();
        var receiver = new Receiver();
        var command = new ConcreteCommand(receiver);
        command.Parameter = "Hello, world!";
        invoker.Command = command;
        invoker.ExecuteCommand();
    }
}

public class Receiver
{
```

```
    public void Action(string message)
    {
        Console.WriteLine("Action called with message, '{0}'.", message);
    }
}

public class Invoker
{
    public CommandBase Command { get; set; }

    public void ExecuteCommand()
    {
        Command.Execute();
    }
}

public abstract class CommandBase
{
    protected readonly Receiver _receiver;

    public CommandBase(Receiver receiver)
    {
        _receiver = receiver;
    }

    public abstract void Execute();
}

public class ConcreteCommand : CommandBase
{
    public string Parameter { get; set; }

    public ConcreteCommand(Receiver receiver) : base(receiver) { }

    public override void Execute()
    {
        _receiver.Action(Parameter);
    }
}
```

Real world example

I have prepared a simple example that demonstrates using of the Command pattern. In this example Command pattern is used to control the robot movement. In this example the client is an application. The receiver of the commands is the robot itself. The `Robot` class has four methods which are for controlling the movement: `Move`, `RotateLeft`, `RotateRight`, `TakeSample`. `RobotCommandBase` is the abstract base class for all concrete command classes. It has a protected `Robot` field which points to the `Robot` object and abstract methods `Move` and `Undo` which must be overridden by the concrete command. The class `RobotController` is in this example the invoker. It contains two methods: `ExecuteCommands` and `UndoCommands`. The first method will execute all commands in a queue and the method `UndoCommands` is the undo functionality to reverse any number of commands as required.

```
public abstract class RobotCommandBase
{
    protected Robot _robot;

    public RobotCommandBase(Robot robot)
    {
```

```
        _robot = robot;
    }

    public abstract void Execute();

    public abstract void Undo();
}

public class MoveCommand:RobotCommandBase
{
    public int ForwardDistance { get; set; }

    public MoveCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.Move(ForwardDistance);
    }

    public override void Undo()
    {
        _robot.Move(-ForwardDistance);
    }
}

public class RotateLeftCommand : RobotCommandBase
{
    public double LeftRotationAngle { get; set; }

    public RotateLeftCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.RotateLeft(LeftRotationAngle);
    }

    public override void Undo()
    {
        _robot.RotateRight(LeftRotationAngle);
    }
}

public class RotateRightCommand : RobotCommandBase
{
    public double RightRotationAngle { get; set; }

    public RotateRightCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.RotateRight(RightRotationAngle);
    }

    public override void Undo()
    {
        _robot.RotateLeft(RightRotationAngle);
    }
}

public class TakeSampleCommand : RobotCommandBase
{
    public bool TakeSample { get; set; }
```

```
    public TakeSampleCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.TakeSample(true);
    }

    public override void Undo()
    {
        _robot.TakeSample(false);
    }
}

public class RobotController
{
    public Queue<RobotCommandBase> Commands;
    private Stack<RobotCommandBase> _undoStack;

    public RobotController()
    {
        Commands = new Queue<RobotCommandBase>();
        _undoStack = new Stack<RobotCommandBase>();
    }

    public void ExecuteCommands()
    {
        Console.WriteLine("EXECUTING COMMANDS.");

        while (Commands.Count > 0)
        {
            RobotCommandBase command = Commands.Dequeue();
            command.Execute();
            _undoStack.Push(command);
        }
    }

    public void UndoCommands(int numUndos)
    {
        Console.WriteLine("REVERSING {0} COMMAND(S).", numUndos);

        while (numUndos > 0 && _undoStack.Count > 0)
        {
            RobotCommandBase command = _undoStack.Pop();
            command.Undo();
            numUndos--;
        }
    }
}

public class Robot
{
    public void Move(int distance)
    {
        if (distance > 0)
            Console.WriteLine("Robot moved forwards {0}mm.", distance);
        else
            Console.WriteLine("Robot moved backwards {0}mm.", -distance);
    }

    public void RotateLeft(double angle)
    {
```

```
        if (angle > 0)
            Console.WriteLine("Robot rotated left {0} degrees.", angle);
        else
            Console.WriteLine("Robot rotated right {0} degrees.", -angle);
    }

    public void RotateRight(double angle)
    {
        if (angle > 0)
            Console.WriteLine("Robot rotated right {0} degrees.", angle);
        else
            Console.WriteLine("Robot rotated left {0} degrees.", -angle);
    }

    public void TakeSample(bool take)
    {
        if(take)
            Console.WriteLine("Robot took sample");
        else
            Console.WriteLine("Robot released sample");
    }
}

class Program
{
    static void Main()
    {
        var robot = new Robot();
        var controller = new RobotController();

        var move = new MoveCommand(robot);
        move.ForwardDistance = 1000;
        controller.Commands.Enqueue(move);

        var rotate = new RotateLeftCommand(robot);
        rotate.LeftRotationAngle = 45;
        controller.Commands.Enqueue(rotate);

        var scoop = new TakeSampleCommand(robot);
        scoop.TakeSample = true;
        controller.Commands.Enqueue(scoop);

        controller.ExecuteCommands();
        controller.UndoCommands(3);
    }
}
```

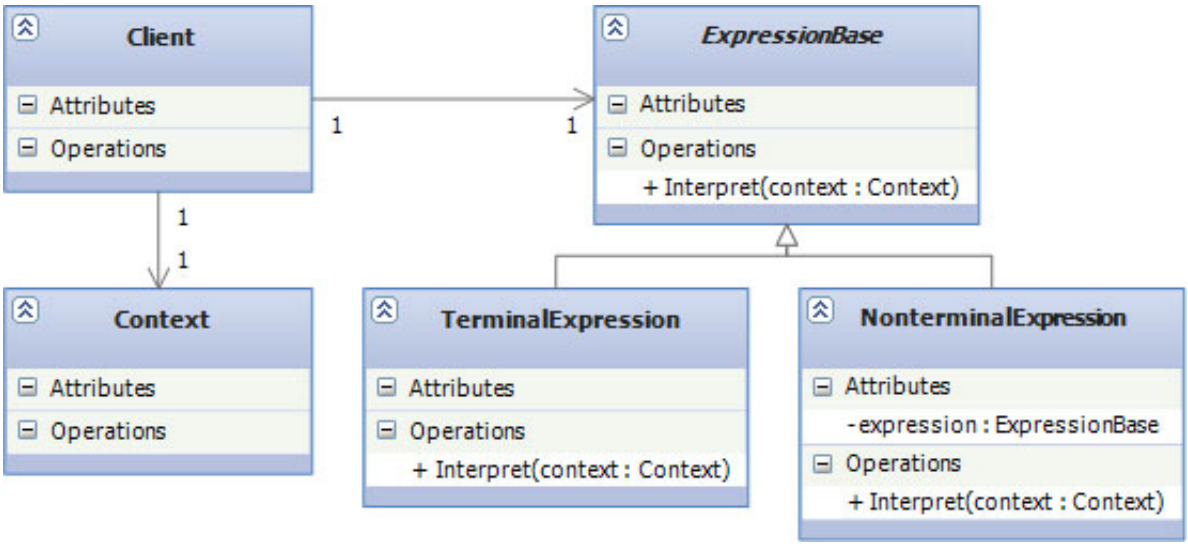
Interpreter pattern

The interpreter pattern is a design pattern that is useful when developing domain-specific languages or notations. The pattern allows the grammar for such a notation to be represented in an object-oriented fashion that can easily be extended.

The Interpreter pattern is another behavioral design pattern that specifies how to evaluate sentences in a language. This pattern is described in terms of formal grammars. It performs some activities based upon a set of expressions. We have two types of expressions: terminal and non-terminal. The difference between these types is very simple. While terminal expressions represent structures which can be immediately evaluated, non-terminal expressions are composed of one or more expressions which could be terminal or

non-terminal. The Interpreter pattern is very similar to the Composite pattern. The terminal expressions represent the leafs of this tree structure and non-terminal ones represent composites. The Interpreter defines the behavior while the Composite defines only the structure.

Structural code example



The UML diagram below describes an implementation of the Interpreter design pattern. This diagram consists of five parts:

- **Client:** builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the `NonterminalExpression` and `TerminalExpression` classes invoking the Interpret operation.
- **Context:** contains information that is global to the interpreter.
- **ExpressionBase:** declares an interface or abstract class for all expressions.
- **TerminalExpresion:** implements an Interpret operation associated with terminal symbols in the grammar. An instance is required for every terminal symbol in the sentence.
- **NonterminalExpression:** Non-terminal expressions are represented using a concrete subclass of `ExpressionBase`. These expressions are aggregates containing one or more further expressions, each of which may be terminal or non-terminal. When a non-terminal expression class' `Interpret` method is called, the process of interpretation includes calls to the `Interpret` method of the expressions it holds.

