Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А. А. Волосевич

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Курс лекций
для студентов специальности
1-40 01 03 Информатика и технологии программирования

Минск 2014

Содержание

1. Базовые сведения о шаблонах проектирования 3
2. Структурные шаблоны 4
3. Порождающие шаблоны 19
4. Шаблоны поведения 32
5. Антипаттерны 57
Литература 61

1. Базовые сведения о шаблонах проектирования

Шаблоны (или *паттерны*) *проектирования* (design patterns) – это многократно используемые решения распространённых проблем, возникающих при разработке программного обеспечения. По мере накопления личного опыта и профессионального роста программист обычно замечает сходство новых задач проектирования с задачами, решёнными им ранее. В дальнейшем приходит осознание того, что решения похожих проблем представляют собой повторяющиеся шаблоны[1]. Зная эти шаблоны, опытные программисты распознают ситуации их применения и сразу используют готовое решение, не тратя времени на предварительный анализ проблемы.

Историю шаблонов проектирования принято начинать с книг архитектора Кристофера Александера (Christopher Alexander). В семидесятых годах двадцатого века он составил набор шаблонов для применения в строительной архитектуре и городском планировании. Однако в области архитектуры эта идея не получила такого развития, как позже в области программной разработки. В 1987 году Кент Бек (Kent Beck) и Уорд Каннингем (Ward Cunningham) использовали некоторые идеи Александера и разработали несколько шаблонов на языке Smalltalk. В 1988 году Эрих Гамма (Erich Gamma) начал писать докторскую работу при Цюрихском университете об общей переносимости методики Александера на разработку программ. В 1991 году Гамма заканчивает свою диссертацию и в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) публикует книгу «Design Patterns: Elements of Reusable Object-Oriented Software»[2]. В книге были описаны 23 шаблона проектирования. «Design Patterns» стала причиной роста популярности шаблонов, а сами шаблоны из книги начали называть классическими. В дальнейшем список шаблонов пополнялся и адаптировался для различных языков программирования.

Рассмотрим аргументы «за» и «против» применения шаблонов проектирования. Главная польза каждого отдельного шаблона состоит в том, что он описывает решение целого класса абстрактных проблем. Тот факт, что каждый шаблон имеет своё имя, облегчает дискуссию между разработчиками. Таким образом, за счёт шаблонов производится унификация терминологии, названий модулей и элементов проекта. Однако есть мнение, что слепое применение шаблонов из справочника, без осмысления причин и предпосылок выделения шаблона, замедляет профессиональный рост программиста, так как подменяет творческую работу механической подстановкой. Люди, придерживающиеся данного мнения, считают, что знакомиться со списками шаблонов следует тогда, когда программист «дорос» до них в профессиональном плане. Хороший критерий нужной степени профессионализма – выделение шаблонов самостоятельно, на основании собственного опыта. При этом, разумеется, знакомство с теорией, связанной с

[1] Алгоритмы не рассматриваются как шаблоны, так как они решают задачи вычисления, а не проектирования.

[2] Команда авторов этой книги известна под названием «Банда четырёх» (Gang of Four – GoF).

шаблонами, полезно на любом уровне профессионализма и направляет развитие программиста в правильную сторону.

Далее будут рассмотрены основные шаблоны проектирования, разбитые на группы. Существует несколько способов группировки шаблонов. Ниже представлена классификация в зависимости от назначения:

1. *Структурные шаблоны* (structural patterns) – показывают, как объекты и классы объединяются для образования сложных структур.
2. *Порождающие шаблоны* (creational patterns) – контролируют процесс создания и жизненный цикл объектов.
3. *Шаблоны поведения* (behavioral patterns) – используются для организации, управления и объединения различных вариантов поведения объектов.

2. Структурные шаблоны

В данном параграфе рассматривается набор классических структурных шаблонов из книги «Design Patterns». Шаблоны упорядочены согласно их русскому названию.

2.1. Адаптер (Adapter)

Шаблон Адаптер нужен для обеспечения коллективной работы классов, которые изначально не были созданы для совместного использования. Такие ситуации часто возникают, когда идёт работа со сторонними библиотеками кода. Нередко их интерфейс не отвечает требованиям клиента[1], но изменить библиотеку нет возможности. Следовательно, появляется задача адаптации библиотеки для клиента.

Два основных варианта дизайна шаблона Адаптер показаны на рис. 1. Класс `Client` работает в соответствии с требованиями своей предметной области, эти требования отражены в интерфейсе `ITarget`. *Адаптируемый класс* `Adaptee` обладает нужной функциональностью, но неподходящим интерфейсом. Класс `Adapter` реализует интерфейс `ITarget` и перенаправляет вызовы от `Client` к `Adaptee`, изменяя при необходимости аргументы и возвращаемые значения. В случае *адаптера класса* `Adapter` одновременно реализует интерфейс `ITarget` и наследуется от класса `Adaptee`. Альтернативой наследованию может стать агрегация объекта `Adaptee`, что соответствует *адаптеру объекта*. При наследовании проще *переопределить* поведение адаптируемого класса, а при агрегации – легче *добавить* к адаптируемому классу новое поведение.

[1] Под термином «клиент» здесь и далее подразумевается некое программное обеспечение (в частном случае – класс, компонент), *использующее* библиотеки кода, интерфейсы, классы.

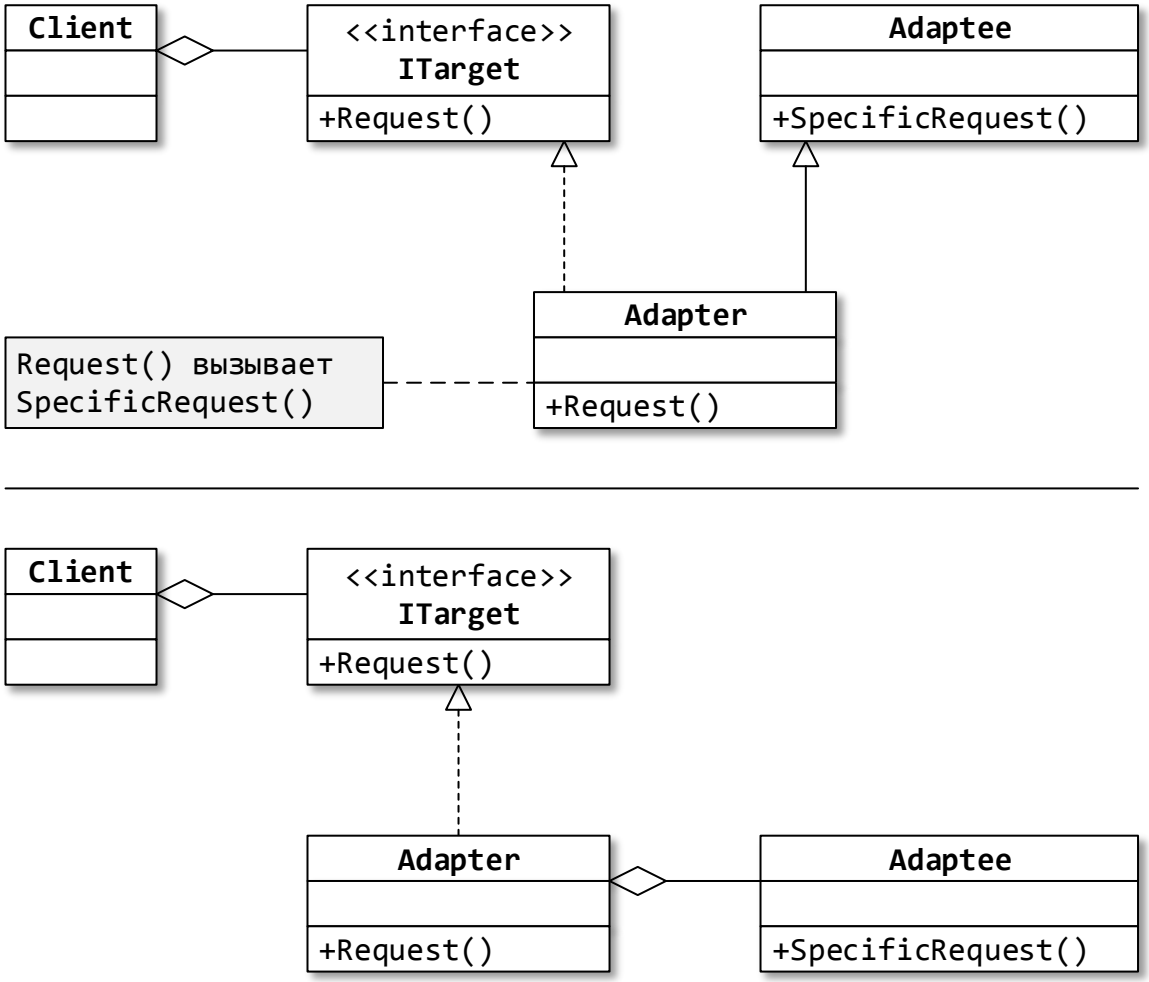

Рис. 1. Дизайн адаптера класса (вверху) и адаптера объекта (внизу).

Рассмотрим пример реализации адаптера объекта. Пусть в приложении необходимо воспроизводить звуковые файлы в формате `wav`. Интерфейс `IAudioPlayer` содержит описание функций простого медиаплеера, осуществляющего загрузку и воспроизведение аудиофайла.

```
public interface IAudioPlayer
{
    // загрузка аудиофайла
    void Load(string file);

    // воспроизведение аудиофайла
    void Play();
}
```

При реализации обратим внимание на тот факт, что платформа .NET уже содержит класс `System.Media.SoundPlayer`, предоставляющий подобные возможности. Однако данный класс не поддерживает заданный интерфейс. Поэтому воспользуемся шаблоном Адаптер.

```
public class SoundPlayerAdapter : IAudioPlayer
{
    // адаптируемый объект
    private readonly SoundPlayer _player = new SoundPlayer();

    public void Load(string file)
    {
        _player.SoundLocation = file;
        _player.Load();
    }

    public void Play()
    {
        _player.Play();
    }
}
```

В отличие от явного использования класса `SoundPlayer`, применённый подход обеспечил независимость кода клиента от конкретной реализации медиаплеера. Например, в дальнейшем можно легко заменить `SoundPlayerAdapter` на другой класс для поддержки файлов иного формата.

2.2. Декоратор (Decorator)

Шаблон Декоратор предназначен для динамического добавления к объекту новой функциональности. Он является гибкой альтернативой наследованию (в том числе множественному). *Декоратор* служит обёрткой для *декорируемого объекта* и реализует тот же интерфейс, что и класс декорируемого объекта. Однако декоратор добавляет свой код до, после или вместо вызовов методов декорируемого объекта. Это приводит к модификации исходного поведения и появлению новых возможностей.

Выделим следующих участников шаблона Декоратор (рис. 2). Интерфейс `IComponent` определяет общие функции исходных компонентов и декораторов[1]. Класс `ConcreteComponent` – это компонент, функциональность которого необходимо модифицировать. Класс `Decorator` – базовый декоратор (как правило, он включает механизм хранения компонента и реализует простую переадресацию). Конкретные декораторы `ConcreteDecoratorA` и `ConcreteDecoratorB` наследуются от базового декоратора, добавляя определённое состояние и (или) поведение.

[1] В шаблоне Декоратор, как и во многих других шаблонах, вместо общего интерфейса допустимо использование общего абстрактного класса (и наоборот).

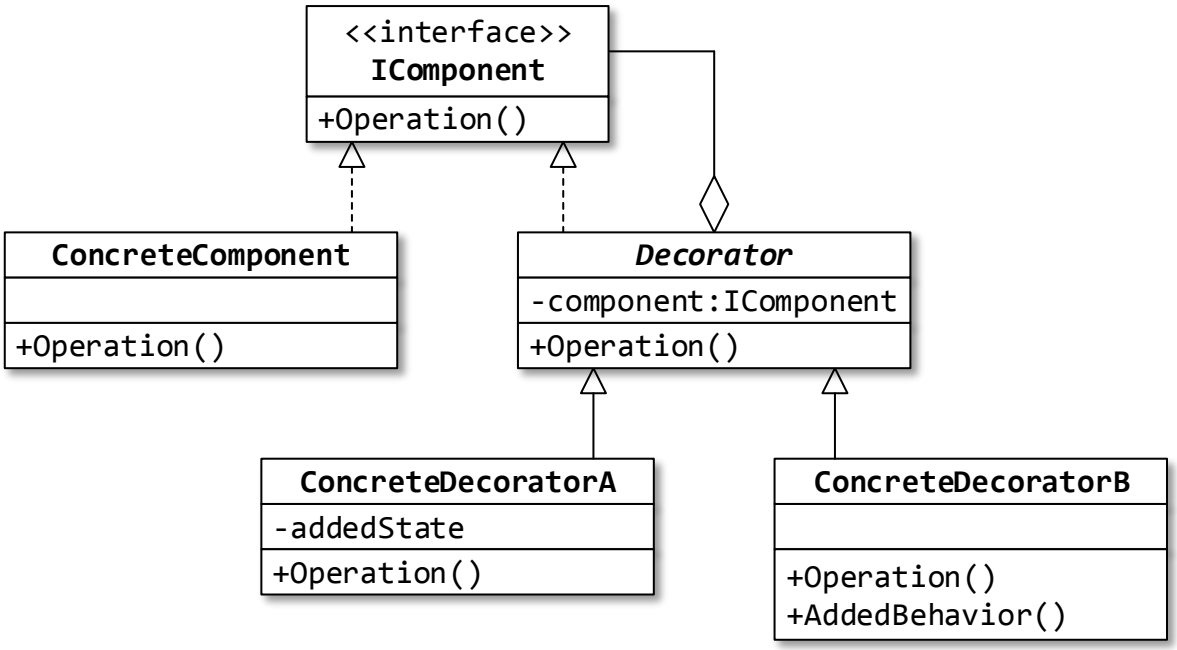

Рис. 2. Схема шаблона Декоратор.

