Объектно-ориентированные технологии программирования и стандарты проектирования

Кирилл Сурков, Дмитрий Сурков, Юрий Четырько

Контакты:

kirill.surkov@gmail.com

http://vk.com/kirill.surkov

Литература

Данная презентация служит методикой изучения предмета и содержит основные понятия ООП и их представление на языке C#.

«C# Language Specification»:
C:\Program Files (x86)\Microsoft Visual Studio 12.0\ VC#\Specifications\1033\CSharp Language Specification.docx

Конспект лекций по языку программирования C++ для студентов, владеющих языком программирования C#, Java или Delphi. Авторы: К.А. Сурков, Д.А. Сурков, Ю.М. Четырько.

«The Java Language Specification»:
http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

Краткое изложение Java для специалистов, владеющих другими объектно-ориентированными языками программирования:
http://en.wikipedia.org/wiki/Java_syntax

«Программирование на языке Delphi». Учебное пособие. Авторы: А.Н. Вальвачев, К.А. Сурков, Д. А. Сурков, Ю.М. Четырько. Книга расположена по адресу http://rsdn.ru/?summary/3165.xml

«The Programming Language Oberon». Niklaus Wirth:
http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf

«Zonnon Language Report». Jurg Gutknecht
http://zonnon.ethz.ch/archive/znnLanguageReportv04y090606draft.pdf

«Язык ДРАКОН». Владимир Пароджанов:
http://drakon-practic.ru/drakon.pdf – краткое описание,
http://drakon.pbworks.com/w/page/18205516/FrontPage – полное описание.

Базовые понятия структурного программирования

Переменная – ячейка данных

Тип данных – допустимые значения и операции переменной

Указатель – переменная, содержащая адрес

Оператор – единица выполнения алгоритма

Процедура – вызываемая на выполнение подпрограмма

Процедурная переменная – указатель на процедуру

Модуль – единица разработки и доставки программы

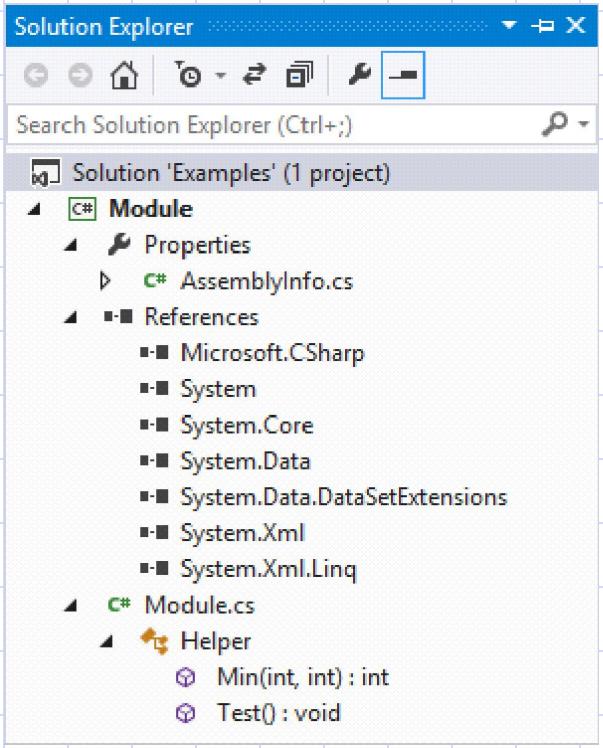
Объект – динамический модуль

Исключение – событие и объект с информацией об ошибке

Модуль

Модуль – единица разработки, применения и поставки.

```
public static class Helper // модуль
{
    public static int Min(int x, int y) // процедура модуля
    {
        int result;
        if (x < y)
            result = x;
        else
            result = y;
        return result;
    }

    public static void Test()
    {
        int n = Min(10, 20);
        Console.WriteLine("Min = {0}", n);
    }
}
```



Разграничение доступа к модулям

Разграничение доступа к модулю осуществляется с помощью ключевых слов:

- public – доступ к модулю получают все;
- protected – доступ к модулю получают данный модуль и модули расширения;
- internal – доступ к модулю получают данный модуль и программы, в которых данный модуль подключается на уровне исходного кода;
- protected internal – доступ к модулю получают модули расширения и программы, в которых данный модуль подключается на уровне исходного кода;
- private – доступ к модулю получают лишь процедуры этого модуля.

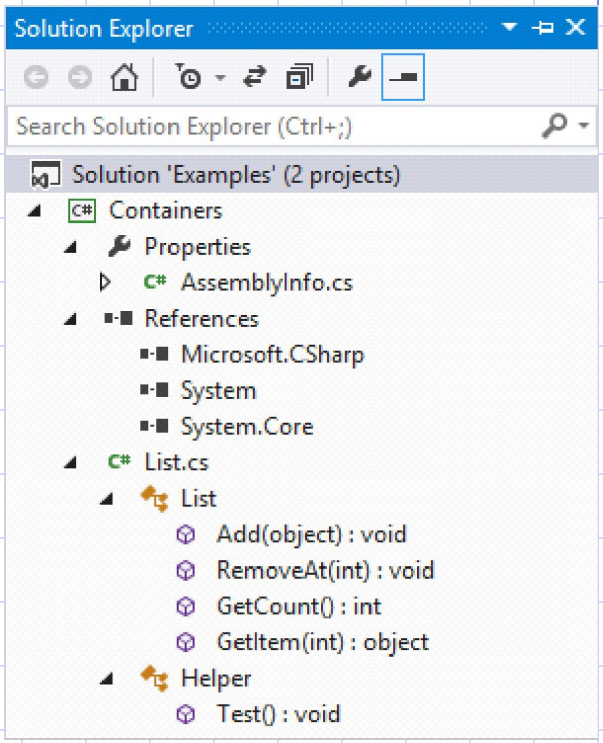
Примеры модулей и процедур с различными режимами доступа:

```
public static class Module1
{
    public static void Procedure1() { ... }
    internal static void Procedure2() { ... }
    private static void Procedure3() { ... }
}

internal static class Module2
{
    public static void Procedure4() { ... }
```

Объект – динамический модуль

Объект – динамический модуль, у которого может быть много экземпляров. Описание динамического модуля – класс.

```
public class List // динамический модуль
{
    public void Add(object item) { ... }
    public void RemoveAt(int index) { ... }
    public int GetCount() { ... }
    public object GetItem(int index) { ... }
}

public static class Helper // статический модуль
{
    public static void Test()
    {
        var list = new List(); // создание модуля
        list.Add("Андрей Ершов");
        list.Add("Niklaus Wirth");
        Console.WriteLine("Count = {0}", list.GetCount());
    }
}
```



Исключение

Исключение – событие и объект с информацией об ошибке.

Исключение создается:

- Аппаратно – в результате ошибки выполнения оператора.
- Программно – с помощью оператора создания исключения:

```
throw new OutOfMemoryException("Не хватает памяти");
```

Некоторые стандартные исключения модуля System:

```
Exception
ArithmeticException
DivideByZeroException
OverflowException
ArrayTypeMismatchException
IndexOutOfRangeException
InvalidCastException
```

```
NullReferenceException
OutOfMemoryException
StackOverflowException
TypeInitializationException
NotSupportedException
NotImplementedException
```

Обработка исключений

Обработка исключения:

```
try
{
    // операторы, подверженные исключениям
}
catch
{
    // операторы обработки исключения
}
```

Частичная обработка исключения:

```
try
{
    // операторы, подверженные исключениям
}
catch
{
    // операторы обработки исключения
    throw; // передача обработки внешнему блоку try...catch
}
```

Классификация исключений

Распознавание класса исключения:

```
try
{
    // арифметические вычисления
}
catch (DivideByZeroException e)
{
    // обработка деления на ноль
}
catch (OverflowException e)
{
    // обработка переполнения
}
catch (ArithmeticException e)
{
    // обработка другой арифметической ошибки
}
catch (Exception e)
{
    // обработка любого иного исключения
}
```

Защита ресурсов от исключений

Защита ресурсов от исключения:

```
// запрос ресурса
try
{
    // работа с ресурсом
}
finally
{
    // освобождение ресурса
}
```

Пример:

```
var stream = new FileStream("MyFile.dat", FileMode.Open, FileAccess.Read);
try
{
    stream.Read(buffer, 0, buffer.Length);
}
finally
{
    stream.Dispose(); // эквивалентно stream.Close();
}
```

Защита ресурсов от исключений

С помощью оператора using:

```
using (var stream = new FileStream(...))
{
    stream.Read(buffer, 0, buffer.Length);
}
```

Комплексный вариант оператора try:

```
// запрос ресурса
try
{
    // работа с ресурсом
}
catch (Exception e)
{
    // обработка исключения
}
finally
{
    // освобождение ресурса
}
```

