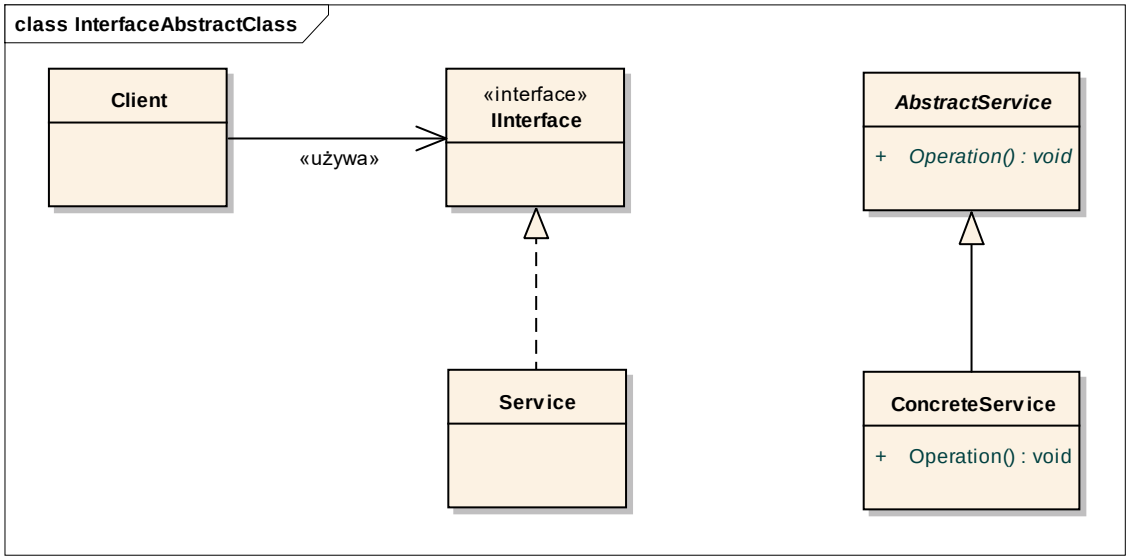
# Projektowanie obiektowe oprogramowania
# Wykład 4 – wzorce projektowe
# cz.I. wzorce podstawowe i kreacyjne
# Wiktor Zychla 2024

## Spis treści

# 1 Wzorce podstawowe

## 1.1 Interface vs Abstract class



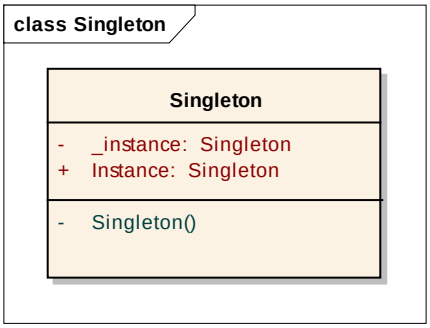

- klasa abstrakcyjna może zawierać implementacje, interfejs nie
- klasa może dziedziczyć tylko z jednej klasy abstrakcyjnej i wielu interfejsów
- przykłady `IEnumerable` vs `Stream`

## 1.2 Delegation (Prefer Delegation over Inheritance)

- dziedziczenie jest relacją statyczną, delegacja może być dynamiczna
- delegujący obiekt może ukrywać metody delegowanego (i ogólniej – zmieniać kontrakt), co jest niemożliwe w przypadku dziedziczenia
- dobra praktyka: klasa domeny nie dziedziczymy z klas użytkowych (`Person` nie dziedziczy z `Hashtable`), ale delegacja jest ok.
- delegacja powoduje że jest więcej kodu – w językach programowania brakuje wsparcia dla delegacji (por. https://github.com/dotnet/roslyn/issues/13952 - dyskusja nad propozycją rozszerzenia składni C# o wsparcie dla delegacji)

# 2 Wzorce kreacyjne

## 2.1 Singleton




- Jedna i ta sama instancja obiektu dla wszystkich klientów

- Często punkt wyjścia dla innych elementów architektury aplikacji

Zalety:

- Uniwersalność
- „Leniwa” konstrukcja

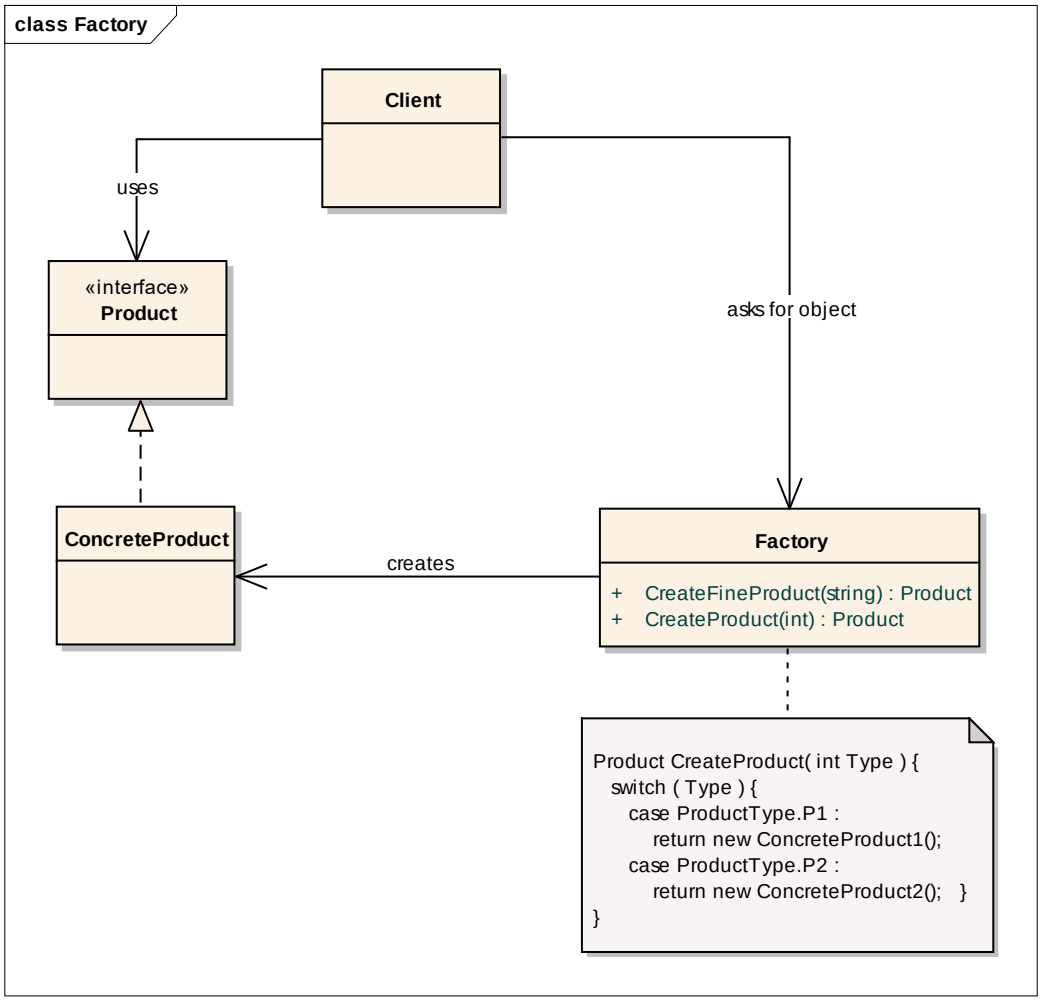
Rozszerzenia:

- Możliwość sterowania czasem życia obiektu „wspierającego” („pseudosingleton”, singleton z określoną polityką czasu życia)
- Singleton parametryzowany (zainicjowanie wymaga parametrów inicjalizacyjnych)

## 2.2 Monostate

- Usuwa ograniczenie liczby instancji w Singletonie, pozostawia właściwość współdzielenia stanu

## 2.3 (Delegate) Factory

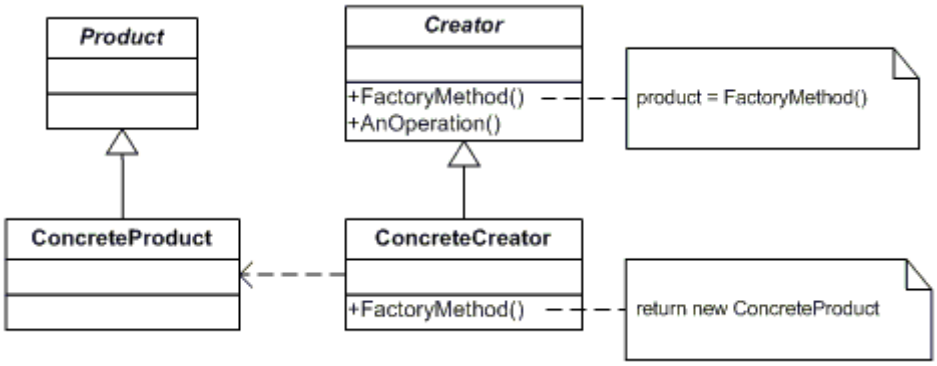



- To jeden z częściej stosowanych wzorców, realizacja odpowiedzialności Creator z GRASP
- Interfejs klasy fabryki może mieć wiele metod, ułatwiających tworzenie konkretnych obiektów (parametryzacja przez typ metody fabryki, przez wiele metod jednej fabryki); fabryka może też zwracać obiekt typu pochodnego względem oczekiwanego, w ten sposób

być przygotowana na zmiany funkcjonalności – klient spodziewa się obiektu typu A, dostaje B dziedziczące z A i korzysta z niego jak z A, ale w rzeczywistości B realizuje swoją odpowiedzialność być może inaczej niż A