Разберём использование шаблона Декоратор на следующем примере. Пусть имеется класс `Element`, описывающий элемент блок-схемы. Для простоты ограничимся текстом элемента и методом его отображения на консоли.

```
public interface IElement
{
    string Text { get; set; }
    void Draw();
}

public class Element : IElement
{
    public string Text { get; set; }

    public void Draw()
    {
        Console.WriteLine("Element text = {0}", Text);
    }
}
```

Создадим базовый декоратор для класса `Element`.

```
public class ElementDecorator : IElement
{
    protected readonly IElement Component;

    public ElementDecorator(IElement component)
    {
        Component = component;
    }
```

```
        public virtual string Text
        {
            get { return Component.Text; }
            set { Component.Text = value; }
        }

        public virtual void Draw()
        {
            Component.Draw();
        }
    }
```

Теперь разработаем два декоратора: один для добавления отступов перед элементом, а второй – для обрамления элемента в скобки.

```
    public class ElementWithIdent : ElementDecorator
    {
        public int Ident { get; set; }

        public ElementWithIdent(IElement component) : base(component)
        {
        }

        public override void Draw()
        {
            MakeIdent();
            base.Draw();
        }

        private void MakeIdent()
        {
            Console.Write(new string(' ', Ident));
        }
    }

    public class ElementWithBrackets : ElementDecorator
    {
        public ElementWithBrackets(IElement component) : base(component)
        {
        }

        public override void Draw()
        {
            Console.WriteLine("[");
            base.Draw();
            Console.WriteLine("]");
        }
    }
```

Все готово к применению компонентов и их декораторов. Вначале создадим исходный компонент, а потом дополним его возможности, используя оба декоратора. Обратите внимание на то, что несколько декораторов могут независимо применяться к одному и тому же объекту, а также на то, что декоратор может декорировать объект, который уже декорирован.

```
// обыкновенный элемент
IElement element = new Element {Text = "Demo"};
element.Draw();

// декорирование отступом
IElement ei = new ElementWithIdent(element) {Ident = 4};
ei.Draw();

// дополнительное декорирование скобками
IElement eb = new ElementWithBrackets(ei);
eb.Draw();
```

2.3. Заместитель (Proxy)

Шаблон Заместитель позволяет контролировать доступ к заданному объекту, перехватывая все вызовы к этому объекту и прозрачно замещая его. Ни интерфейс, ни функциональность замещённого объекта с точки зрения клиента не меняются. Данный шаблон часто используется, если необходимо упростить или оптимизировать взаимодействие с объектом, скрывая несущественные для конкретной задачи подробности реализации.

На рис 3. показан дизайн шаблона Заместитель. И *заместитель* `Proxy`, и класс *замещаемого объекта* `Subject` реализуют общий интерфейс `ISubject`. Заместитель может агрегировать замещаемый объект или порождать локальные экземпляры этого объекта при необходимости. Как правило, клиент не имеет прямого доступа к замещаемому объекту.

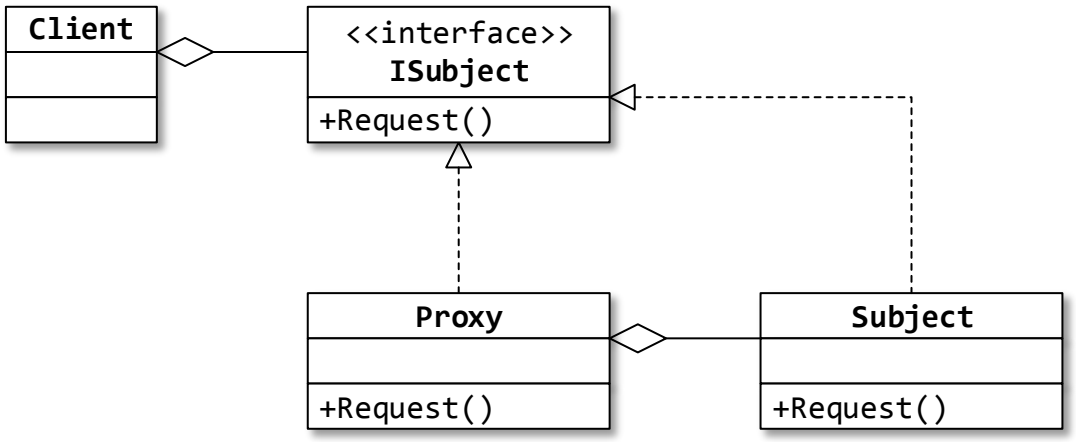

Рис. 3. Дизайн шаблона Заместитель.

Перечислим некоторые разновидности шаблона Заместитель:

– *Удалённый заместитель* (remote proxy) обеспечивает связь с замещаемым объектом, который находится в другом адресном пространстве или на удалённой машине;

– *Виртуальный заместитель* (virtual proxy) реализует создание замещаемого объекта только тогда, когда он действительно необходим;

– *Защищающий заместитель* (protection proxy) проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права.

Рассмотрим примера реализации виртуального заместителя для оптимизации работы с большими изображениями.

```
public interface IImage
{
    void Display();
}

public class RealImage : IImage
{
    private readonly string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadImageFromDisk();
    }

    public void Display()
    {
        Console.WriteLine("Displaying " + _filename);
    }

    private void LoadImageFromDisk()
    {
        Console.WriteLine("Loading " + _filename);
    }
}

public class ProxyImage : IImage
{
    private RealImage _image;
    private readonly string _filename;

    public ProxyImage(string filename)
    {
        _filename = filename;
    }

    public void Display()
    {
        if (_image == null)
```

```
        {
            _image = new RealImage(_filename);
        }
        _image.Display();
    }
}
```

Приведём клиентский код, работающий с виртуальным заместителем:

```
IImage image1 = new ProxyImage("HiRes_10MB_Photo1");
IImage image2 = new ProxyImage("HiRes_20MB_Photo2");

image1.Display();    // выполняется загрузка
image1.Display();    // загрузка не выполняется
image2.Display();    // выполняется загрузка
image2.Display();    // загрузка не выполняется
image1.Display();    // загрузка не выполняется
```

2.4. Компоновщик (Composite)

Шаблон Компоновщик позволяет упростить и стандартизировать взаимодействие между клиентом и группой объектов, представляющих древовидную структуру. Этот шаблон используется, если необходимо, чтобы клиент одинаково обращался как с составным объектом, так и с отдельными его частями.

Рассмотрим элементы шаблона Компоновщик (рис. 4). Интерфейс `IComponent` является общим для составных объектов и их частей. С точки зрения клиента все объекты являются экземплярами `IComponent` – клиент не различает составные объекты и их части. Класс `Leaf` представляет «неделимые» объекты (название взято по аналогии с наименованием элемента древовидной структуры). Класс `Composite` описывает составной объект, он может содержать как неделимые объекты, так и другие составные объекты.

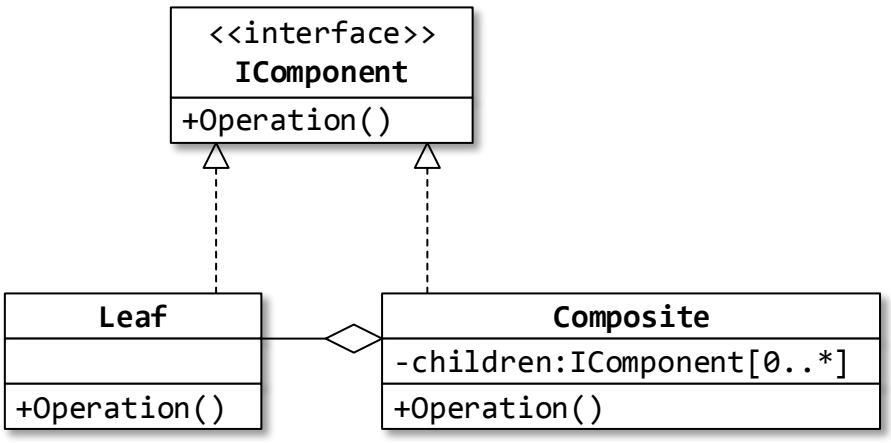

Рис. 4. Дизайн шаблона Компоновщик.

Разберём пример использования шаблона Компоновщик. Пусть необходимо работать с картой, отдельные элементы которой реализуют интерфейс `IComponent`.

```
public interface IComponent
{
    string Title { get; set; }
    void Draw();
    IComponent FindChild(string title);
}

public class MapComponent : IComponent
{
    public string Title { get; set; }

    public void Draw()
    {
        Console.WriteLine(Title);
    }

    public IComponent FindChild(string title)
    {
        return (Title == title) ? this : null;
    }
}
```

Применим класс `MapComposite`, чтобы «собрать» фрагменты карты.

```
public class MapComposite : IComponent
{
    private readonly List<IComponent> _map = new List<IComponent>();

    public string Title { get; set; }

    public void AddComponent(IComponent component)
    {
        _map.Add(component);
    }

    public void Draw()
    {
        Console.WriteLine(Title);
        foreach (IComponent component in _map)
        {
            component.Draw();
        }
    }

    public IComponent FindChild(string title)
    {
        if (Title == title)
        {
            return this;
        }
```

```
            foreach (IComponent component in _map)
            {
                var found = component.FindChild(title);
                if (found != null)
                {
                    return found;
                }
            }
            return null;
        }
    }
```

Покажем, как используются описанные классы.

```
var district = new MapComposite {Title = "District"};
district.AddComponent(new MapComponent {Title = "House 1"});
district.AddComponent(new MapComponent {Title = "House 2"});

var city = new MapComposite {Title = "New city"};
city.AddComponent(district);
city.Draw();

var house = city.FindChild("House 1");
house.Draw();
```

2.5. Мост (Bridge)

Шаблон Мост позволяет разделить объект на *абстракцию* и *реализацию* так, чтобы они могли изменяться независимо друг от друга. При этом основной код, необходимый для функционирования объекта, переносится в реализацию. Всё остальное, включая взаимодействие с клиентом, содержится в абстракции. Методы абстракции при необходимости могут быть изменены или дополнены. Абстракция содержит экземпляр реализации и использует его для обработки поступающих от клиента запросов.

Выделим следующих участников шаблона Мост (рис. 5). Класс `Abstraction` определяет базовую абстракцию. Класс `RefinedAbstraction` наследуется от `Abstraction` и представляет уточнённый функционал взаимодействия. Интерфейс `IImplementor` является интерфейсом реализации. Конкретная реализация описывается классом `ConcreteImplementor`.

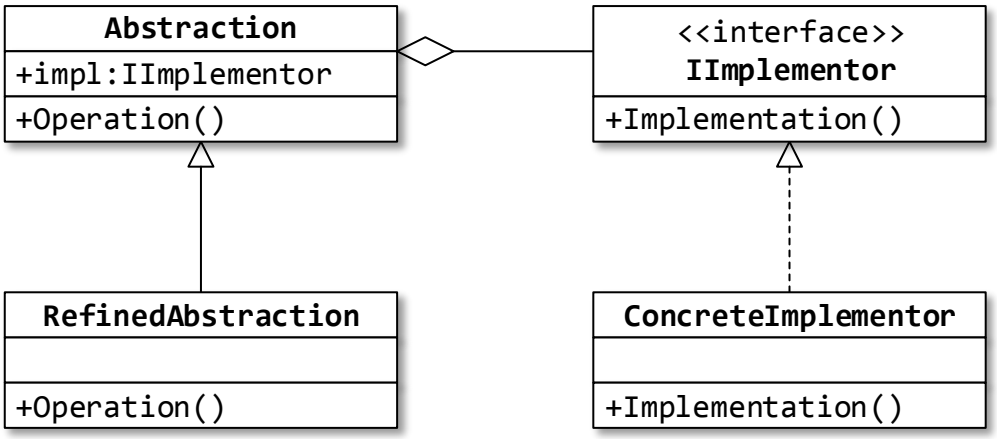

Рис. 5. Дизайн шаблона Мост.

