

1. Введение в паттерны проектирования

Что такое паттерны проектирования?

Паттерны проектирования (design patterns) — это проверенные временем решения типичных задач, которые возникают при разработке программного обеспечения. Это не готовый код, который можно скопировать и вставить, а скорее **шаблоны для решения архитектурных проблем в объектно-ориентированном программировании**.

Паттерны помогают:

- Стандартизировать подход к решению типичных проблем
- Ускорить процесс разработки за счет использования готовых архитектурных решений
- Сделать код более читаемым и понятным для других разработчиков
- Снизить вероятность ошибок в архитектуре приложения

Паттерны не являются решением всех проблем их применение должно быть обоснованным. Неправильное использование паттернов может усложнить код и сделать его менее поддерживаемым.

История: Gang of Four (GoF)

Термин "паттерны проектирования" получил широкое распространение после выхода в 1994 году книги "**Design Patterns: Elements of Reusable Object-Oriented Software**".

Авторы книги — четыре программиста: Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес — получили прозвище "**Банда четырех**" или **GoF (Gang of Four)**. В своей работе они систематизировали и описали 23 классических паттерна, основанных на опыте разработки больших объектно-ориентированных систем. Книга GoF стала классикой в области программирования и до сих пор остается основным источником знаний о паттернах.

Классификация паттернов

Паттерны делятся на три основные категории в зависимости от их назначения:

- **Паттерны поведения (Behavioral Patterns)**
 - **Задача:** определяют взаимодействие между объектами и распределение ответственности.
- **Порождающие паттерны (Creational Patterns)**
 - **Задача:** управляют процессом создания объектов.
- **Структурные паттерны (Structural Patterns)**
 - **Задача:** управляют композицией классов и объектов.

Принципы проектирования SOLID

Помимо паттернов, важную основу качественного объектно-ориентированного проектирования составляют **принципы SOLID**. Это пять принципов, которые помогают создавать гибкую, расширяемую и поддерживаемую архитектуру:

- **SRP** — Single Responsibility Principle (Принцип единственной ответственности)
- **OCP** — Open/Closed Principle (Принцип открытости/закрытости)
- **LSP** — Liskov Substitution Principle (Принцип подстановки Барбары Лисков)
- **ISP** — Interface Segregation Principle (Принцип разделения интерфейса)
- **DIP** — Dependency Inversion Principle (Принцип инверсии зависимостей)

Эти принципы не зависят от конкретного языка программирования и являются фундаментальными правилами хорошего проектирования. Многие паттерны реализуют один или несколько принципов SOLID, обеспечивая их практическое применение.

2. Паттерны поведения

Паттерны поведения определяют взаимодействие между объектами, распределяют обязанности и алгоритмы, они фокусируются на **поведении** системы в ходе выполнения программы.

2.1 Паттерн Стратегия (Strategy)

Описание: Определяет семейство алгоритмов, инкапсулирует каждый из них и делает взаимозаменяемыми. Позволяет выбирать алгоритм во время выполнения.

Пример кода:

```
// Интерфейс стратегии
public interface IShippingStrategy
{
    decimal Calculate(decimal orderTotal);
}

// Конкретные стратегии
public class StandardShipping : IShippingStrategy
{
    public decimal Calculate(decimal orderTotal) => orderTotal * 0.05m;
}

public class ExpressShipping : IShippingStrategy
{
    public decimal Calculate(decimal orderTotal) => 15m + orderTotal *
```

```

0.02m;
}

// Контекст
public class Order
{
    private IShippingStrategy _shippingStrategy;

    public void SetShippingStrategy(IShippingStrategy strategy) =>
    _shippingStrategy = strategy;
    public decimal GetShippingCost(decimal orderTotal) =>
    _shippingStrategy.Calculate(orderTotal);
}

// Использование
var order = new Order();
order.SetShippingStrategy(new ExpressShipping());
var cost = order.GetShippingCost(100m); // 17

```

Где использовать:

- Когда есть несколько алгоритмов выполнения одной задачи (расчет доставки, сортировка, фильтрация)
- Необходимость динамически менять поведение объекта

2.2 Паттерн Шаблонный метод (Template Method)

Описание: Определяет скелет алгоритма в базовом классе, позволяя подклассам переопределять определенные шаги без изменения структуры алгоритма.

