

WZORCE PROJEKTOWE

SPRAWOZDANIE

ZADANIE DECORATOR KAWA

Patryk Figas
Informatyka, programowanie
Grupa 34_Inf_P_NW_6

1. Cel

Celem niniejszego dokumentu jest przedstawienie rozwiązania zadania polegającego na zaprojektowaniu elastycznego systemu zamawiania kawy, który umożliwi dynamiczne dodawanie dodatków do napoju (takich jak mleko, cukier czy czekolada), bez potrzeby tworzenia wielu wariantów klas.

W ramach ćwiczenia zaprojektowano klasę zarządzającą kolejką zadań drukowania za pomocą „pseudokodu”, diagramu UML i implementacji interfejsu i klasy do programu oraz użycie jej w programie Main.

Zaimplementowano system oparty na wzorcu projektowym **Decorator**, który pozwala na rozszerzanie funkcjonalności obiektów w sposób zgodny z zasadą **Open/Closed**.

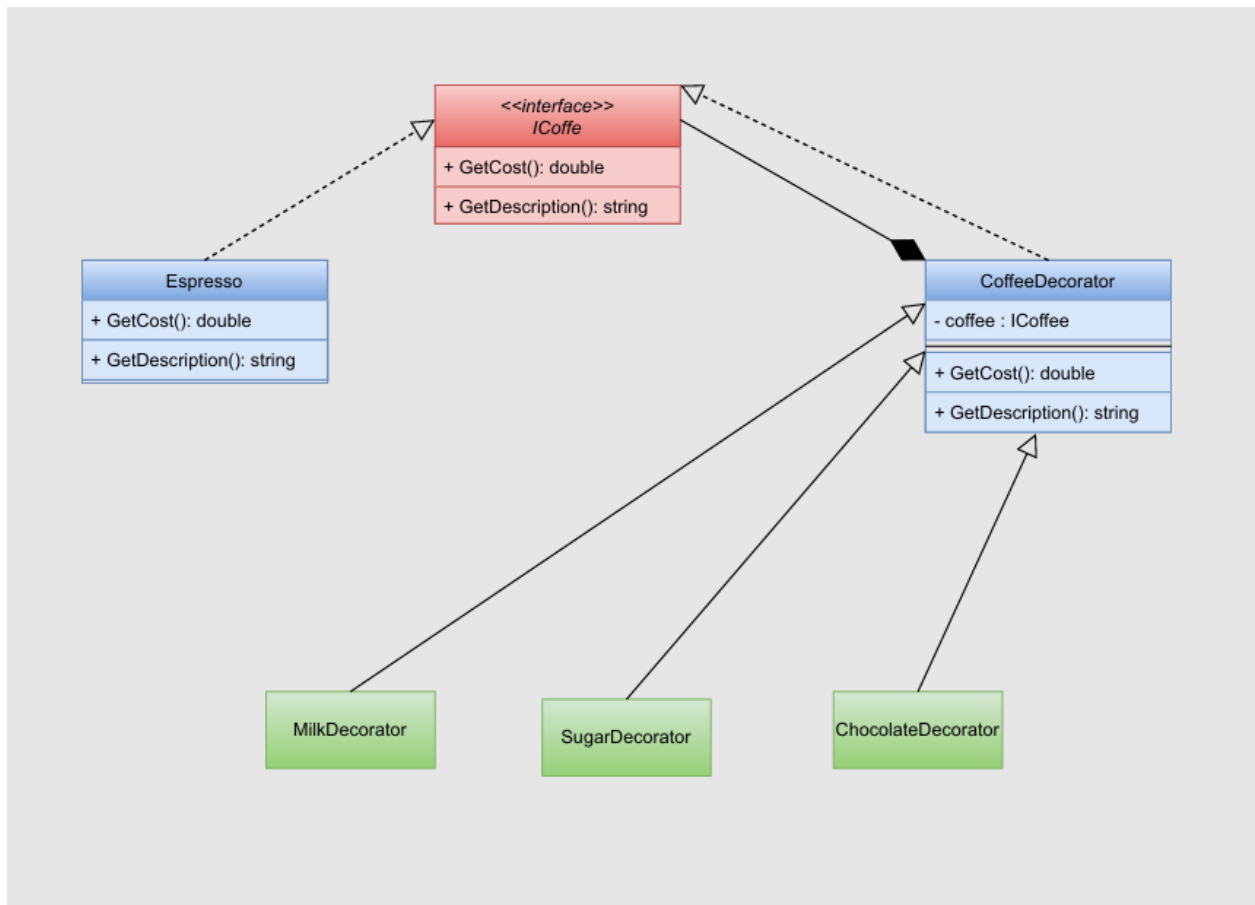
2. Opis rozwiązania

Rozwiązanie opiera się na **interfejsie ICoffee**, który definiuje podstawowe metody: **GetCost()** oraz **GetDescription()**. Klasa Espresso implementuje ten interfejs i reprezentuje bazowy napój.

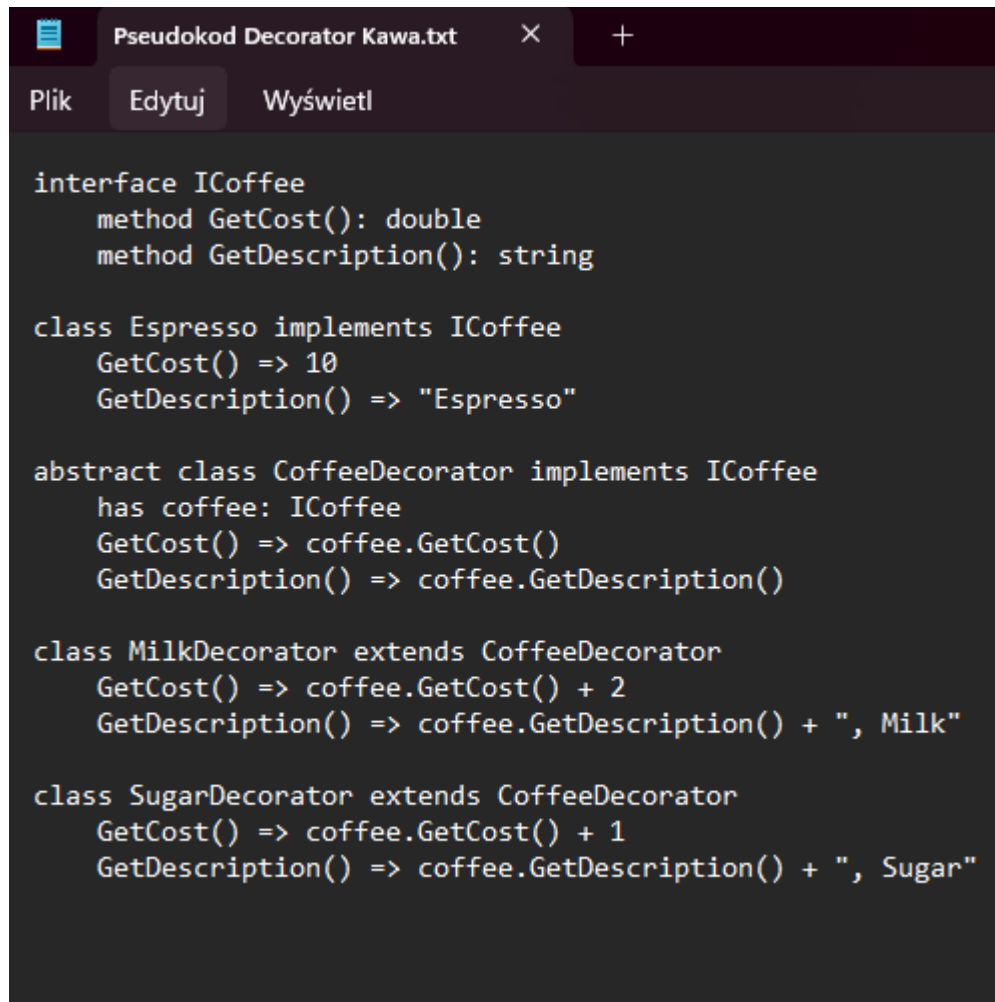
Następnie wprowadzono abstrakcyjną klasę **CoffeeDecorator**, która również implementuje **ICoffee** i przechowuje referencję do innego obiektu typu **ICoffee**. Konkretnie **klasy dekoratorów (MilkDecorator, SugarDecorator, ChocolateDecorator)** rozszerzają **CoffeeDecorator** i modyfikują koszt oraz opis kawy, dodając odpowiednie składniki.