Проиллюстрируем применение шаблона Мост. Пусть имеется класс `RecordBase`, который описывает функции для работы с набором записей.

```
public class RecordBase
{
    public IDataObject<string> DataObject { get; set; }

    public virtual void Next()
    {
        DataObject.NextRecord();
    }

    public virtual void Prior()
    {
        DataObject.PriorRecord();
    }

    public virtual void Add(string name)
    {
        DataObject.AddRecord(name);
    }

    public virtual void Delete(string name)
    {
        DataObject.DeleteRecord(name);
    }

    public virtual void Get()
    {
        var customer = DataObject.GetCurrentRecord();
        Console.WriteLine(customer);
    }
}
```

Класс `Record` уточняет одну из функций класса `RecordBase`:

```
public class Record : RecordBase
{
    public override void Get()
    {
        Console.WriteLine("----------");
        base.Get();
        Console.WriteLine("----------");
    }
}
```

Интерфейс `IDataObject<T>` задаёт методы хранилища записей.

```
public interface IDataObject<T>
{
    void NextRecord();
    void PriorRecord();
    void AddRecord(T t);
    void DeleteRecord(T t);
    T GetCurrentRecord();
}
```

Класс `ListData` описывает конкретное хранилище (список строк).

```
public class ListData : IDataObject<string>
{
    private readonly List<string> _records;
    private int _current;

    public ListData()
    {
        _records = new List<string> {"Jim Jones", "Samuel Jackson"};
    }

    public void NextRecord()
    {
        if (_current < _records.Count - 1)
        {
            _current++;
        }
    }

    public void PriorRecord()
    {
        if (_current > 0)
        {
            _current--;
        }
    }
```

```
        public void AddRecord(string customer)
        {
            _records.Add(customer);
        }

        public void DeleteRecord(string customer)
        {
            _records.Remove(customer);
        }

        public string GetCurrentRecord()
        {
            return _records[_current];
        }
    }
```

Указанные классы можно использовать следующим образом:

```
// создаём объект уточнённого функционала
var record = new Record();

// связываем с конкретной реализацией
record.DataObject = new ListData();

// косвенно вызываем методы конкретной реализации
record.Get();
record.Next();
record.Get();
```

2.6. Приспособленец (Flyweight)

Шаблон Приспособленец применяется для уменьшения затрат при работе с большим количеством объектов. *Приспособленец* – это разделяемый объект, который можно использовать одновременно в нескольких контекстах. Чтобы это стало возможным, у объекта выделяют внутреннее и внешнее состояния. *Внутреннее состояние* (intrinsic state) хранится в приспособленце и состоит из информации, не зависящей от контекста. Именно поэтому оно может разделяться. *Внешнее состояние* (extrinsic state) зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Клиент отвечает за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Опишем компоненты шаблона Приспособленец (рис. 6). Класс `Flyweight` является базовым классом для любого приспособленца и отражает возможность передачи в приспособленец внешнего состояния. От этого класса наследуются конкретные классы приспособленцев – как с внутренним состоянием (`ConcreteFlyweight`), так и без него (`UnsharedFlyweight`). Класс `FlyweightFactory` – это *фабрика приспособленцев*, которая содержит метод для получения необходимого приспособленца по ключу.

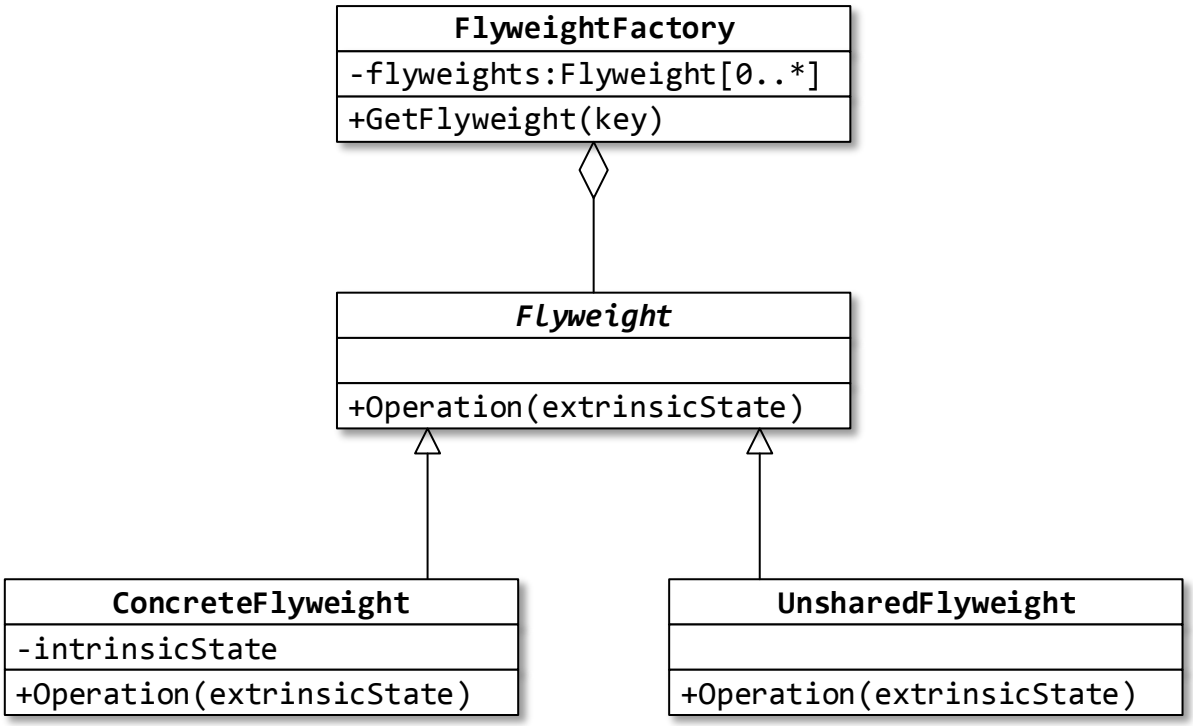

Рис. 6. Диаграмма шаблона Приспособленец.

Приведём пример использования шаблона Приспособленец. Пусть необходимо работать с текстом, состоящим из букв латинского алфавита и пробельных символов. Абстрактный класс `Character` описывает один символ текста, занимающий определённую позицию (она играет роль внешнего состояния).

```
public abstract class Character
{
    public abstract void Display(int position);
}
```

У класса `Character` есть два наследника: класс `Latin` хранит символ латинского алфавита (это его разделяемое внутреннее состояние), класс `Whitespace` представляет пробел (и не имеет разделяемого состояния).

```
public class Latin : Character
{
    public char Symbol { get; set; }

    public override void Display(int position)
    {
        Console.WriteLine("{0} (position {1})", Symbol, position);
    }
}

public class Whitespace : Character
{
    public override void Display(int position)
    {
```

```
            Console.WriteLine("_ (position {0})", position);
        }
    }
```

Для эффективного хранения символов используем класс `CharacterFactory`.

```
public class CharacterFactory
{
    private Dictionary<char, Character> _chars =
                                    new Dictionary<char, Character>();

    public Character GetCharacter(char key)
    {
        if (!_chars.ContainsKey(key))
        {
            if (key == ' ')
            {
                _chars.Add(key, new Whitespace());
            }
            else
            {
                _chars.Add(key, new Latin {Symbol = key});
            }
        }
        return _chars[key];
    }
}
```

Клиентский код использует фабрику, чтобы получить необходимый объект-приспособленец. После этого у приспособленца устанавливается внешнее состояние, а затем этот объект используется.

```
var document = "AA BB AB";
var chars = document.ToCharArray();
var f = new CharacterFactory();

// position - это внешнее состояние
var position = 0;

// используем для каждого символа объект-приспособленец
foreach (var c in chars)
{
    var character = f.GetCharacter(c);
    character.Display(position++);
}
```

2.7. Фасад (Facade)

Шаблон Фасад позволяет скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. Шаблон Фасад строит над набором подсистем интерфейс, причём этот интерфейс может быть абсолютно любым. Допускается существование нескольких фасадов для одного набора подсистем. Клиент вместо прямой работы с подсистемами использует предложенный фасадом интерфейс.

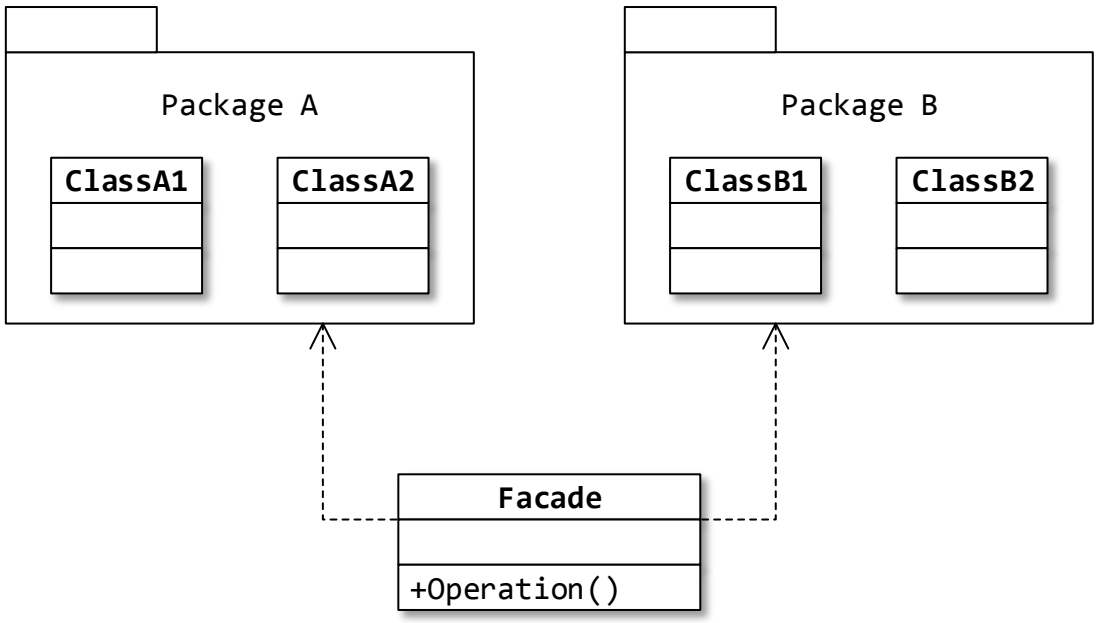

Рис. 7. Дизайн шаблона Фасад.

В качестве иллюстрации применения шаблона Фасад рассмотрим работу с классами системы отправки почтовых сообщений. Письмо, адресат, тело письма, почтовый сервер, присоединённые файлы – все это является отдельным объектами специальных классов. Без применения фасада клиент должен взаимодействовать со всем перечисленным набором объектов, используя их свойства и методы. Фасад берет работу по взаимодействию на себя – теперь клиент использует только объект-фасад, который делегирует работу нужным подсистемам.

Детали реализации шаблона Фасад не представляют трудностей. Из возможных вариантов отметим *прозрачные фасады* (в этом случае компоненты подсистемы могут быть доступны и через фасад, и в обход его) и *статические фасады* (фасад является статическим классом – скрываемые объекты не агрегируются, а создаются в методах фасада по необходимости).

3. Порождающие шаблоны

В параграфе рассматривается набор порождающих шаблонов из книги «Design Patterns» и два дополнительных шаблона: Отложенная инициализация и Пул объектов. Шаблоны упорядочены согласно их русскому названию.

3.1. Абстрактная фабрика (Abstract factory)

Шаблон Абстрактная фабрика предназначен для создания объектов, принадлежащих одному набору классов и используемых совместно. Абстрактная фабрика переопределяется в конкретных классах-фабриках, создающих объекты одного набора. Этот шаблон изолирует имена классов и их определения от клиента: единственный способ для клиента получить необходимый объект – это воспользоваться одной из реализаций абстрактной фабрики.

Дизайн шаблона Абстрактная фабрика включает большое число участников, однако он достаточно прост (рис. 8). Клиент работает с определённым набором объектов, но использует для этого только реализуемые классами объектов интерфейсы (`IProductA` и `IProductB`). Клиент хранит ссылку на конкретную фабрику, реализующую интерфейс `IFactory`. Для получения нужного объекта клиент вызывает один из методов фабрики.

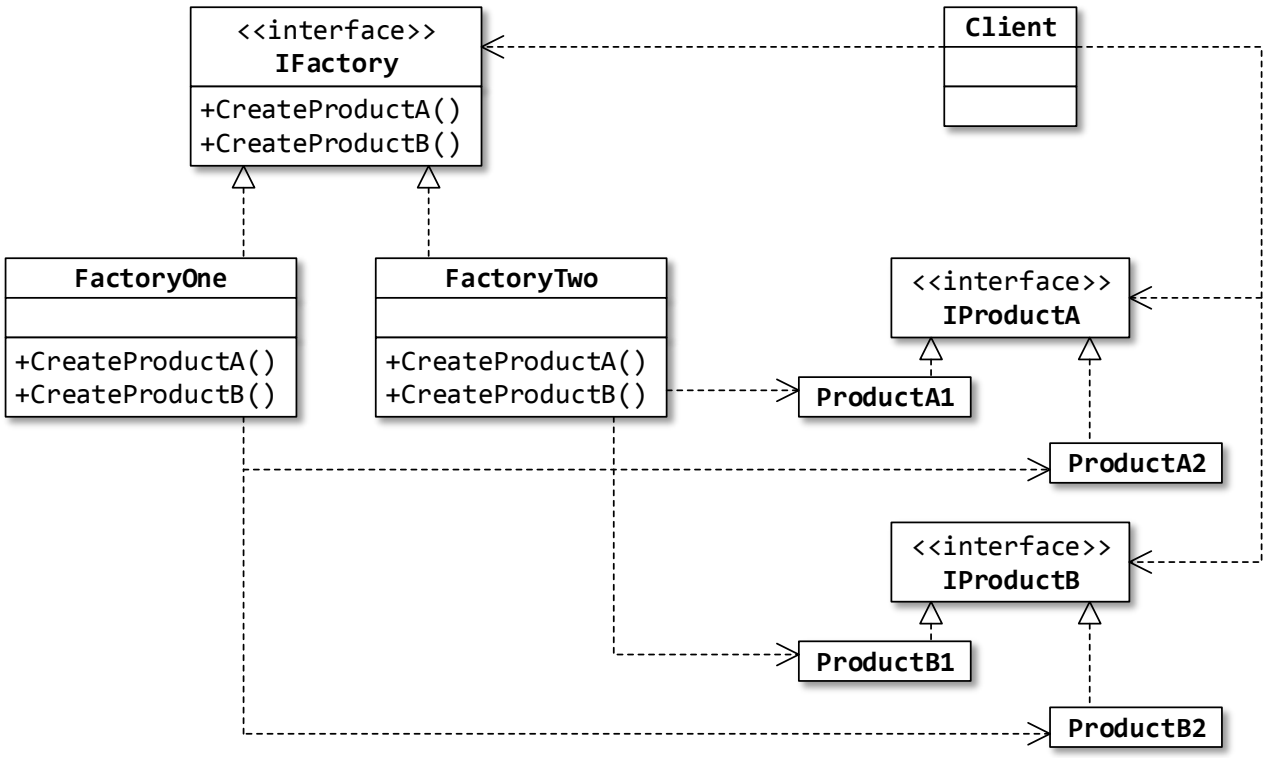

Рис. 8. Шаблон Абстрактная фабрика.

