Масштабирование паттерна Dispose

в рамках проекта

Кирилл Маурин

Стрим

leo.bonart@gmail.com

План

- Паттерн
- Проблемы
- Методы решения
- Отказ от ObjectDisposedException
- Только управляемые ресурсы
- Композиция владения ресурсами
- Внешнее управление временем жизни
- Разделение ресурса, владельца и пользователя

Паттерн Dispose

- Управляемый ресурс с детерминированным высвобождением
- Требуется реализовать интерфейс IDisposable
- В методе Dispose высвободить ресурсы
- Бросать ObjectDisposableException, если Dispose уже был вызван
- В финализаторе высвободить ресурсы аварийно

Проблемы

- Много кода
- Сложно избежать ошибок
- Владение и управляемыми, и неуправляемыми ресурсами
- Дублирование в каждой реализации
- Плохая масштабируемость

Методы решения

- Janitor.Fody
- Отказ от выброса ObjectDisposedException
- Только управляемые ресурсы
- Композиция владения ресурсами
- Внешнее управление временем жизни
- Разделение ресурса и владения

Janitor.Fody

- Автоматически генерирует почти весь шаблонный код
- Лечит симптомы, но не проблему
- Замедляет сборку
- Затрудняет отладку

ObjectDisposableException

- В правильно написанном коде не нужно
- Вносит лишние проверки времени исполнения
- Устранение шумового кода только с помощью метапрограммирования
- Ущерб от исключения выше чем от повторного вызова Dispose

Только управляемые ресурсы

- Каждому неуправляемому ресурсу — персональную управляемую обертку
- Реализацию оберток можно упростить, используя наследование от стандартных классов
- Владеть двумя разными видами ресурсов — антипаттерн

Композиция владения ресурсами

- Класс CompositeDisposable
- Однострочный метод Dispose
- Однострочное добавление собственного ресурса
- Локализация кода захвата и высвобождения ресурса

CompositeDisposable

- Набор управляемых ресурсов как один управляемый ресурс
- Есть в Rx, но при желании несложно реализовать отдельно

Однострочник Dispose

```
public void Dispose()
   => _lifetime.Dispose();

readonly CompositeDisposable _lifetime
   = new CompositeDisposable();
```

Очистка управляемых ресурсов

- Было

```
_resource = new Resource();
...
_resource?.Dispose();
_resource = null;
```

- Стало

```
_lifetime.Add(_resource = new Resource());
```

Отписка от событий

- Было

```
_resource.Event += ResourceEvent;
...
_resource.Event -= ResourceEvent;
```

- Стало

```
_resource.Event += ResourceEvent;
_lifetime.Add(() => _resource.Event -= ResourceEvent);
```

Отписка от IObservable

- Было

```
_subscription = observable.Subscribe(Handler);
...
_subscription?.Dispose();
_subscription = null;
```

- Стало

```
_lifetime.Add(observable.Subscribe(Handler));
```

Произвольная очистка

- Было

```
CreateAction();
...
DisposeAction();
```

- Стало

```
CreateAction();
_lifetime.Add(() => DisposeAction());
```

Проверка на вызов Dispose

```
_lifetime.IsDisposed
```

Пироги и пышки

- Универсальность
- Выразительность
- Прозрачность
- Аддитивность
- CompositeDisposable легко заменим

Синяки и шишки

- Проблема масштабирования в общем случае не решена
- Порядок высвобождения для CompositionDisposable не документирован
- Фактический порядок высвобождения FIFO
- Дополнительная нагрузка на сборщик мусора

Наследование интерфейсов от IDisposable

- Обязывает реализацию высвобождать ресурсы за себя и за свои зависимости
- Прямо противоречит паттерну внедрения зависимостей
- Никогда так не делайте для интерфейсов, отличных от ролевых

DI-контейнеры

- Самостоятельно распознают реализации IDisposable
- Высвобождают все ресурсы в контексте при окончании его времени жизни
- Мало чем могут помочь с транзиентными зависимостями

Ключ к масштабируемости

- Высвобождение ресурсов — отдельная ответственность
- Высвобождение не разрушение
- Пользователь ресурса не владелец
- Когда освобождать решает пользователь
- Что делать с освобожденным решает владелец

Реализация от Autofac

- Owned<T> реализует IDisposable
- Обозначает ресурс с детерминированным высвобождением
- Контейнер обеспечивает автоматическое создание LifetimeScope для каждой такой зависимости

Универсальное решение

- Класс Usable<T> — дополнение к Lazy<T>
- Первое обращение к Lazy.Value — захват ресурса
- Usable.Dispose() — высвобождение ресурса
- Что делать с ресурсом до захвата и после освобождения определяет владелец ресурса
- Ресурс, пользователь ресурса и владелец ресурса — три разных объекта

В чем различие между IDisposable и Usable<T>?