Dzięki temu możliwe jest dynamiczne tworzenie napoju z dowolną liczbą dodatków, bez mnożenia klas i łamania zasad SOLID.

- Propozycja diagramu klas



- Pseudokod



```
interface ICoffee
    method GetCost(): double
    method GetDescription(): string

class Espresso implements ICoffee
    GetCost() => 10
    GetDescription() => "Espresso"

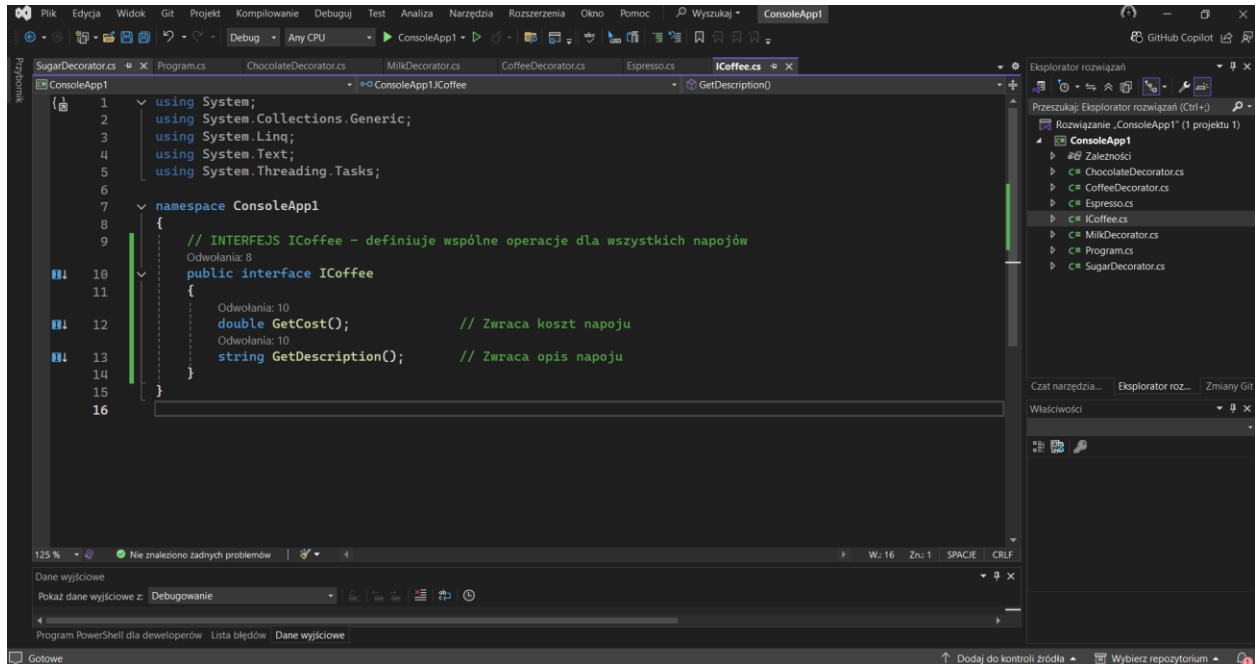
abstract class CoffeeDecorator implements ICoffee