Рассмотрим пример кода с реализацией шаблона Абстрактная фабрика. Пусть клиент работает с интерфейсами для представления сумок и туфель.

```
public interface IBag
{
    string Material { get; }
}

public interface IShoes
{
    int Price { get; }
}
```

Имеется два набора классов, реализующих указанные интерфейсы.

```
// сумка и туфли Gucci
public class GucciBag : IBag
{
    public string Material
    {
        get { return "Crocodile skin"; }
    }
}

public class GucciShoes : IShoes
{
    public int Price
    {
        get { return 1000; }
    }
}

// сумка и туфли Poochy :)
public class PoochyBag : IBag
{
    public string Material
    {
        get { return "Plastic"; }
    }
}

public class PoochyShoes : IShoes
{
    public int Price
    {
        get { return 50; }
    }
}
```

Опишем фабричный интерфейс и реализуем две конкретные фабрики.

```
public interface IFactory
{
    IBag CreateBag();
    IShoes CreateShoes();
}

public class GucciFactory : IFactory
{
    public IBag CreateBag()
    {
        return new GucciBag();
    }
```

```
        public IShoes CreateShoes()
        {
            return new GucciShoes();
        }
    }

    public class ChinaFactory : IFactory
    {
        public IBag CreateBag()
        {
            return new PoochyBag();
        }

        public IShoes CreateShoes()
        {
            return new PoochyShoes();
        }
    }
```

Код клиента один раз создаёт требуемую фабрику, а далее работает только с интерфейсами фабрики и товаров.

```
// можно изменить на factory = new ChinaFactory();
IFactory factory = new GucciFactory();
IBag bag = factory.CreateBag();
IShoes shoes = factory.CreateShoes();
```

3.2. Одиночка (Singleton)

Шаблон Одиночка гарантирует создание единственного экземпляра объекта некоторого класса и предоставляет точку доступа для получения этого экземпляра. Существование единственного объекта часто требуется при организации доступа к аппаратному обеспечению, реализации кэша, систем ведения отладочной информации и т. п.

Общая схема шаблона достаточно проста (рис. 9). Конструктор класса объявляется закрытым – создавать объекты могут только методы самого класса. Для хранения единственного экземпляра используется закрытое статическое поле. Для получения экземпляра служит статический метод.

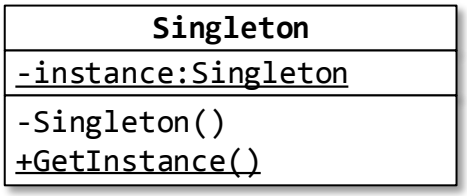

Рис. 9. Схема шаблона Одиночка.

Ниже представлен код, отвечающий описанной схеме шаблона Одиночка.

```
public sealed class Singleton
{
    private static Singleton instance;

    private Singleton()
    {
    }

    public static Singleton GetInstance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Данная реализация шаблона имеет важный недостаток – она неустойчива в многопоточных приложениях. Перепишем пример с учётом этого:

```
public sealed class Singleton
{
    private static Singleton instance;
    private static readonly object Locker = new object();

    private Singleton()
    {
    }

    public static Singleton GetInstance()
    {
        lock (Locker)
        {
            return instance ?? (instance = new Singleton());
        }
    }
}
```

Приведём ещё два варианта реализации шаблона Одиночка, подходящие для применения в многопоточных приложениях. Сначала используем тот факт, что при наличии в классе статического конструктора компилятор C# генерирует CIL-код, заставляющий откладывать инициализацию статических полей до первого использования класса (т.е. инициализация максимально отложена). Важно и то, что стандартный код инициализации статических полей потокобезопасен.

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
```

```
    static Singleton()
    {
    }

    private Singleton()
    {
    }

    public static Singleton GetInstance()
    {
        return instance;
    }
}
```

Следующий вариант реализации использует для отложенной потокобезопасной инициализации стандартный класс `System.Lazy<T>`, представленный в платформе .NET четвёртой версии.

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
                        new Lazy<Singleton>(() => new Singleton());

    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get { return lazy.Value; }
    }
}
```

3.3. Отложенная инициализация (Lazy initialization)

Шаблон Отложенная инициализация позволяет отсрочить действия, связанные с созданием объекта, до момента, когда непосредственно потребуется результат этих действий. Данный шаблон используется, если создание объекта связано с большими затратами ресурсов, или если есть вероятность, что объект или его часть не будут использованы. При использовании шаблона необходимо учитывать факт появления задержки при первом обращении к объекту или некоторым его методам.

Стандартный класс платформы .NET `System.Lazy<T>` служит для поддержки отложенной инициализации объектов. Данный класс содержит булево свойство для чтения `IsValueCreated` и свойство для чтения `Value` типа `T`. Использование `Lazy<T>` позволяет задержать создание объекта до первого обращения к свойству `Value`. Для создания объекта используется либо конструктор типа `T` без параметров, либо функция, передаваемая конструктору `Lazy<T>`.

3.4. Прототип (Prototype)

Шаблон Прототип описывает способ построения новых объектов путём клонирования существующих объектов. Обычно объект создаётся при помощи вызова конструктора. Если же используется шаблон Прототип, то клиент знает только об интерфейсе или базовом классе, содержащем метод клонирования. Реальный класс объекта клиенту неизвестен. Прототипы для клонирования могут использоваться произвольное количество раз, сами они при операции клонирования меняться не должны. Хотя существуют различные варианты дизайна данного шаблона, наиболее гибким является вариант с *менеджером прототипов*, содержащим индексированный список доступных прототипов.

При реализации шаблона Прототип для платформы .NET следует учитывать, что пользовательские объекты могут содержать другие объекты, а значит, требовать дополнительных усилий по получению полной копии. Ниже приводится пример кода, в котором для полного копирования используются возможности сериализации времени выполнения.

```
// тип T также должен быть сериализуемым
[Serializable]
public abstract class Prototype<T>
{
    // поверхностное копирование
    public virtual T Clone()
    {
        return (T) MemberwiseClone();
    }

    // глубокое копирование
    public virtual T DeepCopy()
    {
        using (var stream = new MemoryStream())
        {
            var context =
                 new StreamingContext(StreamingContextStates.Clone);
            var formatter = new BinaryFormatter {Context = context};
            formatter.Serialize(stream, this);
            stream.Position = 0;
            return (T) formatter.Deserialize(stream);
        }
    }
}

// конкретный класс, использующий описанные механизмы копирования
[Serializable]
public class Student : Prototype<Student>
{
    public Guid Id { get; set; }
    public string Name { get; set; }
}
```

```
// использование шаблона Прототип
var student = new Student {Name = "Alex"};
var clone = student.Clone();

// иногда после клонирования нужно переопределить
// часть свойств объекта
clone.Id = Guid.NewGuid();
```

3.5. Пул объектов (Object pool)

Пул объектов – это порождающий шаблон проектирования, который представляет собой набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул (при этом состояние объекта сбрасывается до начального). Шаблон применяется для повышения производительности, если объекты часто создаются и уничтожаются, а также, если создание и уничтожение объекта являются затратными операциями.

При практической реализации шаблона особого внимания заслуживает стратегия поведения при переполнении пула. В этом случае возможен один из трёх вариантов: расширение пула; отказ в создании объекта и аварийный останов; ожидание освобождения одного из объектов в пуле.

Примером реализации шаблона Пул объектов является существующий на платформе .NET пул потоков, для работы с которым используется класс `System.Threading.ThreadPool`.

3.6. Строитель (Builder)

Шаблон Строитель позволяет отделить процесс создания сложного объекта от его реализации. При этом результатом одних и тех же операций могут быть различные объекты. Данный шаблон используется в случае, если процесс создания объекта можно разделить на стадии (шаги). При этом конструирование должно обеспечивать возможность создавать разные объекты.

Шаблон Строитель включает двух участников процесса (рис. 10). *Строитель* (Builder) предоставляет методы для сборки частей объекта, при необходимости преобразовывает исходные данные в нужный вид, создаёт и выдаёт объект. *Распорядитель* (Director) определяет стратегию сборки: собирает данные и определяет порядок вызовов методов строителя. Задача распорядителя – *сокрытие* стратегии сборки. Это позволит, при необходимости, модифицировать или даже полностью менять её, не затрагивая остальной код.

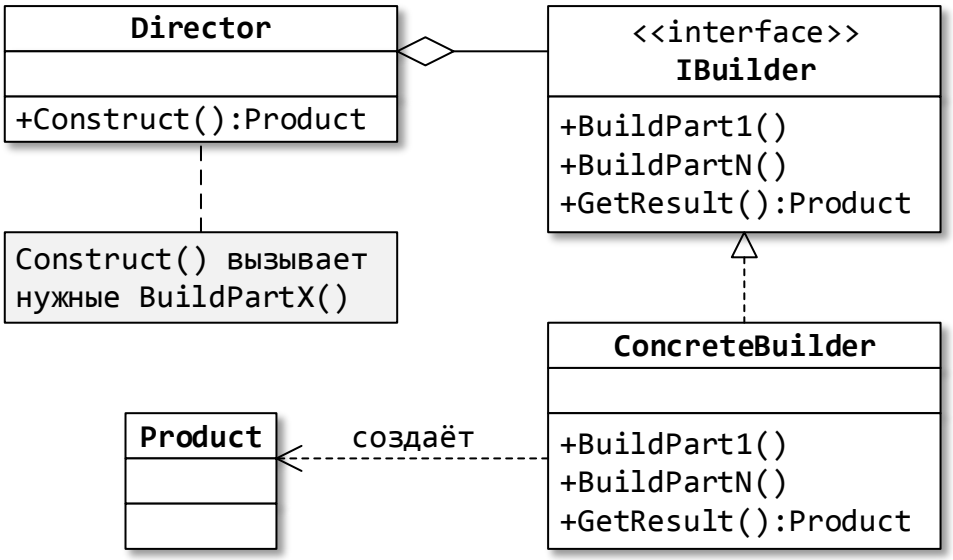

Рис. 10. Дизайн шаблона Строитель.

Разберём пример использования шаблона Строитель. Пусть необходимо генерировать страницу на сайте. Определим шаги конструирования страницы: создаём шапку (Header), добавляем элементы меню (MenuItems), выводим публикации (Post) и завершаем страницу кодом подвала (Footer). Эти четыре шага и метод получения готовой страницы будут определять интерфейс Строителя.

```
public interface IPageBuilder
{
    void BuildHeader(string header);
    void BuildMenu(string menuItems);
    void BuildPost(string post);
    void BuildFooter(string footer);
    string GetResult();
}
```

Два конкретных класса отвечают за построение «нормальной» страницы и версии страницы для печати (эта страница содержит только публикации).

```
public class PageBuilder : IPageBuilder
{
    private string _page = string.Empty;

    public void BuildHeader(string header)
    {
        _page += header + Environment.NewLine;
    }

    public void BuildMenu(string menuItems)
    {
        _page += menuItems + Environment.NewLine;
    }
```

```
        public void BuildPost(string post)
        {
            _page += post + Environment.NewLine;
        }

        public void BuildFooter(string footer)
        {
            _page += footer + Environment.NewLine;
        }

        public string GetResult()
        {
            return _page;
        }
    }

    public class PrintPageBuilder : IPageBuilder
    {
        private string _page = string.Empty;

        public void BuildHeader(string header)
        {
        }

        public void BuildMenu(string menuItems)
        {
        }

        public void BuildPost(string post)
        {
            _page += post + Environment.NewLine;
        }

        public void BuildFooter(string footer)
        {
        }

        public string GetResult()
        {
            return _page;
        }
    }
```

Приступим к реализации распорядителя. Ограничимся одной стратегией создания страницы, а значит, одним классом распорядителя.

