Паттерны

Doc

- Книга: Приемы ООП. Паттерны проектирования (GOF Book)
- Книга: Паттерны пароектирования на платформе .NET
- https://metanit.com/sharp/patterns
- https://refactoring.guru/ru/design-patterns
- https://makedev.org/principles/index.html
- https://sourcemaking.com/design_patterns
- https://www.geeksforgeeks.org/software-design-patterns
- https://www.oodesign.com
- https://www.dofactory.com/net/design-patterns
- https://www.codeproject.com/Articles/145546/Factory-Pattern-using-Generics
- https://stackoverflow.com/questions/39386586/c-sharp-generic-interface-and-factory-pattern
- https://codereview.stackexchange.com/questions/8307/implementing-factory-design-pattern-with-generics
- https://www.codeproject.com/Articles/21043/Generic-Abstract-Factory
- https://stackoverflow.com/questions/1874049/explanation-of-the-uml-arrows

Паттерн - шаблон проектирования. Практически во всех шаблонах инкапсулируется создание / поведение / структурность

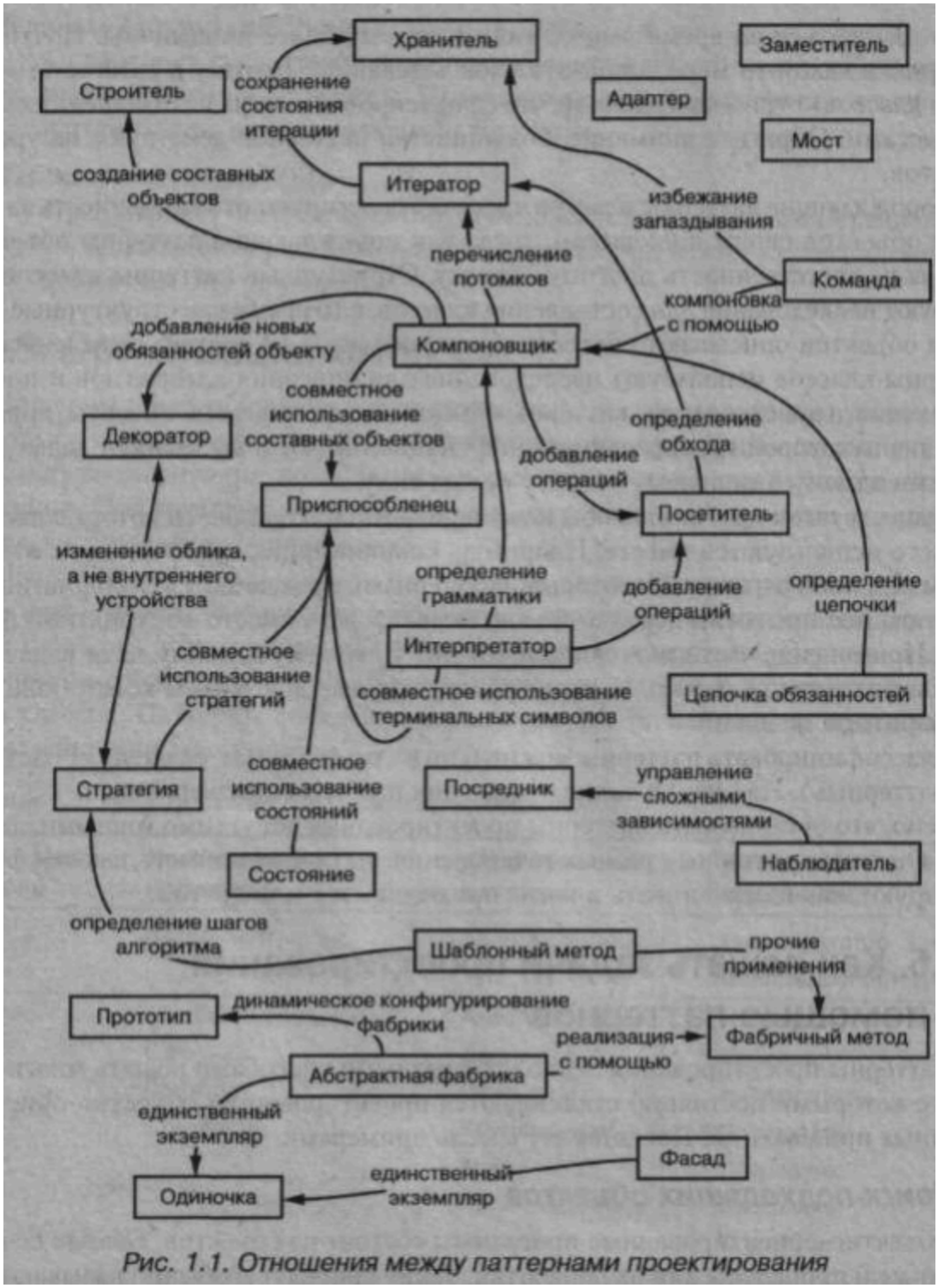
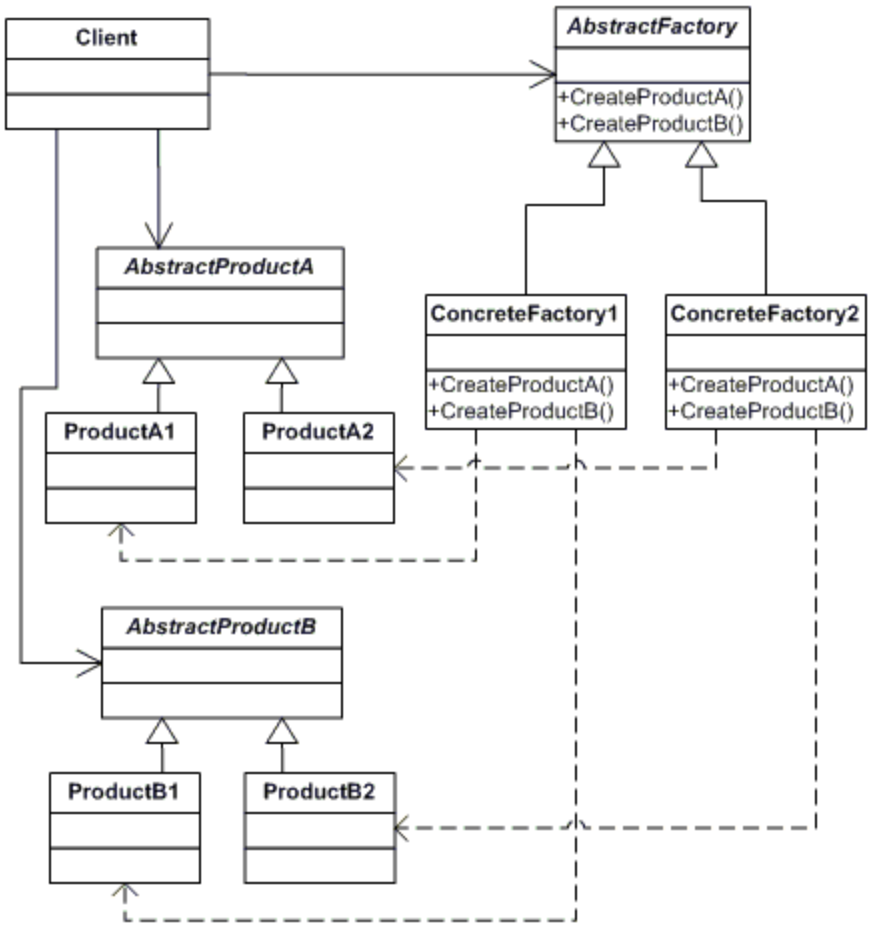


Рис. 1.1. Отношения между паттернами проектирования

Порождающие паттерны (Creational Patterns)

- Абстрактная фабрика (Abstract Factory) - создание разных, но взаимосвязанных объектов (семейство объектов)