    has coffee: ICoffee
    GetCost() => coffee.GetCost()
    GetDescription() => coffee.GetDescription()

class MilkDecorator extends CoffeeDecorator
    GetCost() => coffee.GetCost() + 2
    GetDescription() => coffee.GetDescription() + ", Milk"

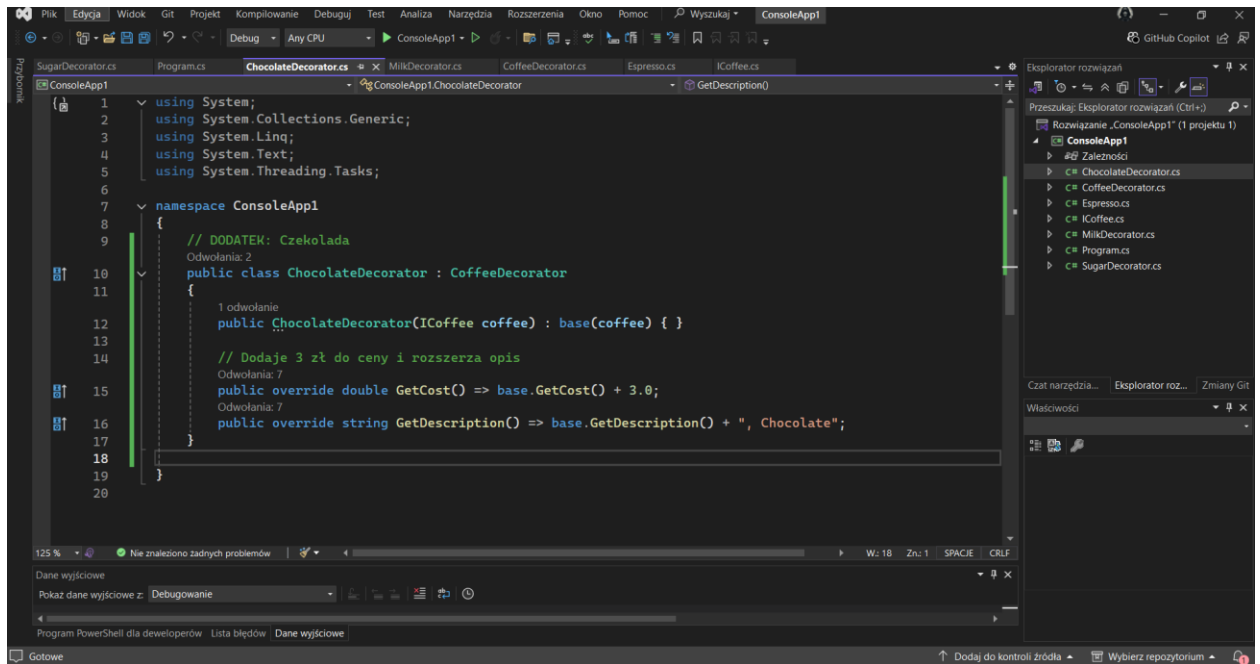
class SugarDecorator extends CoffeeDecorator
    GetCost() => coffee.GetCost() + 1
    GetDescription() => coffee.GetDescription() + ", Sugar"
```

