

объектно-ориентированное программирование

структурные паттерны

адаптер

промежуточный тип, использующий объект
одного типа, для реализации интерфейса
другого типа



адаптер

структура

адаптер

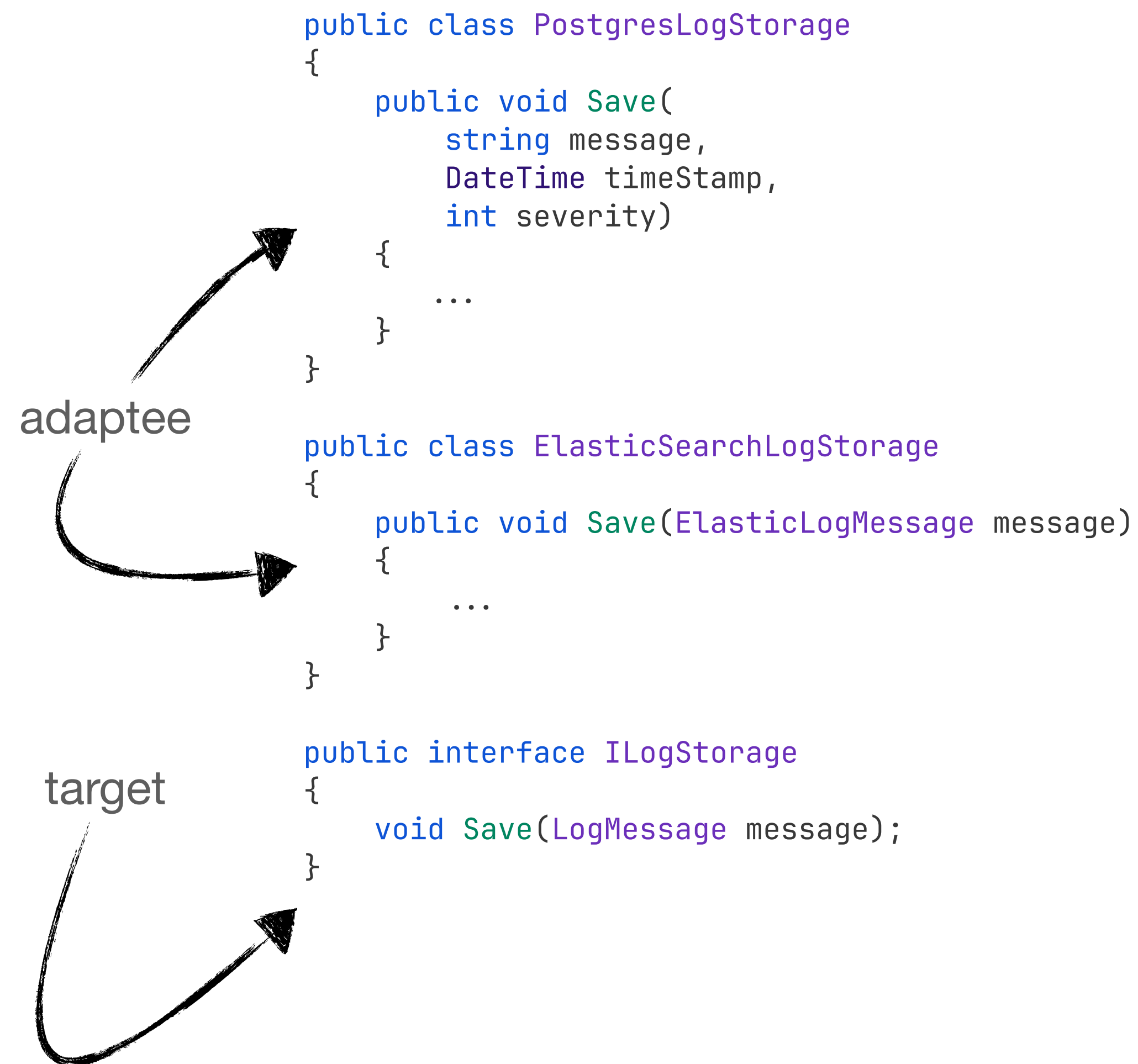
- target
целевой интерфейс, через который мы хотим взаимодействовать с объектом, изначально его не реализующим
- adaptee
адаптируемый тип
- adapter
тип-обёртка, реализует целевой интерфейс, содержит объект адаптируемого типа, перенаправляет в него вызовы поведений целевого интерфейса