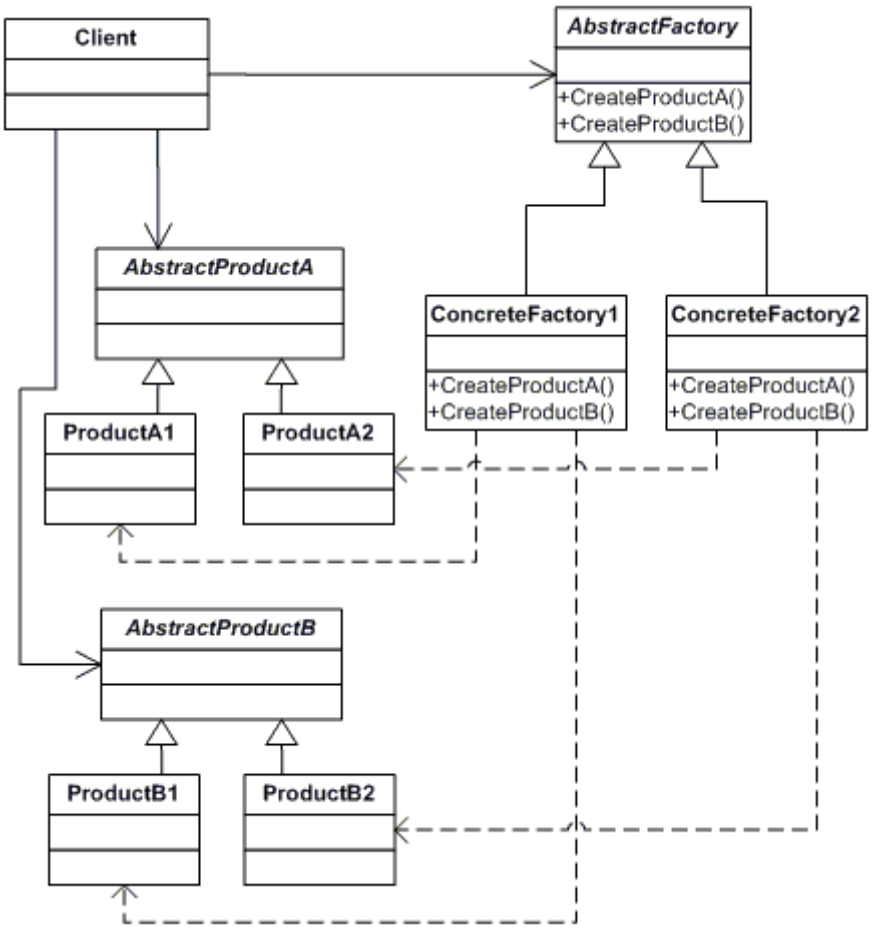
- Fabryka może kontrolować czas życia tworzonych obiektów (zwracając obiekty o różnych czasach życia)
- Fabryka może być przygotowana na rozszerzenia, w ten sposób realizując postulat Open-Closed Principle (przykład z wykładu: Factory + FactoryWorker zamiast „switch")
- W praktyce – zamiast z singletonów i monostates lepiej używać fabryk, są bardziej uniwersalne w implementacji i zapewniają stabilny interfejs dla klienta (w ten sposób fabryka realizuje też postulat **Protected Variations** (Law of Demeter) z GRASP

