Programowanie wizualne

opracował: Wojciech Frohmberg

Lab 7

Zagadnienia do opanowania:

- NUnit
- Test-driven development

Zarys problemu:
Do projektu z poprzednich zajęć chcielibyśmy dodać funkcjonalność równomiernego pod względem czasochłonności podziału nierozdzielnych zadań pomiędzy deweloperów skojarzonego Issue. Problem ten z natury rzeczy jest trudny do rozwiązania w postaci ogólnej (jest jednym z problemów należących do klasy problemów silnie NP-trudnych). Skupimy się zatem na jego podproblemie tj. na problemie podziału zadań między dwóch deweloperów, który wciąż jest NP-trudny, jednak posiada pseudowielomianowy algorytm rozwiązania. Skorzystamy tutaj z metodyki test-driven development tj. spróbujemy opracować testy jednostkowe, które wspomogą nas w implementacji algorytmu. Celem zajęć jest pokazanie, iż pomimo dość skomplikowanej funkcjonalności do zaimplementowania – testy mogą być nie muszą być wcale trudne do napisania!

CZĘŚĆ I Określenie interfejsu algorytmu

W ramach tej części spróbujemy przygotować nasz projekt do dodania funkcjonalności podziału zbioru zadań na podzbiory o równych pracochłonnościach. Dokonamy tego poprzez określenie, w precyzyjny na ile to możliwe sposób, w jaki sposób chcielibyśmy widzieć korzystanie z tej funkcjonalności. Do tego celu w językach obiektowych wysokiego poziomu wykorzystywane są tzw. interfejsy. W interfejsie możemy określić zestaw właściwości oraz metod jeszcze bez implementowania stojących za nimi funkcjonalności. Będą one świadczyły o tym po jakich etykietach będzie można dostać się do tych funkcjonalności.

1. Do utworzonego na poprzednich zajęciach projektu dodaj folder Algorithms.
2. W ramach folderu utwórz interfejs IPartitionAlgorithm sparametryzowany typem T (dla przypomnienia parametryzacja typem odbywa się przy użyciu nawiasów trójkątnych w naszym przypadku zatem interfejs powinien być zadekarowany następująco:

```
public interface IPartitionAlgorithm<T>
{

}
```

   W zamierzeniach za pomocą parametru generycznego T przekażemy informację jaki jest typ elementu kolekcji stanowiącej przedmiot podziału. Zauważmy, że dzięki takiemu uogólnieniu nasza implementacja algorytmu de facto abstrahuje od naszego przypadku użycia i może być wykorzystana gdziekolwiek gdzie będziemy chcieli dokonać równomiernego podziału nierozdzielnych elementów zbioru na podzbiory. Chcemy zatem napisać algorytm na tyle ogólnie, żeby można było go zastosować nie tylko w przypadku podziału nierozdzielnych zadań pomiędzy deweloperów, ale w przypadku dowolnego zbioru elementów, który chcielibyśmy podzielić na podzbiory o równiej sumie określających elementy wag.
3. Do interfejsu dodaj właściwość InterpretAsWeight (z publicznymi elementami get i set), w ramach której przechowasz funktor przyporządkowujący elementowi naszego zbioru –

całkowitoliczbową wartość jego wagi (w naszym przypadku czasochłonności zadania). Możesz przyjąć, że typ zwracany przez funktor to int. Jakiego typu dostępnego ze standardu C# możesz użyć do przechowania funktora? Odpowiedzią jest oczywiście delegacja (delegate). Nie chielibyśmy jednak deklarować delegacji specjalnie dla naszego przypadku, stąd też możemy skorzystać z gotowej generycznej delegacji Func. W tym przypadku konkretnie Func<T, int>.

```
public interface IPartitionAlgorithm<T>
{
    Func<T, int> InterpretAsWeight { get; set; }
}
```

Teraz do celu określenia wagi obiektu „o” typu T skorzystamy po prostu z instrukcji InterpretAsWeight(o).

4. Do interfejsu dodaj właściwość SubsetsCount (z publicznymi elementami get i set), w ramach której przechowasz liczbę podzbiorów, na które ma zostać podzielony zbiór wejściowy przy użyciu algorytmu. Jakiego typu powinna być właściwość?
5. Do folderu Algorithms dodaj folder Exceptions i umieść w nim klasę wyjątku UnsupportedSubsetsCountException dziedziczącą z klasy Exception (klasa może pozostać pusta).

```
public class UnsupportedSubsetsCountException : Exception
{
}
```

Wyrzucenie wyjątku tego typu będzie obrazowało zażądanie by algorytm dokonał podziału elementów zbioru wejściowego na nieobsługiwaną przez algorytm liczbę podzbiorów.

