

Projektowanie obiektowe oprogramowania

Zestaw 4

Wzorce podstawowe i kreacyjne

2025-03-18

Liczba punktów do zdobycia: **5/30**

Zestaw ważny do: 2025-04-01

Uwaga! Obowiązkową częścią każdego rozwiązania są testy jednostkowe.

1. **(1p) (Singleton)** Przygotować implementacje singletonów o następujących politykach czasu życia:

- jedna instancja dla całego procesu
- jedna osobna instancja dla każdego wątku
- jedna instancja na co najwyżej 5 kolejnych sekund

Dostarczyć właściwe testy jednostkowe.

2. **(2p) (Open Delegate Factory)** Powtórzyć przykład z wykładu fabryki otwartej na rozszerzenia:

```
public class ShapeFactory
{
    public RegisterWorker( IShapeFactoryWorker worker ) {
        ...
    }

    public IShape CreateShape( string ShapeName, params object[] parameters ) {
        ...
    }
}

// klient
ShapeFactory factory = new ShapeFactory();
IShape square = factory.CreateShape( "Square", 5 );

// rozszerzenie
factory.RegisterWorker( new RectangleFactoryWorker() );
IShape rect = factory.CreateShape( "Rectangle", 3, 5 );
```

3. **(1p) (Object Pool)** Zaproponowana na wykładzie implementacja wzorca **ObjectPool** jest zgodna z implementacją referencyjną, ale ma pewne wady - istnieje na przykład konieczność sprawdzania czy parametr wywołania metody **ReleaseReusable** jest zasobem pochodzącym z tej puli do której jest zwracany.

W praktyce, pule implementuje się inaczej, klient nie musi używać puli wprost i wywoływać na niej metody **AcquireReusable** i **ReleaseReusable**. Zamiast tego, interfejsem programistycznym klienta jest klasa - nazwijmy ją **BetterReusable** - która dla klienta wygląda jak **Reusable** ale

- wszystkie operacje deleguje do **Reusable**
- w konstruktorze próbuje pozyskać zasób (wywołać **AcquireReusable**)
- w jakiejś jawnej metodzie zwraca zasób (wywołuje **ReleaseReusable**)

```
public class BetterReusable
{
    private Reusable _reusable;

    // konstruktor
    public BetterReusable()
    {
        // tu próba pozyskania Reusable z puli
    }

    // metoda uwalniająca zasób
    public void Release()
    {
        // tu uwolnienie Reusable do puli
    }

    // wszystkie operacje są delegowane
    // z BetterReusable do Reusable
    public void DoWork()
    {
        _reusable.DoWork();
    }
}
```

Dzięki takiej konstrukcji, zamiast

```
var reusable = ObjectPool.Instance.AcquireReusable();

reusable.DoWork();

ObjectPool.Instance.ReleaseReusable( reusable );
```

klient wykona

```
var reusable = new BetterReusable();

reusable.DoWork();

reusable.Release();
```

Zadanie polega na uzupełnieniu implementacji **BetterReusable**, powtórzeniu wszystkich testów jednostkowych które pokazano na wykładzie oraz dodaniu jednego ważnego testu - wykonanie operacji (tu: **DoWork**) na obiekcie który został już zwrócony (wykonano na nim **Release**) powinno kończyć się bezwarunkowym wyjątkiem.

4. (1p) (Builder) Przykład z wykładu

<http://netpl.blogspot.com/2012/02/simple-fluent-and-recursive-tag-builder.html>
rozwinąć o obsługę wcięć (z wcięciami/bez wcięć/głębokość wcięć), po to żeby można było napisać:

```
StringWriter writer = new StringWriter(...);
TagBuilder tag      = new TagBuilder( writer );
tag.IsIndented      = true;
tag.Indentation      = 4;
tag.StartTag( "parent" )
    .AddAttribute( "parentproperty1", "true" )
```

```
.AddAttribute( "parentproperty2", "5" )  
    .StartTag( "child1")  
    .AddAttribute( "childproperty1", "c" )  
    .AddContent( "childbody" )  
    .EndTag()  
    .StartTag( "child2" )  
    .AddAttribute( "childproperty2", "c" )  
    .AddContent( "childbody" )  
    .EndTag()  
.EndTag()  
.StartTag( "script" )  
.AddContent( "$.scriptbody();" )  
.EndTag();
```

Dostarczyć właściwe testy jednostkowe.

Wiktor Zychla