Базовые понятия объектно-ориентированного программирования

Класс – тип данных для создания объектов

Объект – экземпляр класса

Метод – процедура над объектом

Конструктор и деструктор – особые методы

Свойство – виртуальное поле

Наследование – расширение класса

Виртуальный метод – переопределяемый метод

Делегат – ссылка на метод

Событие – список делегатов

Интерфейс – описание класса без реализации

Шаблон – параметризованный класс

Атрибут – метаданные, механизм рефлексии

Класс и объект

Класс – тип данных для создания объектов:

```
public class DelimitedReader
{
    public void Open(string fileName, char delimiter) { ... }
    public bool NextLine() { ... }
    public bool IsEndOfFile() { ... }

    public string[] Items;

    private string FileName;
    private char Delimiter;
}
```

Объект – экземпляр класса:

```
var reader = new DelimitedReader();
reader.Open("MyFile.csv", ';');
while (reader.NextLine())
{
    Console.WriteLine("{0}", reader.Items[0]); // выводим первый столбец
}
```

Метод

Метод:

```
public class DelimitedReader
{
    public void AssignFields()
    {
        FileName = "MyFile.csv";
        this.Delimiter = ';';
    }

    private string FileName;
    private char Delimiter;
}
```

Процедура, эквивалентная методу:

```
public static void AssignFields(this DelimitedReader reader)
{
    reader.FileName = "MyFile.csv";
    reader.Delimiter = ';';
}
```

Конструктор и деструктор

Конструктор и деструктор – особые методы:

```
public class DelimitedReader
{
    public DelimitedReader(string fileName, char delimiter)
    {
        FileName = filename;
        Delimiter = delimiter;
    }

    ~DelimitedReader()
    {
        ...
    }
    ...
}
```

Вызов конструктора:

```
var reader = new DelimitedReader("MyFile.csv", ';');
```

Свойство

Свойство – виртуальное поле:

```
public class DelimitedReader
{
    public bool Active { get { return fActive; } set { SetActive(value); } }

    private bool fActive;
    private void SetActive(bool value) { ... }
    ...
}
```

Работа со свойством внешне не отличается от работы с полем:

```
reader.Active = true;

...

if (reader.Active)
{
    ...
}
```

Назначение свойств – создание побочных эффектов при обращении, например, открытие файла при установке свойства Active в true.

Свойство

Свойство с автоматически создаваемым полем для него:

```
public class DelimitedReader
{
    public char Delimiter { get; set; }
    ...
}
```

Свойство с разными режимами доступа вне и внутри модуля:

```
public class DelimitedReader
{
    public char Delimiter { get; private set; }
    ...
}
```

Свойство с ограничением режима доступа – только чтение:

```
public class DelimitedReader
{
    public char Delimiter { get; }
    ...
}
```

Индексатор

Индексатор – особое свойство, предоставляющее доступ к объекту как к массиву:

```
public class DelimitedReader
{
    public string this[int index]
    {
        get { return fItems[index]; }
        set { fItems[index] = value; OnChange(); }
    }

    private string[] fItems;
    private void OnChange() { ... }
    ...
}
```

Применение индексатора:

```
var reader = new DelimitedReader();
...
reader[0] = "Andrey";  // reader.Items[0]
reader[1] = "Yershov"; // reader.Items[1]
...
```

Наследование – расширение класса

Базовая абстракция объекта для чтения текста в табличном виде:

```
public class TableReader
{
    public string FileName { get; private set; }
    public string[] Items { get { return fItems; } }
    public int ItemCount { get { return fItems.Length; } }

    public TableReader(string fileName) { ... }
    public void Close() { ... }
    public bool NextLine() { ... }
    public bool IsEndOfFile() { ... }

    private StreamReader fStreamReader;
    private string[] fItems;
}
```

* CSV – Comma-Separated Values

Расширение базового класса для чтения таблиц формата CSV *:

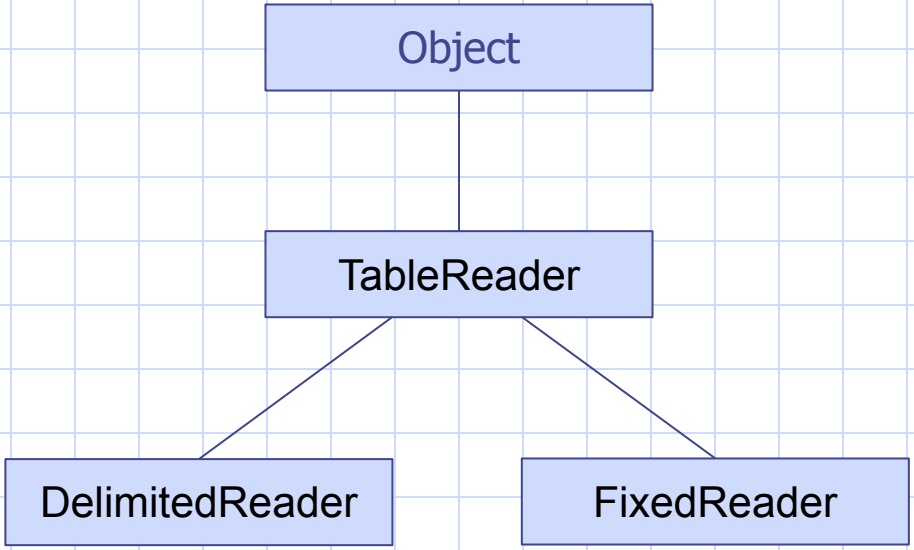
```
public class DelimitedReader : TableReader
{
    public char Delimiter { get; private set; }

    public DelimitedReader(string fileName, char delimiter) { ... }
```

Наследование – расширение класса

Расширение базового класса для чтения таблиц формата TXT:

```
public class FixedReader : TableReader
{
    public int[] ItemWidths { get; private set; }

    public FixedReader(string fileName, params int[] itemWidths) { ... }
    public bool NextLine() { ... }
}
```

Дерево классов:



Наследование – расширение класса

Смысл расширения типа заключается в том, чтобы определить переменную базового типа и присваивать ей объекты любых производных типов.

```
TableReader reader;
reader = new DelimitedReader("MyFile.csv", ';');
reader = new FixedReader("MyFile.csv", 20, 20, 60);
```

Через переменную базового типа можно безопасно обращаться к полям и методам, определенным в базовом типе. Это обеспечивается благодаря бинарной совместимости всех производных типов с базовым типом.

```
reader.NextLine();
```

Наследование – контроль и приведение типа

Контроль типа осуществляется с помощью оператора **is**:

```
TableReader reader;
...
if (reader is DelimitedReader) // или производного класса (!)
{
    ...
}
```

Приведение переменной к ожидаемому типу данных:

```
char c = ((DelimitedReader)reader).Delimiter;
```

Приведение типа с помощью оператора **as**:

```
char c = (reader as DelimitedReader).Delimiter;
```

Оператор **as** отличается от традиционного приведения типа тем, что не создает исключение, если тип не соответствует ожидаемому.

```
DelimitedReader delimitedReader = reader as DelimitedReader;
if (delimitedReader != null)
{
    char c = delimitedReader.Delimiter;
    ...
}
```

Базовый класс Object

Базовый класс System.Object содержит общие для всех объектов методы:

```
namespace System
{
    public class Object
    {
        public Object();
        public virtual bool Equals(object obj);
        public static bool Equals(object objA, object objB);
        public virtual int GetHashCode();
        public Type GetType();
        protected object MemberwiseClone();
        public static bool ReferenceEquals(object objA, object objB);
        public virtual string ToString();
    }
}
```