схема

адаптер



пример использования адаптер



```
public class PostgresLogStorageAdapter : ILogStorage
{
    private readonly PostgresLogStorage _storage;

    public void Save(LogMessage message)
    {
        _storage.Save(
            message.Message,
            message.DateTime,
            message.Severity.AsInteger());
    }
}

public class ElasticLogStorageAdapter : ILogStorage
{
    private readonly ElasticSearchLogStorage _storage;

    public void Save(LogMessage message)
    {
        _storage.Save(message.AsElasticLogMessage());
    }
}
```



адаптивный рефакторинг

адаптер

```
public interface IAsyncLogStorage
{
    Task SaveAsync(LogMessage message);
}

public class AsyncLogStorageAdapter : IAsyncLogStorage
{
    private readonly ILogStorage _storage;

    public Task SaveAsync(LogMessage message)
    {
        _storage.Save(message);
        return Task.CompletedTask;
    }
}
```

адаптивный рефакторинг

адаптер

- позволяет проводить рефакторинг в два шага
 - изменения использований
 - изменение реализации
- позволяет локализовать изменения

МОСТ

разделение объектной модели на абстракции разных уровней
реализации абстракций более высокого уровня,
использующие абстракции более низкого уровня и являются
“МОСТОМ”

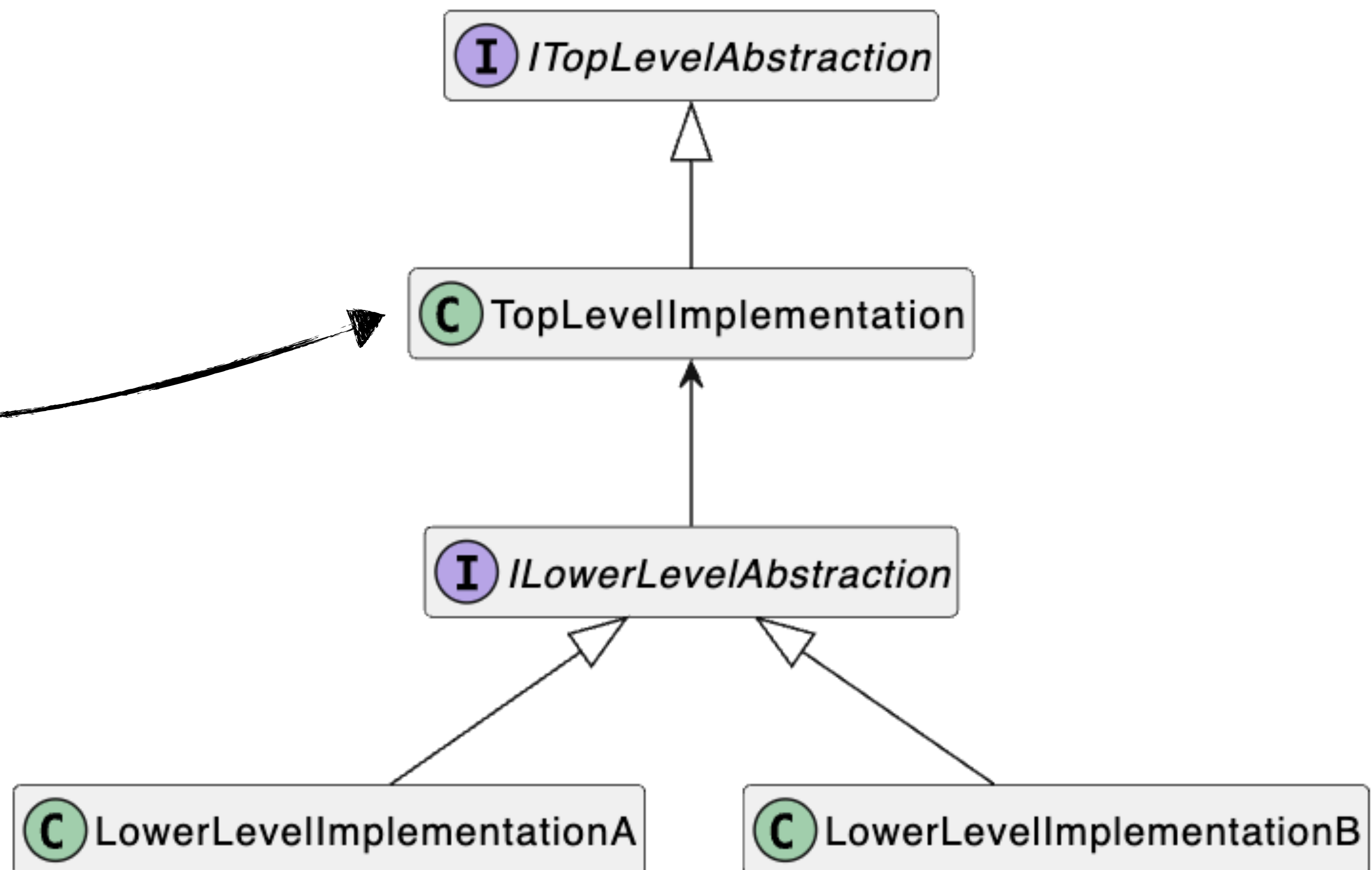


МОСТ

cxema

МОСТ

МОСТ



пример использования

МОСТ

top-level abstraction

```
public interface IControl
{
    void ToggleEnabled();

    void ChannelForward();