```
    public class PageDirector
    {
        private readonly IPageBuilder _builder;
```

```
    public PageDirector(IPageBuilder builder)
    {
        _builder = builder;
    }

    public string BuildPage(int pageId)
    {
        _builder.BuildHeader(GetHeader(pageId));
        _builder.BuildMenu(GetMenuItems(pageId));
        foreach (var post in GetPosts(pageId))
        {
            _builder.BuildPost(post);
        }
        _builder.BuildFooter(GetFooter(pageId));
        return _builder.GetResult();
    }

    private string GetHeader(int pageId)
    {
        // реализация упрощена
        return "Header of page " + pageId;
    }

    private string GetMenuItems(int pageId)
    {
        // реализация упрощена
        return "Menu";
    }

    private IEnumerable<string> GetPosts(int pageId)
    {
        // реализация упрощена
        return new List<string> {"Post 1", "Post 2"};
    }

    private string GetFooter(int pageId)
    {
        // реализация упрощена
        return "Footer of page " + pageId;
    }
}
```

Теперь всё готово к использованию:

```
var pageBuilder = new PageBuilder();
var pageDirector = new PageDirector(pageBuilder);

var page = pageDirector.BuildPage(123);
```

3.7. Фабричный метод (Factory method)

Фабричный метод занимается созданием объектов. Каждый такой объект принадлежит некоему классу, однако все эти классы либо имеют общего предка, либо реализуют общий интерфейс. Фабричный метод сам решает, какой конкретный класс нужно использовать для создания очередного объекта. Решение принимается либо на основе информации, предоставленной клиентом, либо на основе внутреннего состояния метода.

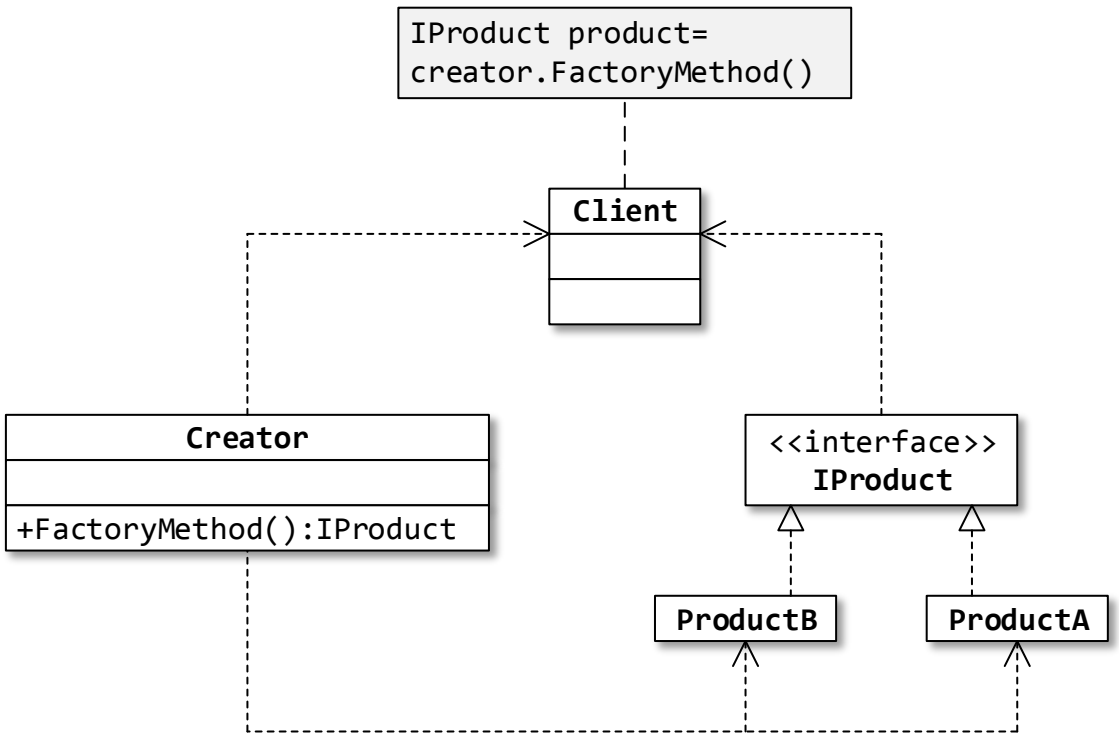

Рис. 11. Диаграмма шаблона Фабричный метод.

Для иллюстрации шаблона Фабричный метод представим магазин, торгующий определённым видом фруктов. Магазин (выступающий в роли клиента) работает с поставщиком (фабричный метод), который в зависимости от времени года импортирует фрукты (объекты) из разных стран (разные классы объектов). Детали операций импорта от клиента скрыты. Далее приведён код, использующий шаблон Фабричный метод и соответствующий описанному выше примеру.

```
public interface IProduct
{
    string ShipFrom();
}

public class ProductFromAfrica : IProduct
{
    public string ShipFrom()
    {
        return "from South Africa";
    }
}
```

```
public class ProductFromSpain : IProduct
{
    public string ShipFrom()
    {
        return "from Spain";
    }
}

public class DefaultProduct : IProduct
{
    public string ShipFrom()
    {
        return "not available";
    }
}

public class Creator
{
    public IProduct FactoryMethod(int month)
    {
        if (month >= 4 && month <= 11)
        {
            return new ProductFromAfrica();
        }
        if (month == 1 || month == 2 || month == 12)
        {
            return new ProductFromSpain();
        }
        return new DefaultProduct();
    }
}

public class FactoryMethodExample
{
    public static void Main()
    {
        var creator = new Creator();
        IProduct product;
        for (var i = 1; i <= 12; i++)
        {
            product = creator.FactoryMethod(i);
            System.Console.WriteLine("Avocados " +
                                     product.ShipFrom());
        }
    }
}
```

4. Шаблоны поведения

В параграфе рассматриваются избранные шаблоны поведения. За основу взят набор шаблонов из книги «Design Patterns», за исключением шаблонов Интерпретатор (Interpreter) и Хранитель (Memento). Дополнительно рассматривается шаблон Нулевой объект (Null object). Шаблоны упорядочены согласно их русскому названию.

4.1. Итератор (Iterator)

Шаблон Итератор обеспечивает последовательный доступ ко всем элементам коллекции, не раскрывая при этом её внутренней реализации. Задача итератора – упростить обход и сделать его однообразным для коллекций различных типов. Причём реализация шаблона может находиться как в объекте, представляющим коллекции, так и отдельно от него.

В шаблоне Итератор задействованы следующие участники. *Интерфейс итератора* определяет методы для доступа к элементам коллекций. *Итератор* – класс, реализующий интерфейс итератора для конкретной коллекции. *Интерфейс составного объекта* (коллекции) задаёт способ получения итератора клиентом. Наконец, *составной объект* – это реализация интерфейса коллекции.

Обычно интерфейс итератора состоит из следующих операций:

`Reset()` – устанавливает первый элемент коллекции в качестве текущего;

`MoveNext()` – переход на следующий элемент;

`IsDone` – проверяет, есть ли ещё элементы в коллекции;

`Current` – возвращает текущий элемент.

Этот список можно изменять. Например, объединить `MoveNext()` и `IsDone` в один метод. Или добавить метод `Skip()`, позволяющий пропустить заданное число элементов. В некоторых случаях может быть полезно свойство, содержащее ссылку на предыдущий элемент.

Разберём пример реализации шаблона Итератор. Вначале опишем интерфейс итератора:

```
public interface IIterator<out T>
{
    void Reset();
    void MoveNext();
    bool IsDone { get; }
    T Current { get; }
}
```

Далее представим интерфейс коллекции:

```
public interface IContainer<out T>
{
    IIterator<T> CreateIterator();
}
```

Сконструируем коллекцию, реализующую интерфейс `IContainer<T>`. При помощи вложенного класса реализуем итератор коллекции:

```
public class ListContainer : IContainer<int>
{
    private readonly int[] _list;

    public class ForwardIterator : IIterator<int>
    {
        private readonly int[] _iteratorList;
        private int _position;

        public ForwardIterator(int[] list)
        {
            _iteratorList = new int[list.Length];
            list.CopyTo(_iteratorList, 0);
        }

        public void Reset()
        {
            _position = 0;
        }

        public void MoveNext()
        {
            _position++;
        }

        public bool IsDone
        {
            get { return _position > _iteratorList.Length - 1; }
        }

        public int Current
        {
            get { return _iteratorList[_position]; }
        }
    }

    public ListContainer(params int[] data)
    {
        _list = new int[data.Length];
        data.CopyTo(_list, 0);
    }

    public IIterator<int> CreateIterator()
    {
        return new ForwardIterator(_list);
    }
}
```

Применить описанные классы и интерфейсы можно следующим образом:

```
IContainer<int> list = new ListContainer(2, 3, 5, 7);
var iterator = list.CreateIterator();
while (!iterator.IsDone)
{
    Console.WriteLine(iterator.Current);
    iterator.MoveNext();
}
```

Поддержка итераторов реализована на платформе .NET и в синтаксисе языка C# при помощи интерфейсов `IEnumerable` и `IEnumerator`, операторов `yield` и `foreach`. При этом `IEnumerator` играет роль интерфейса итератора, а `IEnumerable` выступает в качестве интерфейса составного объекта.

4.2. Команда (Command)

Шаблон Команда обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве аргумента методам, а также возвращать её в виде результата, как и любой другой объект.

Дизайн шаблона Команда показан на рис. 12. Все *команды* – это объекты классов, реализующих общий интерфейс или унаследованных от общего предка. Конкретная команда имеет некое состояние (контекст выполнения). Объект класса `Receiver` – это то, *над чем* выполняется команда. Команда изменяет состояние этого объекта. Класс `Invoker` инкапсулирует команду (или команды) и может выполнять команды по необходимости.

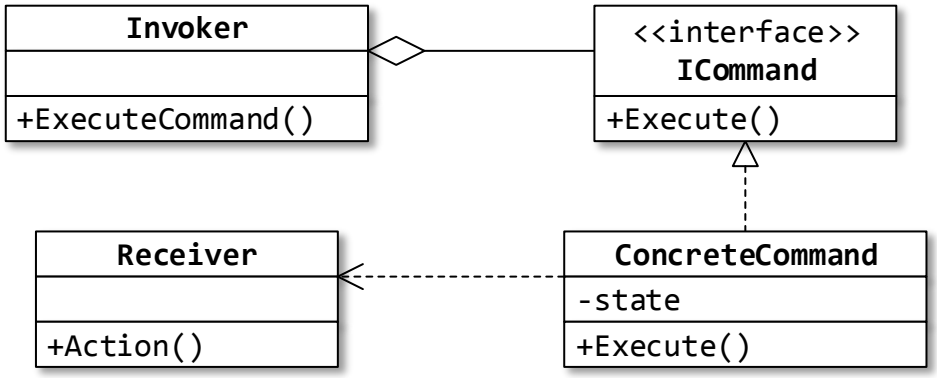

Рис. 12. Дизайн шаблона Команда.

Рассмотрим пример реализации шаблона Команда. Пусть класс `Calculator` обеспечивает выполнение арифметических операций с целыми числами.

```
// хранит целочисленное значение,
// производит с ним арифметические операции
public class Calculator
{
    private int _value;

    public void Operation(char @operator, int operand)
```

```
    {
        switch (@operator)
        {
            case '+':
                _value += operand;
                break;
            case '-':
                _value -= operand;
                break;
            case '*':
                _value *= operand;
                break;
            case '/':
                _value /= operand;
                break;
        }
        Console.WriteLine("Current value = {0,1}", _value);
    }
}
```

Интерфейс команды обеспечивает операции выполнения и отмены.

```
public interface ICommand
{
    void Execute();
    void Undo();
}
```

Создадим класс для описания команды.

```
public class CalculatorCommand : ICommand
{
    private readonly Calculator _calculator;
    private readonly char _operator;
    private readonly int _operand;

    public CalculatorCommand(Calculator calculator, char @operator,
                                                    int operand)
    {
        _calculator = calculator;
        _operator = @operator;
        _operand = operand;
    }

    public void Execute()
    {
        _calculator.Operation(_operator, _operand);
    }

    public void Undo()
```

```
        {
            _calculator.Operation(GetOppositeOperator(_operator),
                                  _operand);
        }

        private static char GetOppositeOperator(char @operator)
        {
            const string operators = "+-*/";
            const string opposite = "-+/*";

            var pos = operators.IndexOf(@operator);
            if (pos == -1)
            {
                throw new ArgumentException("@operator");
            }
            return opposite[pos];
        }
    }
```

Класс `User` поддерживает список команд для реализации функций отмены и повторного выполнения.