```
static class Program
{
    static void Main()
    {
        Client.BuildAndInterpretCommands();
    }
}

public static class Client
{
    public static void BuildAndInterpretCommands()
    {
        var context = new Context("hello world!!!");
        var root = new NonterminalExpression
                        {
                            Expression1 = new TerminalExpression(),
```

```
                        Expression2 = new TerminalExpression()
                    };
        root.Interpret(context);
    }
}

public class Context
{
    public string Name { get; set; }

    public Context(string name)
    {
        Name = name;
    }
}

public abstract class ExpressionBase
{
    public abstract void Interpret(Context context);
}


public class TerminalExpression : ExpressionBase
{
    public override void Interpret(Context context)
    {
        Console.WriteLine("Terminal for {0}.", context.Name);
    }
}


public class NonterminalExpression : ExpressionBase
{
    public ExpressionBase Expression1 { get; set; }

    public ExpressionBase Expression2 { get; set; }

    public override void Interpret(Context context)
    {
        Console.WriteLine("Nonterminal for {0}.", context.Name);
        Expression1.Interpret(context);
        Expression2.Interpret(context);
    }
}
```

Real world example

As an example I decided to implement a simplified Polish notation evaluator. This evaluator allows you to add or subtract simple integer values. The grammar for the Polish notation is very simple. While integer values represent terminals, subtraction and addition operators represent non-terminals. In this example, it is not necessary to create a custom context class, because the context is represented by a string that contains numbers and + and - symbols. In this example, we have an `IExpressionBase` interface that includes the signature for the method `Evaluate` which evaluates the expression and returns the result of the evaluation. Next we have three concrete expression classes which implement the `IExpressionBase` interface. The `IntegerExpression` class represents basic integers. `AdditionExpression` and `SubtractionExpression` classes are non-terminal expression classes. Both of these classes have a constructor with two `IExpressionBase` parameters. Both of these parameters must be evaluated and the result of the evaluation is returned by `AdditionExpression`

and `SubtractionExpressionClasses`.

```
    public interface IExpressionBase
    {
        int Evaluate();
    }

    public class AdditionExpression : ExpressionBase.IExpressionBase
    {
        ExpressionBase.IExpressionBase _expr1;
        ExpressionBase.IExpressionBase _expr2;

        public AdditionExpression(ExpressionBase.IExpressionBase expr1,
ExpressionBase.IExpressionBase expr2)
        {
            _expr1 = expr1;
            _expr2 = expr2;
        }

        public int Evaluate()
        {
            int value1 = _expr1.Evaluate();
            int value2 = _expr2.Evaluate();
            return value1 + value2;
        }

        public override string ToString()
        {
            return string.Format("({0} + {1})", _expr1, _expr2);
        }
    }

    public class SubtractionExpression : ExpressionBase.IExpressionBase
    {
        ExpressionBase.IExpressionBase _expr1;
        ExpressionBase.IExpressionBase _expr2;

        public SubtractionExpression(ExpressionBase.IExpressionBase expr1,
               ExpressionBase.IExpressionBase expr2)
        {
            _expr1 = expr1;
            _expr2 = expr2;
        }

        public int Evaluate()
        {
            int value1 = _expr1.Evaluate();
            int value2 = _expr2.Evaluate();
            return value1 - value2;
        }

        public override string ToString()
        {
            return string.Format("({0} - {1})", _expr1, _expr2);
        }
    }

    public class IntegerExpression:ExpressionBase.IExpressionBase
    {
        int _value;

        public IntegerExpression(int value)
```

```
        {
            _value = value;
        }

        public int Evaluate()
        {
            return _value;
        }

        public override string ToString()
        {
            return _value.ToString();
        }
    }

    static void Main(string[] args)
    {
        var parser = new Parser();

        var commands = new string[]
                                {
                                    "+ 5 6",
                                    "- 6 5",
                                    "+ - 4 5 6",
                                    "+ 4 - 5 6",
                                    "+ - + - - 2 3 4 + - -5 6 + -7 8 9 10"
                                };

        foreach (var command in commands)
        {
            ExpressionBase.IExpressionBase expression = parser.Parse(command);
            Console.WriteLine("{0} = {1}", expression, expression.Evaluate());
        }
    }
}

public class Parser
{
    public ExpressionBase.IExpressionBase Parse(string polish)
    {
        var symbols = new List<string>(polish.Split(' '));
        return ParseNextExpression(symbols);
    }

    public ExpressionBase.IExpressionBase ParseNextExpression(List<string> symbols)
    {
        int value;
        if (int.TryParse(symbols[0], out value))
        {
            symbols.RemoveAt(0);
            return new IntegerExpression(value);
        }
        return ParseNonTerminalExpression(symbols);
    }

    private ExpressionBase.IExpressionBase ParseNonTerminalExpression(List<string> symbols)
    {
        var symbol = symbols[0];
        symbols.RemoveAt(0);

        var expr1 = ParseNextExpression(symbols);
        var expr2 = ParseNextExpression(symbols);
```

```
        switch (symbol)
        {
            case "+":
                return new AdditionExpression(expr1, expr2);
            case "-":
                return new SubtractionExpression(expr1, expr2);
            default:
                string message = string.Format("Invalid Symbol ({0})", symbol);
                throw new InvalidOperationException(message);
        }
    }
}
```

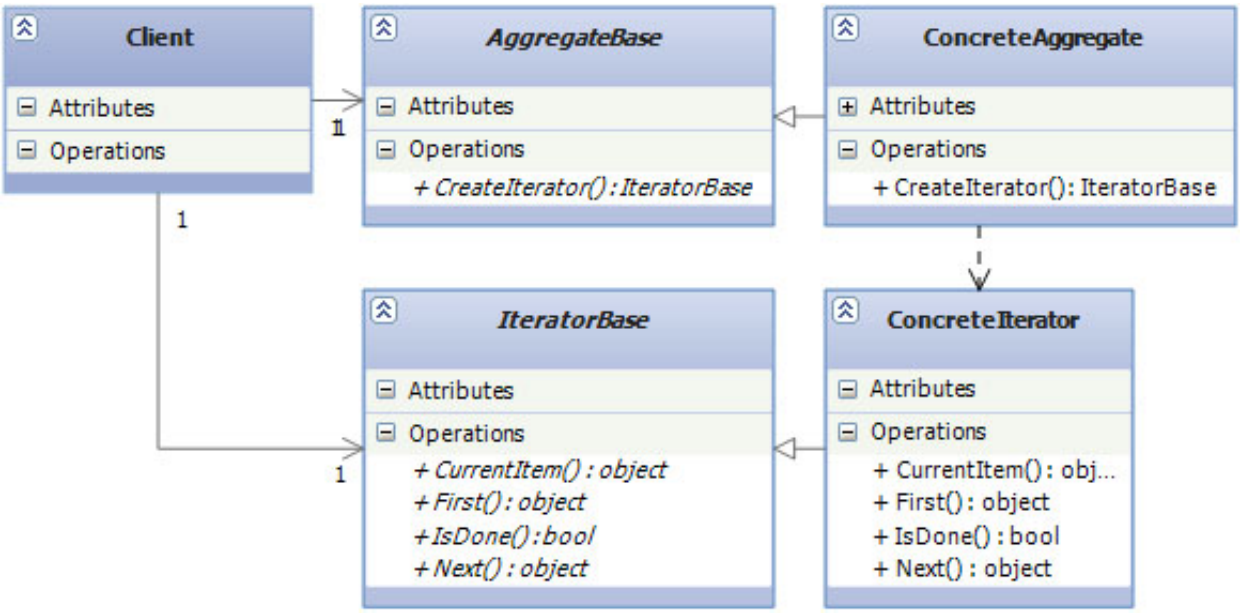
This example was taken from www.blackwasp.co.uk/Interpreter.aspx.

Iterator pattern

The iterator pattern is a design pattern that provides a means for the elements of an aggregate object to be accessed sequentially without knowledge of its structure. This allows traversing of lists, trees and other structures in a standard manner.

One of the most common data structures in software development is a structure which is generically called a collection. A collection is a set of objects. These object can have the same type or they can by all casted to a base type. A collection can by an array, list, and so on. One of the most important things about a collection is its ability to access the elements without exposing the internal structure of the collection. The main intent of the iterator pattern is to provide a simple interface for traversing a collection of items. The iterator takes the responsibility of accessing and passing through the objects of the collection and putting it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

Structural code example



The UML diagram below describes an implementation of the interpreter design pattern. This diagram consists of five parts:

- **Client:** this object requests and consumes the iterator from an aggregate object when it wants to loop through items.
- **AggregateBase:** this is an abstract base class (or interface) for all concrete aggregates. This class contains a method which returns the iterator.
- **ConcreteAggregate:** represents the concrete implementation of `AggregateBase`. It holds items which can be traversed using the iterator.
- **IteratorBase:** this is the abstract class (or interface) for all concrete iterators. It include methods (in case of interface signature of methods) which allows you to traverse through items.
- **ConcreteIterator:** this is the concrete implementation of `IteratorBase`.