    void ChannelBackward();

    void VolumeUp();

    void VolumeDown();
}
```

bridge

```
public class Control : IControl
{
    private readonly IDevice _device;

    public void ToggleEnabled()
        ⇒ _device.IsEnabled = !_device.IsEnabled;

    public void ChannelForward()
        ⇒ _device.Channel += 1;

    public void ChannelBackward()
        ⇒ _device.Channel -= 1;

    public void VolumeUp()
        ⇒ _device.Volume += 10;

    public void VolumeDown()
        ⇒ _device.Volume -= 10;
}
```

lower-level abstraction

```
public interface IDevice
{
    public bool IsEnabled { get; set; }

    public int Channel { get; set; }

    public int Volume { get; set; }
}
```

пример использования

МОСТ

```
public class FaultyControl : IControl
{
    private readonly IDevice _device;

    public void ToggleEnabled()
    {
        TryFault();
        _device.IsEnabled = !_device.IsEnabled;
    }

    public void ChannelForward()
    {
        TryFault();
        _device.Channel += 1;
    }

    ...

    private void TryFault()
    {
        _device.IsEnabled = Random.Shared.NextDouble() < 0.5
            ? _device.IsEnabled
            : !_device.IsEnabled;
    }
}
```

разбиение на другие принципы

МОСТ

- способ реализации OCP
- способ реализации protected variations
- подвид адаптера
отличается тем, что абстракции моста проектируются изначально, а адаптер добавляется в процессе поддержки кода
- полиморфный билдер + директор = мост