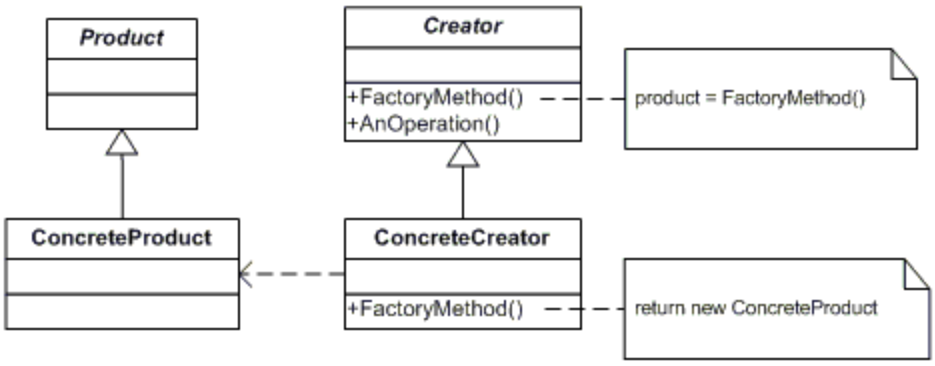


- Абстрактная фабрика
  - интерфейс, который содержит методы создания конкретных продуктов
- Конкретная фабрика
  - реализует интерфейс
- Абстрактный продукт
  - интерфейс конкретного продукта фабрики
- Продукт
  - конкретный продукт конкретной фабрики
- Клиент
  - принимает на вход фабрику и использует ее в дальнейших методах

```
// Abstract Product
public interface IDbConnection
{
    void Open();
}

// Abstract Factory
```

```
public interface IDatabaseFactory
{
   IDbConnection CreateConnection();
}

// Concrete Product
public class SqlConnection : IDbConnection
{
   public void Open()
   {
      Console.WriteLine("SQL Server Connection Opened");
   }
}

// Concrete Factory
public class SqlServerFactory : IDatabaseFactory
{
   public IDbConnection CreateConnection()
   {
      return new SqlConnection();
   }
}

static void Main(string[] args)
{
   // client creates factory
   IDatabaseFactory factory = new SqlServerFactory()

   // factory creates product
   IDbConnection connection = factory.CreateConnection();

   // client uses product
   connection.Open();
}
```

- Фабричный метод (Factory Method) - метод создания объекта (в отличие от абстрактной фабрики продукт один)

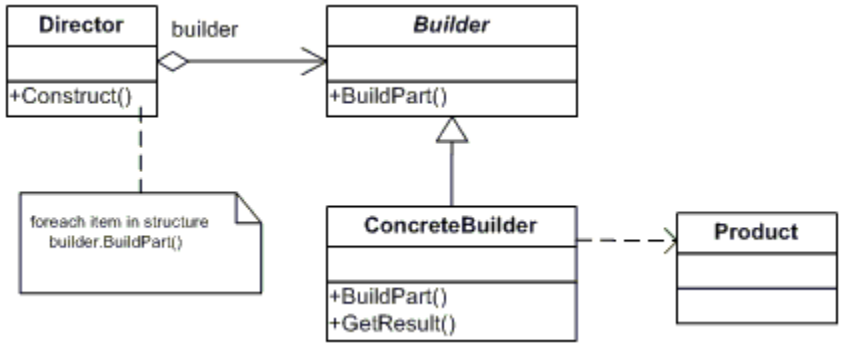


- Product
  - интерфейс продукта
- ConcreteProduct
  - конкретный продукт
- Creator
  - интерфейс создателя объекта, содержит фабричный метод
- ConcreteCreator
  - конкретный создатель

```
class MainApp
{
   static void Main()
   {
      var creator = new ConcreteCreator();

      Product product = creator.FactoryMethod();
   }
}

abstract class Product { }

class ConcreteProduct : Product { }

abstract class Creator
{
   public abstract Product FactoryMethod();
}

class ConcreteCreator : Creator
{
   public override Product FactoryMethod()
   {
      return new ConcreteProduct();
   }
```

}

- Строитель (Builder) - поэтапное создание объекта (в отличие от фабричного метода создание состоит из этапов, количество и названия этапов для создания объекта определяются директором, но реализация этапов у конкретных строителей разная)

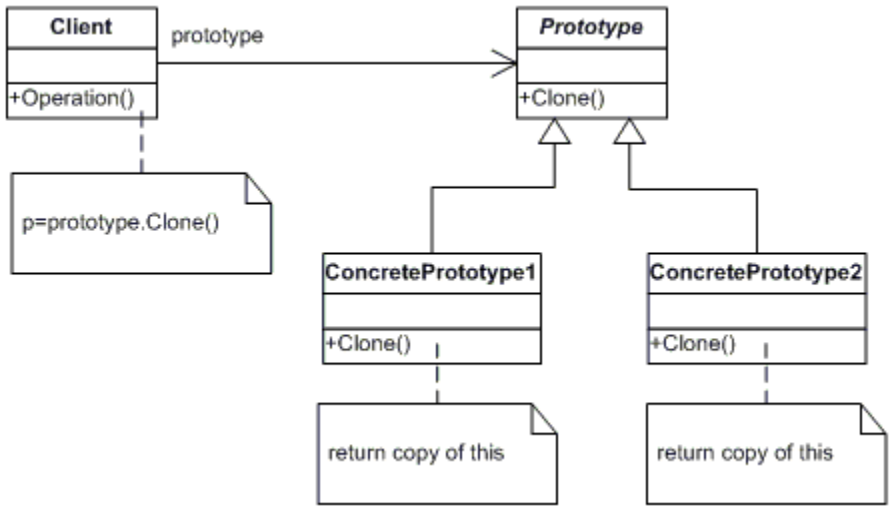


- Строитель
  - интерфейс содержит метод поэтапного создания объекта
- Конкретный строитель
  - создает поэтапно объект
- Директор
  - заказывает объект у строителя, определяет порядок этапов
- Продукт

```
public class MainApp
{
   public static void Main()
   {
      var director = new Director();
      var builder = new ConcreteBuilder();

      director.Construct(builder);
      var product = builder.GetResult();
   }
}

class Product { }

abstract class Builder
{
   // можно добавить дополнительные методы - этапы: BuildPartA, BuildPartB
   public abstract void BuildPart();
   public abstract Product GetResult();
}
```

```
class ConcreteBuilder : Builder
{
    private Product _product = new Product();

    public override void BuildPart() { } // change product

    public override Product GetResult()
    {
        return _product;
    }
}

class Director
{
    public void Construct(Builder builder)
    {
        builder.BuildPart();
    }

}
```

- Прототип (Prototype) - создание новых объектов на основе объекта-прототипа



- Прототип
  - интерфейс, который содержит метод для самоклонирования
- Конкретный прототип
  - класс, реализующий интерфейс
- Клиент
  - создает новый объект путем клонирования объекта-прототипа

Копирование

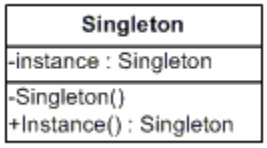
```
public class Company { public string Name { get; set; } }

class Person
{
   public string Name { get; set; } // значимый тип
   public Company Company { get; set; } // ссылочный тип

   public object GetShallowCopy() // неглубокое(поверхностное) копирование
   {
      return this.MemberwiseClone(); // возвращает тип object

      // преобразуется

      return new Person() // упаковывается в тип object неявно
      {
         Name = this.Name, // переменная значимого типа
         Company = this.Company // ссылка на ссылочную переменную
      };
   }

   public object GetDeepCopy() // глубокое копирование
   {
      return new Person() // упаковывается в тип object неявно
      {
         Name = this.Name, // переменная значимого типа
         Company = new Company() { Name = this.Name } // новый экземпляр
      };
   }
}
```

P.S. Для неглубокого копирования: если на момент копирования ссылочный тип равен null (область памяти для переменной не определена), то впоследствии при изменении одного поля другое остается равным null. Вместо методов GetShallowCopy и GetDeepCopy и можно наследоваться от интерфейса ICloneable и реализовать метод object Clone().

```
class MainApp
{
   static void Main()
   {
      var p = new ConcretePrototype();
      var clone = p.Clone();
   }
```

```
}

abstract class Prototype
{
   public abstract Prototype Clone();
}

class ConcretePrototype : Prototype
{
   public override Prototype Clone()
   {
      return (Prototype)this.MemberwiseClone(); // метод доступен только для this
   }
}
```

- [Одиночка (Singleton)]



- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

Singleton - в математике означает множество из одного элемента.

В отличие от статического класса одиночка может наследоваться (например реализовывать интерфейс, это может быть удобно для того, чтобы вызывать разные версии одного и того же метода: версия для класса Singleton, версия для интерфеса 1, версия для интерфеса 2).

Конструктор либо приватный либо защищенный для того, чтобы запретить явное создание через оператор new.

Одиночка может быть запечатанным **sealed** классом, если наследование от одиночки запрещено (но одиночка может наследоваться) и тогда конструктор может быть приватным **private**, иначе одиночка может быть незапечатанным классом, тогда наследование от одиночки разрешено (например для построения иерархии классов) и тогда конструктор может быть защищенным **protected** для наследования конструктора одиночки : base() классом-наследником.