```
static class Program
{
    static void Main()
    {
        var aggregate = new ConcreteAggregate();
        aggregate[0] = "Item 1";
        aggregate[1] = "Item 2";
        aggregate[2] = "Item 3";
        aggregate[3] = "Item 4";


        var itorator = new ConcreteIterator(aggregate);


        object item = itorator.First();
        while (item != null)
        {
            Console.WriteLine(item);
            item = itorator.Next();
        }
    }
}
abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

abstract class Aggregate
{
    public abstract Iterator CreateIterator();
}

class ConcreteIterator : Iterator
{
    private readonly ConcreteAggregate _aggregate;
    private int _current;

    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        _aggregate = aggregate;
    }

    public override object First()
    {
        return _aggregate[0];
    }

    public override object Next()
```

```
    {
        object ret = null;
        if (_current < _aggregate.Count - 1)
        {
            ret = _aggregate[++_current];
        }

        return ret;
    }

    public override object CurrentItem()
    {
        return _aggregate[_current];
    }

    public override bool IsDone()
    {
        return _current >= _aggregate.Count;
    }
}

class ConcreteAggregate : Aggregate
{
    private readonly ArrayList _items = new ArrayList();

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

    public int Count
    {
        get { return _items.Count; }
    }

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

Real world example

In this example I decided to create a custom collection (`PersonAggregate`) of `Person` objects. This custom collection is an aggregate which implements the `IPersonEnumerable` aggregate base interface. This interface contains the signature of the method `GetIterator()` which returns the iterator for this collection. Next I have created the `PersonEnumerator` class which is the concrete implementation of the `IPersonIterator` interface. This class has a constructor which accepts an `IPersonEnumerable` object. The `PersonEnumerator` class has three methods that allow you to traverse though the list of persons. The method `MoveFirst` sets the cursor at the first element of the list. The method `MoveNext` sets the cursor at the next element of the list and returns true if another element exists in the collection after the current item. The last method `Reset` sets the cursor before the first element of the list.

```
public interface IPersonEnumerable
{
    IPersonIterator GetIterator();
```

```
}

public class PersonAggregate : IPersonEnumerable
{
    private List<Person> _persons = new List<Person>();

    public Person this[int index]
    {
        get { return _persons[index]; }
    }

    public IPersonIterator GetIterator()
    {
        return new PersonEnumerator(this);
    }

    public int Count
    {
        get { return _persons.Count; }
    }

    public void Add(Person person)
    {
        _persons.Add(person);
    }
}

public class PersonEnumerator:IPersonIterator
{
    private PersonAggregate _aggregate;
    int _position;

    public PersonEnumerator(PersonAggregate aggregate)
    {
        _aggregate = aggregate;
    }

    public void MoveFirst()
    {
        if(_aggregate.Count==0)
        {
            throw new InvalidOperationException();
        }
        _position = 0;
    }

    public void Reset()
    {
        _position = -1;
    }

    public bool MoveNext()
    {
        _position++;
        return _position < _aggregate.Count;
    }

    public Person Current
    {
        get
        {
            if (_position < _aggregate.Count)
```

```
                return _aggregate[_position];
            throw new InvalidOperationException();
        }
    }
}

public interface IPersonIterator
{
    void MoveFirst();
    void Reset();
    bool MoveNext();
    Person Current { get; }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public double Height { get; set; }
}

static void Main(string[] args)
{
    PersonAggregate aggregate = new PersonAggregate();
    aggregate.Add(new Person(){FirstName = "Robert", Height = 168,LastName = "Kanasz"});
    aggregate.Add(new Person(){FirstName = "John", Height = 181, LastName = "Doe" });
    aggregate.Add(new Person() { FirstName = "Jane", Height = 158, LastName = "Doe" });

    IPersonIterator iterator = aggregate.GetIterator();

    iterator.Reset();

    while (iterator.MoveNext())
    {
        Console.WriteLine(iterator.Current.FirstName + " " + iterator.Current.LastName);
    }
}
```

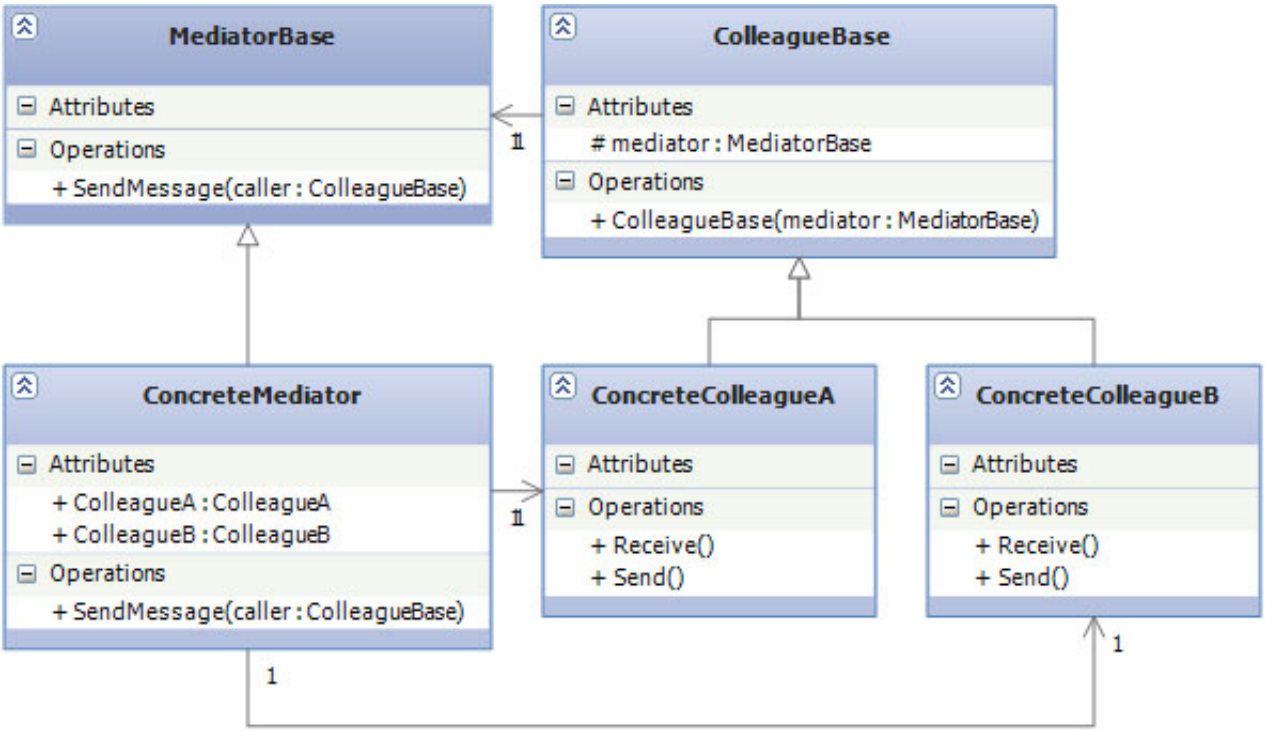
Mediator pattern

The mediator pattern is a design pattern that promotes loose coupling of objects by removing the need for classes to communicate with each other directly. Instead, mediator objects are used to encapsulate and centralize the interactions between classes.

Mediator pattern is another behavioral design pattern. In some situations the program is made up of large classes. This is not unusual but problems arise when these classes need to communicate with each other. Using the traditional approach when classes communicate directly, these classes require to know their internal implementations. This could lead to a very big mess because when the program becoming more and more complex it is harder to read and understand. These classes are tightly coupled which is not good from a design point of view and in future it could cause problems.

Mediator design pattern solves this problem. In this pattern the communication between objects is encapsulated with a mediator object. Instead of classes communicating directly, classes send messages to the mediator and the mediator sends these messages to the other classes.

Structural code example



The UML diagram below describes the implementation of the mediator design pattern. This diagram consists of four parts:

- **ColleagueBase:** this is the abstract class (or interface) for all concrete colleagues. It has a protected field that holds a reference to the mediator object.
- **ConcreteColleague:** this is the implementation of the `ColleagueBase` class. Concrete colleagues are classes which communicate with each other.
- **MediatorBase:** this is the abstract class for the mediator object. This class contains methods that can be consumed by colleagues.
- **ConcreateMediator:** this class implements communication methods of the `MediatorBase` class. This class holds a reference to all colleagues which need to communicate.

```
static class Program
{
    static void Main()
    {
        var m = new ConcreteMediator();

        var c1 = new ConcreteColleague1(m);
        var c2 = new ConcreteColleague2(m);

        m.Colleague1 = c1;
        m.Colleague2 = c2;

        c1.Send("How are you?");
        c2.Send("Fine, thanks");
    }
}
public abstract class ColleagueBase
{
    protected MediatorBase _mediator;

    public ColleagueBase(MediatorBase mediator)
    {
        _mediator = mediator;
```

```
    }
}

public abstract class MediatorBase
{
    public abstract void SendMessage(ColleagueBase caller, string message);
}

class ConcreteMediator : MediatorBase
{
    private ConcreteColleague1 _colleague1;
    private ConcreteColleague2 _colleague2;

    public ConcreteColleague1 Colleague1
    {
        set { _colleague1 = value; }
    }

    public ConcreteColleague2 Colleague2
    {
        set { _colleague2 = value; }
    }


    public override void SendMessage(ColleagueBase caller, string message)
    {
        if (caller == _colleague1)
        {
            _colleague2.Notify(message);
        }
        else
        {
            _colleague1.Notify(message);
        }
    }
}

class ConcreteColleague1 : ColleagueBase
{
    public ConcreteColleague1(MediatorBase mediator)
        : base(mediator)
    {
    }

    public void Send(string message)
    {
        _mediator.SendMessage(this,message);
    }

    public void Notify(string message)
    {
        Console.WriteLine("Colleague1 gets message: " + message);
    }
}

class ConcreteColleague2 : ColleagueBase
{
    public ConcreteColleague2(MediatorBase mediator)
        : base(mediator)
    {
    }
```

```
    public void Send(string message)
    {
        _mediator.SendMessage(this, message);
    }

    public void Notify(string message)
    {
        Console.WriteLine("Colleague2 gets message: "+ message);
    }
}
```

Real world example

Implementing this example in the real world is really simple. Let's imagine you need to create an application which needs to control several aircrafts. Rather than having aircraft to aircraft communication, it is better to implement a centralized mechanism which receives messages from all aircrafts, processes them, and sends them to the other aircrafts.