## 2.4 Factory Method



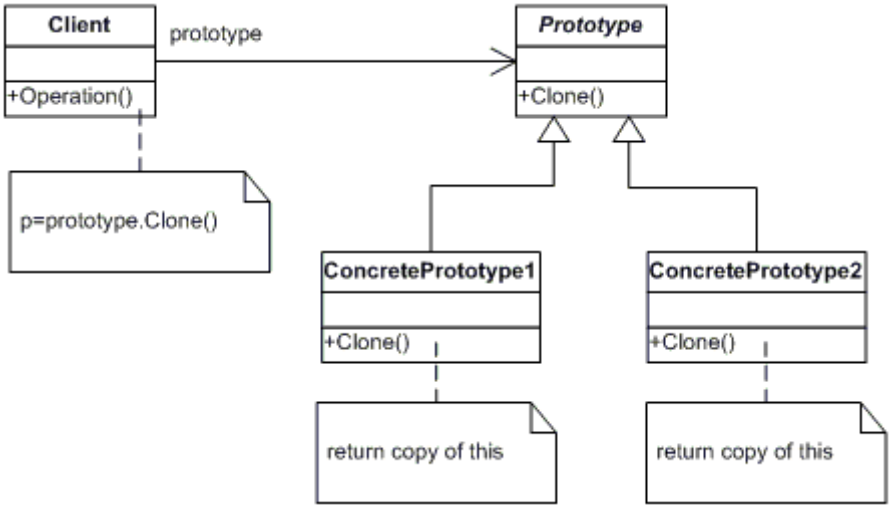

- Delegowanie tworzenia obiektu użytkowego do metody tworzącej, zwykle abstrakcyjnej (`FactoryMethod`)
- Metoda fabrykująca mimo że nie ma implementacji, może już być używana (w `AnOperation`)
- Podklasy dostarczają implementacji metody fabrykującej

## 2.5 Abstract Factory



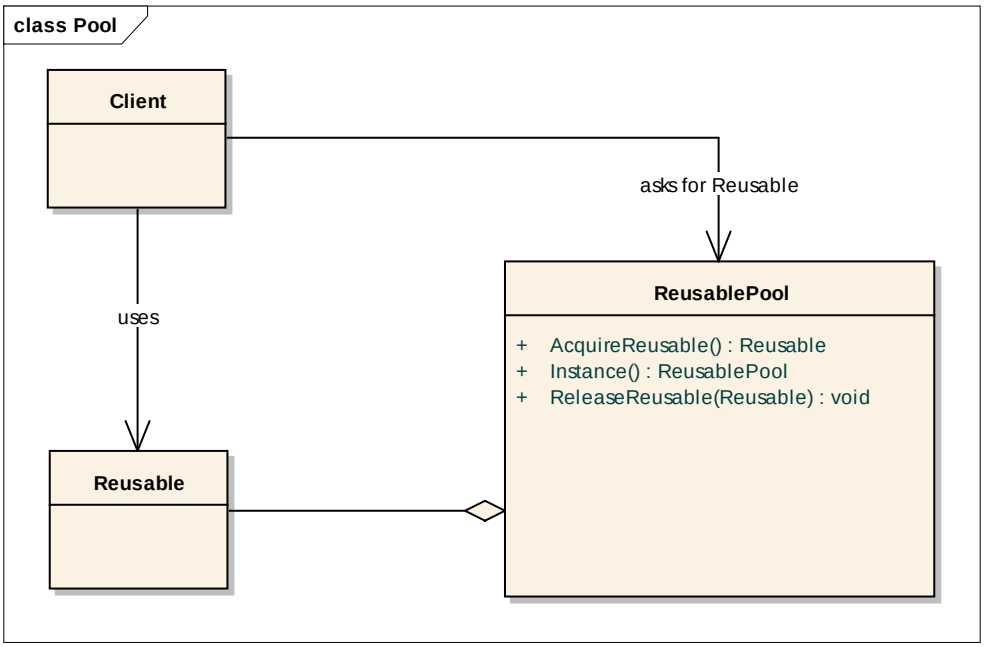

- Nazywany też „tookit”
- Abstrakcyjna fabryka całej rodziny obiektów – klient nie używa abstrakcji ale potrzebuje konkretnej implementacji
- Konstrukcja podobna jak w Factory Metod ale inny zasięg:
    - w FM klasa użytkowa sama realizuje jakieś funkcjonalności, do nich potrzebuje obiektu pomocniczego który sama sobie tworzy, ale na etapie implementacji nie wiadomo jeszcze jak
    - w AF klasa klienta nie tworzy sobie sama obiektu pomocniczego tylko deleguje jego tworzenie do fabryki
    - Creator z FM = Client z AF
- Potrzeba refaktoryzacji FM do AF pojawia się zwykle wtedy, kiedy w klasie implementującej FM pojawia się druga/trzecia (i kolejna) potrzeba wykreowania obiektu pomocniczego o nieznanej implementacji – wtedy wydziela się osobny kontrakt

## 2.6 Prototype




- Istnieje kilka prototypowych instancji obiektów
- Tworzenie nowych polega na kopiowaniu prototypów
- Nie ma znaczenia kto i jak wyprodukował instancje prototypów