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

```
        var s1 = Singleton.Instance();
        var s2 = Singleton.Instance();

        // s1 = s2
    }
}
```

Однопоточная реализация. Инициализация в независимости от использования

```
public sealed class Singleton // запечатанный класс
{
    private static readonly Singleton _instance = new Singleton(); // инициализация

    private Singleton() { } // приватный, так как наследование запрещено

    public static Singleton Instance => _instance; // get accessor
}
```

Однопоточная реализация с отложенной инициализацией

```
class Singleton
{
    private static Singleton _instance;

    protected Singleton() { } // наследование разрешено

    public static Singleton Instance()
    {
        if (_instance == null) // отложенная инициализация
            _instance = new Singleton();

        return _instance;
    }

}
```

Многопоточная реализация с отложенной инициализацией

P.S. вместо volatile можно применять VolatileRead и VolatileWrite, так как после записи (инициализации) следуют в основном операции чтения (кэширование значения).

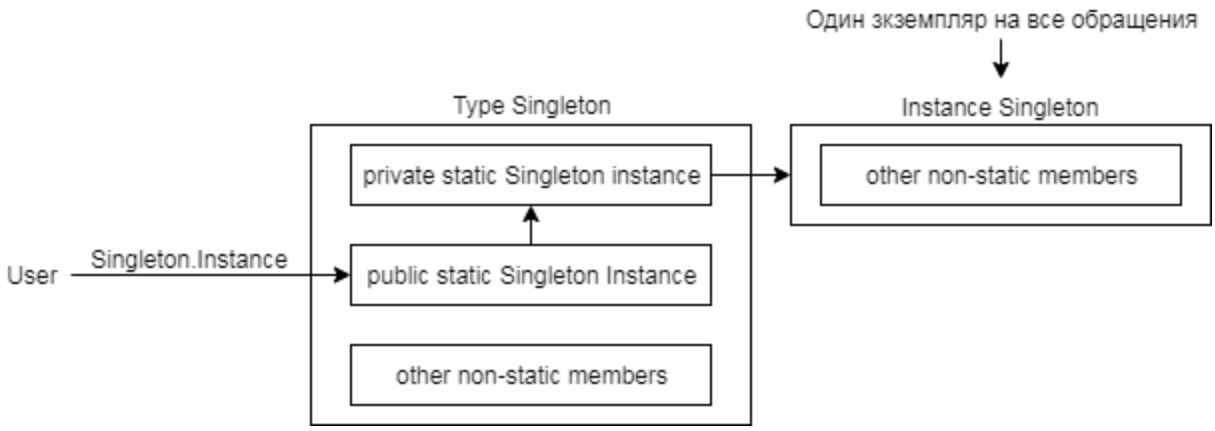
```
public sealed class Singleton
{
  static volatile Singleton _instance; // приватное поле
```

```
  static object _lockObj = new Object(); // приватное поле

  private Singleton() {}

  public static Singleton Instance // публичное свойство
  {
    get
    {
      // после инициализации значения блок Lock не нужно будет выполнять
      if (_instance == null)
      {
        lock (_lockObj) // фрагмент выполняется одним потоком
        {
          // проверка на то что поток не первым выполняет Lock и поток перед ним уже // выполнил блок и
инициализирова значение пока текущий потом был на // проверке перед блоком Lock

          if (_instance == null)
            _instance = new Singleton(); // инициализация значения
        }
      }

      return _instance;
    }
  }
}
```

Приведенный выше шаблон ленивой инициализации безопасной в отношении потоков называется блокировкой с двойной проверкой и временным (volatile) чтением.

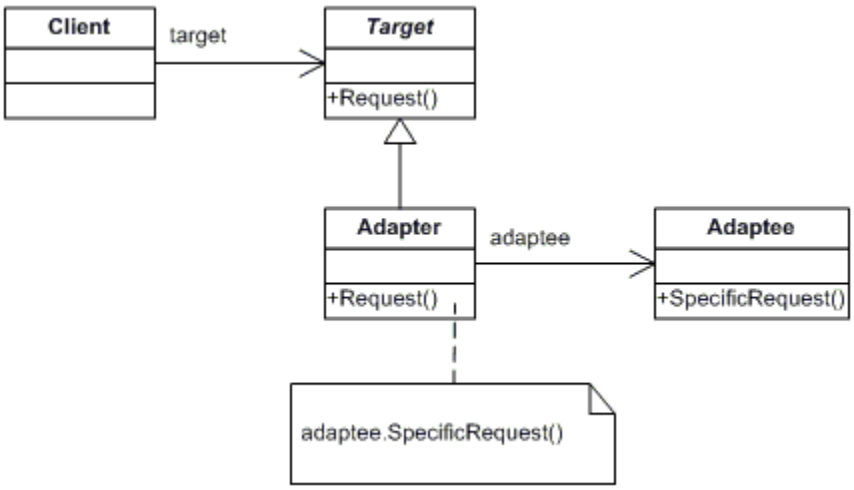
Существует встроенная реализация данного шаблона классом Lazy<T>

```
public sealed class Singleton
{
   static Lazy<Singleton> _instance = new Lazy<Singleton>(
      () => new Singleton(), // делегат инициализации
      true); // безопасно в отношении потоков (для многопоточного приложения)

   private Singleton() { }

   public static Singleton Instance => _instance.Value;
}
```



Структурные паттерны (Structural Patterns)

- [Адаптер (Adapter)]

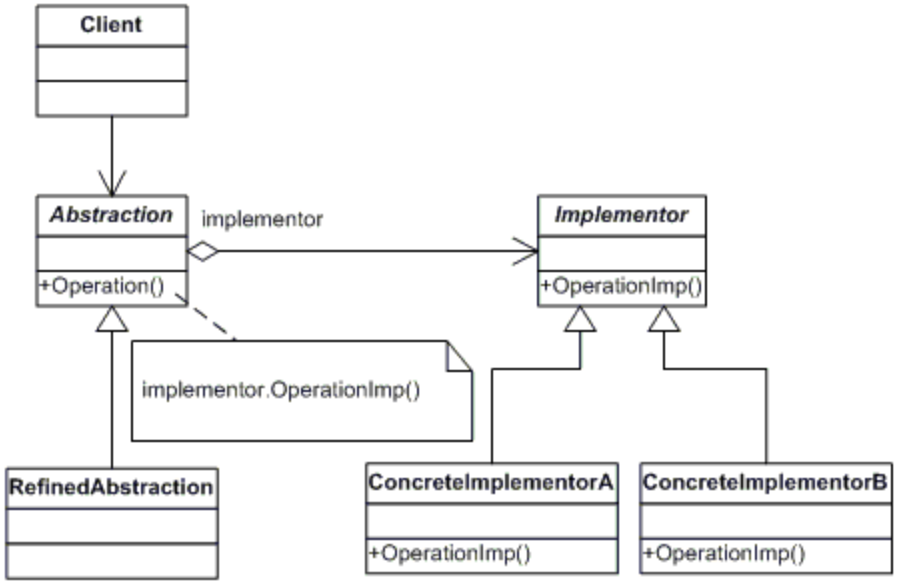


- Target (ChemicalCompound)
  - defines the domain-specific interface that Client uses.
- Adapter (Compound)
  - adapts the interface Adaptee to the Target interface.
- Adaptee (ChemicalDatabank)
  - defines an existing interface that needs adapting.
- Client (AdapterApp)
  - collaborates with objects conforming to the Target interface.

class MainApp

```
{
    static void Main()
    {
        var target = new Adapter();
        target.Request();
    }
}

class Target
{
    public virtual void Request() { }
}

class Adapter : Target
{
    private Adaptee _adaptee = new Adaptee();

    public override void Request()
    {
        _adaptee.SpecificRequest();
    }
}

class Adaptee
{
    public void SpecificRequest() { }
}
```

- Мост (Bridge)