6. Do interfejsu IPartitionAlgorithm dodaj metodę Run, która będzie przyjmowała wejściowy zbiór elementów do podziału w formie IEnumerable<T>. Metoda powinna zwracać List<HashSet<T>> tj. listę podzbiorów, na którą algorytm podzielił nasz zbiór wejściowy:

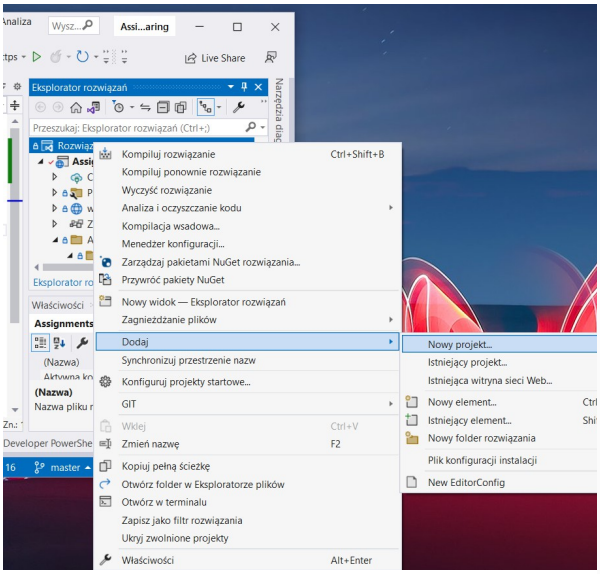
```
List<HashSet<T>> Run(IEnumerable<T> set);
```

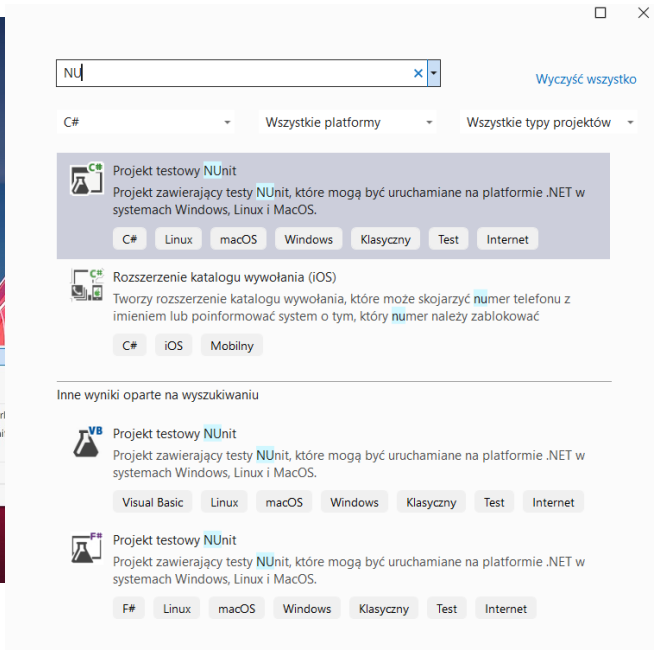
7. Do folderu z wyjątkami dodaj klasę wyjątku InputSetUnsplittableException, która będzie obrazowała sytuację, gdy nie da się dokonać podziału naszego zbioru wejściowego na zadaną liczbę podzbiorów o tych samych sumarycznych wagach. Pamiętaj o wydziedziczeniu jej z klasy Exception.
8. Do folderu dodaj klasę wyjątku InappropriateWeightException, która będzie obrazowała sytuację przekazania w ramach zbioru wejściowego elementu o niewłaściwej wadze (np. ujemnej). Również tutaj zadbaj o wydziedziczenie z klasy Exception.

CZĘŚĆ II Dodanie kodu testującego funkcjonalność

W ramach tej części zanim jeszcze zabierzemy się za implementację funkcjonalności utworzymy testy do niej. Testów powinno być na tyle dużo żeby obejmowały swoim zasięgiem możliwie duży zakres przypadków użycia funkcjonalności. Musimy zatem przemyśleć jakie funkcjonalność którą chcemy przetestować ma przypadki graniczne/brzegowe. Posłużą nam one jako dane wejściowe do kolejnych testów.