```
    public class User
    {
        private readonly Calculator _calculator = new Calculator();
        private readonly List<ICommand> _commands =
                                                  new List<ICommand>();
        private int _current;

        public void Compute(char @operator, int operand)
        {
             var command = new CalculatorCommand(_calculator, @operator,
                                                 operand);
            command.Execute();
            _commands.Add(command);
            _current++;
        }

        public void Undo(int levels)
        {
            for (var i = 0; i < levels; i++)
            {
                if (_current > 0)
                {
                    _commands[--_current].Undo();
                }
            }
        }
```

```
    public void Redo(int levels)
    {
        for (var i = 0; i < levels; i++)
        {
            if (_current < _commands.Count - 1)
            {
                _commands[_current++].Execute();
            }
        }
    }
}
```

Использовать разработанные классы можно следующим образом:

```
// создаём объект для вычислений
var user = new User();

// выполняем несколько команд вычисления
user.Compute('+', 100);
user.Compute('-', 50);
user.Compute('*', 10);
user.Compute('/', 2);

// отменяем четыре команды
user.Undo(4);

// повторно выполняем три команды
user.Redo(3);
```

4.3. Наблюдатель (Observer)

Шаблон Наблюдатель определяет отношение между объектами таким образом, что при изменении состояния одного объекта все зависящие от него объекты оповещаются об этом. Основное назначение шаблона Наблюдатель – реализация системы работы с событиями.

Разберём устройство шаблона Наблюдатель (рис. 13). Объект класса `Subject` – *наблюдаемый объект* – имеет некоторое состояние, изменение которого предполагается отслеживать. Для этого в классе `Subject` описывается коллекция объектов-наблюдателей и методы добавления и удаления элементов коллекции. Каждый объект-*наблюдатель* реализует интерфейс `IObserver`. При изменении состояния наблюдаемого объекта вызывается метод `NotifyObservers()`. Этот метод итерируется по коллекции наблюдателей и вызывает у каждого из них метод `HandleEvent()` (возможно, передавая в качестве аргумента состояние наблюдаемого объекта).

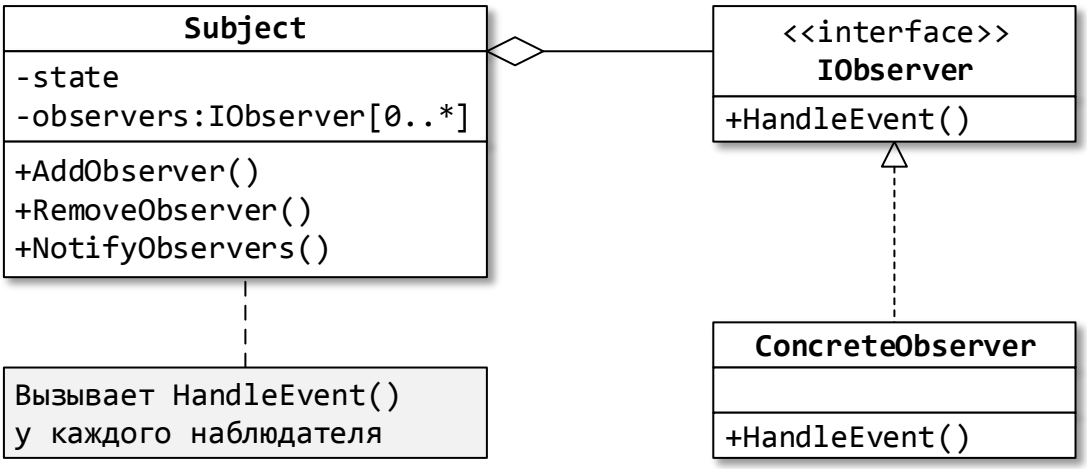

Рис. 13. Дизайн шаблона Наблюдатель.

Далее приводится пример кода, демонстрирующего реализацию шаблона Наблюдатель. Заметим, что в примере намеренно не используются события C#.

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

// аналог Subject
public class Stock
{
    private int _price;
    private IList<IInvestor> _investors = new List<IInvestor>();

    public int Price
    {
        get { return _price; }
        set
        {
            if (_price != value)
            {
                _price = value;
                Notify();
            }
        }
    }

    public Stock(int price)
    {
        _price = price;
    }

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

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

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

// интерфейс, аналогичный IObserver
public interface IInvestor
{
    void Update(Stock stock);
}

// конкретный наблюдатель
public class Investor : IInvestor
{
    public string Name { get; private set; }

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

    public void Update(Stock stock)
    {
        Console.WriteLine("Notified {0} of change to {1}",
                          Name, stock.Price);
    }
}

public class ObserverExample
{
    public static void Main()
    {
        var ibm = new Stock(120);
        ibm.Attach(new Investor("Soros"));
        ibm.Attach(new Investor("Berkshire"));
        ibm.Price = 120;
        ibm.Price = 121;
        ibm.Price = 125;
    }
}
```

Для унификации работы с шаблоном Наблюдатель платформа .NET предлагает два интерфейса. Интерфейс `System.IObserver<T>` реализуется наблюдателем. Этот интерфейс содержит методы, уведомляющие о новом событии, об ошибке в наблюдаемом объекте и о прекращении генерации событий.

```
public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

Интерфейс `System.IObservable<T>` реализуется наблюдаемым объектом. В интерфейсе описан единственный метод `Subscribe()` для добавления наблюдателя. Чтобы удалить наблюдателя, нужно вызвать метод `Dispose()` у объекта, возвращённого методом `Subscribe()`.

```
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
```

Изменим пример, представленный выше, используя типовые интерфейсы.

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

public class Stock : IObservable<int>
{
    private int _price;
    private IList<IInvestor> _investors = new List<IInvestor>();

    public int Price
    {
        get { return _price; }
        set
        {
            if (_price != value)
            {
                _price = value;
                Notify();
            }
        }
    }

    public Stock(int price)
    {
        _price = price;
    }
```

```
    public IDisposable Subscribe(IObserver<int> investor)
    {
        _investors.Add(investor);
        return new Unsubscriber(_investors, investor);
    }

    public void Notify()
    {
        foreach (var investor in _investors)
            investor.OnNext(_price);
    }

    private class Unsubscriber : IDisposable
    {
        private readonly IObserver<int> _investor;
        private readonly IList<IObserver<int>> _investors;

        public Unsubscriber(IList<IObserver<int>> investors,
                            IObserver<int> investor)
        {
            _investors = investors;
            _investor = investor;
        }

        public void Dispose()
        {
            _investors.Remove(_investor);
        }
    }
}

public class Investor : IObserver<int>
{
    public string Name { get; private set; }

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

    public void OnNext(int value)
    {
        Console.WriteLine("Notified of change to {0}", value);
    }

    public void OnError(Exception error) { }

    public void OnCompleted() { }
}
```

```
public class ObserverExample
{
    public static void Main()
    {
        var ibm = new Stock(120);
        ibm.Subscribe(new Investor("Soros"));
        ibm.Subscribe(new Investor("Berkshire"));
        ibm.Price = 120;
        ibm.Price = 121;
        ibm.Price = 125;
    }
}
```

4.4. Нулевой объект (Null object)

Нулевой объект – это объект с заданным нейтральным поведением, использование которого позволяет заменить условные операторы полиморфизмом.

Рассмотрим следующий пример. Пусть имеется система классов для протоколирования и метод-фабрика, возвращающий в зависимости от аргумента объект заданного класса.

```
public abstract class BaseLogger
{
    public abstract void Write(string logEntry);
}

public class FileLogger : BaseLogger
{
    public override void Write(string logEntry)
    {
        Console.WriteLine(logEntry + "is written into the file");
    }
}

public class EmailLogger : BaseLogger
{
    public override void Write(string logEntry)
    {
        Console.WriteLine(logEntry + "is sent by email");
    }
}

public static class LoggerFactory
{
    public static BaseLogger GetLogger(string name)
    {
        switch (name)
        {
            case "file":
                return new FileLogger();
```

```
            case "email":
                return new EmailLogger();
            default:
                return null;
        }
    }
}
```

Использование кода в указанном виде приводит к условным операторам, обеспечивающим корректное поведение в том случае, если метод `GetLogger()` вернул значение `null`.

```
BaseLogger logger = LoggerFactory.GetLogger("abc");
if (logger != null)
{
    logger.Write("message");
}
else
{
    Console.WriteLine("Cannot write in the log");
}
```

Устранить условные операторы поможет введение специального класса с поведением по умолчанию:

```
public class DefaultlLogger : BaseLogger
{
    public override void Write(string logEntry)
    {
        Console.WriteLine("Cannot write in the log");
    }
}

public static class LoggerFactory
{
    public static BaseLogger GetLogger(string name)
    {
        switch (name)
        {
            case "file":
                return new FileLogger();
            case "email":
                return new EmailLogger();
            default:
                return new DefaultlLogger();
        }
    }
}
```

4.5. Посетитель (Visitor)

Шаблон Посетитель служит для выполнения операций над всеми объектами, объединёнными в некоторую структуру. При этом выполняемые операции не являться частью объектов структуры.

Часто в программах встречаются сложные структуры, представляющие собой дерево или граф и состоящие из разнотипных узлов. При этом имеется необходимость обрабатывать все узлы графа или дерева. Очевидное решение – добавить в базовый класс узла виртуальный метод, перекрываемый в наследниках для выполнения нужного действия. Код, который приведён ниже, демонстрирует описанный подход.

```
// базовый класс для узла
public abstract class BaseNode
{
    public string Name { get; private set; }

    protected BaseNode(string name)
    {
        Name = name;
    }

    public abstract void Print(TextWriter writer);
}

// простой узел
public class SimpleNode : BaseNode
{
    public SimpleNode(string name) : base(name)
    {
    }

    public override void Print(TextWriter writer)
    {
        writer.WriteLine(Name + " : Simple");
    }
}

// узел с дочерними узлами
public class CompositeNode : BaseNode
{
    public IList<BaseNode> Nodes { get; private set; }

    public CompositeNode(string name, params BaseNode[] nodes) :
                         base(name)
    {
        Nodes = Array.AsReadOnly(nodes);
    }
```

```
    public override void Print(TextWriter writer)
    {
        writer.WriteLine(Name + " : Composite");
        foreach (var node in Nodes)
        {
            node.Print(writer);
        }
    }
}

// использование классов - построение и печать иерархии
var tree = new CompositeNode("Root",
               new CompositeNode("Level_1",
                   new SimpleNode("Leaf_1_1"),
                   new SimpleNode("Leaf_1_2")),
               new SimpleNode("Leaf_2"));
tree.Print(Console.Out);
```

Решение, применяемое в примере, имеет серьёзный недостаток: в нем структура данных оказывается увязанной с обрабатывающими её алгоритмами. Если нам понадобится алгоритм, отличный от реализованного, то придётся добавлять ещё один виртуальный метод. Ещё хуже, если классы, описывающие дерево, содержатся в недоступном для модификации коде.

Основная идея шаблона Посетитель состоит в том, что каждый элемент объектной структуры содержит виртуальный метод `Accept()`, который принимает на вход в качестве аргумента специальный объект – *посетитель*, реализующий заранее известный интерфейс. Этот интерфейс содержит по одному методу `Visit()` для каждого типа узла. Метод `Accept()` в каждом узле должен вызывать методы `Visit()` для осуществления навигации по структуре.

```
// интерфейс посетителя
public interface IVisitor
{
    void Visit(SimpleNode node);
    void Visit(CompositeNode node);
}

// конкретный посетитель
public class PrintVisitor : IVisitor
{
    private readonly TextWriter _writer;

    public PrintVisitor(TextWriter writer)
    {
        _writer = writer;
    }

    public void Visit(SimpleNode node)
    {
```

```
            _writer.WriteLine(node.Name + " : Simple");
        }

        public void Visit(CompositeNode node)
        {
            _writer.WriteLine(node.Name + " : Composite");
            foreach (var node in node.Nodes)
                node.Accept(this);
        }
    }

    public abstract class BaseNode
    {
        public string Name { get; private set; }

        protected BaseNode(string name)
        {
            Name = name;
        }

        public abstract void Accept(IVisitor visitor);
    }

    public class SimpleNode : BaseNode
    {
        public SimpleNode(string name) : base(name)
        {
        }

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

    public class CompositeNode : BaseNode
    {
        public IList<BaseNode> Nodes { get; private set; }

        public CompositeNode(string name, params BaseNode[] nodes) :
                             base(name)
        {
            Nodes = Array.AsReadOnly(nodes);
        }

        public override void Accept(IVisitor visitor)
        {
            visitor.Visit(this);
        }
    }
```

Приём, использованный в примере, носит название *двойная диспетчеризация*. Обычные виртуальные методы используют *одинарную диспетчеризацию* – то, какая операция будет выполнена, зависит от имени метода и типа объекта-получателя. В двойной диспетчеризации шаблона Посетитель вызываемая операция зависит от имени метода, типа посещаемого элемента и типа конкретного посетителя[1].

4.6. Посредник (Mediator)

Шаблон Посредник служит для обеспечения коммуникации между объектами. Этот шаблон также инкапсулирует протокол, которому должна удовлетворять процедура коммуникации.

Приведём следующую иллюстрацию возможного применения шаблона Посредник. Пусть имеется электронная форма для ввода анкетных данных. Эта форма содержит различные элементы управления – объекты определённых классов. Сценарий работы с формой предполагает изменение состояния одних объектов в зависимости от состояния других. Например, в зависимости от значения поля «Пол» меняется доступность переключателя «Служили ли в армии?». Тривиальное решение предполагает, что каждый из объектов, соответствующих определённому элементу формы, при изменении собственных данных будет изменять состояние связанных с ним объектов. Это приводит к росту зависимостей между объектами и увеличению сложности системы. Использование шаблона Посредник подразумевает выделение отдельного объекта, «дирижирующего» поведением элементов управления в зависимости от их текущего состояния.

Дизайн шаблона Посредник предполагает наличие двух выделенных классов, использующих сообщения для взаимного обмена информацией: `Colleague` (*коллега*) и `Mediator` (*посредник*). Объект `Colleague` регистрирует объект `Mediator` и сохраняет его в своём внутреннем поле. В свою очередь, посредник поддерживает список зарегистрировавших его объектов. Как только один из объектов `Colleague` вызывает свой метод `Send()`, посредник вызывает у остальных зарегистрированных объектов метод `Receive()`.

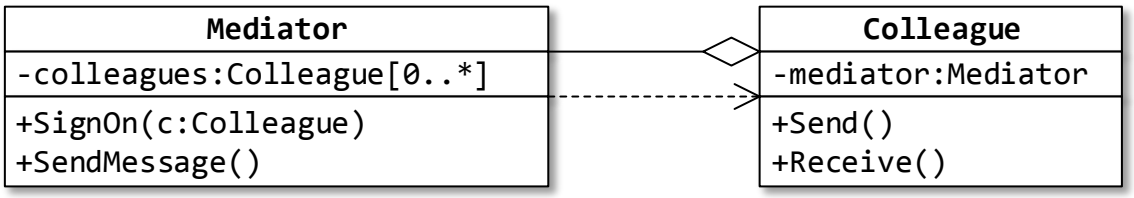

Рис. 14. Дизайн шаблона Посредник.