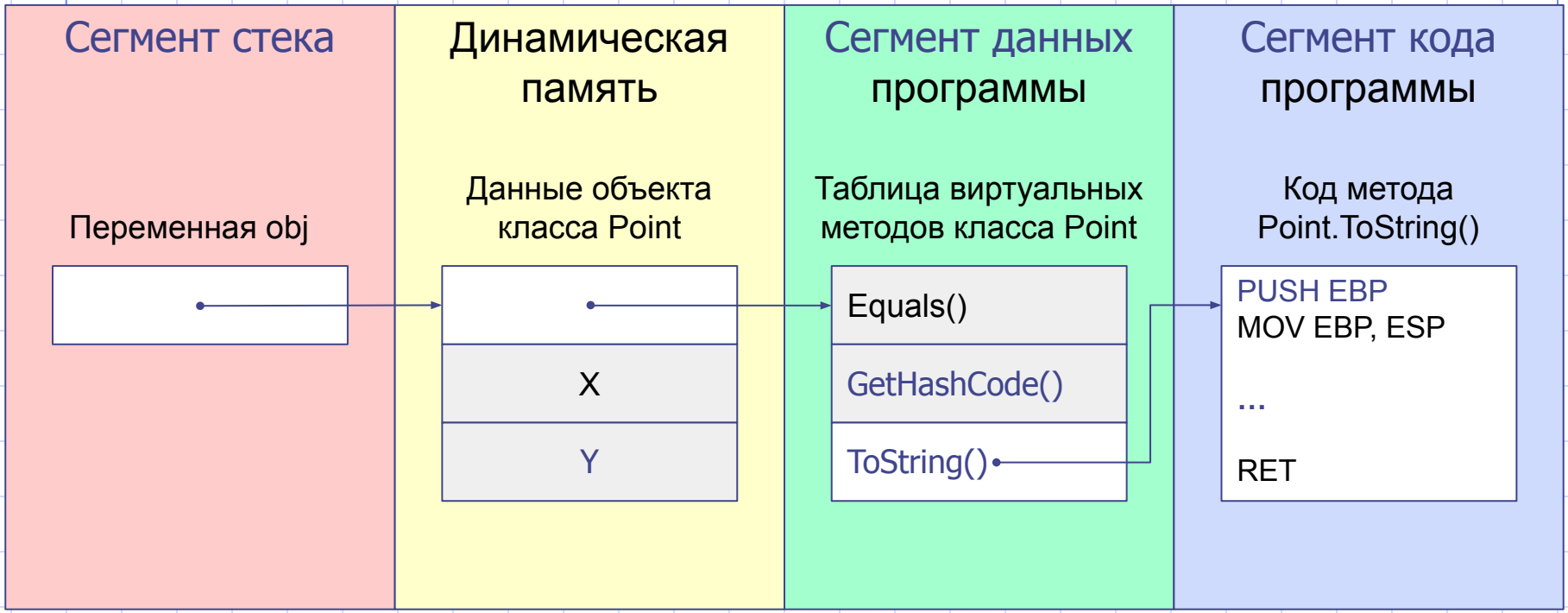
Виртуальный метод

Виртуальный метод – переопределяемый в производных классах метод. Вызов виртуального метода осуществляется в соответствии с фактическим типом объекта, к которому метод применяется:

```
public class Object
{
    ...
    public virtual string ToString();
}

public class Point : Object
{
    public int X { get; protected set; }
    public int Y { get; protected set; }
    public Point(int x, int y) { X = x; Y = y; }
    ...
    public override string ToString()
    {
        return String.Format("X: {0}, Y: {1}", X, Y);
    }
}
...
object obj = new Point(10, 20);
string str = obj.ToString(); // str = "X: 10, Y: 20"
```

Механизм вызова виртуального метода

Схема вызова **obj.ToString();** // obj ссылается на объект класса Point



1. Через объектную переменную выполняется обращение к данным объекта, находящимся в области динамической памяти: MOV EAX, obj
2. Из данных объекта извлекается адрес таблицы виртуальных методов (первые 4 или 8 байт данных, в зависимости от разрядности CPU): MOV EBX, [EAX+0]
3. На основании порядкового номера виртуального метода из таблицы извлекается адрес метода: MOV EAX, [EBX+8] ; значение 8 здесь – смещение до поля ToString().
4. Выполняется вызов кода по извлеченному адресу: CALL EAX

Абстрактный виртуальный метод

Абстрактный метод – это виртуальный метод без реализации. Для такого метода в таблице виртуальных методов резервируется поле адреса, и оно устанавливается в null. Экземпляры классов, в которых существует хотя бы один не перекрытый абстрактный метод, создавать нельзя.

```
public class TableReader
{
    ...
    public abstract bool NextLine();
}
```

В производных классах абстрактный метод перекрывается обычным образом, как и любой виртуальный метод:

```
public class DelimitedReader : TableReader
{
    ...
    public override bool NextLine() { ... }
}
```

Чтобы можно было создать экземпляр класса, все его абстрактные методы должны быть перекрыты.

Динамический виртуальный метод

В некоторых языках программирования разновидностью виртуального метода является динамический метод:

```
public class TableReader
{
    ...
    public dynamic bool NextLine(); // не поддерживается в C# (!)
}
```

В производных классах динамический метод перекрывается обычным образом – с помощью зарезервированного слова **override**:

```
public class DelimitedReader : TableReader
{
    ...
    public override bool NextLine() { ... }
}
```

Различие между динамическим и виртуальным методами состоит лишь в механизме вызова. Виртуальный метод вызывается максимально быстро, но ценой избыточных затрат памяти на таблицы виртуальных методов. Динамический метод вызывается дольше, но таблицы динамических методов имеют значительно более компактный вид, что способствует экономии памяти.

Виртуальное свойство

Свойство можно сделать виртуальным. Тогда в производных классах перекрываются его методы чтения и записи:

```
public class Point
{
    public virtual int X { get; protected set; }
    public virtual int Y { get; protected set; }
    ...
}

public class Rectangle : Point
{
    public override int X { get { ... } protected set { ... } }
    public override int Y { get { ... } protected set { ... } }
    ...
}
```

При перекрытии свойства нельзя изменять атрибуты доступа методов чтения и записи.

Запрет на расширение

Наследование от класса можно запретить с помощью ключевого слова **sealed** – «запечатанный». «Запечатывание» позволяет компилятору в некоторых случаях построить более производительный программный код. Например, вызовы всех виртуальных методов «запечатанного» класса можно выполнять так, будто они не виртуальные. Это дает некоторый выигрыш в производительности. Пример «запечатанного класса»:

```
public sealed class DelimitedReader : TableReader
{
    ...
}
```

«Запечатать» можно отдельный виртуальный метод класса:

```
public class TableReader
{
    public virtual bool NextLine() { ... }
}

public class DelimitedReader : TableReader
{
    public sealed override bool NextLine() { ... }
}
```

Делегат – ссылка на метод

Можно создать процедурную переменную, содержащую адрес метода. Предварительно для такой переменной определяется тип данных, называемый делегатом:

```
public delegate bool NextLineDelegate();
```

Теперь можно объявить переменную с таким типом и присвоить ей ссылку на метод какого-то объекта (сигнатура метода должна совпадать с сигнатурой делегата):

```
NextLineDelegate NextLineDelegate = Form1.NextLine;
```

Наконец, можно сделать вызов через процедурную переменную:

```
NextLineDelegate();
```

В результате этого оператора у объекта Form1 будет вызван метод bool NextLineNotification(). Это произойдет благодаря тому, что в переменной-делегате (NextLineDelegate) хранится пара указателей: указатель на объект и указатель на код метода. Когда делегату присваивается значение (NextLineDelegate = Form1.NextLine), в нем устанавливаются сразу оба указателя.

Пример применения делегата

```
public delegate void NextLineDelegate();

public class DelimitedReader
{
    private NextLineDelegate fNextLineDelegate;

    public DelimitedReader(string fileName, NextLineDelegate nextLineDelegate)
    {   ...; fNextLineDelegate = nextLineDelegate; }

    public bool NextLine()
    {   ...
        if (fNextLineDelegate != null)
            fNextLineDelegate();
    }
}

public class Form1
{
    public void ReadTable()
    {
        var reader = new DelimitedReader("MyFile.csv", this.NextLine);
        ...
    }
```

Событие – список делегатов

Бывает необходим список переменных-делегатов для уведомления целого множества объектов. Он называется событием. Предварительно для переменной-события определяют тип-делегат:

```
public delegate bool NextLineDelegate();
```

Затем определяют переменную-событие:

```
public event NextLineDelegate NextLineEvent;
```

Добавление и исключение методов в списке события осуществляется с помощью операторов += и –=:

```
NextLineEvent += Form1.NextLine;
```

Наконец, можно сделать вызов через процедурную переменную:

```
NextLineEvent();
```

Вызываются все методы, зарегистрированные в списке события.

Пример применения события

```
public delegate void NextLineDelegate();

public class DelimitedReader
{
    public event NextLineDelegate NextLineEvent;

    public DelimitedReader(string fileName) { ... }

    public bool NextLine()
    {   ...;
        NextLineDelegate nextLineEvent = NextLineEvent;
        if (nextLineEvent != null)
            nextLineEvent();
    }
}

public class Form1
{
    public void ReadTable()
    {
        var reader = new DelimitedReader("MyFile.csv");
        reader.NextLineEvent += this.NextLine;
        ...
```

Принятый формат для событий

Для событий существует удобный стандартный тип-делегат, который решено применять для всех событий:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Первый параметр предусмотрен для того, чтобы вызываемая процедура могла обрабатывать события нескольких объектов и могла их различить при работе. Второй параметр предусмотрен для передачи любых дополнительных данных в процедуру обработки события. Таким образом, все обработчики события оказываются совместимы друг с другом.