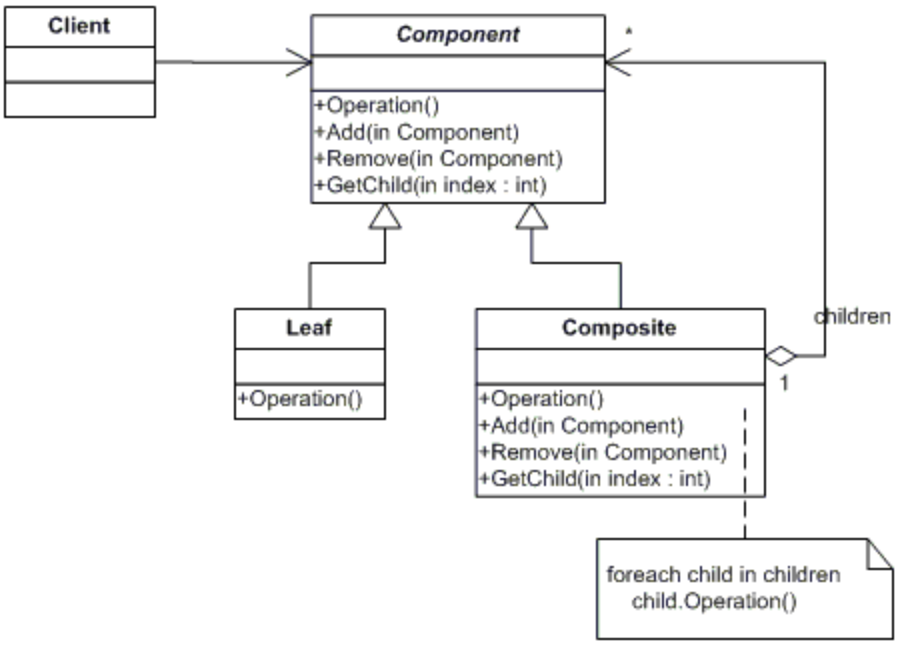


- Abstraction (BusinessObject)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- RefinedAbstraction (CustomersBusinessObject)
  - extends the interface defined by Abstraction.
- Implementor (DataObject)
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor (CustomersDataObject)
  - implements the Implementor interface and defines its concrete implementation.

```
class MainApp
{
    static void Main()
    {
        var a = new ConcreteAbstraction();

        a.Implementor = new ConcreteImplementor();
        a.Operation();
    }
}

abstract class Implementor
{
    public abstract void Operation();
}
```

```
class ConcreteImplementor : Implementor
{
    public override void Operation() { }
}

class Abstraction
{
    protected Implementor implementor;

    public Implementor Implementor {
        set { implementor = value; }
    }

    public virtual void Operation()
    {
        implementor.Operation();
    }
}

class ConcreteAbstraction : Abstraction
{
    public override void Operation() { }
}
```

- Компоновщик (Composite)



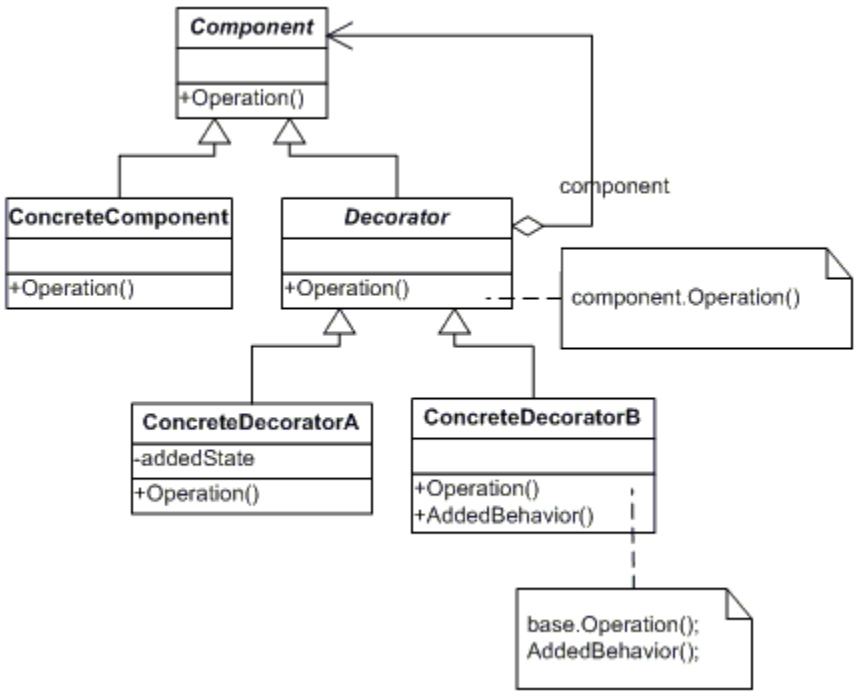
- Component (DrawingElement)
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf (PrimitiveElement)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite (CompositeElement)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client (CompositeApp)
  - manipulates objects in the composition through the Component interface.

```
class MainApp
{
    static void Main()
    {
        var root = new Composite("root");
        root.Add(new Leaf("root leaf"));

        var branch = new Composite("branch");
        branch.Add(new Leaf("branch leaf"));
```

```
        root.Add(branch);

        root.Display(0);

        Output:
        root
            root leaf
            branch
                branch leaf
    }
}

abstract class Component
{
    protected string name;

    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Add(Component c);
    public abstract void Remove(Component c);

    public virtual void Display(int depth)
    {
        Console.WriteLine(new String('\t', depth) + name); // new String (char, count)
    }
}

class Leaf : Component
{
    public Leaf(string name)
      : base(name) { }

    public override void Add(Component c) { }

    public override void Remove(Component c) { }
}

class Composite : Component
{
    private List<Component> _children = new List<Component>();
```

```
    public Composite(string name)
      : base(name) { }

    public override void Add(Component component)
    {
        _children.Add(component);
    }

    public override void Remove(Component component)
    {
        _children.Remove(component);
    }

    public override void Display(int depth)
    {
        base.Display(depth);

        foreach (var component in _children)
            component.Display(depth + 1);
    }
}
```

- [Декоратор (Decorator)]



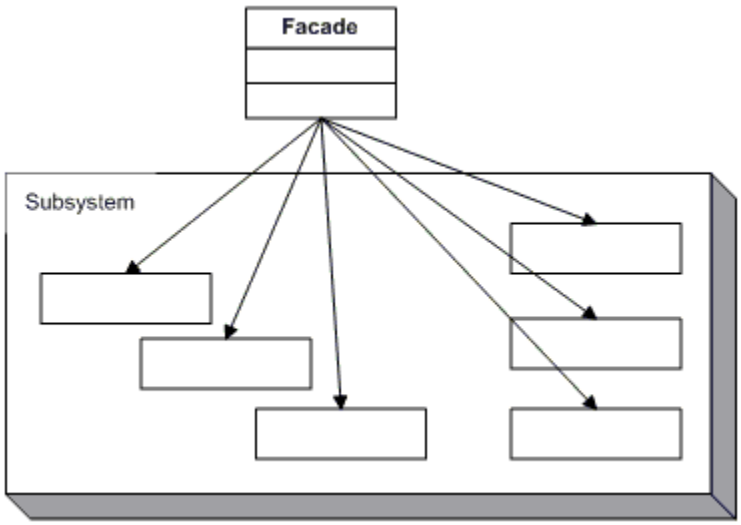
- Component (LibraryItem)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent (Book, Video)
  - defines an object to which additional responsibilities can be attached.
- Decorator (Decorator)
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator (Borrowable)
  - adds responsibilities to the component.

Декоратор - обертка над классом (реализует поведение класса + дополнительный функционал - например обеспечивает многопоточность для класса)

```
class MainApp
{
    static void Main()
    {
        var c = new ConcreteComponent();

        var d = new ConcreteDecorator();
        d.SetComponent(c);

        d.Operation();
    }
}

abstract class Component
{
    public abstract void Operation();
}

class ConcreteComponent : Component
{
    public override void Operation() { }
}

abstract class Decorator : Component
{
    protected Component component;

    public void SetComponent(Component component)
    {
        this.component = component;
    }
```

```
    public override void Operation()
    {
        if (component != null)
            component.Operation();
    }
}

class ConcreteDecorator : Decorator
{
    public override void Operation()
    {
        base.Operation();
        AddedBehavior();
    }

    void AddedBehavior() { }
}
```

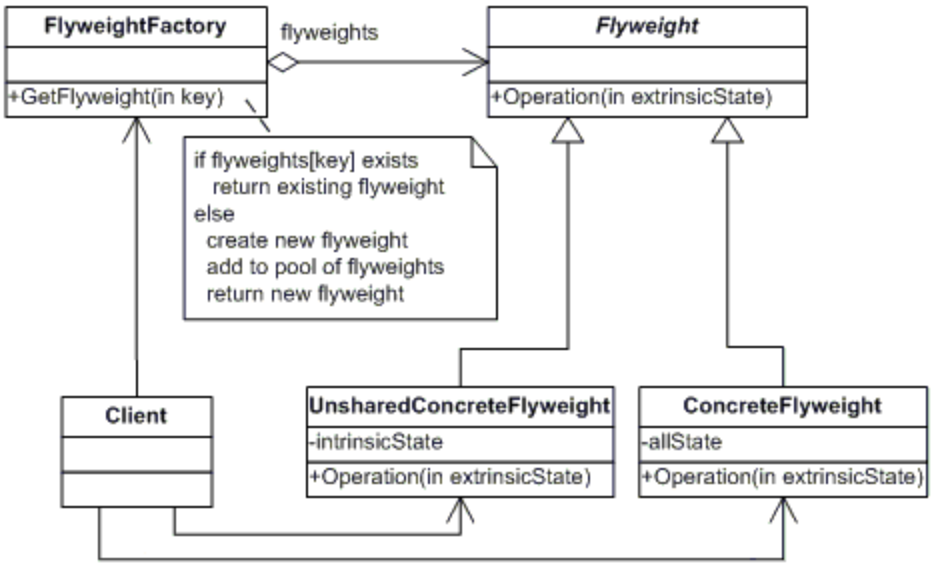
- Фасад (Facade)