Далее представлен код, демонстрирующий использование шаблона.

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

[1] В статье http://code.logos.com/blog/2010/03/the_visitor_pattern_and_dynamic_in_c_4.html рассматривается использование типа `dynamic` для реализации варианта шаблона Посетитель.

```
public class Mediator
{
    private ISet<Colleague> _colleagues = new HashSet<Colleague>();

    public void SignOn(Colleague colleague)
    {
        _colleagues.Add(colleague);
    }

    public void Send(string message, string from)
    {
        foreach (var colleague in _colleagues)
        {
            colleague.Receive(message, from);
        }
    }
}

public class Colleague
{
    private readonly Mediator _mediator;
    private readonly string _name;

    public Colleague(Mediator mediator, string name)
    {
        _mediator = mediator;
        _name = name;
        mediator.SignOn(this);
    }

    public virtual void Receive(string message, string from)
    {
        Console.WriteLine("{0} received from {1}: {2}",
                          _name, from, message);
    }

    public void Send(string message)
    {
        Console.WriteLine("Send (from {0}): {1}", _name, message);
        _mediator.Send(message, _name);
    }
}

public class MediatorExample
{
    public static void Main()
    {
        var mediator = new Mediator();
        var john = new Colleague(mediator, "John");
        var lucy = new Colleague(mediator, "Lucy");
```

```
            var david = new Colleague(mediator, "David");
            john.Send("Meeting on Tuesday, please all ack");
            david.Send("Ack");
            john.Send("Still awaiting some acks");
            lucy.Send("Ack");
            john.Send("Thanks all");
        }
    }
```

4.7. Состояние (State)

При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста[1].

При помощи шаблона Состояние можно эффективно реализовать такую абстракцию как *конечный автомат*. Сделаем небольшое отступление и разберём понятие конечного автомата подробнее. Любую функцию можно рассматривать как некий преобразователь информации. Аргумент функции – *входной сигнал* – преобразуется, согласно определённому правилу, в результат функции – *выходной сигнал*. Функциональные преобразователи обладают важным свойством – их поведение не зависит от предыстории. В реальности, однако, имеется достаточно примеров преобразователей, реакция которых зависит не только от входа в данный момент, но и от того, что было на входе раньше, от *входной истории*. Такие преобразователи называются *автоматами*. Так как количество разных входных историй потенциально бесконечно, на множестве предысторий вводится отношение эквивалентности, и один класс эквивалентности предысторий называется *состоянием автомата*. Состояние автомата меняется только при получении очередного входного сигнала. При этом автомат не только выдаёт информацию на выход как функцию входного сигнала и текущего состояния, но и меняет своё состояние, поскольку входной сигнал изменяет предысторию.

Рассмотрим пример конечного автомата. Опишем поведение отца, отправившего сына в школу. Сын получает хорошие отметки («десятки») и плохие отметки («двойки»). Отец не хочет хвататься за ремень каждый раз, как только сын получит очередную двойку, и выбирает более тонкую тактику воспитания. Чтобы описать модель поведения отца, используем граф, в котором вершины соответствуют состояниям, а дуга x/y из вершины s в вершину q проводится, когда автомат из состояния s под воздействием входного сигнала x переходит в состояние q с выходной реакцией y. Граф автомата, моделирующего поведение родителя, представлен на рис. 15.

[1] Шаблон Состояние часто называют *динамической версией* шаблона Стратегия.

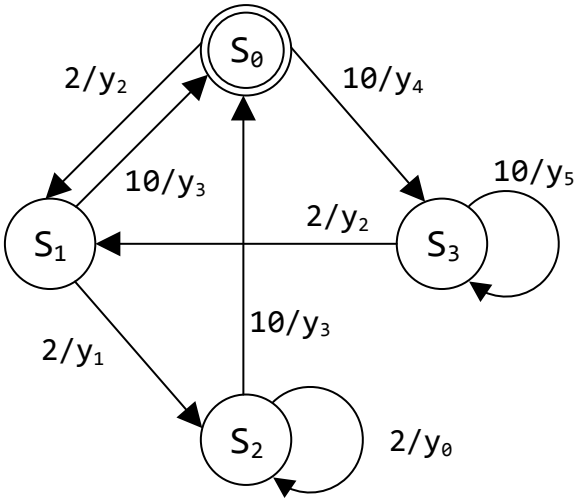

Рис. 15. Автомат, описывающий поведение отца.

Этот автомат имеет четыре состояния $\{s_0, s_1, s_2, s_3\}$ и два входных сигнала $\{2,10\}$ (оценки, полученные сыном в школе). Стартуя из начального состояния s_0 (помечено особо), автомат под воздействием входных сигналов переходит из одного состояния в другое и выдаёт выходные сигналы – реакции на входы. Выходы автомата будем интерпретировать как действия родителя так: y_0 – «брать ремень»; y_1 – «ругать»; y_2 – «успокаивать»; y_3 – «надеяться»; y_4 – «радоваться»; y_5 – «ликовать».

При наивной программной реализации конечного автомата порождаются наборы конструкций `switch-case`, которые, как правило, вложены в друг друга. Использование шаблона Состояние позволяет упростить код. Все состояния конечного автомата описываются отдельными классами, которые обладают набором виртуальных методов, соответствующих входным сигналам. Получение контекстом очередного входного сигнала означает вызов метода того объекта, экземпляр которого находится в поле `State`. При этом сам вызываемый метод может поместить в `State` другой объект-состояние (переключить состояние).

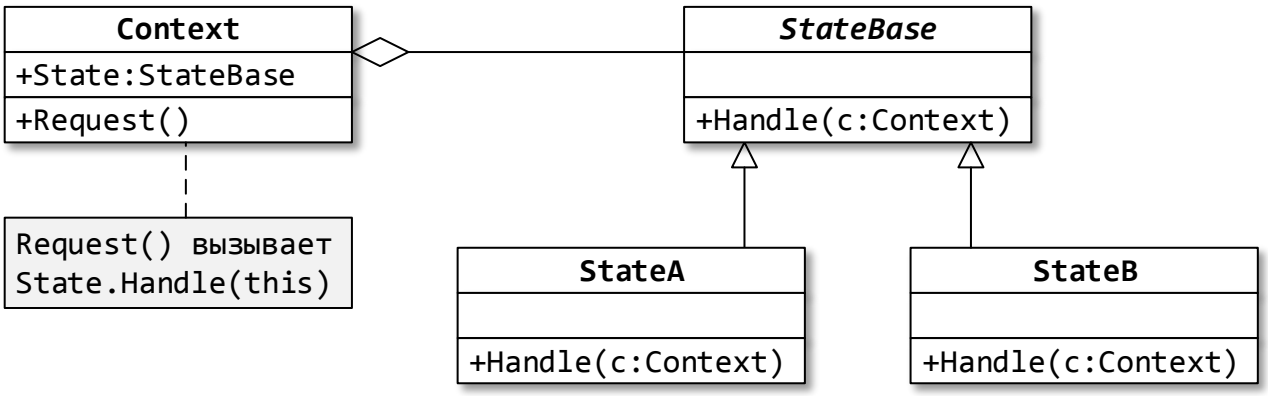

Рис. 16. Дизайн шаблона Состояние.

При практической реализации шаблона Состояние отдельные объекты-состояния могут быть оформлены с применением шаблона Одиночка. Далее приведён код, использующий именно этот подход и соответствующий примеру конечного автомата, рассмотренного выше.

```
using System;

public class SmartParent
{
    public AbstractState State = State0.Instance;

    public void HandleTwo()
    {
        State.HandleTwo(this);
    }

    public void HandleTen()
    {
        State.HandleTen(this);
    }
}

public abstract class AbstractState
{
    public abstract void HandleTwo(SmartParent parent);
    public abstract void HandleTen(SmartParent parent);
}


// здесь используется короткая, но не совсем корректная
// реализация шаблона Одиночка (см. соответствующий шаблон)
public class State0 : AbstractState
{
    public static readonly State0 Instance = new State0();

    public override void HandleTwo(SmartParent parent)
    {
        Console.WriteLine("Успокаивать");
        parent.State = State1.Instance;
    }

    public override void HandleTen(SmartParent parent)
    {
        Console.WriteLine("Радоваться");
        parent.State = State3.Instance;
    }
}
```

```
public class State1 : AbstractState
{
    public static readonly State1 Instance = new State1();

    public override void HandleTwo(SmartParent parent)
    {
        Console.WriteLine("Ругать");
        parent.State = State2.Instance;
    }

    public override void HandleTen(SmartParent parent)
    {
        Console.WriteLine("Надеяться");
        parent.State = State0.Instance;
    }
}

public class State2 : AbstractState
{
    public static readonly State2 Instance = new State2();

    public override void HandleTwo(SmartParent parent)
    {
        Console.WriteLine("Брать ремень");
        parent.State = State2.Instance;
    }

    public override void HandleTen(SmartParent parent)
    {
        Console.WriteLine("Надеяться");
        parent.State = State0.Instance;
    }
}

public class State3 : AbstractState
{
    public static readonly State3 Instance = new State3();

    public override void HandleTwo(SmartParent parent)
    {
        Console.WriteLine("Успокаивать");
        parent.State = State1.Instance;
    }

    public override void HandleTen(SmartParent parent)
    {
        Console.WriteLine("Ликовать");
        parent.State = State3.Instance;
    }
}
```

4.8. Стратегия (Strategy)

При помощи шаблона Стратегия из клиента выделяется алгоритм, который затем инкапсулируется в типах, наследуемых от общего класса (реализующих общий интерфейс). Это позволяет клиенту выбирать нужный алгоритм путём создания объектов необходимых типов. Кроме этого, шаблон допускает изменение набора доступных алгоритмов со временем.

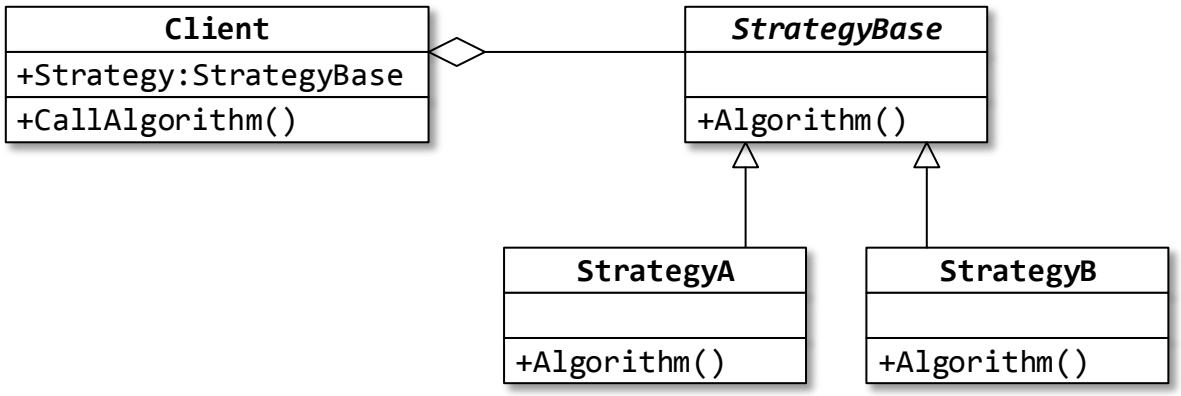

Рис. 17. Диаграмма шаблона Стратегия.

Предположим, что требуется разработать программу, которая показывает календарь. Одно из требований к программе – она должна отображать праздники, отмечаемые различными нациями и религиозными группами. Это требование можно выполнить, помещая логику генерирования каждого набора праздников в отдельный класс. Основная программа будет выбирать необходимый класс из набора, исходя, например, из действий пользователя (или конфигурационных настроек). Ниже приведён пример кода, реализующий поставленную задачу при помощи шаблона Стратегия.

```
using System.Collections.Generic;

public interface IHolidaySet
{
    List<string> GetHolidays();
}

public class USAHolidays : IHolidaySet
{
    public List<string> GetHolidays()
    {
        return new List<string>
            {
                "01.01.13",
                "21.01.13",
                "18.02.13",
                "27.05.13",
                "04.07.13",
                "02.09.13",
                "14.10.13",
```

```
                "11.11.13",
                "28.11.13",
                "25.12.13"
            };
    }
}

public class RussiaHolidays : IHolidaySet
{
    public List<string> GetHolidays()
    {
        return new List<string>
            {
                "01.01.13",
                "07.01.13",
                "23.02.13",
                "08.03.13",
                "11.03.13",
                "01.05.13",
                "09.05.13",
                "12.06.13",
                "04.11.13"
            };
    }
}

public class Client
{
    private readonly IHolidaySet _holidaySetStrategy;

    public Client(IHolidaySet strategy)
    {
        _holidaySetStrategy = strategy;
    }

    public bool CheckForHoliday(string date)
    {
        return _holidaySetStrategy.GetHolidays().Contains(date);
    }
}

public class StrategyExample
{
    public static void Main()
    {
        var client = new Client(new USAHolidays());
        var result = client.CheckForHoliday("04.07.13");
    }
}
```

4.9. Цепочка обязанностей (Chain of responsibility)

Шаблон Цепочка обязанностей работает со списком объектов, называемых *обработчиками*. Каждый из обработчиков имеет естественные ограничения на множество запросов, которые он в состоянии поддержать. Если текущий обработчик не может поддержать запрос, он передаёт его следующему обработчику в списке. Так продолжается, пока запрос не будет обработан, или пока список не закончится.