- IDisposable — разрушение
- Usable<T> — высвобождение
- IDisposable — разрушает никому уже не нужное
- Usable<T> — определяет что делать с уже не нужным конкретному потребителю
- IDisposable — реализует сам ресурс
- Usable<T> — реализует владелец ресурса

Типовые варианты использования

- Прозрачное связывание Stream и Reader
- Прозрачная организация счетчика ссылок
- Прозрачный пул объектов
- Любой захват ресурса, требующий что-то вызвать для высвобождения
- Все это и многое другое для пользователя выглядит как Func<Usable<T>>

Реализация

```
public sealed class Usable<T> : IDisposable
{
    internal Usable(T resource, Action dispose) => (_dispose, _resource) = (dispose, resource);

    public void Dispose()
    {
        (_dispose ?? throw new ObjectDisposedException("")).Invoke();
        _dispose = null;
        _resource = default;
    }

    internal T Resource => _dispose == null ? throw new ObjectDisposedException("") : _resource;

    public override string ToString()
       => _dispose != null ? $"Usable{{{_resource}}}" : $"Disposed<{typeof(T).Name}>";

    Action _dispose;
    T _resource;
}
```

Зачем скрывать конструктор?

- C# не умеет выводить типы для конструкторов
- Сравните частоту использования new List() и ToList()
- Сигнатуру конструктора можно менять, не затрагивая имеющиеся зависимости

Зачем скрывать свойство Resource?

- Этот ресурс — заемный, он у вас на время, а не навсегда
- Писать хороший код должно быть проще, чем плохой
- Каждое обращение к свойству — проверка на то, что метод Dispose еще не вызван

Как создавать?

```
public static Usable<T> ToSelfUsable<T>(this T resource) where T : IDisposable
    => resource.ToUsable(resource);

public static Usable<T> ToUsable<T>(this T resource, IDisposable usageTime)
    => new Usable<T>(resource, usageTime.Dispose);

public static Usable<T> ToUsable<T>(this T resource)
    => resource.ToUsable(DoNothing);

public static Usable<T> ToUsable<T>(this T resource, Func<T, IDisposable> usageTimeFactory)
    => resource.ToUsable(usageTimeFactory(resource));

public static Usable<T> ToUsable<T>(this T resource, Action dispose)
    => new Usable<T>(resource, dispose);

public static Usable<T> ToUsable<T>(this T resource, Action<T> dispose)
    => resource.ToUsable(() => dispose(resource));
```

Как использовать?

```
public static void Using<T>(this Usable<T> usable, Action<T> action)
{
    using (usable)
        action(usable.Resource);
}

public static TResult Using<T, TResult>(this Usable<T> usable, Func<T, TResult> func)
{
    using (usable)
        return func(usable.Resource);
}

public static Usable<T> Do<T>(this Usable<T> usable, Action<T> action)
{
    action(usable.Resource);
    return usable;
}

public static TResult Unwrap<T, TResult>(this Usable<T> usable, Func<T, TResult> func)
    => func(usable.Resource);
```

Как комбинировать?

```
public static Usable<T> Wrap<T>(this Usable<T> resource, Action dispose)
    => new Usable<T>(resource.Resource, () =>
    {
        resource.Dispose();
        dispose();
    });

public static Usable<IEnumerable<T>> ToAggregatedUsable<T>(
    this IEnumerable<Usable<T>> source)
{
    var disposables = source.ToImmutableList();
    return ((IEnumerable<T>)disposables
        .Select(u => u.Resource)
        .ToImmutableList()).ToUsable(new CompositeDisposable(disposables));
}
```

Как LINQфицировать?

```
public static Usable<T> Select<TSource, T>(
    this Usable<TSource> source, Func<TSource, T> selector)
    => selector(source.Resource).ToUsable(source);

public static Usable<T> SelectMany<TSource, T>(
    this Usable<TSource> source, Func<TSource, Usable<T>> selector)
    => selector(source.Resource).Wrap(source);
```

Как раздать?

```
public static Func<Usable<T>> ToRefCount<T>(this Usable<T> source)
{
     var refCount = new RefCountDisposable(source);
     return () =>
     {
         var disposable = refCount.GetDisposable();
         refCount.Dispose();
         return source.Resource.ToUsable(disposable);
     };
}
```

Пироги и пышки

- Универсальность
- Выразительность
- Прозрачность
- Компонуемость
- Аддитивность

Синяки и шишки

- Контринтуитивность
- Нагрузка на сборщик мусора

ValueUsable<T>

- Не нагружает сборщик мусора
- Не контролирует повторное высвобождение
- Требует дублирования библиотечного кода

Источники вдохновения

1. MSDN https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/dispose-pattern
2. Janitor.Fody https://github.com/Fody/Janitor
3. Моя реализация Usable https://github.com/Kirill-Maurin/FluentHelium/blob/master/FluentHelium.Base/Usable.cs
4. Моя статья «Самая простая и надежная реализация шаблона проектирования Dispose» https://habr.com/post/270929/
5. Моя статья «Disposable без границ» https://habr.com/post/272497/https://habr.com/post/272497/
6. Станислав Сидристый «Реализация IDisposable: правильное использование» https://habr.com/post/341864/
7. Станислав Сидристый «Шаблон Lifetime: для сложного Disposing» https://www.youtube.com/watch?v=F5oOYKTFpcQ

Моя реализация Usable

Станислав Сидристый «Шаблон Lifetime: для сложного Disposing»

Вопросы?

Кирилл Маурин

leo.bonart@gmail.com