In this example there are several aircrafts represented by classes: `Airbus380`, `Boeing747`, and `LearJet45`. These classes inherit the `Aircraft` abstract base class and acts as colleagues. The `RegionalAirTrafficControll` class is the concrete implementation of the mediator class and implements the `IAirTrafficControl` interface (MediatorBase). The `RegionalAirTraffic` control object holds the reference to all aircrafts that need to communicate with each other.

Every time the aircraft changes its flight altitude, it sends a message to the mediator (`RegionalAirTraffincControl`). The mediator finds whether any of the aircrafts is in the same flight corridor of the message sender and if yes, it sends to the aircraft a notification message and the sender is ordered to increase its altitude.

```
public abstract class Aircraft
{
    private readonly IAirTrafficControl _atc;
    private int _altitude;

    public string RegistrationNumber { get; set; }

    public int Altitude
    {
        get { return _altitude; }
        set
        {
            _altitude = value;
            _atc.SendWarningMessage(this);
        }
    }

    public Aircraft(string registrationNumber, IAirTrafficControl atc)
    {

        RegistrationNumber = registrationNumber;
        _atc = atc;
        _atc.RegistrerAircraft(this);
    }

    public void Climb(int heightToClimb)
    {
        Altitude += heightToClimb;
    }
```

```
    public void ReceiveWarning(Aircraft reportingAircraft)
    {
        Console.WriteLine("ATC: ({0}) - {1} is at your flight altitude!!!",
          this.RegistrationNumber,reportingAircraft.RegistrationNumber);
    }
}

public class Airbus380:Aircraft
{
    public Airbus380(string registrationNumber, IAirTrafficControl atc) :
base(registrationNumber, atc)
    {
    }
}

public class Boeing747: Aircraft
{
    public Boeing747(string registrationNumber, IAirTrafficControl atc) :
base(registrationNumber, atc)
    {
    }
}

public class LearJet45:Aircraft
{
    public LearJet45(string registrationNumber, IAirTrafficControl atc) :
base(registrationNumber, atc)
    {
    }
}

public interface IAirTrafficControl
{
    void RegistrerAircraft(Aircraft aircraft);

    void SendWarningMessage(Aircraft aircraft);
}

public class RegionalAirTrafficControl : IAirTrafficControl
{
    readonly List<Aircraft> _registeredAircrafts = new List<Aircraft>();

    public void RegistrerAircraft(Aircraft aircraft)
    {
        if (!_registeredAircrafts.Contains(aircraft))
        {
            _registeredAircrafts.Add(aircraft);
        }
    }

    public void SendWarningMessage(Aircraft aircraft)
    {
        var list = from a in _registeredAircrafts
                   where a != aircraft &&
                         Math.Abs(a.Altitude - aircraft.Altitude) < 1000
                   select a;
        foreach (var a in list)
        {
            a.ReceiveWarning(aircraft);
            aircraft.Climb(1000);
        }
```

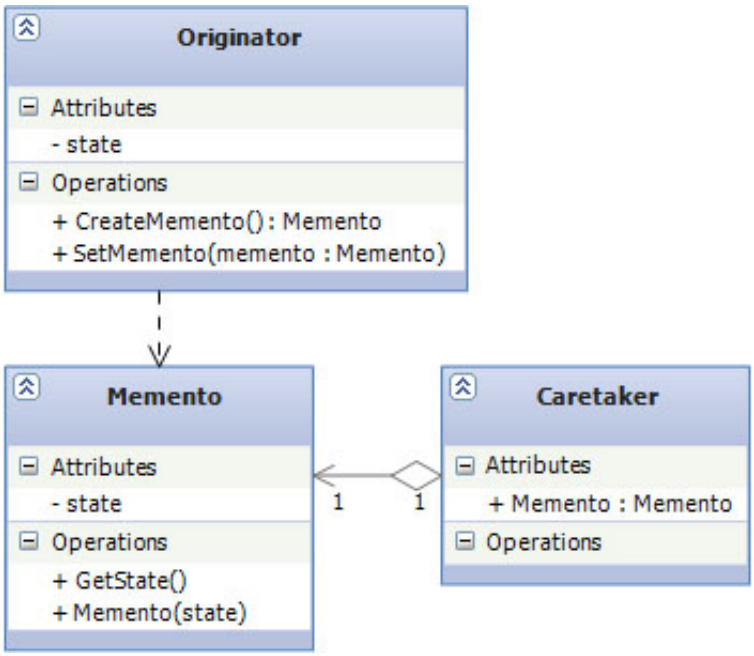
```
    }
}

class Program
{
    static void Main(string[] args)
    {
        var regionalATC = new RegionalAirTrafficControl();
        var aircraft1 = new Airbus380("AI568", regionalATC);
        var aircraft2 = new Boeing747("BA157", regionalATC);
        var aircraft3 = new Airbus380("LW111", regionalATC);

        aircraft1.Altitude += 100;

        aircraft3.Altitude = 1100;
    }
}
```

Memento pattern

The memento pattern is a design pattern that permits the current state of an object to be stored without breaking the rules of encapsulation. The originating object can be modified as required but can be restored to the saved state at any time.

It is sometimes necessary to capture the internal state of an object at some point in time and have the ability to restore that state later at run time. This is the right place where you can use the memento pattern. This pattern is used to provide an application the undo functionality. Sometimes it is called undo with rollback.

Structural code example



The UML diagram below describes an implementation of the memento design pattern. This diagram consists of three parts:

- **Originator:** Creates a memento object capturing the originator's internal state. Originator uses the memento object to restore its previous state.
- **Memento:** Stores internal state of the Originator object. The state can include any number of state variables. The Memento must have two interfaces. An interface to the caretaker: This interface must not allow any operation or any access to the internal state stored by the memento and thus honors encapsulation. The other interface is to the originator and allows the originator to access any state variables necessary to for the originator to restore a previous state.
- **Caretaker:** Is responsible for keeping the memento. The memento is opaque to the caretaker, and the caretaker must not operate on it.

```
static class Program
{
    static void Main()
    {
        var originator = new Originator {State = "State A"};
        Console.WriteLine(originator.State);
        var caretaker = new Caretaker {Memento = originator.CreateMemento()};
        originator.State = "State B";
        Console.WriteLine(originator.State);
        originator.SetMemento(caretaker.Memento);
        Console.WriteLine(originator.State);
    }
}

class Originator
{
    private string _state;

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

    public Memento CreateMemento()
    {
        return (new Memento(_state));
    }

    public void SetMemento(Memento memento)
    {
        State = memento.GetState();
    }
}

class Memento
{
    private readonly string _state;

    public Memento(string state)
    {
        _state = state;
    }

    public string GetState()
    {
        return _state;
```

```
    }
}

class Caretaker
{
    private Memento _memento;

    public Memento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}
```

Real world example

In this example we want to tract state at a point in time of the `Person` class. The state at point in time is stored in the memento class `PersonMemento` which has the same properties (or state variables) as the `Person` class, but here is a little difference. All of the state variables of the memento class are set via a constructor and once they are filled we can't change them. When memento is created it is stored using the `Add` method in the caretaker object's internal stack. `Caretaker` has another method call `GetMemento` which removes and returns the object at the top of the internal stack.

```
public class PersonCaretaker
{
    readonly Stack<PersonMemento> _mementos = new Stack<PersonMemento>();

    public PersonMemento GetMemento()
    {
        return _mementos.Pop();
    }

    public void Add(PersonMemento memento)
    {
        _mementos.Push(memento);
    }
}

public class PersonMemento
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public string CellPhone { get; private set; }
    public string Address { get; private set; }

    public PersonMemento(string fName, string lName, string cell, string address)
    {
        FirstName = fName;
        LastName = lName;
        CellPhone = cell;
        Address = address;
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string CellPhone { get; set; }
```