3. Implementacja

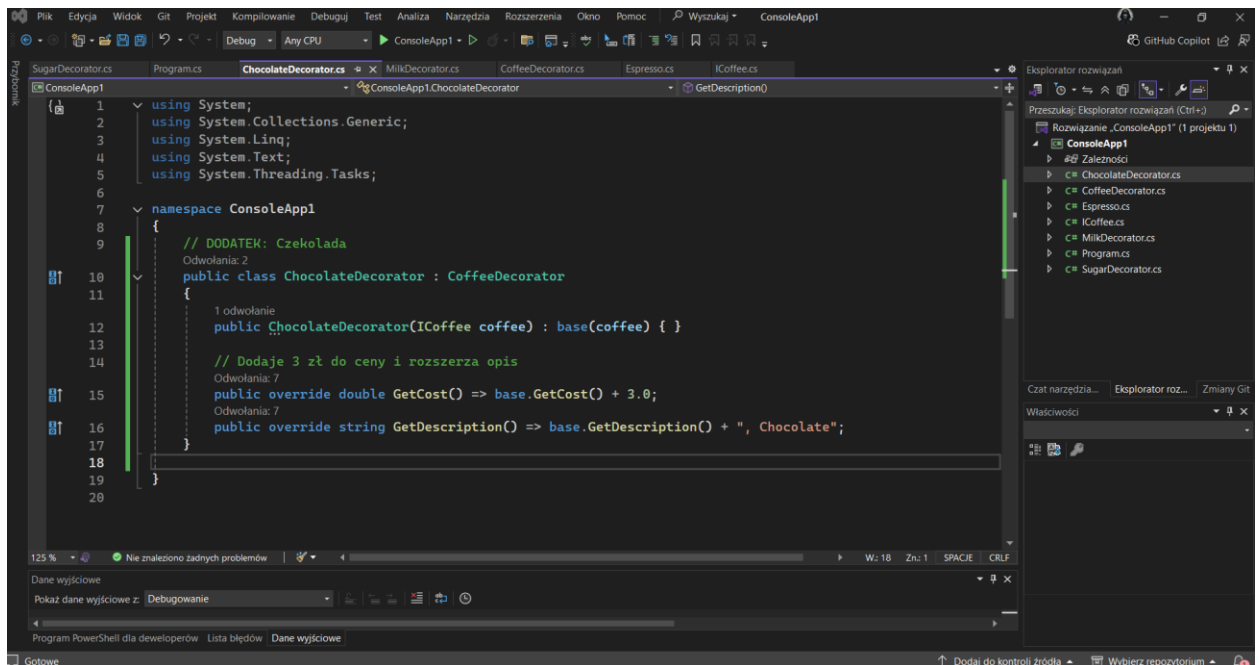
- Kod interfejsu `ICoffee.cs`



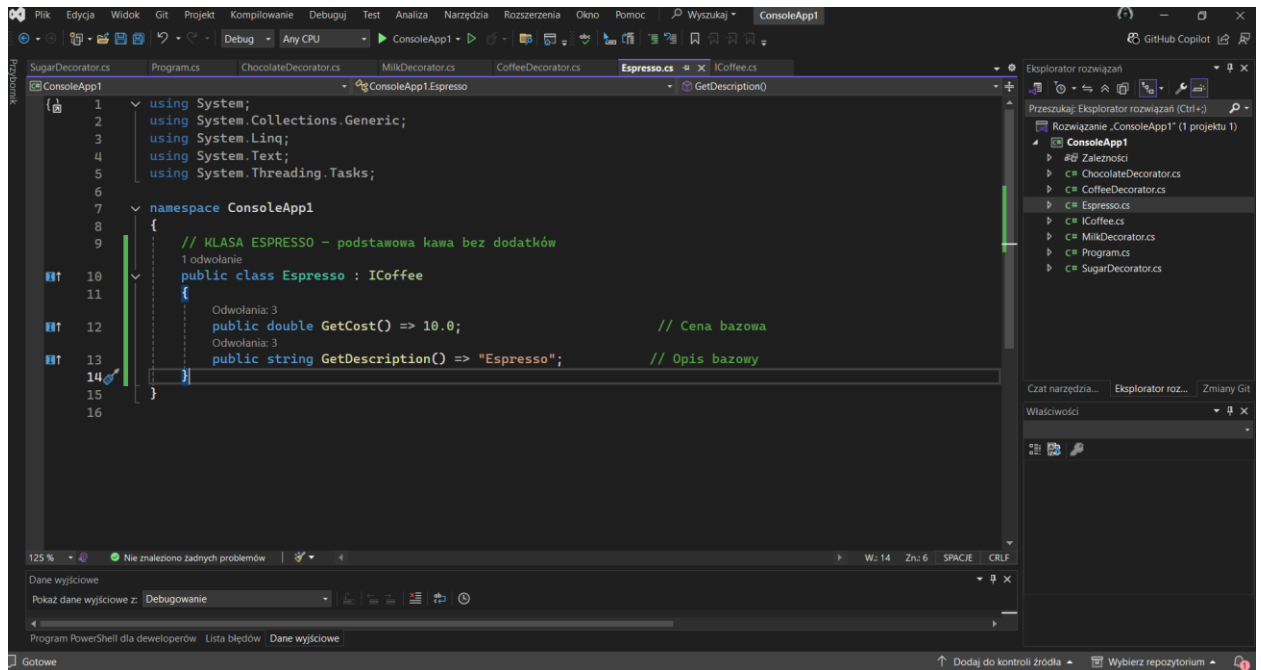
- Kod klasy **ChocolateDecorator.cs**



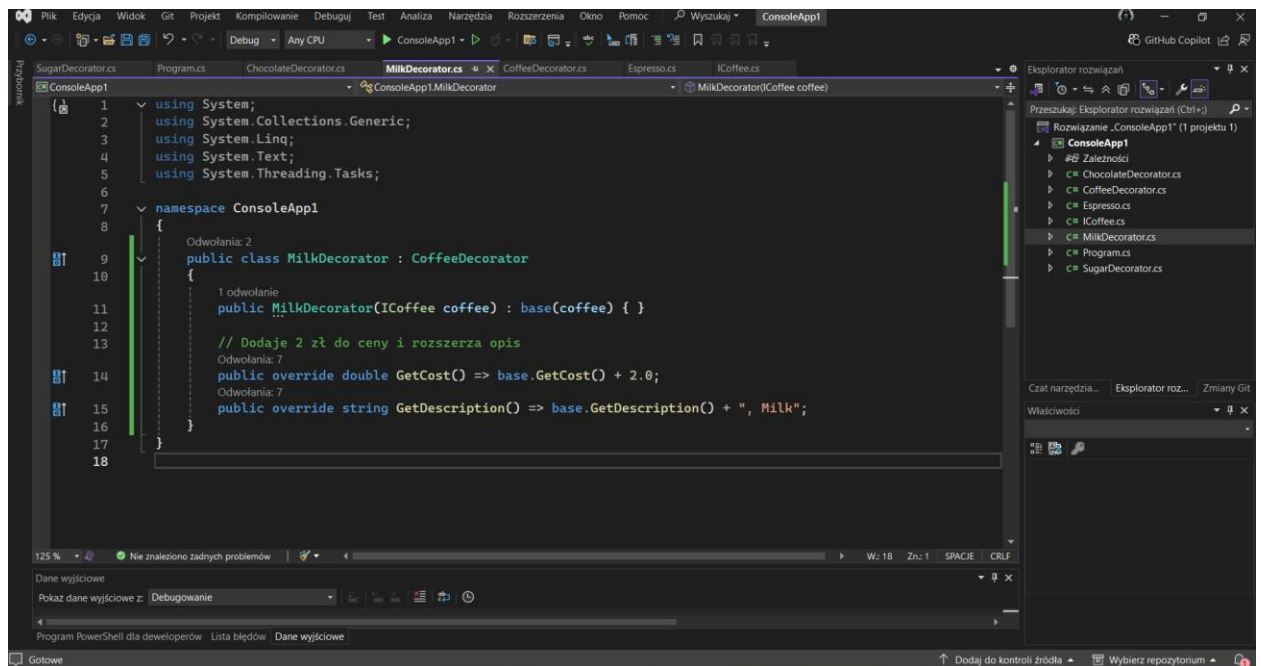
- Kod klasy **CoffeeDecorator.cs**



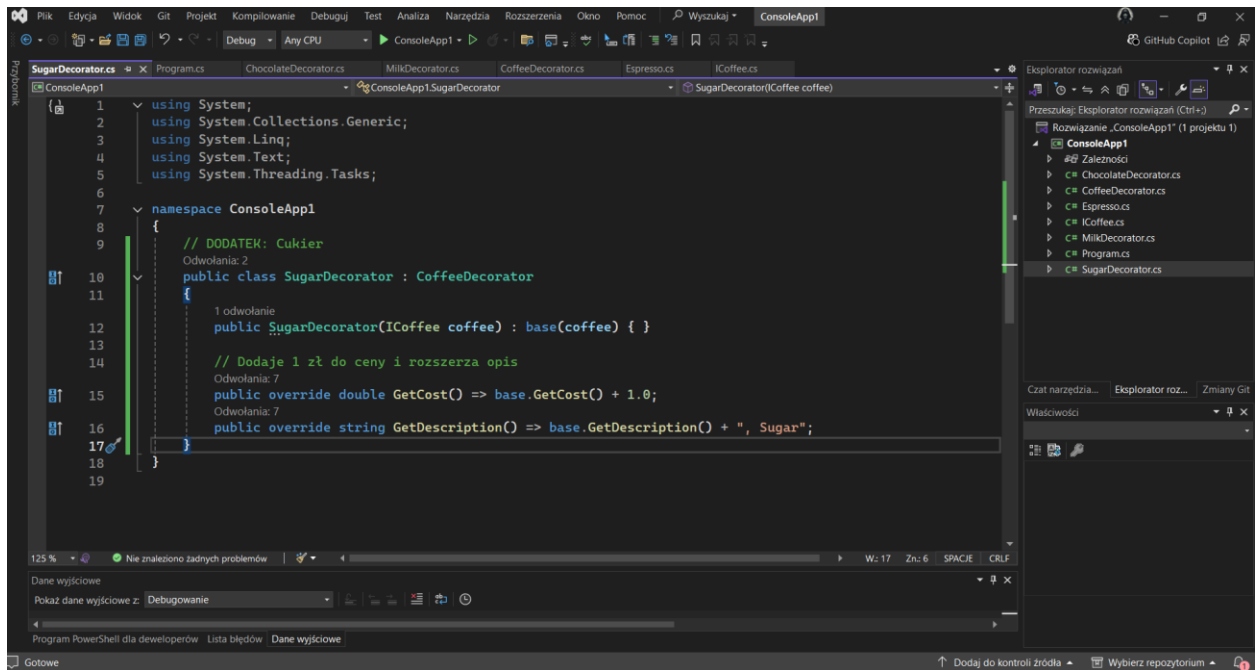
- Kod klasy Espresso.cs



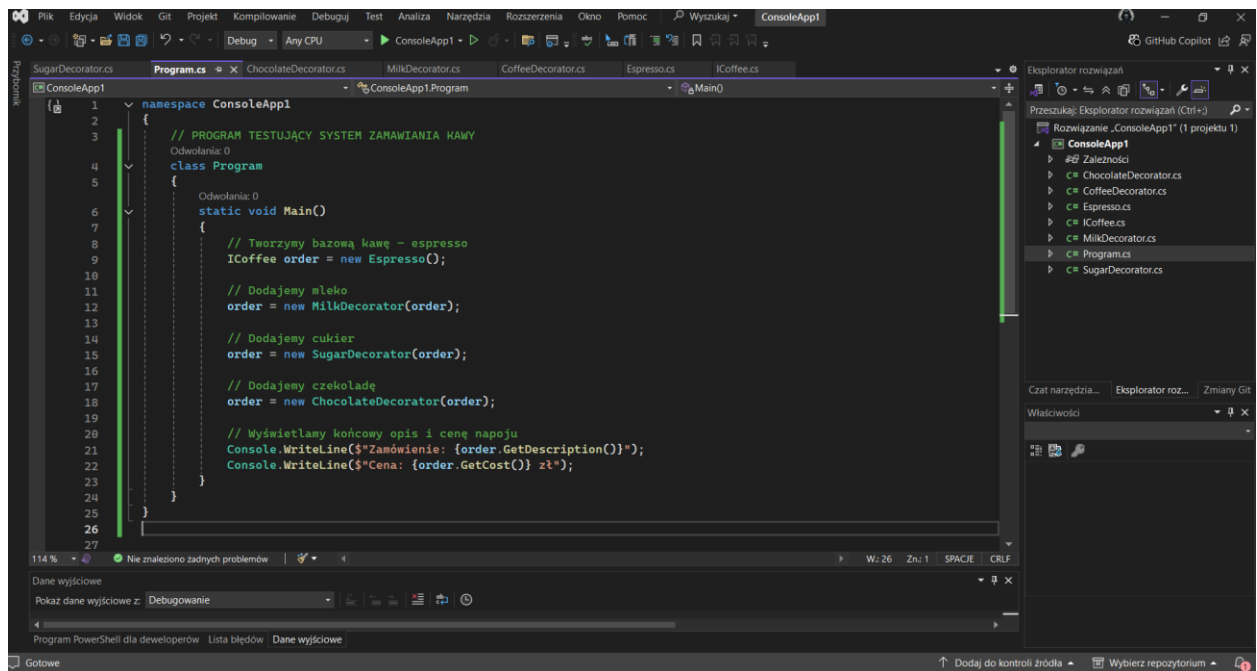
- Kod klasy MilkDecorator.cs



- Kod klasy `SugarDecorator.cs`