Пример кода:

```

public abstract class DataParser
{
    // Шаблонный метод
    public void ParseData()
    {
        Open();
        Read();
        Parse();
        Close();
    }

    protected abstract void Open();
    protected abstract void Read();

    protected virtual void Parse() => Console.WriteLine("Standard parsing");
}

```

```

    protected abstract void Close();
}

public class XmlParser : DataParser
{
    protected override void Open() => Console.WriteLine("Open XML file");
    protected override void Read() => Console.WriteLine("Read XML content");
    protected override void Parse() => Console.WriteLine("Parse XML
specifically");
    protected override void Close() => Console.WriteLine("Close XML file");
}

```

Где использовать:

- Когда алгоритм состоит из повторяющихся и уникальных шагов
- Извлечение общей логики в базовый класс

2.3 Паттерн Посредник (Mediator)

Описание: Определяет объект, который инкапсулирует взаимодействие между множеством объектов, снижая их связанность.

Пример кода:

```

public interface IMediator
{
    void Send(string message, Component sender);
}

public class ChatMediator : IMediator
{
    private List<User> _users = new();

    public void Register(User user) => _users.Add(user);

    public void Send(string message, Component sender)
    {
        foreach (var user in _users.Where(u => u != sender))
            user.Receive(message);
    }
}

public abstract class Component
{
    protected IMediator _mediator;
    public void SetMediator(IMediator mediator) => _mediator = mediator;
}

public class User(string name) : Component
{
}

```

```

{
    public string Name { get; } = name;

    public void Send(string message) => _mediator.Send(message, this);
    public void Receive(string message) => Console.WriteLine($"{Name}
received: {message}");
}

```

Где использовать:

- Сложное взаимодействие множества объектов (чаты, системы бронирования)
- Когда нужно уменьшить связанность компонентов

2.4 Паттерн Итератор (Iterator)

Описание: Предоставляет последовательный доступ к элементам составного объекта, не раскрывая его внутреннее представление.

Пример кода:

```

public class BookCollection : IEnumerable<string>
{
    private string[] _books = ["Book1", "Book2", "Book3"];

    public IEnumerator<string> GetEnumerator()
    {
        for (int i = 0; i < _books.Length; i++)
            yield return _books[i];
    }
}

// Использование
var collection = new BookCollection();
foreach (var book in collection) // Благодаря GetEnumerator
{
    Console.WriteLine(book);
}

```

Где использовать:

- Единый интерфейс обхода для разных типов коллекций
- Поддержка foreach в пользовательских коллекциях

2.5 Паттерн Наблюдатель (Observer)

Описание: Определяет зависимость "один-ко-многим" при изменении состояния одного объекта все подписанные объекты получают уведомление.

Пример кода:

```
public class WeatherStation
{
    public event EventHandler<WeatherInfo>? WeatherChanged;

    private WeatherInfo _currentWeather;

    public void UpdateWeather(WeatherInfo weather)
    {
        _currentWeather = weather;
        WeatherChanged?.Invoke(this, weather);
    }
}

public class WeatherDisplay(string name)
{
    public void Subscribe(WeatherStation station)
    {
        station.WeatherChanged += (s, e) => Console.WriteLine($"{name} shows: {e.Temperature}°C");
    }
}

public record WeatherInfo(decimal Temperature, string Condition);
```

Где использовать:

- GUI: обновление UI при изменении модели
- Подписки и уведомления
- Событийные системы

2.6 Паттерн Посетитель (Visitor)

Описание: Позволяет добавлять новые операции к объектам без изменения их классов. Отделяет алгоритмы от объектов, к которым они применяются.