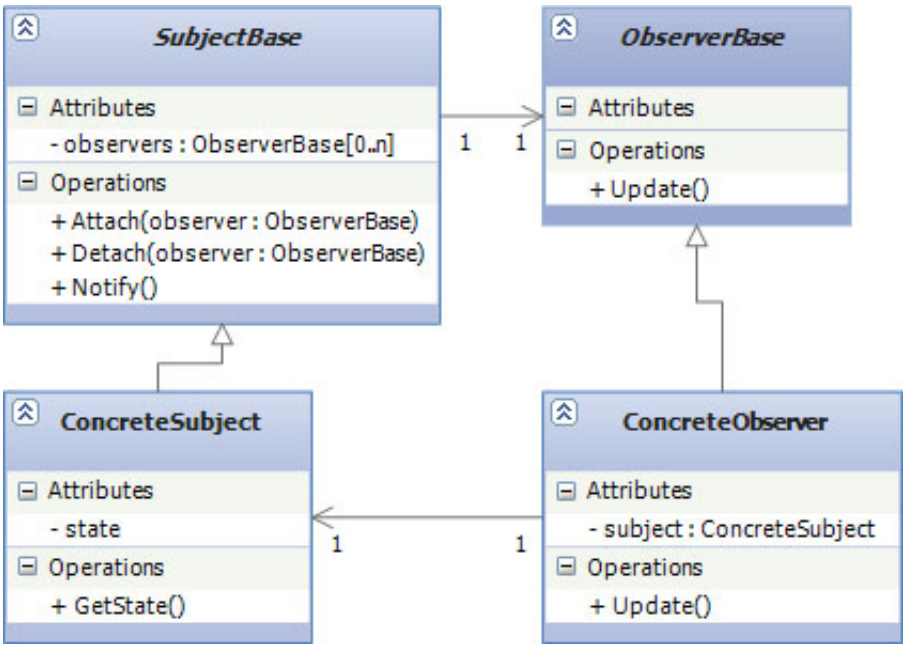
```
    public string Address { get; set; }

    public PersonMemento CreateUnDo()
    {
        return new PersonMemento(FirstName,LastName,CellPhone,Address);
    }

    public  void UnDo(PersonMemento memento)
    {
        FirstName = memento.FirstName;
        LastName = memento.LastName;
        CellPhone = memento.CellPhone;
        Address = memento.Address;
    }

    public void ShowInfo()
    {
        Console.WriteLine("FIRST NAME: {0}",FirstName);
        Console.WriteLine("LAST NAME: {0}", LastName);
        Console.WriteLine("ADDRESS: {0}", Address);
        Console.WriteLine("CELLPHONE: {0}", CellPhone);
    }
}

class Program
{
    static void Main()
    {
        var person = new Person
                        {
                               Address = "Under the Bridge 171",
                               CellPhone = "122011233185",
                               FirstName = "John",
                               LastName = "Doe"
                           };

        var caretaker = new PersonCaretaker();
        caretaker.Add(person.CreateUnDo());

        person.ShowInfo();
        Console.WriteLine();

        person.Address = "Under the Tree 119";
        caretaker.Add(person.CreateUnDo());

        person.ShowInfo();
        Console.WriteLine();

        person.CellPhone = "987654321";
        person.ShowInfo();
        Console.WriteLine();

        person.UnDo(caretaker.GetMemento());
        person.ShowInfo();
        Console.WriteLine();

        person.UnDo(caretaker.GetMemento());
        person.ShowInfo();
        Console.WriteLine();
    }
}
```

Observer pattern

The observer pattern is a design pattern that defines a link between objects so that when one object's state changes, all dependent objects are updated automatically. This pattern allows communication between objects in a loosely coupled manner.

Using this design pattern allows you to monitor and publish changes of a single object called subject. When a change occurs other objects called observers can be automatically notified by calling one of their methods. This pattern gives you a loose coupling between subjects and observers. The subjects hold references to all of the observers which must be notified when changes occur.

Structural code example



The UML diagram below describes an implementation of the observer design pattern. This diagram consists of four parts:

- **SubjectBase:** this is the base class for all subjects. It contains a protected list of observers that are subscribed to the subject and methods allow to add or remove observers. It also contains a method `Notify` which loops through the list of observers and calls their `Notify` method.
- **ConcreteSubject:** this is the concrete implementation of the `SubjectBase` class. It maintains its own state and when a change occurs it calls the `Notify` method which loops through all of the observers and calls their `Notify` method.
- **ObserverBase:** this is the abstract class for all observers. It contains the method which is called when a subject's state changes.
- **ConcreteObserver:** this is the concrete implementation of `ObserverBase`.

```
static class Program
{
    static void Main()
    {
        var subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject,"Observer 1"));
        subject.Attach(new ConcreteObserver(subject,"Observer 2"));
```

```
        subject.Attach(new ConcreteObserver(subject,"Observer 3"));
        subject.SetState("STATE");
    }
}

public abstract class SubjectBase
{
    private ArrayList _observers = new ArrayList();

    public void Attach(ObserverBase o)
    {
        _observers.Add(o);
    }

    public void Detach(ObserverBase o)
    {
        _observers.Remove(o);
    }

    public void Notify()
    {
        foreach (ObserverBase o in _observers)
        {
            o.Update();
        }
    }
}

public class ConcreteSubject : SubjectBase
{
    private string _state;

    public string GetState()
    {
        return _state;
    }

    public void SetState(string newState)
    {
        _state = newState;
        Notify();
    }
}

public abstract class ObserverBase
{
    public abstract void Update();
}


public class ConcreteObserver : ObserverBase
{
    private ConcreteSubject _subject;
    private string _name;

    public ConcreteObserver(ConcreteSubject subject, string name)
    {
        _subject = subject;
        _name = name;
    }

    public override void Update()
```

```
    {
        string subjectState = _subject.GetState();
        Console.WriteLine(_name + ": " +subjectState);
    }
}
```

Real world example

Let's assume we have a stock system which provides data from several companies. In this example we have a two observers: `AllStockObserver` and `IBMStockObserver`. `AllStockObserver` responds to every change of the `ConcreteStockTicker` class. The `IBMStockTicker` class responds only when stocks of the IBM company comes out from `ConcreteStockTicker`.

```
public class AllStockObserver : IStockObserverBase
{
    public AllStockObserver(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Notify(Stock stock)
    {
        Console.WriteLine("Notified {0} of {1}'s " +
                          "change to {2:C}", Name, stock.Code, stock.Price);
    }
}

public class IBMStockObserver:IStockObserverBase
{
    public IBMStockObserver(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Notify(Stock stock)
    {
        if(stock.Code=="IBM")
            Console.WriteLine("Notified {0} of {1}'s " +
                          "change to {2:C}", Name, stock.Code, stock.Price);
    }
}

public class ConcreteStockTicker:StockTickerBase
{
    private Stock stock;
    public Stock Stock
    {
        get { return stock; }
        set {
            stock = value;
            Notify();
        }
    }

    public override void Notify()
```

```
    {
        foreach (var observer in _observers)
        {
            observer.Notify(stock);
        }
    }
}

public class Stock
{
    public decimal Price { get; set; }
    public string Code { get; set; }
}

public interface IStockObserverBase
{
    void Notify(Stock stock);
}

public abstract class StockTickerBase
{
    readonly protected List<IStockObserverBase> _observers = new List<IStockObserverBase>();

    public void Register(IStockObserverBase observer)
    {
        if(!_observers.Contains(observer))
        {
            _observers.Add(observer);
        }
    }

    public void Unregister(IStockObserverBase observer)
    {
        if (_observers.Contains(observer))
        {
            _observers.Remove(observer);
        }
    }

    public abstract void Notify();
}

class Program
{
    static void Main(string[] args)
    {
        var stockTicker = new ConcreteStockTicker();
        var ibmObserver = new IBMStockObserver("ROBER KANASZ");
        var allObserver = new AllStockObserver("IVOR LOTOCASH");

        stockTicker.Register(ibmObserver);
        stockTicker.Register(allObserver);

        foreach (var s in StockData.getNext())
            stockTicker.Stock = s;
    }
}

public static class StockData
{
    private static decimal[] samplePrices =
       new decimal[] { 10.00m, 10.25m, 555.55m, 9.50m, 9.03m, 500.00m, 499.99m, 10.10m, 10.01m
```

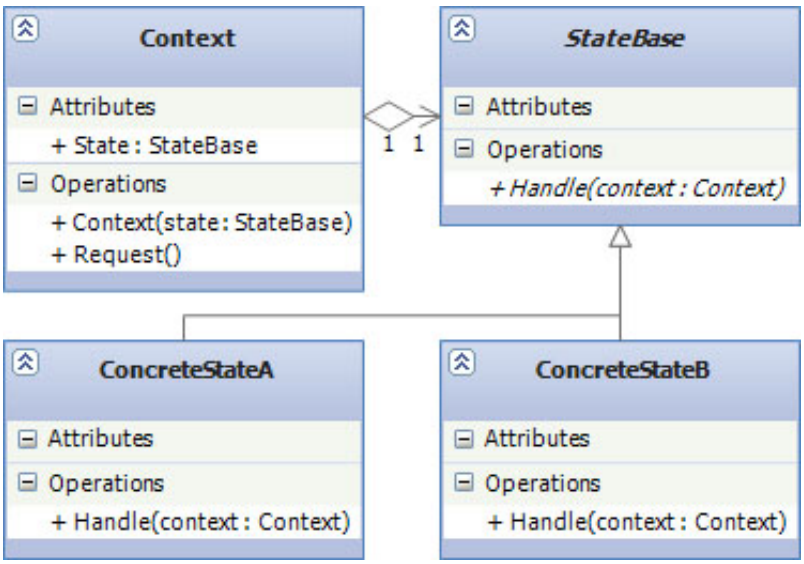
```
};
    private static string[] sampleStocks = new string[] { "MSFT", "MSFT",
      "GOOG", "MSFT", "MSFT", "GOOG",
      "GOOG", "MSFT", "IBM" };
    public static IEnumerable<Stock> getNext()
    {
        for (int i = 0; i < samplePrices.Length; i++)
        {
            Stock s = new Stock();
            s.Code = sampleStocks[i];
            s.Price = samplePrices[i];
            yield return s;
        }
    }
}
```

State pattern

The state pattern is a design pattern that allows an object to completely change its behavior depending upon its current internal state. By substituting classes within a defined context, the state object appears to change its type at run-time.