- Facade (MortgageApplication)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- Subsystem classes (Bank, Credit, Loan)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade and keep no reference to it.

```
class MainApp
{
    public static void Main()
    {
        var facade = new Facade();

        facade.MethodA();
    }
}

class SubSystemOne
{
    public void MethodOne() { }
}

class SubSystemTwo
{
    public void MethodTwo() { }
}

class Facade
{
    private SubSystemOne _one;
    private SubSystemTwo _two;

    public Facade()
    {
        _one = new SubSystemOne();
        _two = new SubSystemTwo();
    }

    public void MethodA()
    {
        _one.MethodOne();
        _two.MethodTwo();
    }
}
```

- Приспособленец (Flyweight)

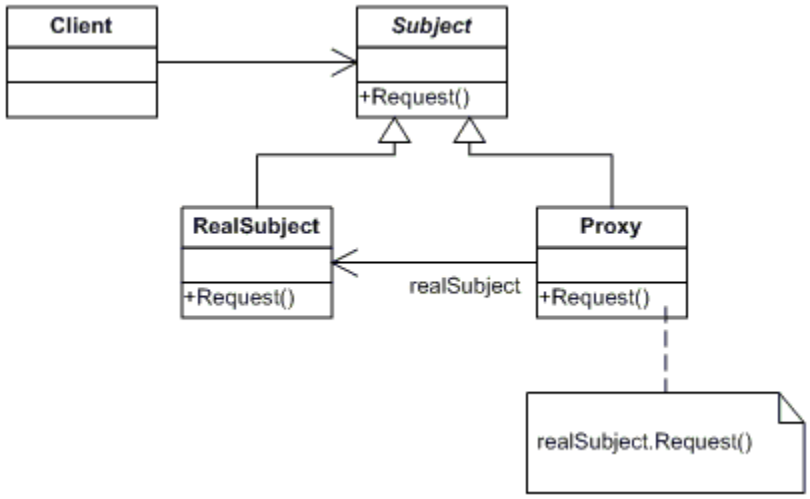


- Flyweight (Character)
  - declares an interface through which flyweights can receive and act on extrinsic state.
- ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.
- UnsharedConcreteFlyweight (not used)
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- FlyweightFactory (CharacterFactory)
  - creates and manages flyweight objects
  - ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects assets an existing instance or creates one, if none exists.
- Client (FlyweightApp)2
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

```
class MainApp
{
    static void Main()
    {
        int extrinsicstate = 22;

        var factory = new FlyweightFactory();

        var fx = factory.GetFlyweight("X");
        fx.Operation(--extrinsicstate);
    }
```

```
}

class FlyweightFactory
{
    private Hashtable flyweights = new Hashtable();

    public FlyweightFactory()
    {
        flyweights.Add("X", new ConcreteFlyweight());
    }

    public Flyweight GetFlyweight(string key)
    {
        return ( (Flyweight) flyweights [ key ] );
    }
}

abstract class Flyweight
{
    public abstract void Operation(int extrinsicstate);
}

class ConcreteFlyweight : Flyweight
{
    public override void Operation(int extrinsicstate) { }
}
```

- [Заместитель (Proxy)]



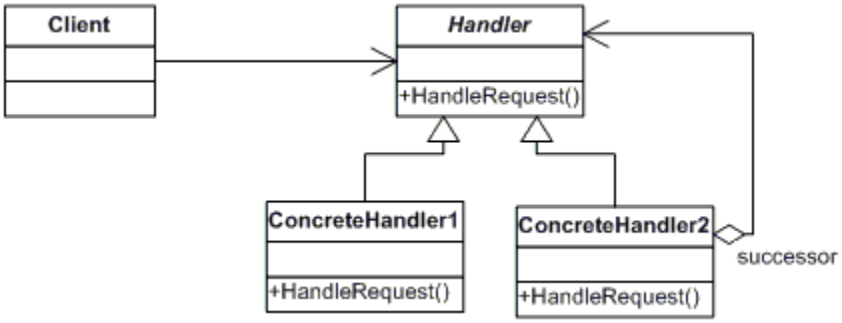
- Proxy (MathProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilites depend on the kind of proxy:
  - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
  - protection proxies check that the caller has the access permissions required to perform a request.
- Subject (IMath)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Math)
  - defines the real object that the proxy represents.

```
class MainApp
{
    static void Main()
    {
        var proxy = new Proxy();
        proxy.Request();
    }
}

abstract class Subject
{
    public abstract void Request();
}

class RealSubject : Subject
{
    public override void Request() { }
}

class Proxy : Subject
{
    private RealSubject _realSubject;

    public override void Request()
    {
        if (_realSubject == null)
```

```
            _realSubject = new RealSubject();

        _realSubject.Request();
    }
}
```

Паттерны поведения (Behavioral Patterns)

- Цепочка обязанностей (Chain of Responsibility)



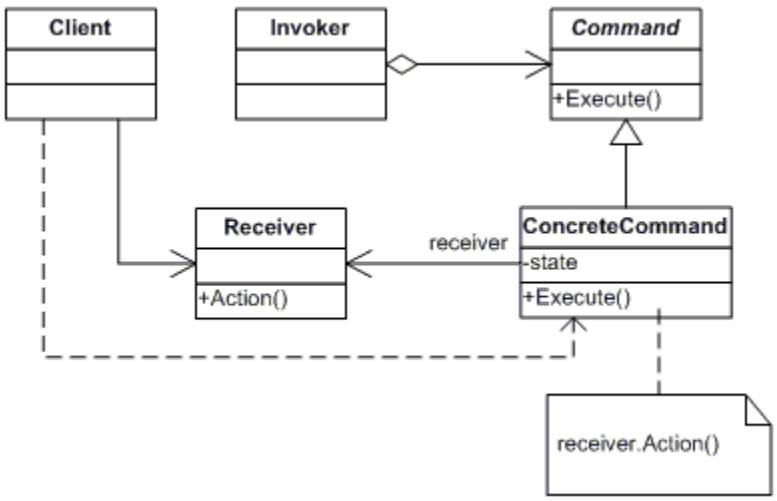
- Handler (Approver)
  - defines an interface for handling the requests
  - (optional) implements the successor link
- ConcreteHandler (Director, VicePresident, President)
  - handles requests it is responsible for
  - can access its successor
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- Client (ChainApp)
  - initiates the request to a ConcreteHandler object on the chain

```
class MainApp
{
    static void Main()
    {
        var h1 = new ConcreteHandler1();
        var h2 = new ConcreteHandler2();
        var h3 = new ConcreteHandler3();

        h1.SetSuccessor(h2);
        h2.SetSuccessor(h3);

        int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };
```

```
        foreach (int request in requests)
        {
            h1.HandleRequest(request);
        }
    }
}

abstract class Handler
{
    protected Handler successor;

    public void SetSuccessor(Handler successor)
    {
        this.successor = successor;
    }

    public abstract void HandleRequest(int request);
}

class ConcreteHandler1 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 0 && request < 10)
        {
            Console.WriteLine("{0} handled request {1}",
              this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}

class ConcreteHandler2 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 10 && request < 20)
        {
            Console.WriteLine("{0} handled request {1}",
              this.GetType().Name, request);
        }
```

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

class ConcreteHandler3 : Handler
{
    public override void HandleRequest(int request)
    {
        if (request >= 20 && request < 30)
        {
            Console.WriteLine("{0} handled request {1}",
              this.GetType().Name, request);
        }
        else if (successor != null)
        {
            successor.HandleRequest(request);
        }
    }
}
```

- [Команда (Command)]



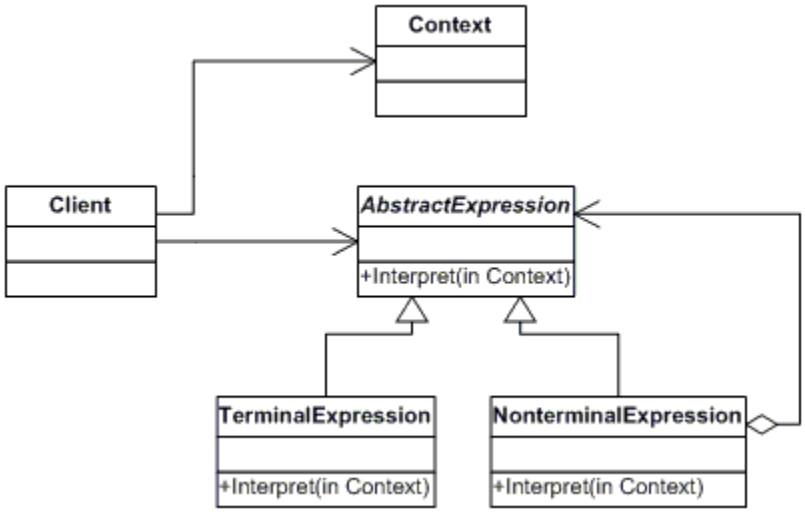
- Command (Command)
  - declares an interface for executing an operation
- ConcreteCommand (CalculatorCommand)
  - defines a binding between a Receiver object and an action

- implements Execute by invoking the corresponding operation(s) on Receiver
- Client (CommandApp)
- creates a ConcreteCommand object and sets its receiver
- Invoker (User)
- asks the command to carry out the request
- Receiver (Calculator)
- knows how to perform the operations associated with carrying out the request.