## 2.7 Object Pool




- Reużywanie / współdzielenie obiektów które są kłopotliwe w tworzeniu (np. czasochłonne)
- Metoda tworzenia/pobierania obiektu bywa parametryzowana

Implementację rozpoczniemy od zestawu testów

- Próba konstrukcji puli o niewłaściwym rozmiarze powinna powodować wyjątek
- Konstrukcja puli o poprawnym rozmiarze powinna się udać, a pula powinna zwrócić poprawny obiekt
- Pula powinna wyrzucać wyjątek przy próbie pobrania obiektu w sytuacji gdy przekroczono rozmiar

- Obiekt powinno dać się zwrócić do puli i pobrać ponownie
- Nie powinno dać się zwrócić do puli obiektu który do niej nie należy

```
[TestClass]
public class ObjectPoolTests
{
    [TestMethod]
    public void InvalidSize()
    {
        Assert.ThrowsException<ArgumentException>(
            () =>
            {
                var pool = new ObjectPool(0);
            } );
    }

    [TestMethod]
    public void ValidSize()
    {
        var pool = new ObjectPool(1);
        var reusable = pool.AcqureReusable();

        Assert.IsNotNull( reusable );
    }

    [TestMethod]
    public void CapacityDepleted()
    {
        var pool = new ObjectPool(1);
        var reusable = pool.AcqureReusable();

        Assert.ThrowsException<ArgumentException>(
            () =>
            {
                var reusable2 = pool.AcqureReusable();
            } );
    }

    [TestMethod]
    public void ReusedInstance()
    {
        var pool = new ObjectPool(1);
        var reusable = pool.AcqureReusable();

        pool.ReleaseReusable( reusable );

        var reusable2 = pool.AcqureReusable();

        Assert.AreEqual( reusable, reusable2 );
    }

    [TestMethod]
    public void ReleaseInvalidInstance()
    {
        var pool = new ObjectPool(1);

        var reusable = new Reusable();
```

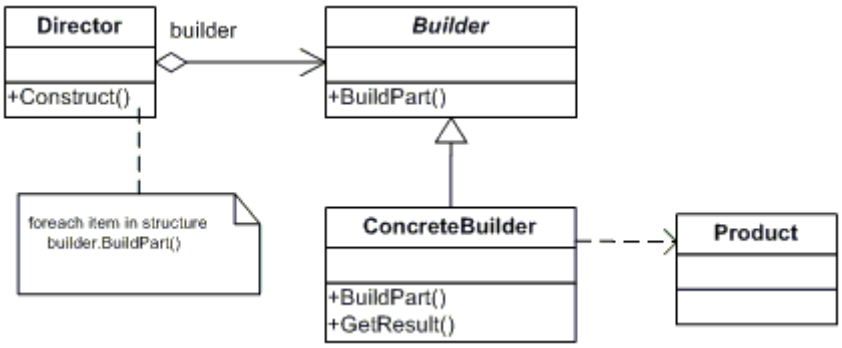
```
        Assert.ThrowsException<ArgumentException>(
            () =>
            {
                pool.ReleaseReusable( reusable );
            } );
    }
}
```

Ten zestaw testów spełnia przykładowa implementacja

```
public class Reusable
{

}

public class ObjectPool
{
    int _poolSize;

    List<Reusable> _pool     = new List<Reusable>();
    List<Reusable> _acquired = new List<Reusable>();

    public ObjectPool( int PoolSize )
    {
        if ( PoolSize <= 0 )
        {
            throw new ArgumentException();
        }

        this._poolSize = PoolSize;
    }

    public Reusable AcqureReusable()
    {
        if ( _acquired.Count() == this._poolSize )
        {
            throw new ArgumentException();
        }

        // tworzenie elementu jeśli pool pusty
        if ( _pool.Count() == 0 )
        {
            var reusable = new Reusable();
            _pool.Add( reusable );
        }

        var element = _pool[0];
        _pool.Remove( element );
        _acquired.Add( element );

        return element;
    }

    public void ReleaseReusable( Reusable reusable )
    {
        if ( !_acquired.Contains( reusable ) )
        {
```

```
            throw new ArgumentException();
        }

        _acquired.Remove( reusable );
        _pool.Add( reusable );
    }
}
```

## 2.8 Builder




- Ukrywanie szczegółów kodu służącego do kreowania obiektu/obiektów
- Ukrywanie wewnętrznej struktury obiektu
- Przykład – `XmlTextWriter`
- Przykład z wykładu: http://www.wiktorzychla.com/2012/02/simple-fluent-and-recursive-tag-builder.html