Пример кода:

```
public interface IVisitor
{
    void Visit(BankAccount account);
    void Visit(RealEstate property);
}

public interface IAsset
{
    void Accept(IVisitor visitor);
```

```

}

public class TaxVisitor : IVisitor
{
    public void Visit(BankAccount account) => Console.WriteLine($"Tax for
account: {account.Balance * 0.1m}");
    public void Visit(RealEstate property) => Console.WriteLine($"Tax for
property: {property.Value * 0.15m}");
}

public class BankAccount(decimal balance) : IAsset
{
    public decimal Balance { get; } = balance;
    public void Accept(IVisitor visitor) => visitor.Visit(this);
}

public class RealEstate(decimal value) : IAsset
{
    public decimal Value { get; } = value;
    public void Accept(IVisitor visitor) => visitor.Visit(this);
}

```

Где использовать:

- Когда нужно часто добавлять новые операции

2.7 Паттерн Команда (Command)

Описание: Превращает запрос в объект, содержащий всю информацию о нем. Позволяет параметризовать клиентов, ставить запросы в очередь, поддерживать отмену.

Пример кода:

```

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

public class TransferCommand(decimal amount, Account from, Account to) : 
ICommand
{
    public void Execute()
    {
        from.Withdraw(amount);
        to.Deposit(amount);
    }
}

```

```

    public void Undo()
    {
        to.Withdraw(amount);
        from.Deposit(amount);
    }
}

public class CommandInvoker
{
    private Stack< ICommand> _history = new();

    public void Execute(ICommand command)
    {
        command.Execute();
        _history.Push(command);
    }

    public void Undo() => _history.Pop().Undo();
}

```

Где использовать:

- Отмена/повтор операций (Undo/Redo)
- Очереди задач
- Параметризация методов командами

2.8 Паттерн Состояние (State)

Описание: Позволяет объекту менять поведение при изменении внутреннего состояния. Создает иллюзию изменения класса объекта.

Пример кода:

```

public interface IOrderState
{
    void Process(Order order);
    void Cancel(Order order);
}

public class NewOrderState : IOrderState
{
    public void Process(Order order) => order.SetState(new
PaidOrderState());
    public void Cancel(Order order) => order.SetState(new
CancelledOrderState());
}

public class PaidOrderState : IOrderState

```

```

{
    public void Process(Order order) => order.SetState(new
ShippedOrderState());
    public void Cancel(Order order) => throw new
InvalidOperationException("Paid order cannot be cancelled");
}

...
public class Order
{
    private IOrderState _state = new NewOrderState();

    public void SetState(IOrderState state) => _state = state;
    public void Process() => _state.Process(this);
    public void Cancel() => _state.Cancel(this);
}

```

Где использовать:

- Состояния заказов, платежей, бронирований
- Конечные автоматы

2.9 Паттерн Цепочка обязанностей (Chain of Responsibility)

Описание: Позволяет передавать запросы по цепочке обработчиков, пока один из них не обработает запрос. Избегает жесткой привязки отправителя запроса к получателю.

Пример кода:

```

public abstract class Handler
{
    private Handler? _next;

    public Handler SetNext(Handler next)
    {
        _next = next;
        return next;
    }

    public virtual void Handle(string request)
    {
        _next?.Handle(request);
    }
}

public class ManagerHandler : Handler

```

```

{
    public override void Handle(string request)
    {
        if (request == "vacation")
            Console.WriteLine("Manager approved vacation");
        else
            base.Handle(request);
    }
}

public class DirectorHandler : Handler
{
    public override void Handle(string request)
    {
        if (request == "raise")
            Console.WriteLine("Director approved raise");
        else
            base.Handle(request);
    }
}

// Использование
var chain = new ManagerHandler();
chain.SetNext(new DirectorHandler());
chain.Handle("vacation"); // Manager approved
chain.Handle("raise");    // Director approved

```

Где использовать:

- Проверка запросов (валидация, авторизация)
- Обработка ошибок (try-catch цепочки)
- Логирование разных уровней

3. Порождающие паттерны

Порождающие паттерны управляют процессом создания объектов, делая его более гибким и независимым от конкретных классов. Они абстрагируют логику создания, позволяя программе создавать объекты различных типов в зависимости от контекста, не зная точных деталей их реализации.

3.1 Паттерн Синглтон (Singleton)

Описание: Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему. Контролирует создание объекта, предотвращая множественную инициализацию.