- Kod klasy Program.cs

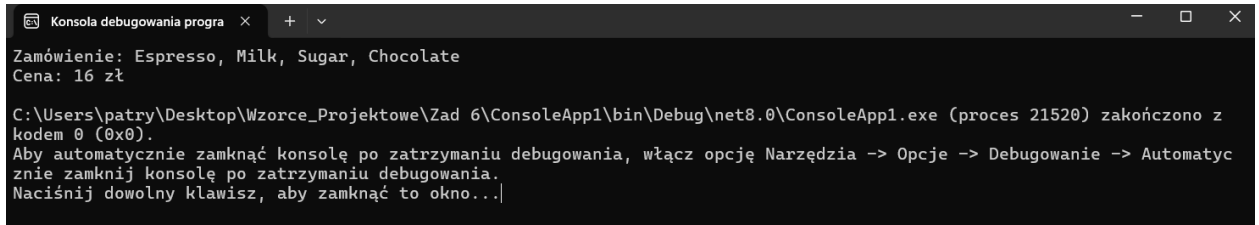


4. Podsumowanie

W zadaniu zastosowano wzorec projektowy **Decorator**, który pozwala na dynamiczne rozszerzanie funkcjonalności obiektów – w tym przypadku: dodawanie składników do kawy. Dzięki niemu nie trzeba tworzyć wielu klas dla każdej możliwej kombinacji dodatków.

Implementacja powiodła się, ponieważ:

- system działa zgodnie z założeniami,
- spełnia zasadę **Open/Closed** (rozszerzanie bez modyfikowania),
- umożliwia łatwe dodawanie nowych dekoratorów.



```
Konsola debugowania progra x + v
Zamówienie: Espresso, Milk, Sugar, Chocolate
Cena: 16 zł

C:\Users\patry\Desktop\Wzorce_Projektowe\Zad 6\ConsoleApp1\bin\Debug\net8.0\ConsoleApp1.exe (proces 21520) zakończono z
kodem 0 (0x0).
Aby automatycznie zamknąć konsolę po zatrzymaniu debugowania, włącz opcję Narzędzia -> Opcje -> Debugowanie -> Automatyc
znie zamknij konsolę po zatrzymaniu debugowania.
Naciśnij dowolny klawisz, aby zamknąć to okno...|
```

Inny wzorec projektowy – np. **Builder** – mógłby się sprawdzić przy bardziej złożonych zamówieniach, np. z konfiguracją przez użytkownika. Jednak dla potrzeb prostego roszszerzania funkcji obiektu **Decorator jest najlepszym wyborem**.

Lista załączników

Repozytorium GITHUB z projektem:

<https://github.com/PatrykFigas/Wzorce-projektowe.git>