9. Do rozwiązania dodaj projekt testowy NUnit i nazwij go





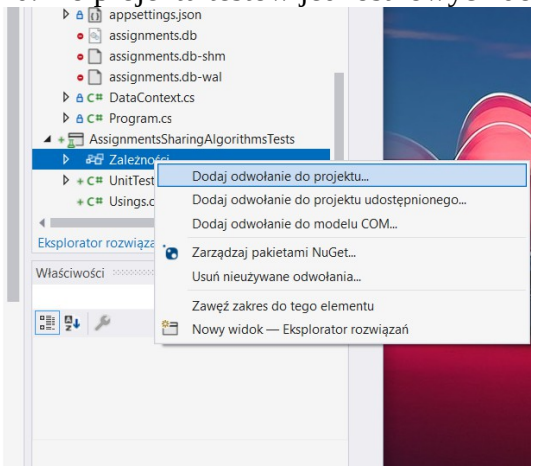
AssignmentsSharingAlgorithmsTests.

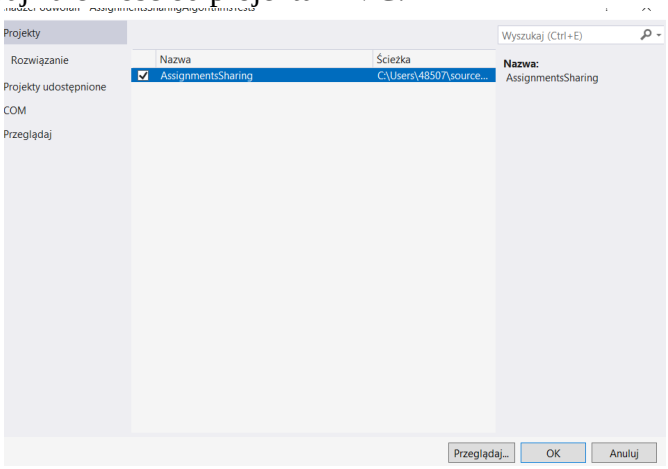
Jeśli korzystasz ze środowiska innego niż VS2022 dodawanie projektu do rozwiązania sprowadza się do komend:

```
dotnet new sln
dotnet new nunit --name <Nazwa projektu>
dotnet sln add <Folder projektu>
```

Przy czym pierwsza komenda tworzy rozwiązanie, druga dodaje nowy projekt typu NUnit, a trzecia dodaje projekt do rozwiązania.

10. Do projektu testów jednostkowych dodaj zależność od projektu MVC.





Zależność określa jednostronną możliwość odnoszenia się do wszystkich elementów zadeklarowanych w zależności. Przy czym nie ma możliwości tworzenia zależności cyklicznych.
W przypadku, gdy korzystasz z innego środowiska niż VS2022 skorzystaj z komendy:

```
dotnet add <Nazwa projektu celu> reference <Nazwa projektu referencji>
```

Pamiętaj żeby wywołując komendę być w folderze rozwiązania.

11. W ramach projektu testów jednostkowych utwórz klasę testową AssignmentsBinaryPartitionAlgorithmsTests sparametryzowaną typem T. W ramach przekazanego parametru będziemy przekazywali algorytm do przetestowania:

```
public class AssignmentsBinaryPartitionAlgorithmsTests<T>
{

}
```

12. Na parametr klasy nałóż ograniczenie spełniania interfejsu IPartitionAlgorithm<Assignment> oraz możliwości utworzenia instancji klasy poprzez domyślny bezparametrowy konstruktor (ograniczenie new()):

    ```
    where T : IPartitionAlgorithms<Assignment>, new()
    ```

    Dzięki temu ograniczeniu po pierwsze użytkownik klasy nie będzie mógł zainicjować parametru generycznego jeśli nie spełnia on wymagań (czyli nie implementuje interfejsu lub też nie ma konstruktora bezparametrowego) po drugie wewnątrz klasy generycznej będziemy w stanie interpretować obiekty parametru generycznego jakby spełniały te założenia tj. będziemy przykładowo w stanie tworzyć nową instancję typu parametru przy użyciu bezparametrowego konstruktora.
13. Utwórz metodę inicjalizacyjną testów. Może być to metoda o dowolnej nazwie określona atrybutem [SetUp]. W jej ramach zawrzyj tworzenie instancji algorytmu i wstawianie jej do prywatnego pola klasy typu IPartitionAlgorithm<Assignment>. Istotnym jest że nie można skorzystać z konstruktora tego interfejsu bezpośrednio, bo najbanalniej w świecie nie można tworzyć obiektów interfejsów, a jedynie klas implementujących ten interfejs. Stąd jedyną opcją jest tutaj utworzenie instancji klasy przekazanej w parametrze generycznym. Na szczęście zabezpieczyliśmy się na taką sytuację i akceptujemy jedynie takie typy, które mają domyślny bezparametrowy konstruktor, a zatem tworzenie obiektu sprowadzi się tutaj do wywołania „new T()”. Pamiętaj również o zainicjalizowaniu właściwości InterpretAsWeight. Żeby to zrobić skorzystaj z wyrażenia lambda konwertującego obiekt typu IPartitionAlgorithm<Assignment> na obiekt typu int np. poprzez:

    ```
    _algorithm.InterpretAsWeight = a => a.TimeCost;
    ```

    Zwróć uwagę, że dzięki podejściu z przekazaną realną implementacją algorytmu w parametrze na tym etapie możesz zupełnie abstrahować od faktycznej klasy podlegającej testom, wystarczy że klasa trzyma się wskazanego, zaproponowanego a priori interfejsu. W teorii zatem możesz przetestować wskazanym kodem wiele alternatywnych implementacji algorytmu.
14. Utwórz zbiór testów, które dla zakresu n większego od 2 i mniejsze lub równego 11 (z krokiem 1) zweryfikuje czy próba ustawienia pod właściwość SubsetsCount wartości n skutkuje wyrzuceniem wyjątku. Wyrzucenie wyjątku jest tutaj pożądane z racji, że testujemy funkcjonalność binarnego podziału zbioru.
15. Utwórz zbiór testów, w ramach których testowane będzie wyrzucanie wyjątku w przypadku przekazania do zbioru wartości niedodatniej (być może jako pojedynczy element spośród wielu dodatnich). Posłuż się do tego celu atrybutem TestCaseSource, który powinien odpowiadać za wygenerowanie (w pętli) 100 zbiorów z losowymi elementami (niekoniecznie niedodatnimi). Każdy spośród zbiorów ponienien posiadać 3 elementy. W ramach testu skorzystaj z klasy Assume, żeby zweryfikować czy test dla podanego zbioru wejściowego powinien być uruchomiony tj. czy posiada przynajmniej jedną wartość niedodatnią.

16. Utwórz zbiór 5 testów ze zbiorem, elementów którego jesteś pewna/pewny, że nie da się podzielić. Np. Zbiór jednoelementowy. Zadbaj by każdy spośród podanych zbiorów posiadał inne właściwości, które chcesz przetestować. Do przeprowadzenia testów skorzystaj z atrybutu TestCase umieszczonego na metodzie testującej wielokrotnie. Zbadaj czy w ramach metody Run zostanie wyrzucony wyjątek braku możliwości podziału zbioru na podzbiory.
17. W ramach klasy testującej utwórz metodę generującą losowy zbiór, który na pewno da się podzielić na dwa podzbiory o równej sumie. Niech metoda przyjmuje w parametrze pożądaną sumę podzbioru oraz wartość maksymalną pojedynczego elementu. Metoda powinna w dwukrotnej iteracji dodawać do wynikowego zbioru losowe elementy typu Task o czasochłonności nieprzekraczającej maksymalnego elementu przekazanego w parametrze, jednak w taki sposób żeby suma tego elementu i poprzednich nie przekraczała pożądanej sumy podzbioru. W tym przypadku, gdy przekroczy – element przed dodaniem elementu należy zmniejszyć jego czasochłonność tak by dopełnił sumę.
18. Skorzystaj z metody do wygenerowania 50 testów weryfikujących poprawność działania algorytmu, zadbaj by przed uruchomieniem testu liczby zostały „przetasowane”.
19. Zaimplementuj algorytm partycjonowania BinaryPartitionAlgorithm sparametryzowany typem T pojedynczego elementu zbioru do podziału. Niech algorytm implementuje interfejs IPartitionAlgorithm<T>. Do celu implementacji algorytmu możesz skorzystać z zewnętrznych źródeł (np. https://www.geeksforgeeks.org/partition-a-set-into-two-subsets-such-that-the-difference-of-subset-sums-is-minimum/). W międzyczasie implementacji weryfikuj co jakiś czas, które spośród testów jednostkowych dla klasy algorytmu zaczynają przechodzić pozytywnie. Do tego celu skorzystaj z odpowiedniego atrybutu (TestFixture, za pomocą którego przekażesz BinaryPartitionAlgorithm do generycznego parametru testu).

Jeśli używasz środowiska innego niż VS2022 do przeprowadzania testów możesz używać komendy:

```
dotnet test
```