This design pattern allows you to alter the behavior of an object as a response to changes of its internal state. It allows to change behavior at run-time without changing the interface used to access the object. This change is hidden in the context of this object. This pattern is very useful when creating software state machines, where the functionality of an object changes fundamentally according to its state.

Structural code example



The UML diagram below describes an implementation of the state design pattern. This diagram consists of three parts:

- **Context:** this class holds the concrete state object that provides the behavior according to its current state.
- **StateBase:** this is the abstract class for all concrete state classes. It defines the interface which is consumed by the context class.
- **ConcreteState:** this is the concrete implementation of the `StateBase` class. It provides a real

functionality consumed by the context class.

```
static class Program
{
    static void Main()
    {
        var context = new Context(new ConcreteStateA());
        context.Request();
        context.Request();
        context.Request();
        context.Request();
    }
}

public class Context
{
    private StateBase _state;

    public Context(StateBase state)
    {
        _state = state;
        State = _state;
    }

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

    public StateBase State
    {
        set
        {
            _state = value;
            Console.WriteLine("Current state: {0}",_state.GetType());
        }
    }
}

public abstract class StateBase
{
    public abstract void Handle(Context context);
}

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

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

Real world example

The DVD player was chosen as a real world example on which I want to demonstrate the State design pattern. Let's assume that the DVD player has two buttons: Play and Menu. The DVD player will behave differently when you push this button according to its selected state. The following table describes the behavior of the DVD player according to its state:

State	Play button	Menu button
`MoviePlayingState`	Stop playing. State switched to `MoviePausedState`.	State switched to `MenuState`.
`MoviePausedState`	Start playing. State switched to `MoviePlayingState`.	State switched to `MenuState` `MenuState`.
`StandByState`	Start playing. State switched to `MoviePlayingState`.	State switched to `MenuState` `MenuState`.
`MenuState`	No state change.	State switched to `MenuState` `StandByState`.

```
public class MenuState:DVDPlayerState
{
    public MenuState()
    {
        Console.WriteLine("MENU");
    }

    public override void PlayButtonPressed(DVDPlayer player)
    {
        Console.WriteLine("  Next Menu Item Selected");
    }

    public override void MenuButtonPressed(DVDPlayer player)
    {
        player.State = new StandbyState();
    }
}

public class MoviePausedState:DVDPlayerState
{
    public MoviePausedState()
    {
        Console.WriteLine("MOVIE PAUSED");
    }

    public override void PlayButtonPressed(DVDPlayer player)
    {
        player.State = new MoviePlayingState();
    }

    public override void MenuButtonPressed(DVDPlayer player)
    {
        player.State = new MenuState();
    }
}

public class MoviePlayingState:DVDPlayerState
{
    public MoviePlayingState()
    {
```

```
        Console.WriteLine("MOVIE PLAYING");
    }

    public override void PlayButtonPressed(DVDPlayer player)
    {
        player.State = new MoviePausedState();
    }

    public override void MenuButtonPressed(DVDPlayer player)
    {
        player.State = new MenuState();
    }
}

public class StandbyState:DVDPlayerState
{
    public StandbyState()
    {
        Console.WriteLine("STANDBY");
    }

    public override void PlayButtonPressed(DVDPlayer player)
    {
        player.State = new MoviePlayingState();
    }

    public override void MenuButtonPressed(DVDPlayer player)
    {
        player.State = new MenuState();
    }
}

public class DVDPlayer
{
    private DVDPlayerState _state;

    public DVDPlayer()
    {
        _state = new StandbyState();
    }

    public DVDPlayer(DVDPlayerState state)
    {
        _state = state;
    }

    public void PressPlayButton()
    {
        _state.PlayButtonPressed(this);
    }

    public void PressMenuButton()
    {
        _state.MenuButtonPressed(this);
    }

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

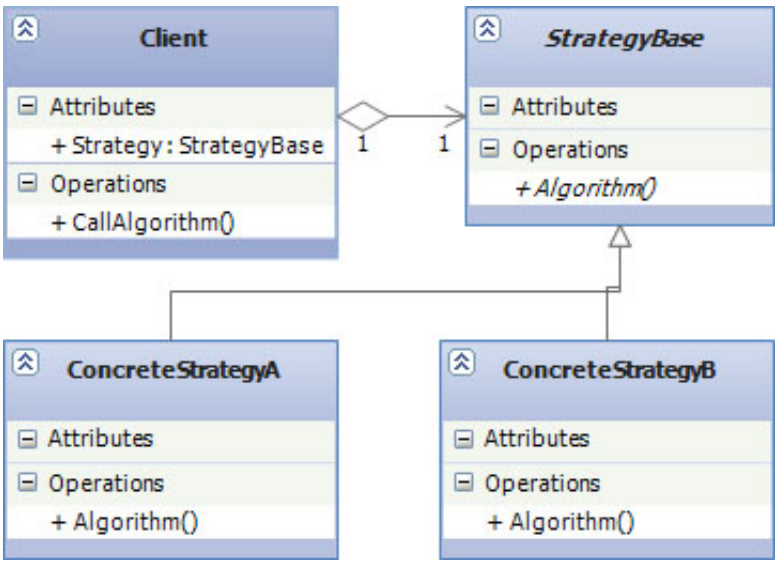
```
public abstract class DVDPlayerState
{
public abstract void PlayButtonPressed(DVDPlayer player);

public abstract void MenuButtonPressed(DVDPlayer player);
}

static void Main()
{
    var player = new DVDPlayer();

    player.PressPlayButton();
    player.PressMenuButton();
    player.PressPlayButton();
    player.PressPlayButton();
    player.PressMenuButton();
    player.PressPlayButton();
    player.PressPlayButton();
}
```

Strategy pattern

The strategy pattern is a design pattern that allows a set of similar algorithms to be defined and encapsulated in their own classes. The algorithm to be used for a particular purpose may then be selected at run-time according to your requirements.

This design pattern can be used in common situations where classes differ only in behavior. In this case it is a good idea to separate these behavior algorithms in separate classes which can be selected at run-time. This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This separation allows behaviors may vary independently of clients that use it. It also increases the flexibility of the application allowing to add new algorithms in future.

Structural code example



The UML diagram below describes an implementation of the strategy design pattern. This diagram consists of three parts:

- **Client:** this class uses interchangeable algorithms. It maintains a reference to the `StrategyBase` object. Sometimes it defines an interface that lets `StrategyBase` access its data.
- **StrategyBase:** declares an interface common to all supported algorithms. The client uses this interface to call the algorithm defined by a `ConcreteStrategy`.
- **ConcreteStrategy:** this is the concrete strategy class inherited from the `StrategyBase` class. Each instance provides a different algorithm that may be used by the client.

```
static class Program
{
    static void Main()
    {
        var client = new Client {Strategy = new ConcreteStrategyA()};
        client.CallAlgorithm();
        client.Strategy = new ConcreteStrategyB();
        client.CallAlgorithm();
    }
}

public class Client
{
    public StrategyBase Strategy { private get; set; }

    public void CallAlgorithm()
    {
        Strategy.Algorithm();
    }
}

public abstract class StrategyBase
{
    public abstract void Algorithm();
}

public class ConcreteStrategyA : StrategyBase
{
    public override void Algorithm()
    {
        Console.WriteLine("Concrete Strategy A");
    }
}

public class ConcreteStrategyB : StrategyBase
{
    public override void Algorithm()
    {
        Console.WriteLine("Concrete Strategy B");
    }
}
```

Real world example

In the following example I will show you how to use the Strategy design pattern. This example application will simulate a different robot behavior. In this example I have created several `Robot` class instances with different behavior. I have created four instances of the `Robot` class. To each of the instance is passed a behavior algorithm using the constructor. We have four kinds of behavior algorithm classes which implement the `IRobotBehaviour` interface.

```
public class AggresiveBehaviour:IRobotBehaviour
{
    public void Move()
    {
        Console.WriteLine("\tAggresive Behaviour: if find another robot, attack it");
    }
}

public class BorgBehaviour:IRobotBehaviour
{
    public void Move()
    {
        Console.WriteLine("\tBorg Behaviour: if find another robot, assimilate it");
    }
}

public class DefensiveBehaviour:IRobotBehaviour
{
    public void Move()
    {
        Console.WriteLine("\tDefensive Behaviour: if find another robot, run from it");
    }
}

public class NormalBehaviour:IRobotBehaviour
{
    public void Move()
    {
        Console.WriteLine("\tNormal Behaviour: if find another robot, ignore it");
    }
}

public interface IRobotBehaviour
{
    void Move();
}

public class Robot
{
    private readonly IRobotBehaviour _behaviour;
    private readonly string _name;
    public Robot(string name, IRobotBehaviour behaviour)
    {
        _behaviour = behaviour;
        _name = name;
    }

    public void Move()
    {
        Console.WriteLine(_name);
        _behaviour.Move();
    }
}

class Program
{
    static void Main()
    {
        var robot1 = new Robot("4of11", new BorgBehaviour());
        var robor2 = new Robot("Military Robot", new AggresiveBehaviour());
        var robot3 = new Robot("rotoROBOt", new NormalBehaviour());
```