```
class MainApp
{
    static void Main()
    {
        var receiver = new Receiver();
        var command = new ConcreteCommand(receiver);
        var invoker = new Invoker();

        invoker.SetCommand(command);
        invoker.ExecuteCommand();
    }
}

abstract class Command
{
    protected Receiver receiver;

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

    public abstract void Execute();
}

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

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

```
class Receiver
{
    public void Action() { }
}

class Invoker
{
    private Command _command;

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

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

- Интерпретатор (Interpreter)

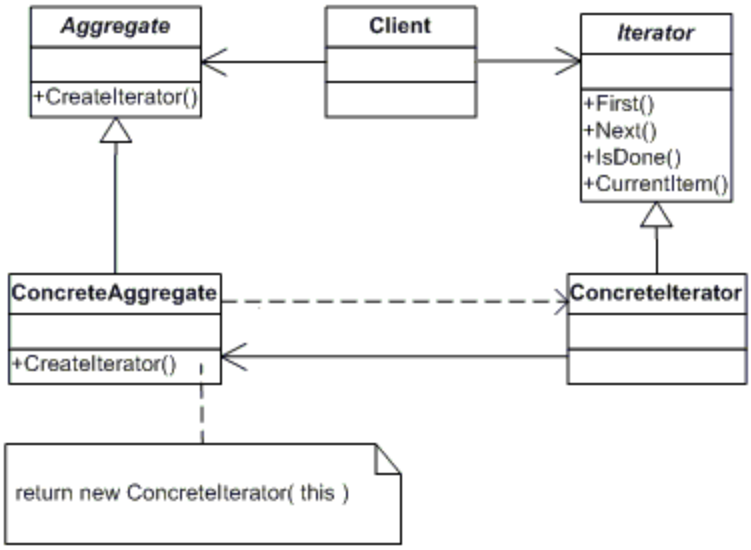


- AbstractExpression (Expression)
  - declares an interface for executing an operation
- TerminalExpression
  - implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in the sentence.
- Context (Context)
  - contains information that is global to the interpreter

- Client (InterpreterApp)
  - 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
  - invokes the Interpret operation

```
class MainApp
{
    static void Main()
    {
        var context = new Context();
        var exp = new TerminalExpression();

        exp.Interpret(context);
    }
}

class Context { }

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

class TerminalExpression : AbstractExpression
{
    public override void Interpret(Context context) { }
}
```

- Итератор (Iterator) - последовательный доступ к элементам объекта



- Iterator
  - интерфейс доступа к элементам
- ConcreteIterator
  - реализует интерфейс итератора
- Aggregate
  - интерфейс получения итератора из объекта
- ConcreteAggregate
  - реализует интерфейс агрегатора

```
class MainApp
{
    static void Main()
    {
        var a = new ConcreteAggregate(); // List<T>
        a[0] = "Item A";
        a[1] = "Item B";
        a[2] = "Item C";
        a[3] = "Item D";

        var i = a.CreateIterator(); // List<T>.Enumerator

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

        object item = i.First();

        while (item != null)
        {
            Console.WriteLine(item);
            item = i.Next();
```

```
        }
    }
}

abstract class Aggregate // IList<T>
{
    public abstract Iterator CreateIterator();
}

class ConcreteAggregate : Aggregate // List<T>
{
    private ArrayList _items = new ArrayList();

    public override Iterator CreateIterator() // GetEnumerator
    {
        return new ConcreteIterator(this); // List<T>.Enumerator
    }

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

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

abstract class Iterator
{
    public abstract object First();
    public abstract object Next();
    public abstract bool IsDone();
    public abstract object CurrentItem();
}

class ConcreteIterator : Iterator
{
    private ConcreteAggregate _aggregate;
    private int _current = 0;

    public ConcreteIterator(ConcreteAggregate aggregate)
    {
        this._aggregate = aggregate;
```

```
    }

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

    public override object Next() // MoveNext
    {
        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;
    }
}
```

Реализация паттерна для List<T>

```
public class List<T> : IList<T>, System.Collections.IList, IReadOnlyList<T>
{
    private T[] _items;
    private int _size;
    private int _version; // индикатор изменения коллекции
    private Object _syncRoot;
    static readonly T[] _emptyArray = new T[0];

    public List()
    {
        _items = _emptyArray;
    }

    public void Add(T item)
```

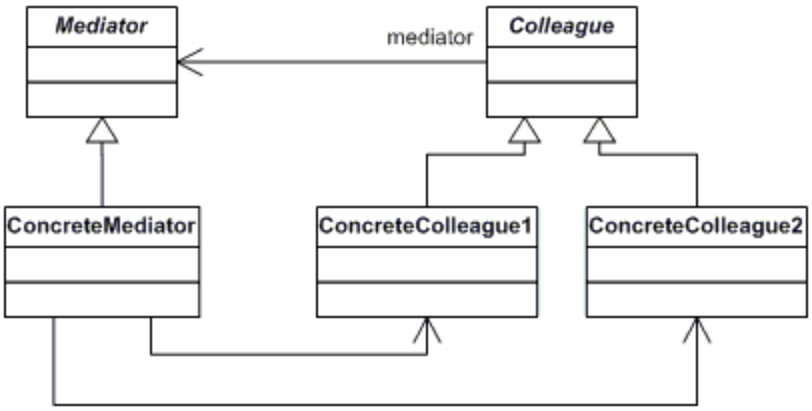
```
{
    if (_size == _items.Length) EnsureCapacity(_size + 1);
    _items[_size++] = item;
    _version++; // обновление версии
}

public struct Enumerator : IEnumerator<T>, System.Collections.IEnumerator
{
    private List<T> list;
    private int index;
    private int version;
    private T current;

    internal Enumerator(List<T> list)
    {
        this.list = list;
        index = 0;
        version = list._version;
        current = default(T);
    }

    public void Dispose() { }
```

MoveNext генерирует exception если коллекция была изменена для того чтобы предотвратить неверный вывод элементов коллекции (пример: A, B, C, D - вставка после B тогда когда index на C - тогда C будет выведена дважды)
P.S. Генерация исключения реализована не во всех языках.

```
    public bool MoveNext()
    {
        List<T> localList = list; // новая переменная но та же ссылка

        if (version == localList._version && ((uint)index < (uint)localList._size))
        {
            current = localList._items[index];
            index++;
            return true;
        }

        return MoveNextRare();
    }

    private bool MoveNextRare()
    {
```

```
            if (version != list._version)
                throw new InvalidOperationException("Collection was modified");

            index = list._size + 1;
            current = default(T);
            return false;
        }

        public T Current {
            get {
                return current;
            }
        }

        Object System.Collections.IEnumerator.Current { // Для ArrayList
            get {
                if (index == 0 || index == list._size + 1)
                    throw new InvalidOperationException("Run MoveNext firstly");

                return Current;
            }
        }

        void System.Collections.IEnumerator.Reset()
        {
            if (version != list._version)
                throw new InvalidOperationException("Collection was modified");

            index = 0;
            current = default(T);
        }
    }
}
```

- Посредник (Mediator) - управление взаимодействием множества объектов, явный посредник - объекты содержат ссылку на посредника, неявный посредник - объекты не содержат ссылку на посредника, в обоих вариантах посредник знает об объектах.



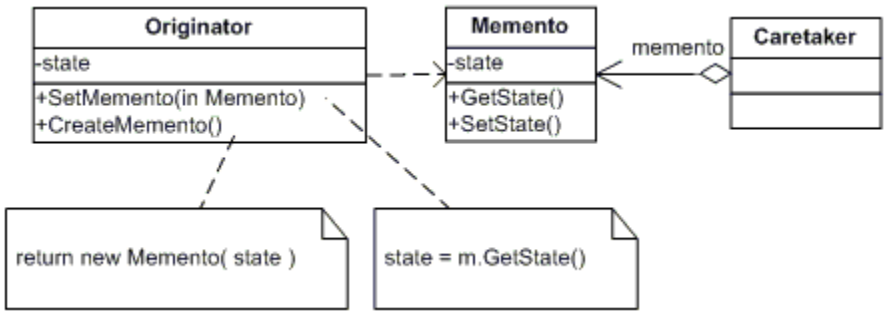
- Mediator
  - интерфейс, который содержит методы взаимодействия объектов
- ConcreteMediator
  - содержит ссылки на объекты
  - реализует интерфейс (метод взаимодействия объектов)
- Colleague classes
  - каждый объект содержит ссылку на посредника
  - взаимодействует с другими объектами через посредника