Пример кода:

```
public class ConfigurationManager
{
    private static ConfigurationManager? _instance;

    private readonly Dictionary<string, string> _settings = new();

    public static ConfigurationManager Instance
    {
        get
        {
            _instance ??= new ConfigurationManager();
            return _instance;
        }
    }

    public void Set(string key, string value) => _settings[key] = value;
    public string? Get(string key) => _settings.TryGetValue(key, out var value) ? value : null;
}

// Использование
ConfigurationManager.Instance.Set("connection", "server=localhost");
var conn = ConfigurationManager.Instance.Get("connection");
```

Где использовать:

- Конфигурационные данные приложения
- Пулы соединений к базе данных
- Доступ к файлам и ресурсам, требующим единого контроля

3.2 Паттерн Абстрактная фабрика (Abstract Factory)

Описание: Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, без указания их конкретных классов. Обеспечивает создание совместимых продуктов.

Пример кода:

```
// Абстрактные продукты
public interface IButton { void Render(); }
public interface ICheckbox { void Render(); }

// Конкретные продукты Windows
public class WinButton : IButton
{
    public void Render() => Console.WriteLine("Windows button");
```

```

}

public class WinCheckbox : ICheckbox
{
    public void Render() => Console.WriteLine("Windows checkbox");
}

// Конкретные продукты Mac
public class MacButton : IButton
{
    public void Render() => Console.WriteLine("Mac button");
}

public class MacCheckbox : ICheckbox
{
    public void Render() => Console.WriteLine("Mac checkbox");
}

// Абстрактная фабрика
public interface IUiFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

// Конкретные фабрики
public class WindowsFactory : IUiFactory
{
    public IButton CreateButton() => new WinButton();
    public ICheckbox CreateCheckbox() => new WinCheckbox();
}

public class MacFactory : IUiFactory
{
    public IButton CreateButton() => new MacButton();
    public ICheckbox CreateCheckbox() => new MacCheckbox();
}

// Использование
public class Application(IUiFactory factory)
{
    private readonly IButton _button = factory.CreateButton();
    private readonly ICheckbox _checkbox = factory.CreateCheckbox();

    public void Render() { _button.Render(); _checkbox.Render(); }
}

```

Где использовать:

- Создание UI для разных платформ (Windows, Mac, Web)
- Работа с разными базами данных (SQL Server, PostgreSQL)
- Поддержка форматов файлов (JSON, XML, CSV)
- Кросс-платформенные приложения
- Темы и стили в приложениях

3.3 Паттерн Строитель (Builder)

Описание: Отделяет конструирование сложного объекта от его представления, позволяя создавать различные представления одного и того же объекта пошагово. Упрощает создание объектов с множеством параметров.

Пример кода:

```
// Продукт
public class Report
{
    public string Title { get; set; } = "";
    public string Content { get; set; } = "";
    public List<string> Charts { get; set; } = new();
    public bool IncludeSummary { get; set; }
}

// Строитель
public interface IReportBuilder
{
    IReportBuilder SetTitle(string title);
    IReportBuilder AddContent(string content);
    IReportBuilder AddChart(string chart);
    IReportBuilder IncludeSummary(bool include);
    Report Build();
}

// Конкретный строитель
public class PdfReportBuilder : IReportBuilder
{
    private readonly Report _report = new();

    public IReportBuilder SetTitle(string title) { _report.Title = title;
    return this; }

    public IReportBuilder AddContent(string content) { _report.Content += content;
    return this; }

    public IReportBuilder AddChart(string chart) {
    _report.Charts.Add(chart); return this; }

    public IReportBuilder IncludeSummary(bool include) {
    _report.IncludeSummary = include; return this; }

    public Report Build() => _report;
}
```

```
// Использование
var builder = new PdfReportBuilder();
var report = builder
    .SetTitle("Custom Report")
    .AddContent("Analysis results")
    .AddChart("Trend analysis")
    .Build();
```

Где использовать:

- Конструирование сложных объектов (отчеты, документы)
- Создание HTTP-запросов с множеством параметров
- Формирование SQL-запросов
- Создание объектов конфигурации
- Парсинг данных пошагово

4. Структурные паттерны

Структурные паттерны управляют композицией классов и объектов, определяя как они объединяются для создания более крупных структур. Эти паттерны упрощают проектирование системы, определяя простые способы реализации взаимоотношений между объектами, обеспечивая при этом гибкость и повторное использование кода.