События принято определять на основе типа EventHandler:

```
public event EventHandler NextLineEvent;
```

Или на основе параметризованного типа EventHandler<T>:

```
public event EventHandler<NextLineEventArgs> NextLineEvent;

public class NextLineEventArgs : EventArgs { ... }
```

Обновленный пример применения события

```
public class DelimitedReader
{
    public event EventHandler NextLineEvent;

    public DelimitedReader(string fileName) { ... }

    public bool NextLine()
    {   ...;
        NextLineDelegate nextLineEvent = NextLineEvent;
        if (nextLineEvent != null)
            nextLineEvent(this, EventArgs.Empty);
    }
}

public class Form1
{
    public void ReadTable()
    {
        var reader = new DelimitedReader("MyFile.csv");
        reader.NextLineEvent += this.NextLine;
        ...
    }
```

Методы регистрации события

Определяя событие, можно указать методы, которые будут вызываться при добавлении делегата в список и при исключении делегата из списка.

```
public class DelimitedReader
{
    public event EventHandler NextLineEvent
    {
        add { AddDelegate(NextLineEventKey, value); }
        remove { RemoveDelegate(NextLineEventKey, value); }
    }

    protected void AddDelegate(object key, Delegate delegate) { ... }
    protected void RemoveDelegate(object key, Delegate delegate) { ... }

    private static readonly object NextLineEventKey = new object();

    ...
}
```

Интерфейс = объект – реализация

Из класса TableReader можно выделить программный интерфейс:

```
public interface ITableReader
{
    string[] Items { get; }
    int ItemCount { get; }
    bool NextLine();
    bool IsEndOfFile();
}
```

Класс TableReader можно определить с поддержкой этого интерфейса:

```
public class TableReader : Object, ITableReader
{
    public string FileName { get; private set; }
    public string[] Items { get { return fItems; } } // ITableReader
    public int ItemCount { get { return fItems.Length; } } // ITableReader

    public TableReader(string fileName) { ... }
    public void Close() { ... }
    public bool NextLine() { ... } // ITableReader
    public bool IsEndOfFile() { ... } // ITableReader

    private StreamReader fStreamReader;
```

Интерфейс

Интерфейс не может содержать поля, конструкторы, деструктор. Все элементы интерфейса по определению являются общедоступными (public) и абстрактными (abstract).

```
public interface ITableReader
{
    string[] Items { get; }
    int ItemCount { get; }
    bool NextLine();
    bool IsEndOfFile();
}
```

Интерфейс выступает дополнительной точкой доступа к объекту. Можно объявить интерфейсную переменную и присвоить ей объект, поддерживающий интерфейс переменной:

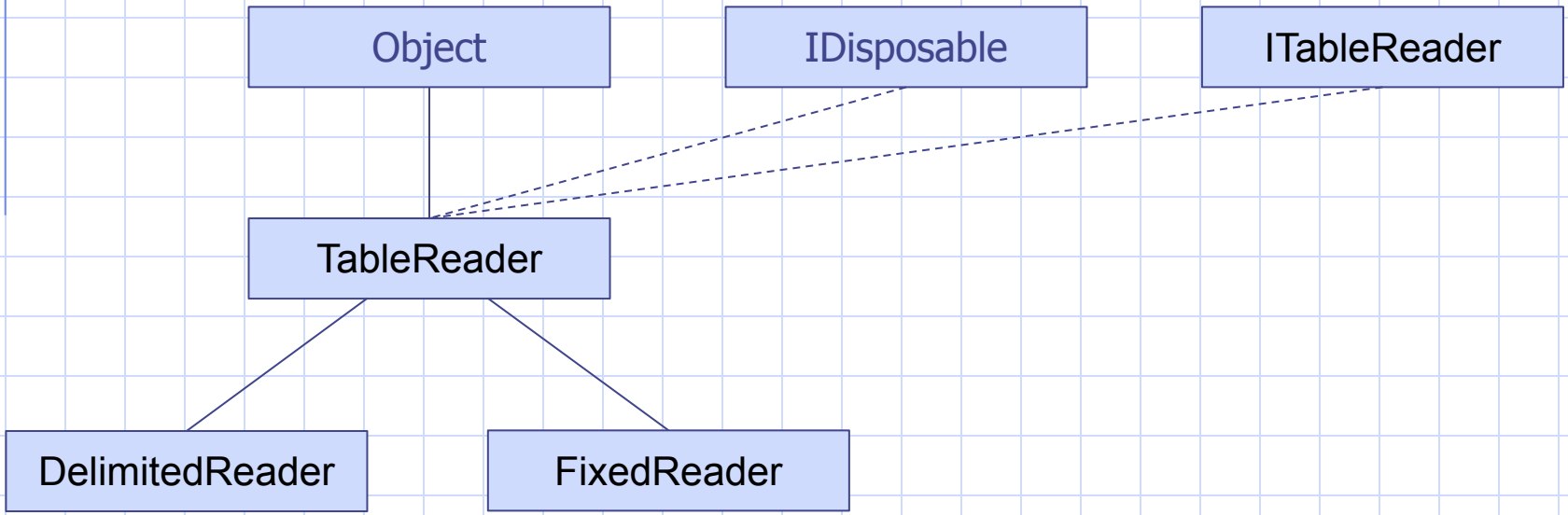
```
TableReader reader = new TableReader(...);
ITableReader intf = reader;
```

Через интерфейсную переменную можно вызывать методы интерфейса. В результате вызываются методы объекта:

```
intf.NextLine(); // reader.NextLine();
```

Интерфейс

Объект может поддерживать несколько интерфейсов, что эффективно заменяет множественное наследование:



Интерфейс

Объект может поддерживать несколько интерфейсов, что эффективно заменяет множественное наследование:

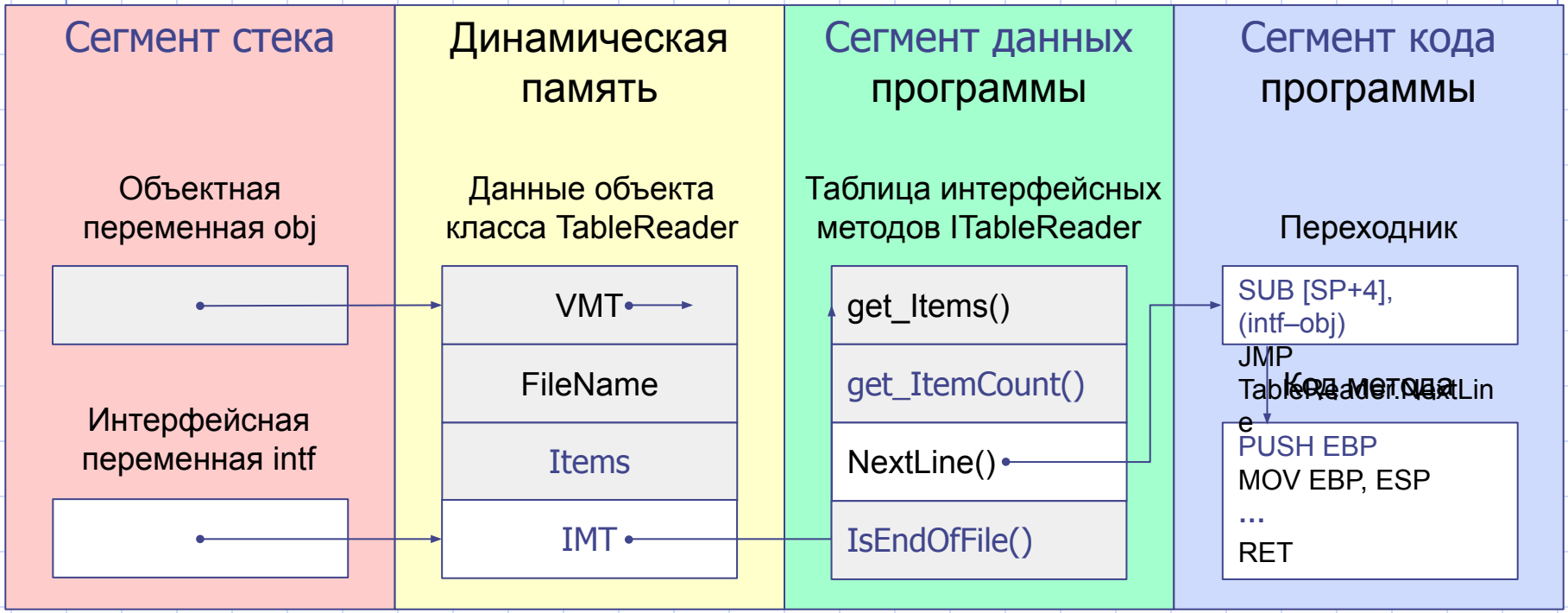
```
public class TableReader : Object, IDisposable, ITableReader
{
    public void Dispose() { ... } // IDisposable
    public string[] Items { get { return fItems; } }
    public int ItemCount { get { return fItems.Length; } }
    public bool NextLine() { ... }
    public bool IsEndOfFile() { ... }
    ...
}
```