```
class MainApp
{
    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");
    }
}

abstract class Mediator
{
    public abstract void Send(string message, Colleague colleague); // msg, from
}

class ConcreteMediator : Mediator
```

```
{
   private ConcreteColleague1 _colleague1;
   private ConcreteColleague2 _colleague2;

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

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

   public override void Send(string message, Colleague colleague)
   {
      if (colleague == _colleague1)
         _colleague2.Notify(message);
      else
         _colleague1.Notify(message);
   }
}

abstract class Colleague
{
   protected Mediator mediator;

   public Colleague(Mediator mediator)
   {
      this.mediator = mediator;
   }
}

class ConcreteColleague1 : Colleague
{
   public ConcreteColleague1(Mediator mediator)
     : base(mediator) { }

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

   public void Notify(string message) { }
}
```

```
class ConcreteColleague2 : Colleague
{
    public ConcreteColleague2 (Mediator mediator)
      : base(mediator) { }

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

    public void Notify(string message) { }
}
```

- Хранитель (Memento)



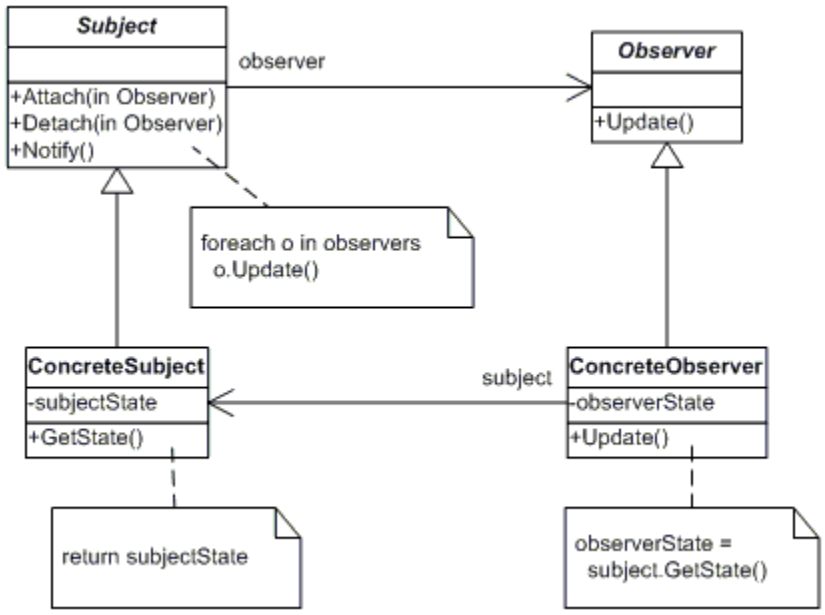
- Memento (Memento)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- Originator (SalesProspect)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- Caretaker (Caretaker)
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

```
class MainApp
{
    static void Main()
    {
        var o = new Originator();
```

```
        o.State = "On";

        var c = new Caretaker();
        c.Memento = o.CreateMemento();
        o.State = "Off";

        o.SetMemento(c.Memento);
    }
}

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.State;
    }
}

class Memento
{
    private string _state;

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

    public string State {
        get { return _state; }
    }
}
```

```
class Caretaker
{
    private Memento _memento;

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

- [Наблюдатель (Observer)]



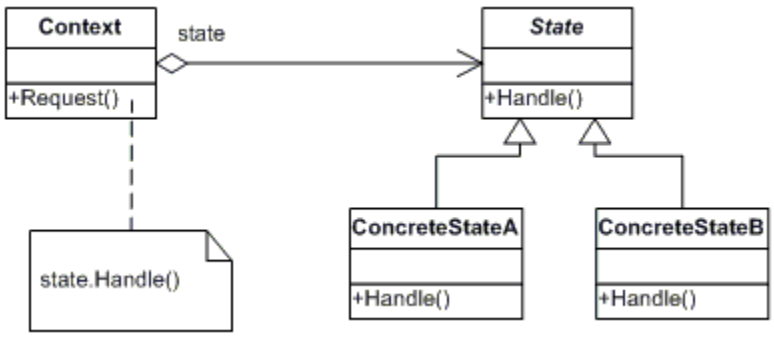
- Subject (Stock)
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects.
- ConcreteSubject (IBM)
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes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

```
class MainApp
{
```

```
    static void Main()
    {
        var s = new ConcreteSubject();

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

        s.SubjectState = "ABC";
        s.Notify();
    }
}

abstract class Subject
{
    private List<Observer> _observers = new List<Observer>();

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

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

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

class ConcreteSubject : Subject
{
    private string _subjectState;

    public string SubjectState {
        get { return _subjectState; }
        set { _subjectState = value; }
    }
```

```
}

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

class ConcreteObserver : Observer
{
    private string _name;
    private string _observerState;
    private ConcreteSubject _subject;

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

    public override void Update()
    {
        _observerState = _subject.SubjectState;
    }

    public ConcreteSubject Subject {
        get { return _subject; }
        set { _subject = value; }
    }
}
```

- Состояние (State)



-
- Context (Account)

- defines the interface of interest to clients
- maintains an instance of a ConcreteState subclass that defines the current state.

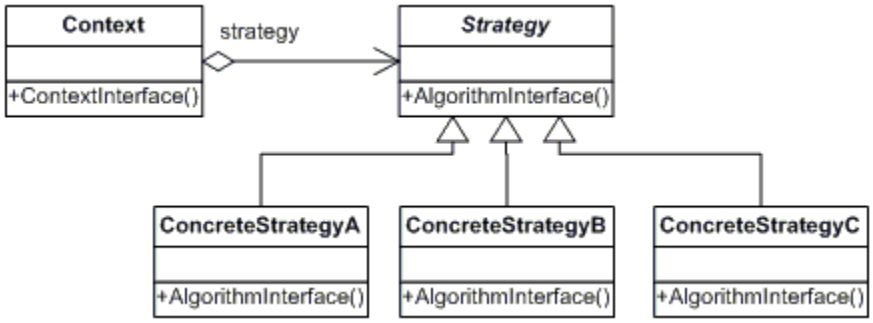
- State (State)
- defines an interface for encapsulating the behavior associated with a particular state of the Context.

- Concrete State (RedState, SilverState, GoldState)
- each subclass implements a behavior associated with a state of Context

```
class MainApp
  {
    static void Main()
    {
      var c = new Context(new ConcreteStateA());

      c.Request();
      c.Request();
    }
  }

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

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

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

  class Context
  {
    private State _state;

    public Context(State state)
```

```
    {
        this.State = state;
    }

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

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

- Стратегия (Strategy) - аналог фабричного метода (в отличие от фабричного метода поведение не создания, а выполнения действия на основе аргумента конструктора)

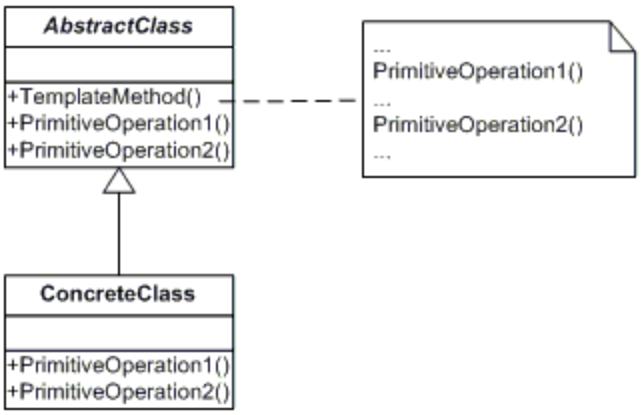


- Strategy (SortStrategy)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy (QuickSort, ShellSort, MergeSort)
  - implements the algorithm using the Strategy interface
- Context (SortedList)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

```
class MainApp
{
    static void Main()
    {
        Context context;
```

```
        context = new Context(new ConcreteStrategyA());
        context.ContextInterface();
    }
}

abstract class Strategy
{
    public abstract void AlgorithmInterface();
}

class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterface() { }
}

class Context
{
    private Strategy _strategy;