4.1 Паттерн Адаптер (Adapter)

Описание: Преобразует интерфейс одного класса в интерфейс, ожидаемый клиентом. Позволяет классам с несовместимыми интерфейсами работать вместе. Работает как "переходник" между системами.

Пример кода:

```
// Существующий сторонний сервис с несовместимым интерфейсом
public class LegacyPrinter
{
    public void PrintDocument(string content) => Console.WriteLine($"Legacy:{content}");
}

// Целевой интерфейс, который ожидает клиент
public interface IModernPrinter
{
    void Print(string text);
    void Cancel();
}
```

```

// Адаптер
public class LegacyPrinterAdapter(LegacyPrinter legacyPrinter) :
IModernPrinter
{
    private readonly LegacyPrinter _legacyPrinter = legacyPrinter;

    public void Print(string text) => _legacyPrinter.PrintDocument(text);

    public void Cancel() => Console.WriteLine("Legacy printer doesn't
support cancel");
}

// Использование
IModernPrinter printer = new LegacyPrinterAdapter(new LegacyPrinter());
printer.Print("Document"); // Legacy: Document

```

Где использовать:

- Интеграция со сторонними библиотеками и API
- Совместимость с устаревшим кодом
- Подключение разнородных систем (платежные шлюзы, базы данных)
- Миграция между версиями API
- Преобразование данных между форматами

4.2 Паттерн Фасад (Facade)

Описание: Предоставляет упрощенный интерфейс к сложной системе классов, библиотеке или фреймворку. Скрывает сложность взаимодействия с подсистемой за единым простым интерфейсом.

Пример кода:

```

// Сложная подсистема
public class Cpu { public void Freeze() => Console.WriteLine("CPU freeze");
}
public class Memory { public void Load() => Console.WriteLine("Memory
load"); }
public class HardDrive { public void Read() => Console.WriteLine("HDD
read"); }

// Фасад
public class Computer
{
    private readonly Cpu _cpu = new();
    private readonly Memory _memory = new();
    private readonly HardDrive _hardDrive = new();
}
```

```

public void Start()
{
    _cpu.Freeze();
    _memory.Load();
    _hardDrive.Read();
    Console.WriteLine("Computer started");
}

// Использование
var computer = new Computer();
computer.Start(); // Простой вызов вместо трех сложных

```

Где использовать:

- Упрощение API сложных библиотек
- Клиенты для HTTP, баз данных, внешних сервисов
- Фреймворки, требующие множества взаимосвязанных вызовов
- Предоставление простого интерфейса к подсистемам
- Уменьшение зависимостей клиентского кода

4.3 Паттерн Компоновщик (Composite)

Описание: Позволяет работать с иерархиями объектов как с единым объектом.

Объединяет объекты в древовидные структуры для представления частей и целого.

Клиент работает с компонентами одинаково.

Пример кода:

```

// Компонент
public interface IComponent
{
    void Operation();
    decimal GetPrice();
}

// Лист
public class Product(string name, decimal price) : IComponent
{
    private readonly string _name = name;
    private readonly decimal _price = price;

    public void Operation() => Console.WriteLine($"Product: {_name}");
    public decimal GetPrice() => _price;
}

// Контеинер
public class Box : IComponent

```

```

{
    private readonly List<IComponent> _children = new();

    public void Add(IComponent component) => _children.Add(component);
    public void Remove(IComponent component) => _children.Remove(component);

    public void Operation()
    {
        Console.WriteLine("Box contains:");
        foreach (var child in _children)
            child.Operation();
    }

    public decimal GetPrice() => _children.Sum(c => c.GetPrice());
}

// Использование
var box = new Box();
box.Add(new Product("Phone", 500));
box.Add(new Product("Case", 20));

var subBox = new Box();
subBox.Add(new Product("Cable", 10));
box.Add(subBox);

Console.WriteLine($"Total: {box.GetPrice()}"); // 530

```

Где использовать:

- Деревья файловых систем
- UI-компоненты (панели, контейнеры)
- Иерархии организационных структур
- Комплексные товары (коробки с товарами)

4.4 Паттерн Заместитель (Proxy)

Описание: Предоставляет суррогат или заменитель для другого объекта для контроля доступа к нему. Перехватывает вызовы и может добавлять дополнительную логику (кэширование, безопасность, логирование).