КОМПОНОВЩИК

**представление древовидной структуры
объектов в виде одного композитного объекта**

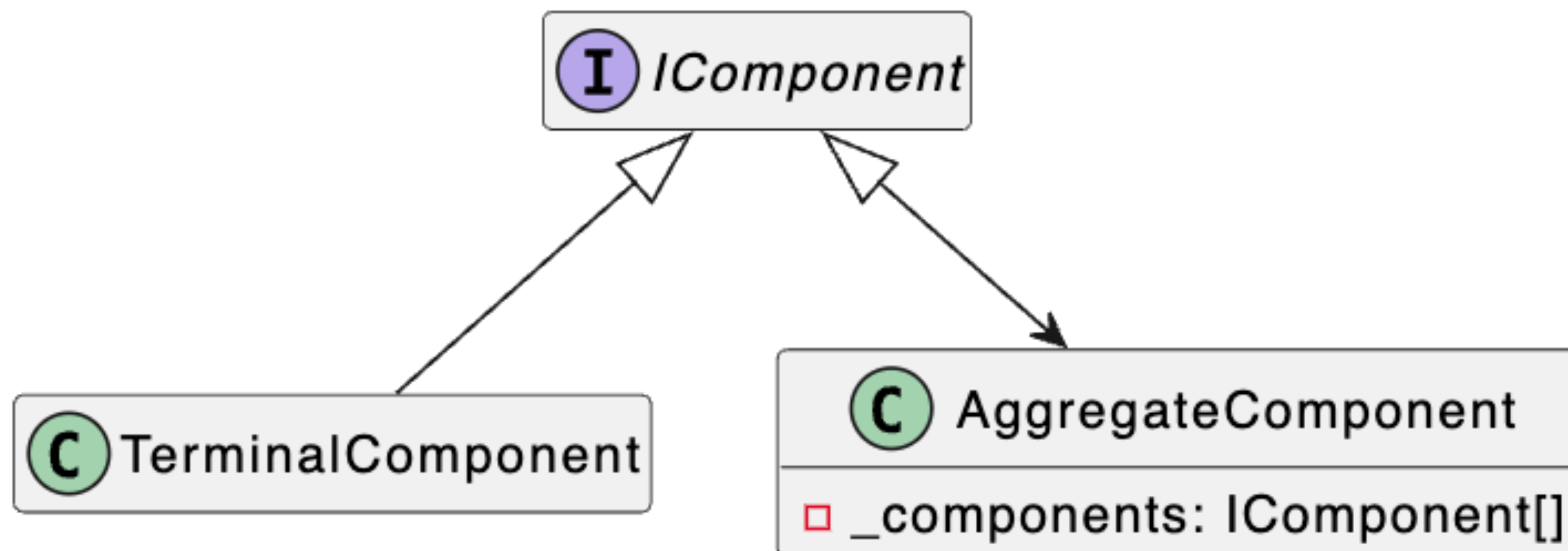


КОМПОНОВЩИК

схема

КОМПОНОВЩИК

- терминальные узлы – не содержат другие узлы
- агрегатные узлы – содержат другие узлы



пример использования

КОМПОНОВЩИК

```
public readonly record struct Position(int X, int Y);

public interface IGraphicComponent
{
    void MoveBy(int x, int y);

    void Draw();
}
```

```
public class Circle : IGraphicComponent
{
    private Position _position;

    public void MoveBy(int x, int y)
    {
        _position = new Position(_position.X + x, _position.Y + y);
    }

    public void Draw()
    {
        Console.WriteLine($"Circle: {_position.X}, {_position.Y}");
    }
}
```

```
public class Square : IGraphicComponent
{
    private Position _position;

    public void MoveBy(int x, int y)
    {
        _position = new Position(_position.X + x, _position.Y + y);
    }

    public void Draw()
    {
        Console.WriteLine($"Square: {_position.X}, {_position.Y}");
    }
}
```