Реализация свойств и методов интерфейса может быть закрыта внутри класса:

```
public class TableReader : Object, IDisposable, ITableReader
{
    void IDisposable.Dispose() { ... } // IDisposable
    string[] ITableReader.Items { get { return fItems; } }
    int ITableReader.ItemCount { get { return fItems.Length; } }
    bool ITableReader.NextLine() { ... }
    bool ITableReader.IsEndOfFile() { ... }
    ...
```

Механизм вызова метода через интерфейс

Схема вызова **intf.NextLine();** // intf ссылается на объект TableReader



1. Через интерфейсную переменную выполняется обращение к данным объекта: MOV EAX, intf
2. Из объекта извлекается адрес таблицы методов интерфейса: MOV EBX, [EAX]
3. На основании порядкового номера метода интерфейса из таблицы извлекается адрес метода: MOV EAX, [EBX+8]; значение 8 здесь – смещение до поля NextLine().
4. Выполняется вызов кода по извлеченному адресу: CALL EAX
5. На вершине стека корректируется аргумент this, чтобы из значения intf он превратился в ожидаемое значение obj: SUB [SP+4], (intf–obj); значение 4 здесь – смещение до this.

Шаблон – параметризованный класс

Можно определить класс, параметризованный типом данных. Такой класс называют шаблоном или обобщенным классом:

```
public class List<T>
{
    public List();
    public void Add(T item);
    public void AddRange(List<T> collection);
    public bool Remove(T item);
    public T[] ToArray();
    ...
}
```

На основе шаблона можно сконструировать тип и создать переменную сконструированного типа:

```
var stringList = new List<string>();
var integerList = new List<int>();
```

Представление в памяти сконструированного типа зависит от типа-параметра. Представление оптимизировано для каждого отдельного скалярного типа-параметра (например, int, double и т.д.) и для всех ссылочных типов (например, object, string и т.д.).

Шаблон

При описании шаблона можно ограничить возможные значения параметра-типа:

```
public class List<T>
    where T: ListItem<T>
{
    ...
    public void Add(T item); // Внутри шаблона - List<T>.Add()
    ...
}
```

Вариант шаблона с несколькими параметрами и ограничениями:

```
public class Dictionary<K, V>
    where K: IComparable<K>, ISerializable
    where V: ISerializable
{
    ...
    public void Add(K key, V value); // Dictionary<K, V>.Add()
    ...
}
```

Атрибут – метаданные

Атрибуты – это метаданные, которые можно назначать элементам программы. Эти метаданные представляются как объекты, производные от класса Attribute.

```
public class HelpAttribute : Attribute
{
    public string Url { get; private set; }
    public string Topic { get; set; }

    public HelpAttribute(string url)
    {
        Url = url;
    }
}
```

Применение атрибута:

```
[Help("http://company.com/help/Widget.htm")]
public class Widget
{
    [Help("http://company.com/help/WidgetDisplay.htm", Topic = "Dislay")]
    public void Display(string text) { ... }
}
```

Атрибут – механизм рефлексии

С помощью механизма рефлексии для каждого элемента программы – класса, поля, свойства, метода, параметра – можно получить список атрибутов и воспользоваться данными атрибутов.

```
using System;
using System.Reflection;
using System.Diagnostics;

public class Program
{
    static void Main()
    {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }

    static void ShowHelp(MemberInfo member)
    {
        HelpAttribute attr = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        Process.Start(attr.Url);
    }
}
```

Дополнительные понятия

Переменная с непостоянным типом значений

Анонимная функция

Переменная с непостоянным типом значений

Для поддержки скриптовых языков в язык C# внесли возможность создавать переменные с непостоянным типом значений. Для таких переменных допустимые операции зависят от присвоенного в данный момент значения. На этапе компиляции их можно использовать в любых выражениях. Переменные с непостоянным типом значений можно интерпретировать как объекты произвольных типов. Например, можно вызывать неизвестные методы:

```
public static void Test(dynamic obj)
{
    obj.OpenFile("MyFile.txt");
    obj.NextLine();
    obj.Close();
}
```

Результат выполнения этой процедуры зависит от того, что передано в параметре obj. Если передан объект, содержащий такие методы, тогда вызовы пройдут успешно. Если это какой-то другой объект, произойдет исключение.

Анонимная функция

```
public delegate int FunctionDelegate(int x);

public class Program
{
    public void Test()
    {
        CallFunction(this.Negate);
        CallFunction(delegate(int x) { return -x; });
        CallFunction(delegate { return 0; });
        CallFunction((int x) => { return -x; }); // лямбда-выражение
        CallFunction((int x) => -x); // предпочтительно
        CallFunction((x) => -x);
        CallFunction(x => -x);
    }

    public int Negate(int x) { return -x; }

    public void CallFunction(FunctionDelegate f)
    {
        Console.WriteLine("F(10) = {0}", f(10));
    }
}
```

Анонимная функция

Вместо вручную объявленного делегата:

```
public delegate int FunctionDelegate(int x);
```

можно использовать готовый шаблон:

```
public delegate TResult Func<in T, out TResult>(T arg);
```

Прежний вариант:

```
public void CallFunction(FunctionDelegate f)
{
    ...
}
```

Новый вариант:

```
public void CallFunction(Func<int, int> f)
{
    ...
}
```

Анонимная функция

Создадим процедурную переменную, которой присвоим ссылку на анонимную функцию:

```
Func<int, int> func = (int x) => -x; // код
```

Существует возможность представить анонимную функцию как данные с помощью шаблона System.Linq.Expressions.Expression<D>:

```
Expression<Func<int, int>> exp = func; // данные
```

Получение анонимного делегата из сконструированного в переменной типа Expression<D> выражения:

```
Func<int, int> func = exp.Compile(); // теперь код
```

Вызов:

```
int y = func(10);
```

Конструируемое выражение не может содержать некоторые элементы языка: блок { } и операторы присваивания.

Приемы программирования

Итератор – абстрактный продвигаемый вперед указатель на элемент контейнера

Одиночка – объект, создаваемый в единственном экземпляре

Заместитель – объект, перенаправляющий вызовы к другому объекту

Компоновщик – объект, компонующий набор других однотипных объектов в одно целое

Мост – делегирование функциональности метода другому объекту через интерфейс, чтобы иметь возможность независимо менять реализацию интерфейса

Наблюдатель – объект, получающий уведомления от других объектов

Посетитель – объект, передаваемый другому объекту для вызова своих методов

Фабричный метод – виртуальный метод, создающий объект

Фабрика классов – интерфейс с виртуальными методами, создающими объекты различных классов

Пул объектов – кэш заранее созданных объектов

Итератор (Iterator/Enumerator)

Итератор – абстрактный продвигаемый вперед указатель на элемент контейнера:

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Контейнеры поддерживают интерфейс создания итератора:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable
{
```

Итератор

Для удобства пользования итератором существует оператор foreach:

```
public static void Test(List<string> args)
{
    foreach (string s in args)
        Console.WriteLine(s);
}
```

Показанный выше оператор foreach транслируется в следующий код:

```
IEnumerator<string> e = args.GetEnumerator();
try
{
    while (e.MoveNext())
    {
        string s = e.Current;
        Console.WriteLine(s);
    }
}
finally
{
    e.Dispose();
}
```

Итератор

Если функция возвращает значение типа IEnumerable, то такую функцию можно сделать итератором:

```
public static void Test(List<List<string>> lists)
{
    foreach (string s in PlainList(lists))
    {
        Console.WriteLine(s);
    }
}

public static IEnumerable<string> PlainList(List<List<string>> lists)
{
    foreach (List<string> list in lists)
        foreach (string s in list)
            yield return s;
    yield break; // здесь можно опустить
}
```

Одиночка (Singleton)

Одиночка – объект, создаваемый в единственном экземпляре.

Пример – статическое (static) поле класса, инициализированное объектом с помощью оператора new:

```
public class Program
{
    public static Window Desktop = new Desktop();
    ...
}
```

Объект Program.Desktop является глобальной переменной, создаваемой один раз при первом обращении к ней.

В примере кроется потенциальная проблема. Если конструктор класса Desktop создаст исключение, объект не будет создан, и переменная Program.Desktop останется равна null. Выход – использовать функцию для создания объекта-одиночки.

Статья MSDN по объектам-одиночкам:
https://msdn.microsoft.com/en-us/library/ee817670.aspx

Одиночка