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

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

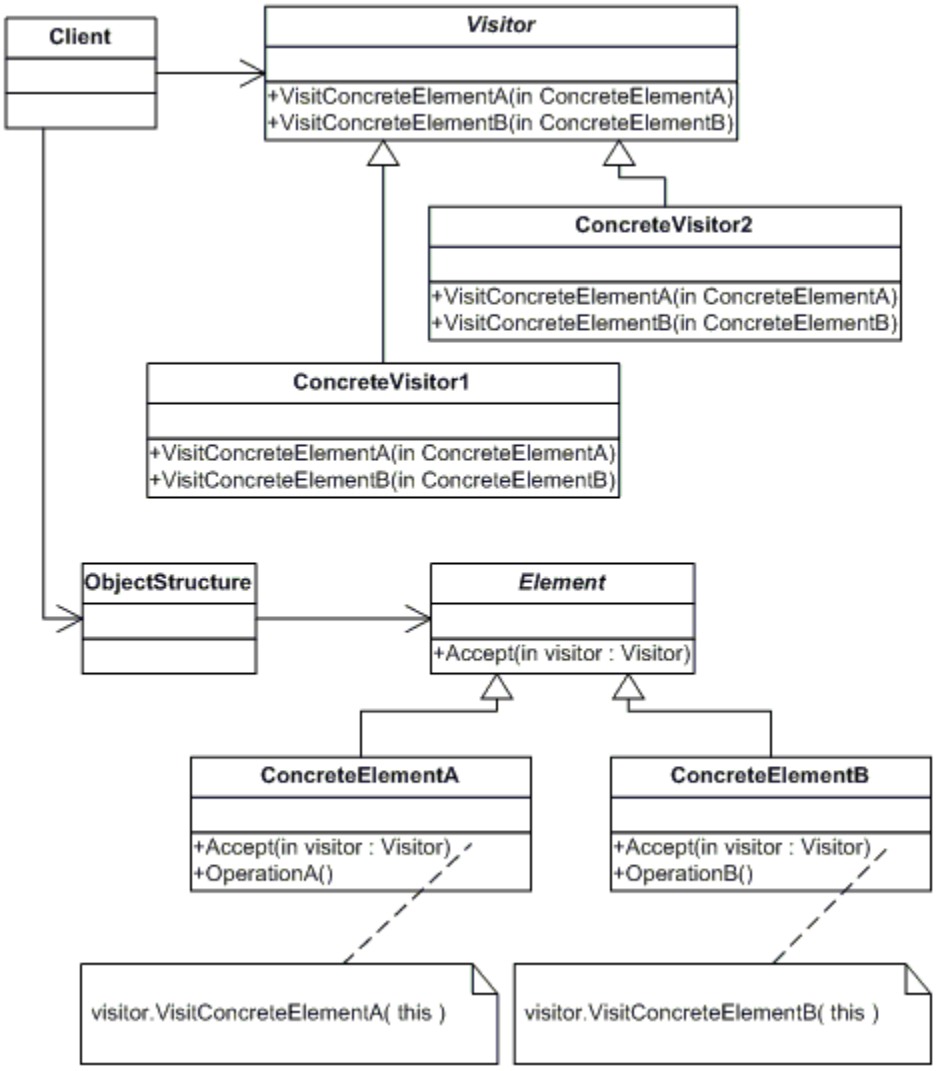
- Шаблонный метод (Template Method) - в отличие от строителя метод содержащий этапы определен не у директора, а у класса (строителя) который и реализует этапы, но порядок и название этапов по прежнему одинаковое (может быть задано в абстрактном классе - строителе)



- AbstractClass
  - реализует шаблонный метод (порядок этапов)
  - определяет этапы
- ConcreteClass
  - реализует этапы

```
class MainApp
{
    static void Main()
    {
        var cA = new ConcreteClassA();
        cA.TemplateMethod();
    }
}

abstract class AbstractClass
{
    protected abstract void PrimitiveOperation1();
    protected abstract void PrimitiveOperation2();

    public void TemplateMethod() // последовательность шагов наследуется
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
    }
}

class ConcreteClassA : AbstractClass
{
    protected override void PrimitiveOperation1() { } // шаг переопределяется
    protected override void PrimitiveOperation2() { } // шаг переопределяется
}
```

- Посетитель (Visitor)



- The visitor (Visitor)
  - 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 (IncomeVisitor, VacationVisitor)
  - implements each operation declared by 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.
- Element (Element)
  - defines an Accept operation that takes a visitor as an argument.
- ConcreteElement (Employee)
  - implements an Accept operation that takes a visitor as an argument
- ObjectStructure (Employees)

    - can enumerate its elements
- may provide a high-level interface to allow the visitor to visit its elements
- may either be a Composite (pattern) or a collection such as a list or a set

```
class MainApp
  {
    static void Main()
    {
      var o = new ObjectStructure();
      o.Attach(new ConcreteElementA());

      var v1 = new ConcreteVisitor1();
      o.Accept(v1);
    }
  }

  abstract class Visitor
  {
    public abstract void VisitConcreteElementA(ConcreteElementA concreteElementA);
  }

  class ConcreteVisitor1 : Visitor
  {
    public override void VisitConcreteElementA(ConcreteElementA concreteElementA) { }
  }

  abstract class Element
  {
    public abstract void Accept(Visitor visitor);
  }

  class ConcreteElementA : Element
  {
    public override void Accept(Visitor visitor)
    {
      visitor.VisitConcreteElementA(this);
    }
  }

  class ObjectStructure
  {
    private List<Element> _elements = new List<Element>();

    public void Attach(Element element)
```

```
    {
        _elements.Add(element);
    }

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

    public void Accept(Visitor visitor)
    {
        foreach (Element element in _elements)
        {
            element.Accept(visitor);
        }
    }
}
```

Паттерн Dispose

The IDisposable interface is a mechanism to release unmanaged resources, if not implemented correctly this could result in resource leaks or more severe bugs.

This rule raises an issue when the recommended dispose pattern, as defined by Microsoft, is not adhered to. See the Compliant Solution section for examples.

Satisfying the rule's conditions will enable potential derived classes to correctly dispose the members of your class:

- sealed classes are not checked.
- If a base class implements IDisposable your class should not have IDisposable in the list of its interfaces. In such cases it is recommended to override the base class's protected virtual void Dispose(bool) method or its equivalent.
- The class should not implement IDisposable explicitly, e.g. the Dispose() method should be public.
- The class should contain protected virtual void Dispose(bool) method. This method allows the derived classes to correctly dispose the resources of this class.
- The content of the Dispose() method should be invocation of Dispose(true) followed by GC.SuppressFinalize(this)
- If the class has a finalizer, i.e. a destructor, the only code in its body should be a single invocation of Dispose(false).
- If the class inherits from a class that implements IDisposable it must call the Dispose, or Dispose(bool) method of the base class from within its own implementation of Dispose or Dispose(bool), respectively. This ensures that all resources from the base class are properly released.

Compliant Solution

```
// Sealed class
public sealed class Foo1 : IDisposable
{
```

```
    public void Dispose()
    {
        // Cleanup
    }
}

// Simple implementation
public class Foo2 : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Cleanup
    }
}

// Implementation with a finalizer
public class Foo3 : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Cleanup
    }

    ~Foo3()
    {
        Dispose(false);
    }
}

// Base disposable class
public class Foo4 : DisposableBase
{
```

```
    protected override void Dispose(bool disposing)
    {
        // Cleanup
        // Do not forget to call base
        base.Dispose(disposing);
    }
}
```

Implementation

P.S. Если неуправляемые ресурсы не используются, то деструктор не переопределяется

```
private bool disposed;

/// <inheritdoc/>
public void Dispose()
{
	Dispose(true);
	GC.SuppressFinalize(this);
}

/// <summary>
/// Освобождение ресурсов.
/// </summary>
/// <param name="releaseManagedResources">Освобождение управляемых ресурсов.</param>
protected override void Dispose(bool releaseManagedResources)
{
	if (disposed)
	{
		return;
	}

	try
	{
		if (releaseManagedResources)
		{
			OrganizationContext.Set(Option<Guid>.Empty);
		}

		disposed = true;
	}
	finally
	{
		base.Dispose(releaseManagedResources);
```

```
    }
}
```

Принципы проектирования

Doc

https://makedev.org/principles/index.html

- Don't Repeat Yourself (DRY, рус. Не повторяйся)
- KISS (keep it short and simple)
- S.O.L.I.D.
    - 1. SPR (Single responsibility principle) - Принцип единственной обязанности
    - 2. OCP (Open/closed principle) - Принцип открытости/закрытости
    - 3. LSP (Liskov substitution principle) - Принцип подстановки Барбары Лисков
    - 4. ISP (Interface segregation principle) - Принцип разделения интерфейса
    - 5. DIP (Dependency inversion principle) - Принцип инверсии зависимостей
- YAGNI (англ. You Ain't Gonna Need It — «Вам это не понадобится»)
- Общие принципы проектирования