пример использования

КОМПОНОВЩИК

```
public class GraphicComponentGroup : IGraphicComponent
{
    private readonly IReadOnlyCollection<IGraphicComponent> _components;

    public GraphicComponentGroup(IReadOnlyCollection<IGraphicComponent> components)
    {
        _components = components;
    }

    public void MoveBy(int x, int y)
    {
        foreach (var component in _components)
            component.MoveBy(x, y);
    }

    public void Draw()
    {
        foreach (var component in _components)
            component.Draw();
    }
}
```

пример использования

КОМПОНОВЩИК

```
var circle = new Circle();  
circle.MoveBy(1, 2);
```

```
var square = new Square();  
square.MoveBy(10, 0);
```

```
var composite = new GraphicComponentGroup([circle, square]);  
composite.Draw();  
Console.WriteLine("----");
```

```
composite.MoveBy(1, 1);  
composite.Draw();
```

```
Circle: 1, 2  
Square: 10, 0  
----  
Circle: 2, 3  
Square: 11, 1
```

декоратор

тип-обёртка над объектом абстракции, которую он
реализует
добавляет к поведению объекта новую логику



декоратор

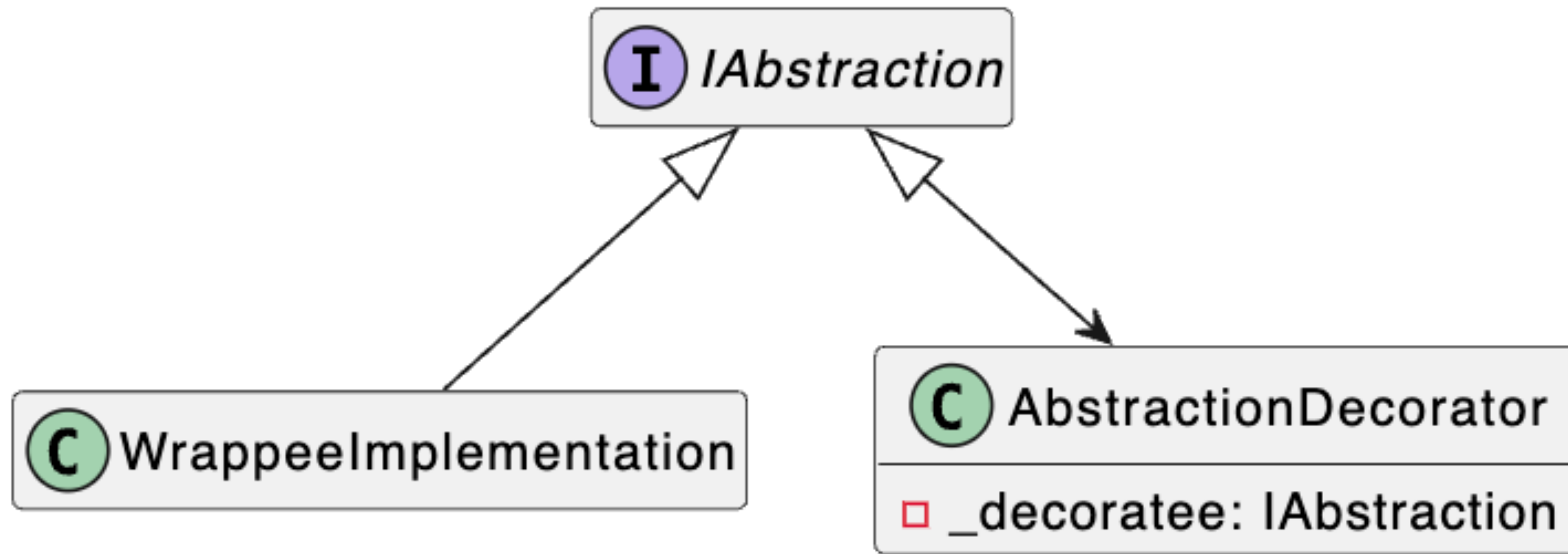
структура

декоратор

- абстракция
какой-либо интерфейс, определяющий поведения
- декоратор
тип, реализующий абстракцию, содержащий объект данной абстракции
- decoratee
объект, типа, реализующего абстракцию, оборачиваемый в декоратор