Пример кода:

```

// Интерфейс реального объекта
public interface IImage
{
    void Display();
}

```

```

// Реальный объект (тяжелый)
public class RealImage : IImage
{
    private readonly string _filename;

    public RealImage(string filename)
    {
        _filename = filename;
        LoadFromDisk(); // Времяемкая операция
    }

    private void LoadFromDisk() => Console.WriteLine($"Loading {_filename}");

    public void Display() => Console.WriteLine($"Displaying {_filename}");
}

// Заместитель
public class ProxyImage : IImage
{
    private readonly string _filename;
    private static readonly Dictionary<string, RealImage> _cache = new();

    public ProxyImage(string filename) => _filename = filename;

    public void Display()
    {
        if (!_cache.ContainsKey(_filename))
        {
            _cache[_filename] = new RealImage(_filename);
        }
        _cache[_filename].Display();
    }
}

// Использование
IImage image1 = new ProxyImage("photo.jpg");
IImage image2 = new ProxyImage("photo.jpg");

// Загрузка произойдет только один раз
image1.Display(); // Loading + Displaying
image2.Display(); // Только Displaying (кэш)

```

Где использовать:

- Отложенная инициализация (ленивая загрузка)
- Кэширование результатов запросов
- Контроль доступа (защита)

- Логирование действий
 - Удаленные прокси (удаленный доступ к объектам)
-

5. Принципы проектирования SOLID

SOLID — это акроним пяти фундаментальных принципов объектно-ориентированного проектирования, которые помогают создавать гибкую, поддерживаемую и расширяемую архитектуру. Эти принципы были сформулированы Робертом Мартином и являются основой качественного кода.

5.1 SRP — Single Responsibility Principle (Принцип единственной ответственности)

Описание: Класс должен иметь только одну причину для изменения, то есть одну обязанность или ответственность. Разделение ответственостей делает код более модульным и понятным.

Пример кода:

```
// ✗ Нарушение SRP
public class Employee
{
    public string Name { get; set; } = "";

    // Ответственность 1: бизнес-логика
    public decimal CalculateSalary() => 1000m;

    // Ответственность 2: сохранение в БД
    public void SaveToDatabase() => Console.WriteLine("Saving to DB...");

    // Ответственность 3: генерация отчета
    public string GenerateReport() => "Employee Report";
}

// ✓ Соблюдение SRP
public class Employee
{
    public string Name { get; set; } = "";
}

public class SalaryCalculator
{
    public decimal Calculate(Employee employee) => 1000m;
}

public class EmployeeRepository
```

```

{
    public void Save(Employee employee) => Console.WriteLine("Saving to DB..."); 
}

public class ReportGenerator
{
    public string Generate(Employee employee) => "Employee Report";
}

```

Что делает пример: Разделяет класс Employee на специализированные сервисы, каждый из которых отвечает только за одну задачу.

5.2 OCP — Open/Closed Principle (Принцип открытости/закрытости)

Описание: Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации. Поведение класса можно изменять без изменения его исходного кода.

Пример кода:

```

// ✗ Нарушение OCP
public class AreaCalculator
{
    public double Calculate(object shape)
    {
        if (shape is Circle circle) return Math.PI * circle.Radius * circle.Radius;
        if (shape is Rectangle rect) return rect.Width * rect.Height;
        // При добавлении новой фигуры нужно изменять этот метод!
        throw new Exception("Unknown shape");
    }
}

// ✓ Соблюдение OCP
public abstract class Shape
{
    public abstract double Area();
}

public class Circle(double radius) : Shape
{
    public double Radius { get; } = radius;
    public override double Area() => Math.PI * Radius * Radius;
}

public class Rectangle(double width, double height) : Shape

```

```

{
    public double Width { get; } = width;
    public double Height { get; } = height;
    public override double Area() => Width * Height;
}

public class AreaCalculator
{
    public double Calculate(Shape shape) => shape.Area(); // Для новой
фигуры просто наследуем Shape
}

```

Что делает пример: Вместо модификации условий в калькуляторе, мы создаем иерархию фигур, где каждая реализует свой расчет площади.