Дизайн шаблона Цепочка обязанностей достаточно прост (рис. 18). Каждый из обработчиков принадлежит к одному из классов, имеющих общего предка. Кроме метода (или методов) для обработки запроса, каждый обработчик имеет поле, указывающее на следующий обработчик в цепочке.

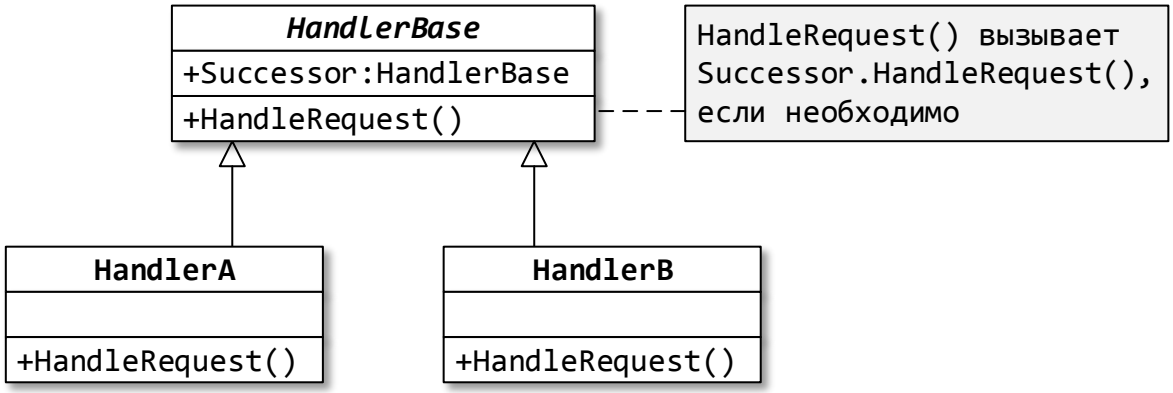

Рис. 18. Диаграмма шаблона Цепочка обязанностей.

Ниже приведён код, иллюстрирующий применение шаблона Цепочка обязанностей.

```
public class Handler
{
    public int Limit { get; private set; }
    public Handler Next { get; private set; }

    public Handler(int limit, Handler handler)
    {
        Limit = limit;
        Next = handler;
    }

    public string HandleRequest(int data)
    {
        if (data < Limit)
        {
            return string.Format("Request for {0} handled", data);
        }
        if (Next != null)
        {
            return Next.HandleRequest(data);
        }
```

```
            return "Request handled BY DEFAULT";
        }
    }

    public class ChainOfResponsibilityExample
    {
        public static void Main()
        {
            Handler start = null;
            for (var i = 5; i > 0; i--)
            {
                start = new Handler(i*1000, start);
            }
            int[] data = {50, 2000, 1500, 10000, 175, 4500};
            foreach (var i in data)
            {
                System.Console.WriteLine(start.HandleRequest(i));
            }
        }
    }
```

4.10. Шаблонный метод (Template method)

Предположим, что программная логика некоего алгоритма представлена в виде набора вызовов методов. При использовании *шаблонного метода* создаётся абстрактный класс, который реализует только часть методов программной логики, оставляя детали реализации остальных методов своим потомкам. Благодаря этому общая структура алгоритма остаётся неизменной, в то время как некоторые конкретные шаги могут изменяться.

В качестве иллюстрации применения шаблонного метода рассмотрим класс с методом сортировки. Пусть этот метод вызывает отдельный метод сравнения элементов сортируемого набора. Классы-наследники переопределяют метод сравнения, позволяя, например, реализовать сортировку по убыванию или по возрастанию.

```
    using System;

    public abstract class Sorter
    {
        private readonly int[] data;

        protected Sorter(params int[] source)
        {
            data = new int[source.Length];
            Array.Copy(source, data, source.Length);
        }

        protected abstract bool Compare(int x, int y);
```

```
    public void Sort()
    {
        for (var i = 0; i < data.Length - 1; i++)
        {
            for (var j = i + 1; j < data.Length; j++)
            {
                if (Compare(data[j], data[i]))
                {
                    var temp = data[i];
                    data[i] = data[j];
                    data[j] = temp;
                }
            }
        }
    }

    public int[] GetData()
    {
        var result = new int[data.Length];
        Array.Copy(data, result, data.Length);
        return result;
    }
}

public class GreaterFirstSorter : Sorter
{
    public GreaterFirstSorter(params int[] source) : base(source){ }

    protected override bool Compare(int x, int y)
    {
        return x > y;
    }
}

public class TemplateMethodExample
{
    public static void Main()
    {
        var sorter = new GreaterFirstSorter(1, 3, -10, 0);
        sorter.Sort();
        var result = sorter.GetData();
    }
}
```

5. Антипаттерны

Антипаттерны (anti-patterns), также известные как *ловушки* (pitfalls) – это классы наиболее часто внедряемых плохих решений проблем. Они изучаются в случае, когда их хотят избежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем.

Термин *антипаттерн* построен на основе термина pattern – шаблон проектирования. То есть, антипаттерны – это как бы противоположность шаблонам, представляющим хорошие методы программирования.

Далее будет приведён список некоторых наиболее распространённых антипаттернов, разбитый на категории. Для каждого элемента списка указывается название и короткое определение. Также для некоторых элементов будет дан своеобразный «рецепт лечения», указывающий, как устранить антипаттерн.

Антипаттерны в программировании.

– *Ненужная сложность* (Accidental complexity) – внесение ненужной сложности в решение. Ненужная сложность не обусловлена действительной сложностью решаемой проблемы, а искусственно внесена при разработке архитектуры и дизайна ПО.

– *Действие на расстоянии* (Action at a distance) – неожиданное взаимодействие между широко разделёнными частями системы. Антипаттерн может проявиться при использовании в программе глобальных переменных.

– *Слепая вера* (Blind faith) – недостаточная проверка корректности исправления ошибки или результата работы подпрограммы. Альтернативой антипаттерну является разработка через тестирование и, в частности, модульные тесты.

– *Лодочный якорь* (Boat anchor) – сохранение более не используемой части системы. Чтобы уменьшить влияние данного антипаттерна, рекомендуется устаревшие компоненты системы выносить в отдельные библиотеки, дабы не захламлять основной набор классов и методов.

– *Активное ожидание* (Busy spin) – потребление процессорного времени во время ожидания события, обычно при помощи постоянно повторяемой проверки, вместо того, чтобы использовать систему сообщений.

– *Кэширование ошибки* (Caching failure) – забывать сбросить флаг ошибки после её обработки. Согласно своему названию, антипаттерн обычно возникает в системах с кэшированием. Один из способов устранения – полная очистка кэша в случае возникновения ошибки.

– *Проверка типа вместо интерфейса* (Checking type instead of interface) – проверка того, что объект имеет специфический тип в то время, когда требуется только определённый интерфейс.

– *Кодирование путём исключения* (Coding by exception) – добавление нового кода для поддержки каждого специального распознанного случая.

– *Таинственный код* (Cryptic code) – использование аббревиатур вместо мнемонических имён.

– *Жёсткое кодирование* (Hard code) – внедрение предположений об окружении системы в слишком большом количестве точек её реализации. Примером является явная запись имени и пути к файлу в коде. Для устранения антипаттерна следует вместо явных значений использовать именованные константы или параметры из конфигурационных файлов.

– *Мягкое кодирование* (Soft code) – патологическая боязнь жёсткого кодирования, приводящая к тому, что настраивается всё что угодно, при этом конфигурирование системы само по себе превращается в программирование.

– *Поток лавы* (Lava flow) – сохранение нежелательного (излишнего или низкокачественного) кода по причине того, что его удаление слишком дорого или будет иметь непредсказуемые последствия.

– *Магические числа* (Magic numbers) – включение чисел в алгоритмы без объяснений. Для устранения антипаттерна следует использовать именованные константы, причём имя должно отражать смысл константы (см. антипаттерн Таинственный код).

– *Процедурный код* (Procedural code) – ситуация, когда другая парадигма программирования является более подходящей для решения задачи.

– *Спагетти-код* (Spaghetti code) – системы, чья структура редко понятна, особенно потому что структура кода используется неправильно. Антипаттерн обычно проявляется в методах большого размера (20 строк кода и более). Для устранения антипаттерна выполняют выделение фрагментов кода в отдельные функции.

– *Мыльный пузырь* (Soap bubble) – класс, инициализированный мусором, максимально долго притворяется, что содержит какие-то данные.

Антипаттерны в объектно-ориентированном программировании.

– *Базовый класс-утилита* (Base Bean) – наследование функциональности из класса-утилиты вместо делегирования к нему.

– *Вызов предка* (Call Super) – для реализации прикладной функциональности методу класса-потомка требуется в обязательном порядке вызывать те же методы класса-предка. Вместо данного антипаттерна следует использовать паттерн Шаблонный метод.

– *Божественный объект* (God object) – концентрация слишком большого количества функций в одиночной части дизайна (классе). Любой класс должен иметь единственное назначение, которое можно описать несколькими словами. Большие классы следует разбить на мелкие, возможно с применением агрегирования, делегирования и наследования.

– *Объектная клоака* (Object cesspool) – антипод шаблона Пул объектов. Антипаттерн возникает тогда, когда объекты, возвращаемые в пул, не приводятся в своё первоначальное состояние.

– *Полтергейст* (Poltergeist) – объекты, чьё единственное предназначение – передавать информацию другим объектам. Антипаттерн проявляется в виде короткоживущих объектов, лишённых состояния. Эти объекты часто используются для инициализации других, более устойчивых объектов. Устранение антипаттерна заключается в переносе функций в объекты- «долгожители».

– *Проблема йо-йо* (Yo-yo problem) – структура, которая тяжело понятна вследствие избыточной фрагментации.

– *Синглетонизм* (Singletonitis) – избыточное использование шаблона Одиночка.

Антипаттерны в разработке ПО.

– *Большой комок грязи* (Big ball of mud) – система с нераспознаваемой структурой.

– *Бензиновая фабрика* (Gas factory) – необязательная сложность дизайна.

– *Затычка на ввод данных* (Input kludge) – забывчивость в спецификации и выполнении поддержки возможного неверного ввода. Антипаттерн устраняется продуманным алгоритмом проверки (валидации) пользовательского ввода.

– *Раздувание интерфейса* (Interface bloat) – изготовление интерфейса очень мощным и очень трудным для осуществления. Альтернативой этому антипаттерну служит использование таких шаблонов, как Адаптер или Посетитель.

– *Магическая кнопка* (Magic pushbutton) – выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, в системах типа Delphi это написание прикладной логики в обработчиках нажатий на кнопку.

– *Дымоход* (Stovepipe system) – редко поддерживаемая сборка плохо связанных компонентов.

– *Гонки* (Race hazard) – ошибка в определении последовательности различных порядков событий.

Методологические антипаттерны.

– *Программирование методом копирования-вставки* (Copy and paste programming) – копирование (и лёгкая модификация) существующего кода вместо создания общих решений. Симптом этого антипаттерна: после внесения изменений программа в некоторых случаях ведёт себя также, как и раньше. Для устранения антипаттерна требуется выделить повторяющийся код в отдельный метод.

– *Дефакторинг* (De-Factoring) – процесс уничтожения функциональности и замены её документацией.

– *Золотой молоток* (Golden hammer) – сильная уверенность в том, что любимое решение универсально применимо. Название происходит от английской поговорки «когда в руках молоток, все проблемы кажутся гвоздями».

– *Фактор невероятности* (Improbability factor) – предположение о невозможности того, что сработает известная ошибка.

– *Преждевременная оптимизация* (Premature optimization) – оптимизация на основе недостаточной информации.

– *Изобретение колеса* (Reinventing the wheel) – ошибка адаптации существующего решения.

– *Изобретение квадратного колеса* (Reinventing the square wheel) – создание плохого решения, когда существует хорошее.

Литература

1. Влиссидес, Д. Применение шаблонов проектирования. Дополнительные штрихи. / Д. Влиссидес. – М. : Издат. дом «Вильямс», 2003. – 144 с.

2. Гамма, Э. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – Спб. : Питер, 2013. – 368 с.: ил.

3. Мартин, Р. С. Принципы, паттерны и методики гибкой разработки на языке C#. / Р. С. Мартин, М. Мартин. – Спб. : ООО «Издательство «Символ-плюс», 2011. – 768 с.

4. Нильссон, Дж. Применение DDD и шаблонов проектирования. Проблемно-ориентированное проектирование приложений с примерами на C# и .NET. / Дж. Нильссон. – М. : Издат. дом «Вильямс», 2008. – 560 с.

5. Цвалина, К. Инфраструктура программных проектов: соглашения, идиомы и шаблоны для многократно используемых библиотек .NET. / К. Цвалина, Б. Адамс. – М. : Издат. дом «Вильямс», 2011. – 416 с.

6. Фаулер, М. Шаблоны корпоративных приложений. / М. Фаулер. – М. : Издат. дом «Вильямс», 2011. – 544 с.

7. Фаулер, М. Рефакторинг. Улучшение существующего кода. / М. Фаулер. – Спб. : ООО «Издательство «Символ-плюс», 2013. – 432 с.