```
public class Config
{
    private static volatile Config instance = null;
    private static readonly object syncRoot = new object();

    private Config() { ... }

    public static Config Instance() // применение: Config.Instance().ToString()
    {
        if (instance == null)
        {
            lock (syncRoot) // лишь один поток может находиться в этом блоке
            {
                if (instance == null)
                {
                    instance = new Config();
                }
            }
        }
        return instance;
    }
}
```

Заместитель

Прокси (Proxy) или Суррогат (Surrogate) – легковесный объект-заместитель, перенаправляющий вызовы к замещаемому тяжеловесному объекту.

Обертка (Wrapper) или Декоратор (Decorator) – объект-заместитель, содержащий в себе замещаемый объект и предоставляющий, по сравнению с ним, новые функции.

Адаптер (Adapter) – объект, реализующий некоторый интерфейс путем обращения к другому объекту через свойственный ему интерфейс.

Прокси (Proxy) / Суррогат (Surrogate)

Прокси (Proxy) или Суррогат (Surrogate) – легковесный объект-заместитель, перенаправляющий вызовы к замещаемому тяжеловесному объекту.

Представьте, что у вас есть интерфейс IBookStorage и стандартная реализация этого интерфейса для работы с веб-сервисом, которую вы используете для создания, получения, обновления и удаления (CRUD) книг в вашей электронной библиотеке. Вы решили, что хорошо было бы кэшировать полученные данные на какое-то время. Как добавить кэш?

Можно модифицировать существующий код для работы с кэшем. А что если он должен быть опциональным? А что если другой команде нужен ваш компонент, но без всяких намеков на кэш?

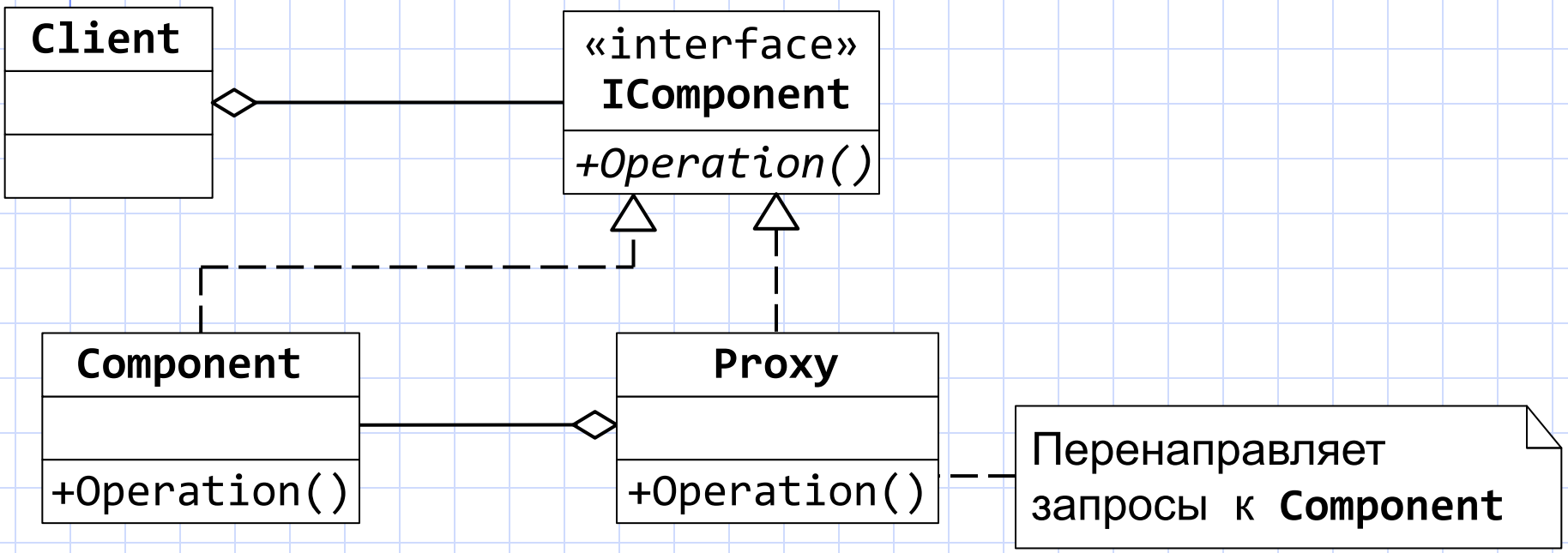
Задача решается созданием промежуточного объекта – прокси, реализующего интерфейс IConfig (IStream) путем обращения к объекту Dictionary (массиву byte[]) через свойственный ему интерфейс.

Прокси / Суррогат

```
public interface IWebClient
{
    void Request();
}

public class WebClient : IWebClient
{
    public WebClient() { ... }
    public void Request() { ... }
}

public class ProxyClient : IWebClient // Прокси для WebClient
{
    private WebClient fClient;