схема

декоратор



пример использования декоратор

```
public interface IService
{
    void DoStuff(DoStuffArgs args);
}

public class Service : IService
{
    public void DoStuff(DoStuffArgs args) { }
}
```

```
public class LoggingServiceDecorator : IService
{
    private readonly IService _decoratee;
    private readonly ILogger _logger;

    public void DoStuff(DoStuffArgs args)
    {
        _logger.Log(ArgsToLogMessage(args));
        _decoratee.DoStuff(args);
    }

    private static string ArgsToLogMessage(DoStuffArgs args) { ... }
}
```

прокси

тип-обёртка, реализующий логику контроля
доступа к объекту, реализующему абстракцию,
которую реализует он сам



прокси

ВИДЫ прокси

```
public interface IService
{
    void DoOperation(OperationArgs args);
}

public class Service : IService
{
    public void DoOperation(OperationArgs args) { }
}
```

virtual proxy

прокси

- реализует ленивую инициализацию
- позволяет абстрагировать эту логику от потребителя

```
public class VirtualServiceProxy : IService
{
    private readonly Lazy<IService> _service = new Lazy<IService>(() => new Service());

    public void DoOperation(OperationArgs args)
    {
        _service.Value.DoOperation(args);
    }
}
```

defensive proxy

прокси

- реализует логику валидации, проверки допустимости действий

```
public class ServiceAuthorizationProxy : IService
{
    private readonly IService _service;
    private readonly IUserInfoProvider _userInfoProvider;

    public void DoOperation(OperationArgs args)
    {
        if (_userInfoProvider.GetUserInfo().IsAuthenticated)
            _service.DoOperation(args);
    }
}
```

caching proxy

прокси

- реализует кеширование/мемоизацию
- абстрагирует кеширование от пользователя

```
public class CachingServiceProxy : IService
{
    private readonly IService _service;
    private readonly Dictionary<OperationArgs, OperationResult> _cache;

    public OperationResult DoOperation(OperationArgs args)
    {
        if (_cache.TryGetValue(args, out var result))
            return result;

        return _cache[args] = _service.DoOperation(args);
    }
}
```

decorator vs proxy

прокси

- ВИДЫ КОМПОЗИЦИИ
прокси – агрегация/ассоциация
декоратор – только агрегация
- extended dispatch vs controlled dispatch
прокси – контролирует обрачиваемый объект
декоратор – только расширяет логику обрачиваемого объекта
- наличие обрачиваемого объекта
прокси – может имитировать наличие объекта
декоратор – объект должен существовать

фасад

**оркестрация одной или набора
сложных операций в каком-либо типе**



фасад

недостатки

фасад

- риск сделать god-class
- потеря абстракций засчёт переиспользования логики внутри фасада
- тяжесть рефакторинга и декомпозиции
- стоит приводить к request-response модели

легковес

декомпозиция объектов, выделение тяжёлых и повторяющихся данных в отдельные модели для дальнейшего переиспользования



легковес

пример использования

легковес

```
public record Particle(int X, int Y, byte[] Model);

public class ParticleFactory
{
    private readonly IAssetLoader _assetLoader;

    public Particle Create(string modelName)
    {
        var model = _assetLoader.Load(modelName);
        return new Particle(0, 0, model);
    }
}
```

пример использования легковес

```
public record ModelData(byte[] Value);

public record Particle(int X, int Y, ModelData Model);

public class ParticleFactory
{
    private readonly IAssetLoader _assetLoader;
    private readonly Dictionary<string, ModelData> _cache;

    public Particle Create(string modelName)
    {
        var model = _cache.TryGetValue(modelName, out var data)
            ? data
            : _cache[modelName] = new ModelData(_assetLoader.Load(modelName));

        return new Particle(0, 0, model);
    }
}
```