5.3 LSP — Liskov Substitution Principle (Принцип подстановки Барбары Лисков)

Описание: Объекты подклассов должны заменять объекты базового класса без изменения свойств программы. Поведение производного класса должно соответствовать ожиданиям от базового класса.

Пример кода:

```

// ✗ Нарушение LSP
public class Rectangle
{
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }
    public double Area() => Width * Height;
}

public class Square : Rectangle
{
    public override double Width
    {
        set { base.Width = value; base.Height = value; }
    }
    public override double Height
    {
        set { base.Width = value; base.Height = value; }
    }
}

Rectangle r = new Square();
r.Width = 5;
r.Height = 6;
Console.WriteLine(shape.Area());
// Ожидаем площадь 30, получаем 36!

```

```

// ✅ Соблюдение LSP
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle(double width, double height) : Shape
{
    public double Width { get; } = width;
    public double Height { get; } = height;
    public override double Area() => Width * Height;
}

public class Square(double side) : Shape
{
    public double Side { get; } = side;
    public override double Area() => Side * Side;
}

Shape shape = new Square(5);
Console.WriteLine(shape.Area()); // 25

```

Что делает пример: Вместо наследования Square от Rectangle (где поведение противоречит логике), оба класса наследуются от общего Shape, сохраняя корректную математическую семантику.

5.4 ISP — Interface Segregation Principle (Принцип разделения интерфейса)

Описание: Клиенты не должны зависеть от интерфейсов, которые они не используют. Лучше иметь несколько специализированных интерфейсов, чем один универсальный.

Пример кода:

```

// ❌ Нарушение ISP
public interface IWorker
{
    void Work();
    void Eat();
}

public class Robot : IWorker
{
    public void Work() => Console.WriteLine("Robot working");
    public void Eat() => throw new Exception("Robot doesn't eat"); // Лишний
    метод!
}

```

```

// ✅ Соблюдение ISP
public interface IWorkable
{
    void Work();
}

public interface IFeedable
{
    void Eat();
}

public class Human : IWorkable, IFeedable
{
    public void Work() => Console.WriteLine("Human working");
    public void Eat() => Console.WriteLine("Human eating");
}

public class Robot : IWorkable // Только нужный метод
{
    public void Work() => Console.WriteLine("Robot working");
}

```

Что делает пример: Разделяем "толстый" интерфейс на два специализированных, чтобы Robot не был вынужден реализовывать ненужный метод Eat() .

5.5 DIP — Dependency Inversion Principle (Принцип инверсии зависимостей)

Описание: Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей, детали должны зависеть от абстракций.

Пример кода:

```

// ❌ Нарушение DIP
public class UserController
{
    private readonly SqlDatabase _database = new(); // Жесткая зависимость
    public void SaveUser(User user) => _database.Save(user);
}

public class SqlDatabase
{
    public void Save(User user) => Console.WriteLine("Saving to SQL");
}

// ✅ Соблюдение DIP

```

```
public interface IDatabase
{
    void Save(User user);
}

public class UserController
{
    private readonly IDatabase _database;

    // Зависимость внедряется (DI)
    public UserController(IDatabase database) => _database = database;

    public void SaveUser(User user) => _database.Save(user);
}

public class SqlDatabase : IDatabase
{
    public void Save(User user) => Console.WriteLine("Saving to SQL");
}

public class MongoDB : IDatabase
{
    public void Save(User user) => Console.WriteLine("Saving to MongoDB");
}

// Использование
IDatabase db = new MongoDB(); // Легко заменить
var controller = new UserController(db);
controller.SaveUser(new User());
```

Что делает пример: Контроллер зависит от интерфейса `IDatabase`, а не от конкретной реализации. Это позволяет легко заменить базу данных, тестировать с мок-объектами и соблюдать гибкость архитектуры.