    public void Request()
    {
        if (fClient == null)
            fClient = new WebClient();
        fClient.Request();
    }
}
```

Прокси / Суррогат

Прокси (Proxy) – легковесный объект-заместитель, перенаправляющий вызовы к замещаемому тяжеловесному объекту (обычно через сеть).



Обертка (Wrapper) / Декоратор (Decorator)

Обертка (Wrapper) или Декоратор (Decorator) – объект-заместитель, содержащий в себе замещаемый объект и предоставляющий, по сравнению с ним, новые функции.

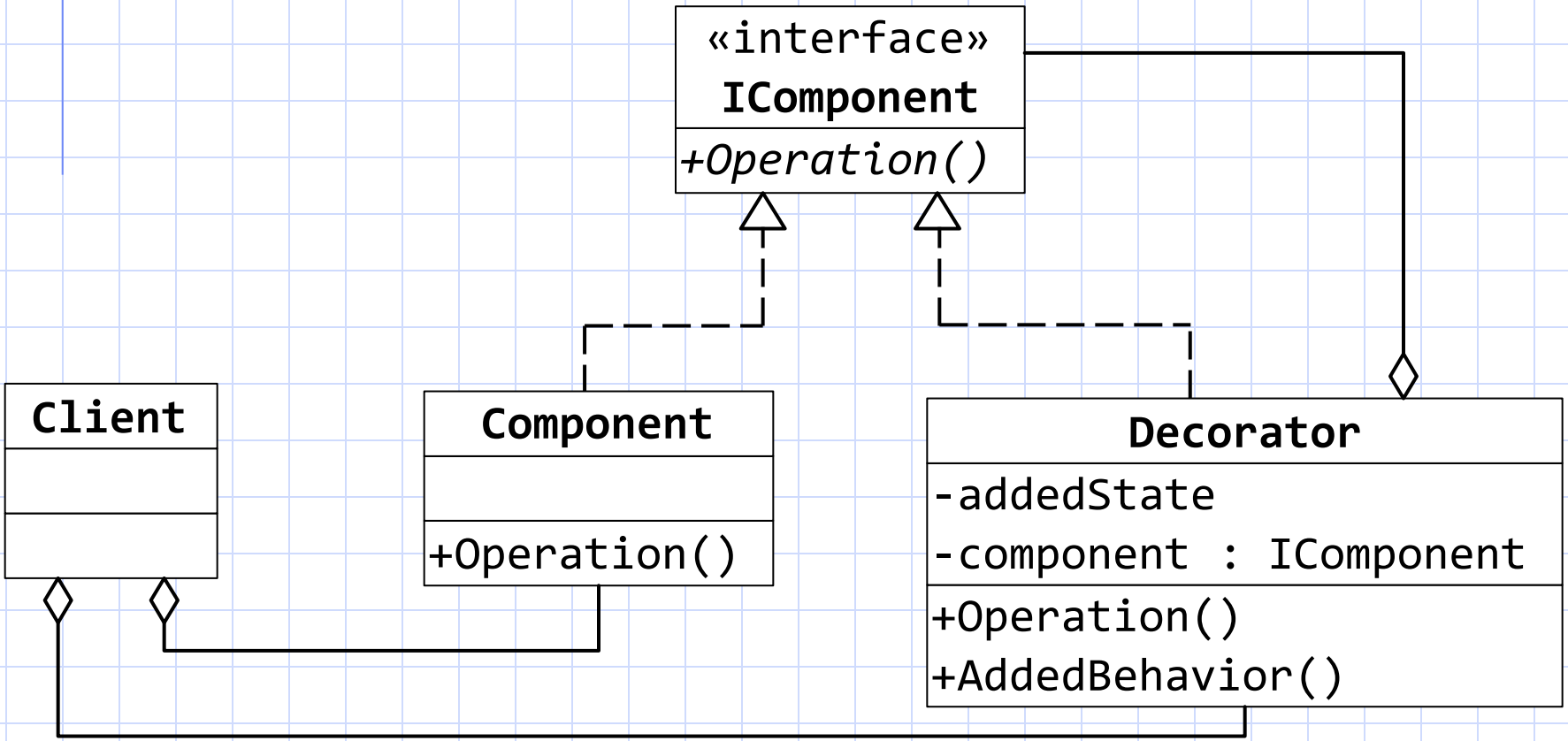
Представьте, что у вас есть интерфейс IStream (поток) и стандартная реализация этого интерфейса для чтения и записи файла (вы ее используете для сохранения пользовательских настроек). Появилось требование, чтобы данные записывались в файл в сжатом виде.

Можно вставить логику сжатия данных в место, где используется IStream (непосредственно до вызова метода Write и сразу после вызова метода Read). Но в таком случае, работать со сжатыми файлами становится неудобно – код сжатия и разжатия приходится дублировать, поддерживать программу становится сложно. Можно модифицировать код методов Write и Read. А если он должен быть опциональным? А если другой команде нужен поток, но без сжатия?

Задача решается созданием промежуточного объекта – обертки / декоратора, содержащего внутри себя поток и выполняющего сжатие и разжатие данных.

Обертка (Wrapper) / Декоратор (Decorator)

Обертка (Wrapper) или Декоратор (Decorator) – объект-заместитель, содержащий в себе замещаемый объект и предоставляющий, по сравнению с ним, новые функции.



Адаптер (Adapter)

Адаптер – объект, реализующий некоторый интерфейс путем обращения к другому объекту через свойственный ему интерфейс.

Представьте, что у вас есть интерфейс IStream (поток) и реализация этого интерфейса для чтения файла. Вам нужно сделать так, чтобы другая реализация IStream читала из массива byte[]. Но массив byte[] не поддерживает интерфейс IStream. Так что же делать?

Представьте, что у вас есть интерфейс IConfig с методом string GetValue(string key) и реализация этого интерфейса для считывания из системного реестра значений по заданным ключам. Вам для тестирования нужно сделать так, чтобы другая реализация IConfig забирала значения из вашего Dictionary<string, string>. Но Dictionary не поддерживает интерфейс IConfig. Так что же делать?

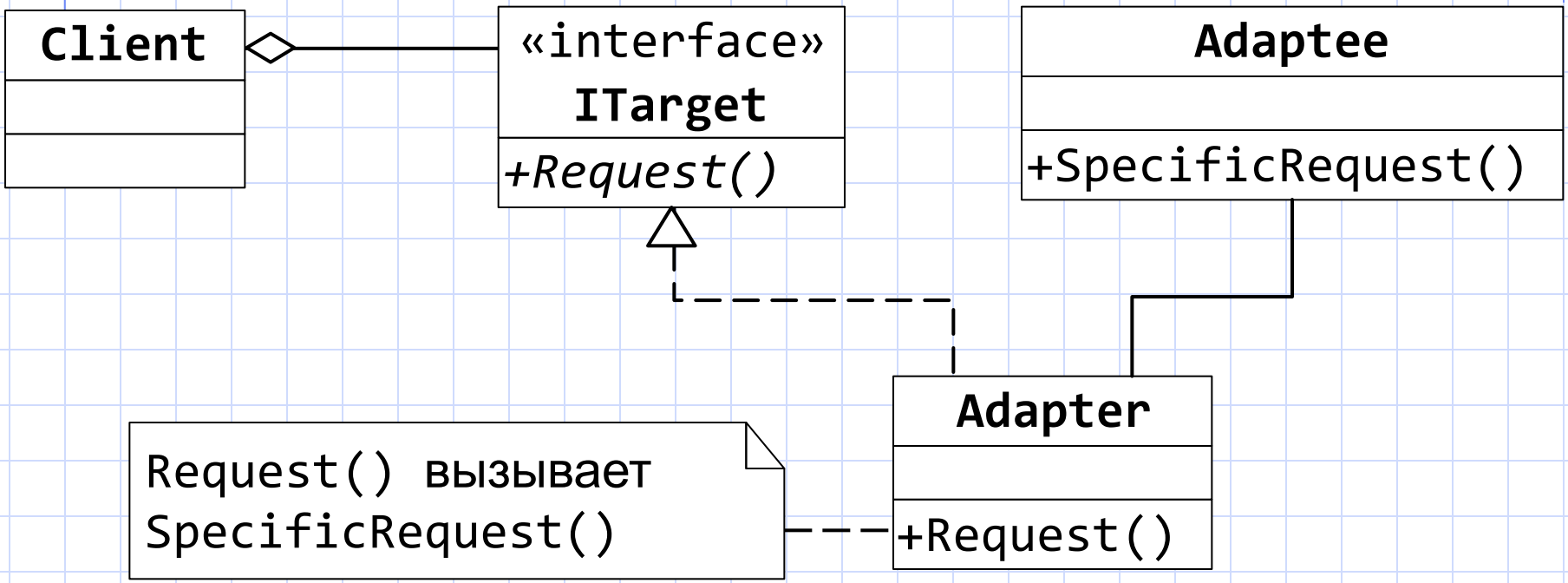
Задача решается созданием промежуточного объекта – адаптера, реализующего интерфейс IConfig (IStream) путем обращения к объекту Dictionary (массиву byte[]) через свойственный ему интерфейс.

Адаптер

```
interface IConfig
{
    string GetValue(string key);
}

class DictionaryConfig : IConfig // Адаптер для Dictionary<string, string>
{
    private Dictionary<string, string> fDictionary;

    public DictionaryConfig(Dictionary<string, string> dictionary)
    {
        fDictionary = dictionary;
    }

    public string GetValue(string key)
    {
        return fDictionary[key];
    }
}
```

Адаптер

Адаптер – объект, реализующий некоторый интерфейс путем обращения к другому объекту через свойственный ему интерфейс.



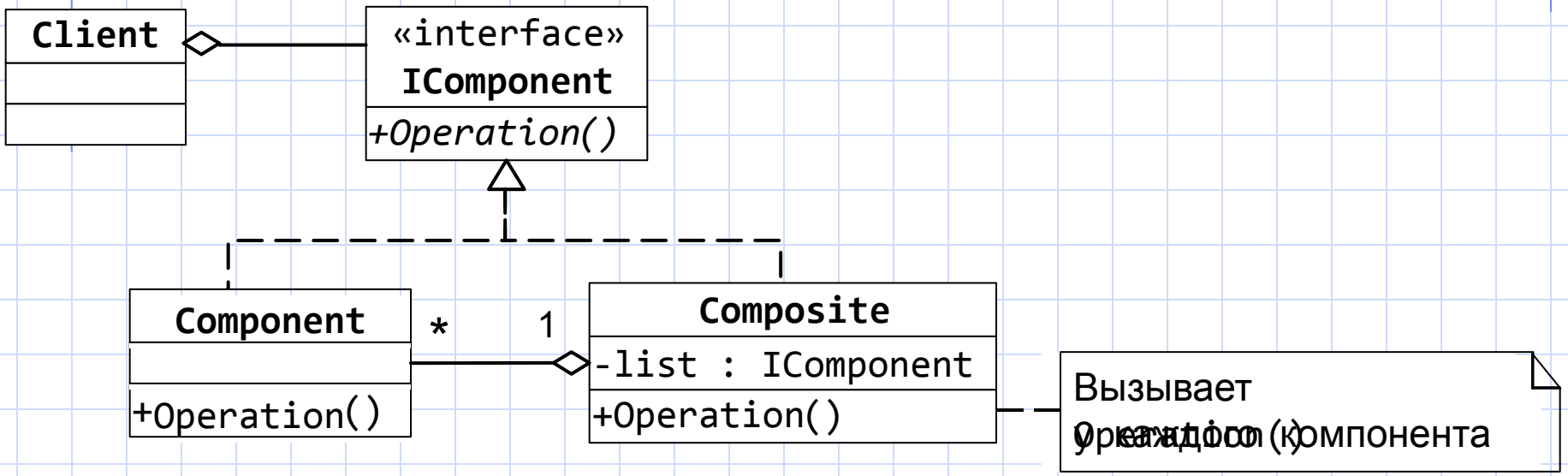
Компоновщик (Composite)

Компоновщик – объект, компонующий набор других однотипных объектов в одно целое.