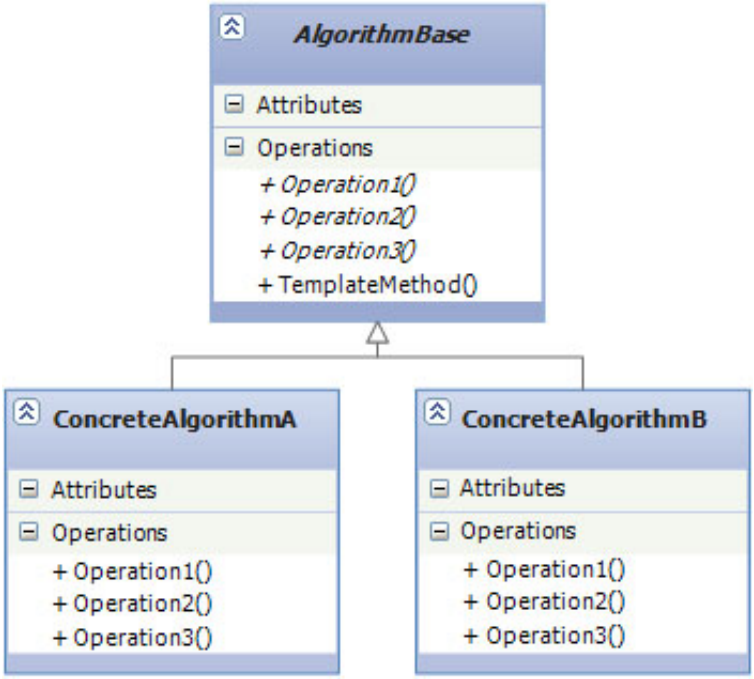
```
        robot1.Move();
        robor2.Move();
        robot3.Move();
    }
}
```

Template method pattern

The template method pattern is a design pattern that allows a group of interchangeable, similarly structured, multi-step algorithms to be defined. Each algorithm follows the same series of actions but provides a different implementation of the steps.

This design pattern allows you to change the behavior of a class at run-time. It is almost the same as Strategy design pattern but with a significant difference. While the Strategy design pattern overrides the entire algorithm, Template method allows you to change only some parts of the behavior algorithm using abstract operations (the entire algorithm is separated into several operations).

Structural code example



The UML diagram below describes an implementation of the template method design pattern. This diagram consists of two parts:

- **AlgorithmBase:** defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm. Implements a template method which defines the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in `AbstractClass` or those of other objects.
- **ConcreteAlgorithm:** implements the primitive operations to carry out subclass-specific steps of the algorithm. When a concrete class is called the template method code will be executed from the base class while for each method used inside the template method will be called the implementation from the derived class.

```
static class Program
```

```
{
    static void Main()
    {
        AlgorithmBase algorithmBase1 = new ConcreteAlgorithmA();
        algorithmBase1.TemplateMethod();

        AlgorithmBase algorithmBase2 = new ConcreteAlgorithmB();
        algorithmBase2.TemplateMethod();
    }
}

public abstract class AlgorithmBase
{
    public void TemplateMethod()
    {
        Step1();
        Step2();
        Step3();
    }

    protected abstract void Step1();

    protected abstract void Step2();

    protected abstract void Step3();
}

public class ConcreteAlgorithmA : AlgorithmBase
{
    protected override void Step1()
    {
        Console.WriteLine("ConcreteAlgorithmA.Step1()");
    }

    protected override void Step2()
    {
        Console.WriteLine("ConcreteAlgorithmA.Step2()");
    }

    protected override void Step3()
    {
        Console.WriteLine("ConcreteAlgorithmA.Step3()");
    }
}


public class ConcreteAlgorithmB : AlgorithmBase
{
    protected override void Step1()
    {
        Console.WriteLine("ConcreteAlgorithmB.Step1()");
    }

    protected override void Step2()
    {
        Console.WriteLine("ConcreteAlgorithmB.Step2()");
    }

    protected override void Step3()
    {
        Console.WriteLine("ConcreteAlgorithmB.Step3()");
    }
```

```
}
```

Real world example

This example is a little notification application that uses different kinds of notification senders. The consumer of the sender is the `Message` class. Each of the senders inherit from the `NotificationSenderBase` class which has an abstract method called `Notify()`.

```
public abstract class NotificationSenderBase
{
    protected SystemOperator _operator;

    public NotificationSenderBase(SystemOperator systemOperator)
    {
        _operator = systemOperator;
    }

    protected abstract string GetNotificationMessageText();


    public abstract void Notify();

}

public class MailNotificationSender : NotificationSenderBase
{
    public MailNotificationSender(SystemOperator systemOperator) : base(systemOperator)
    {
    }

    protected override string GetNotificationMessageText()
    {
        return "[UNSPECIFIED ERROR OCCURED]";
    }

    public override void Notify()
    {
        Console.WriteLine("EMAIL MESSAGE:{0} WAS SET TO: {1}", GetNotificationMessageText(),
_operator.Email);
    }
}

public class Message
{
    public NotificationSenderBase Sender { get; set; }

    public void Send()
    {
        Sender.Notify();
    }
}

public class SmsNotificationSender: NotificationSenderBase
{
    public SmsNotificationSender(SystemOperator systemOperator) : base(systemOperator)
    {
    }

    protected override string GetNotificationMessageText()
    {
```

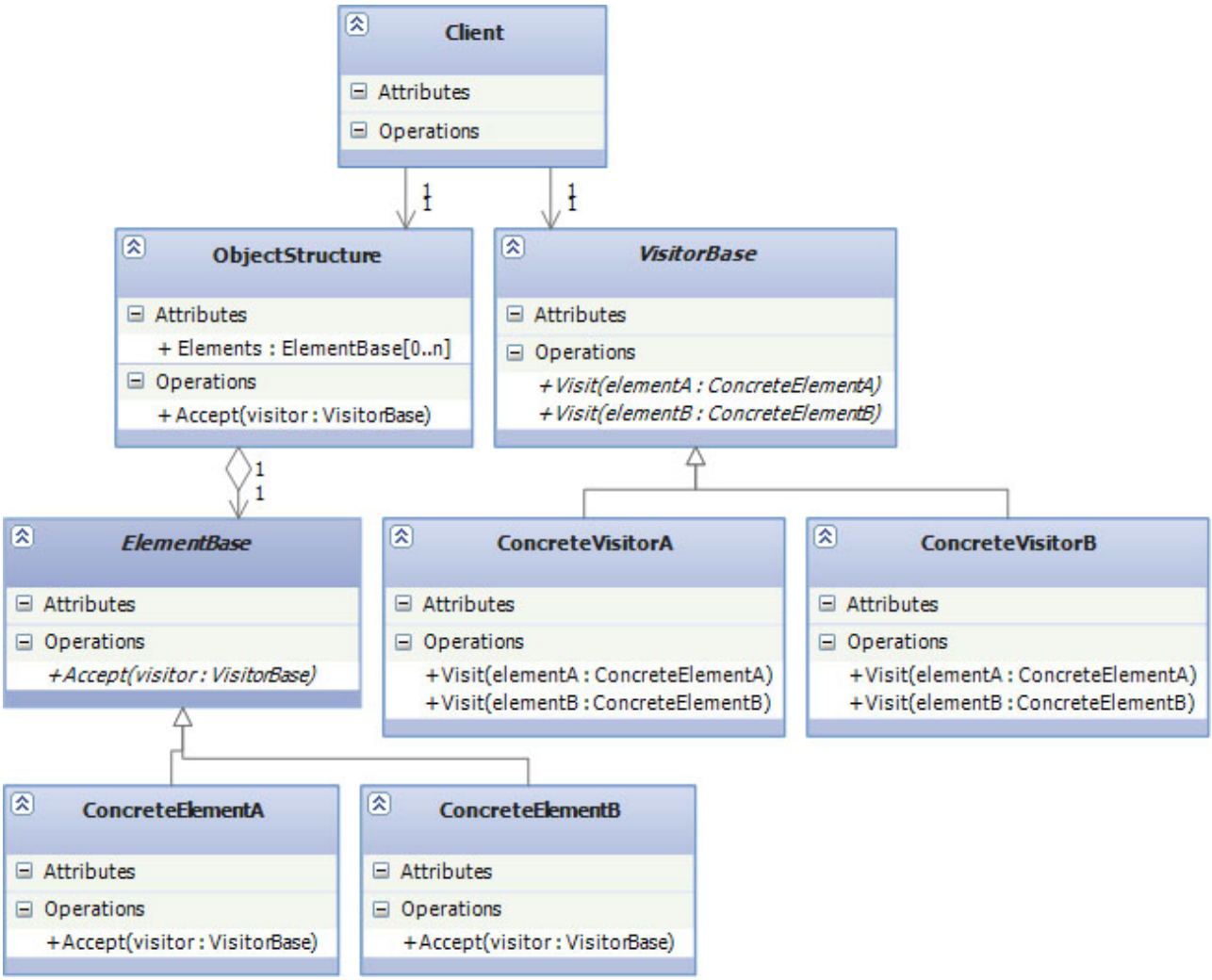
```
        return "UNSPECIFIED ERROR OCCURED";
    }

    public override void Notify()
    {
        Console.WriteLine("SMS MESSAGE:{0} WAS SET TO: {1}",
                GetNotificationMessageText(),_operator.CellPhone);
    }
}

public class SystemOperator
{
    public string Name { get; set; }
    public string Pager { get; set; }
    public string CellPhone { get; set; }
    public string Email { get; set; }
}

class Program
{
    static void Main()
    {

        var systemOperator = new SystemOperator
                                            {
                                                CellPhone = "145616456",
                                                Email = "system@operator.com",
                                                Name = "Super Operator",
                                                Pager = "465565456"
                                            };
        var message = new Message();

        message.Sender = new SmsNotificationSender(systemOperator);
        message.Send();

        message.Sender = new MailNotificationSender(systemOperator);
        message.Send();
    }
}
```