```
public class CompositeStream : Stream // Компоновщик
{
    private readonly Stream[] fStreams;

    public CompositeStream(params Stream[] streams)
    {
        fStreams = streams;
    }

    public void CopyTo(Stream destination)
    {
        foreach (Stream stream in fStreams)
            stream.CopyTo(destination);
    }
}
```

Компоновщик

Компоновщик – объект, компонующий набор других однотипных объектов в одно целое.



Мост (Bridge)

Мост – делегирование функциональности метода другому объекту через интерфейс, чтобы иметь возможность независимо менять реализацию интерфейса.

```
public class Painter
{
    private readonly IPaintDevice fDevice;

    public Painter(IPaintDevice device)
    {
        fDevice = device;
    }

    public void DrawCircle(int x, int y, int radius)
    {
        fDevice.DrawCircle(x, y, radius);
    }
}

public interface IPaintDevice
{
    void DrawCircle(int x, int y, int radius);
}
```

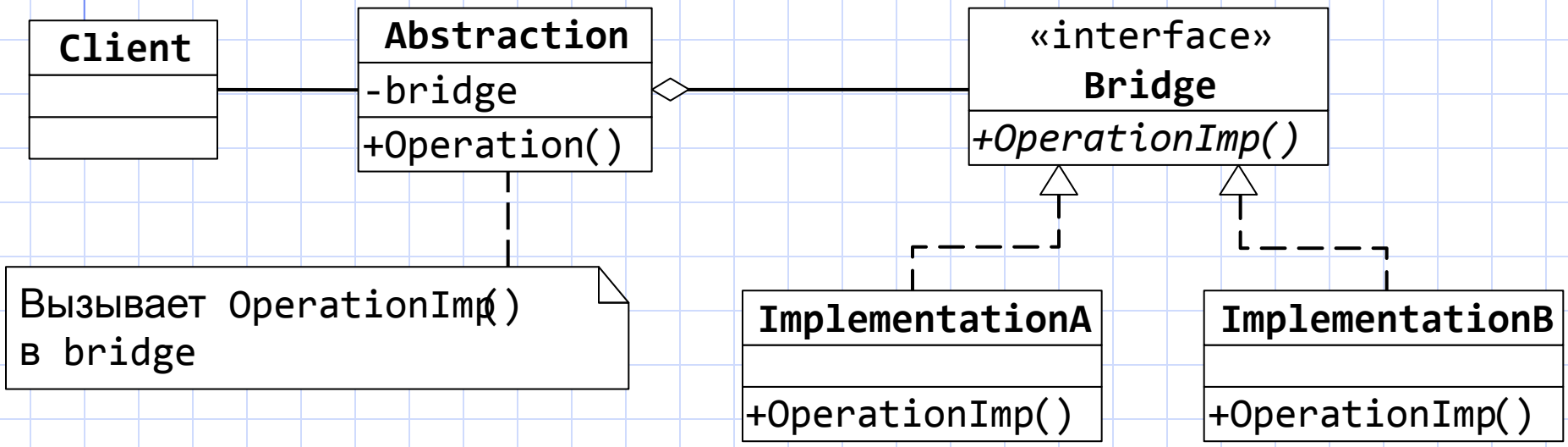
Мост

Мост через интерфейс создается к отдельной иерархии классов.

```
public class LowQualityPaintDevice : IPaintDevice
{
    public void DrawCircle(int x, int y, int radius) { ... }
}

public class HighQualityPaintDevice : IPaintDevice
{
    public void DrawCircle(int x, int y, int radius) { ... }
}

public static void TestBridge()
{
    var painter1 = new Painter(new LowQualityPaintDevice());
    painter1.DrawCircle(10, 10, 5);

    var painter2 = new Painter(new HighQualityPaintDevice());
    painter2.DrawCircle(10, 10, 5);
}
```

Мост

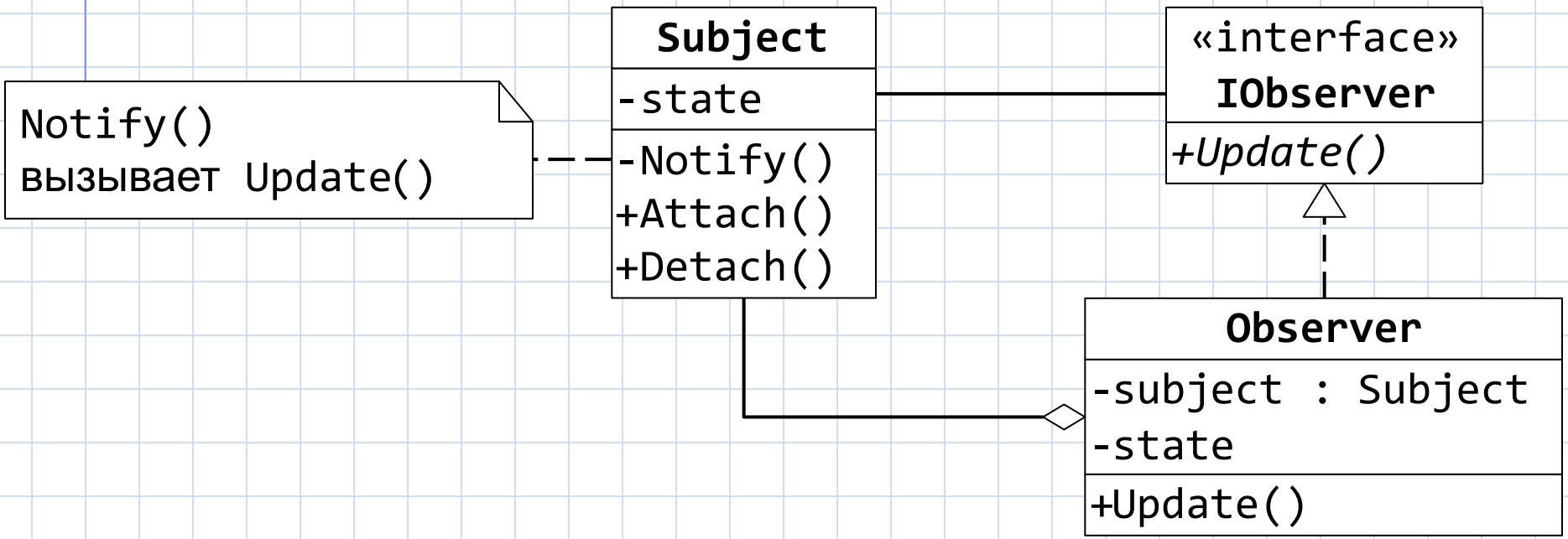
Мост – делегирование функциональности метода другому объекту через интерфейс, чтобы иметь возможность независимо менять реализацию интерфейса.



Наблюдатель (Observer/Listener)

Наблюдатель – объект, получающий уведомления от других объектов.

Наблюдатель применяется в языках программирования, не имеющих процедурных переменных (делегатов, событий).



Посетитель (Visitor)

Посетитель – объект, передаваемый другому объекту для вызова своих методов.

Например, в библиотеке визуальных компонентов существует элементы пользовательского интерфейса (Controls), которые умеют нарисовать себя на какой-то поверхности (Canvas) в какой-то позиции (X, Y). В каждом производном от Control классе перекрыт виртуальный метод void Draw(Canvas canvas, int x, int y), который рисует элемент пользовательского интерфейса.

В окне (Form) существует список элементов пользовательского интерфейса. При перерисовке окна у каждого элемента пользовательского интерфейса вызывается метода Draw с передачей ему поверхности рисования окна (Form.Canvas).

Объект, передающий себя (или какой-то свой объект) другому объекту для вызова у себя методов, и называется посетителем.

Фабричный метод (Factory Method)

Фабричный метод – виртуальный метод, создающий объект.

Фабрика классов (Factory)

Цель фабрики классов – создание объектов. Почему не создать его просто через new? – вот в чем вопрос. И здесь у каждого своя причина. Вот некоторые их них:

Проблема расширяемости конструктора. Вы хотите расширить класс, но для этого ему нужно передать в конструктор еще один интерфейс. Проблема в том, что этот класс инстанциируется во многих местах, и теперь вам придется «пофиксить» все эти места. Решение – использовать фабрику, которая будет создавать объекты этого типа. Современный подход – использование IoC контейнеров для данных целей.

Поговорим о C++. Проблема в том, что вы не можете создать объект класса просто потому, что он находится где-то в другой dll, и класс не экспортируется через __declspec(dllexport). Это обычно делается для расширяемого программирования (для плагинов). Обычно экспортируют лишь одну функцию, например GetFactory, которая возвращает IFactory, через которую вы можете создавать объекты тех или иных классов. В C# эту проблему можно обойти рефлексией, инстанциируя классы из другой dll через Activator.CreateInstance (очень распространенный подход).

Пул объектов (Object Pool)

Пул объектов – кэш заранее созданных объектов