Visitor pattern

The visitor pattern is a design pattern that separates a set of structured data from the functionality that may be performed upon it. This promotes loose coupling and enables additional operations to be added without modifying the data classes.

The Visitor pattern allows you to separate an algorithm from a relatively complex object structure on which it operates. The result of this separation is a data model with limited functionality and a set of visitors that perform operations upon the data. Another benefit is the ability to add a new visitor without modifying the existing structure. The classes of data structures are created with members that can be consumed by the visitor object. Usually this data object is passed to the visitor as a parameter of its method (the convention is to call this method `Visit()`).

Structural code example



The UML diagram below describes an implementation of the visitor design pattern. This diagram consists of six parts:

- **Client:** this class is a consumer of the Visitor pattern. It manages the object structure and instructs the objects within the structure when to accept a visitor.
- **ObjectStructure:** this is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. In can be a complex structure, such as a composite object.
- **ElementBase:** is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object. Each object from a collection should implement this abstraction in order to be able to be visited.
- **ConcreteElement:** Those classes inherit from the base abstract class `ElementBase` or implements an interface and defines the accept operation. The visitor object is passed to this object using the accept operation.
- **VisitorBase:** declares a `Visit` operation for each class of `ConcreteElement` in the object structure. The operation's name and signature identifies the class that sends the `Visit` request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the elements directly through its particular interface.
- **ConcreteVisitor:** implements each operation declared by a `Visitor`. Each operation implements a fragment of the algorithm defined for the corresponding class or object in the structure. `ConcreteVisitor` provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

```
static class Program
{
    static void Main()
    {
        var objectStructure = new ObjectStructure();
        objectStructure.Attach(new ConcreteElementA());
        objectStructure.Attach(new ConcreteElementB());
        objectStructure.Accept(new ConcreteVisitorA());
        objectStructure.Accept(new ConcreteVisitorB());
    }
}

public abstract class ElementBase
{
    public abstract void Accept(VisitorBase visitor);
}

public abstract class VisitorBase
{
    public abstract void Visit(ConcreteElementA element);

    public abstract void Visit(ConcreteElementB element);
}

public class ObjectStructure
{
    private List<ElementBase> _elements;

    public ObjectStructure()
    {
        _elements = new List<ElementBase>();
    }

    public void Accept(VisitorBase visitor)
    {
        foreach (ElementBase element in _elements)
        {
            element.Accept(visitor);
        }
    }

    public void Attach(ElementBase element)
    {
        _elements.Add(element);
    }

    public void Detach(ElementBase element)
    {
        _elements.Remove(element);
    }
}

public class ConcreteElementA : ElementBase
{
    public override void Accept(VisitorBase visitor)
    {
        visitor.Visit(this);
    }

    public string Name { get; set; }
}
```

```
public class ConcreteElementB : ElementBase
{
    public override void Accept(VisitorBase visitor)
    {
        visitor.Visit(this);
    }

    public string Title { get; private set; }
}

public class ConcreteVisitorA : VisitorBase
{
    public override void Visit(ConcreteElementA element)
    {
        Console.WriteLine("VisitorA visited ElementA : {0}", element.Name);
    }

    public override void Visit(ConcreteElementB element)
    {
        Console.WriteLine("VisitorA visited ElementB : {0}", element.Title);
    }
}

public class ConcreteVisitorB : VisitorBase
{
    public override void Visit(ConcreteElementA element)
    {
        Console.WriteLine("VisitorB visited ElementA : {0}", element.Name);
    }

    public override void Visit(ConcreteElementB element)
    {
        Console.WriteLine("VisitorB visited ElementB : {0}", element.Title);
    }
}
```

Real world example

In this example we have a `Car` class which represents the object structure. This class contains a generic list of `IElementBase` objects which represent car parts. To support the Visitor design pattern, every concrete implementation of the `IElementBase` object contains a method called `Accept` that attaches a car part to the concrete visitor. We have a concrete visitor which displays information about car parts. Every visitor in this case must implement the `IVisitorBase` interface. This interface contains a signature for four `Visit` methods. Each of these methods consumes a different kind of car part.

```
public class Body : IElementBase
{

    public Body(string  name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Accept(IVisitorBase visitor)
    {
        visitor.Visit(this);
```

```
    }
}

public class Engine : IElementBase
{

    public Engine(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Accept(IVisitorBase visitor)
    {
        visitor.Visit(this);
    }
}

public class Transmission : IElementBase
{
    public Transmission(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    public void Accept(IVisitorBase visitor)
    {
        visitor.Visit(this);
    }
}

public class Wheel : IElementBase
{

    public Wheel(string  name)
    {
        Name = name;
    }

    public string Tire { get; set; }

    public string Name { get; set; }

    public void Accept(IVisitorBase visitor)
    {
        visitor.Visit(this);
    }

    void IElementBase.Accept(IVisitorBase visitor)
    {
        throw new System.NotImplementedException();
    }
}

public class CarElementPrintVisitor:IVisitorBase
{
    public void Visit(Wheel wheel)
    {
        Console.WriteLine("Wheel ({0}): {1}",wheel.Name,wheel.Tire);
```

```
    }

    public void Visit(Body body)
    {
        Console.WriteLine("Body: {0}",body.Name);
    }

    public void Visit(Engine engine)
    {
        Console.WriteLine("Engine: {0}",engine.Name);
    }

    public void Visit(Transmission transmission)
    {
        Console.WriteLine("Transmission: {0}",transmission.Name);
    }
}

public interface IElementBase
{
    void Accept(IVisitorBase visitor);
}

public class Car : IElementBase
{
    private readonly List<IElementBase> _parts;

    public Car()
    {
        _parts=new List<IElementBase>
                  {
                      new Wheel("Front Left"){Tire = "Michelin Pilot Super Sport"},
                      new Wheel("Front Right"){Tire = "Michelin Pilot Super Sport"},
                      new Wheel("Back Left"){Tire = "Michelin Pilot Super Sport"},
                      new Wheel("Back Right"){Tire = "Michelin Pilot Super Sport"},
                      new Engine("3.3 TDI 225"),
                      new Body("4-door sedan"),
                      new Transmission("5-speed manual")
                  };
    }

    public void Accept(IVisitorBase visitor)
    {
        foreach (var part in _parts)
        {
            part.Accept(visitor);
        }
    }
}

public interface IVisitorBase
{
    void Visit(Wheel wheel);
    void Visit(Body wheel);
    void Visit(Engine wheel);
    void Visit(Transmission wheel);
}

class Program
{
    static void Main()
    {
```

```
        var visitor = new CarElementPrintVisitor();

        var car = new Car();
        car.Accept(visitor);
    }
}
```

History

- 7 September - Original version posted
- 7 October - `SubtractionExpression`/`ToString` method fixed.

License

About the Author

Kanasz Robert
Architect The Staffing Edge &
Marwin Cassovia Soft
Slovakia

My name is Robert Kanasz and I have been working with ASP.NET, WinForms and C# for several years.

MCTS - .NET Framework 3.5, ASP.NET Applications
- SQL Server 2008, Database Development
- SQL Server 2008, Implementation and Maintenance
- .NET Framework 4, Data Access
- .NET Framework 4, Service Communication Applications
- .NET Framework 4, Web Applications
MCPD - ASP.NET Developer 3.5
- Web Developer 4
MCITP - Database Administrator 2008
- Database Developer 2008

Open source projects: DBScripter - Library for scripting SQL Server database objects

Please, do not forget vote

Comments and Discussions

102 messages have been posted for this article Visit **http://www.codeproject.com/Articles/455228/Design-Patterns-3-of-3-Behavioral-Design-Patterns** to post and view comments on this article, or click **here** to get a print view with messages.