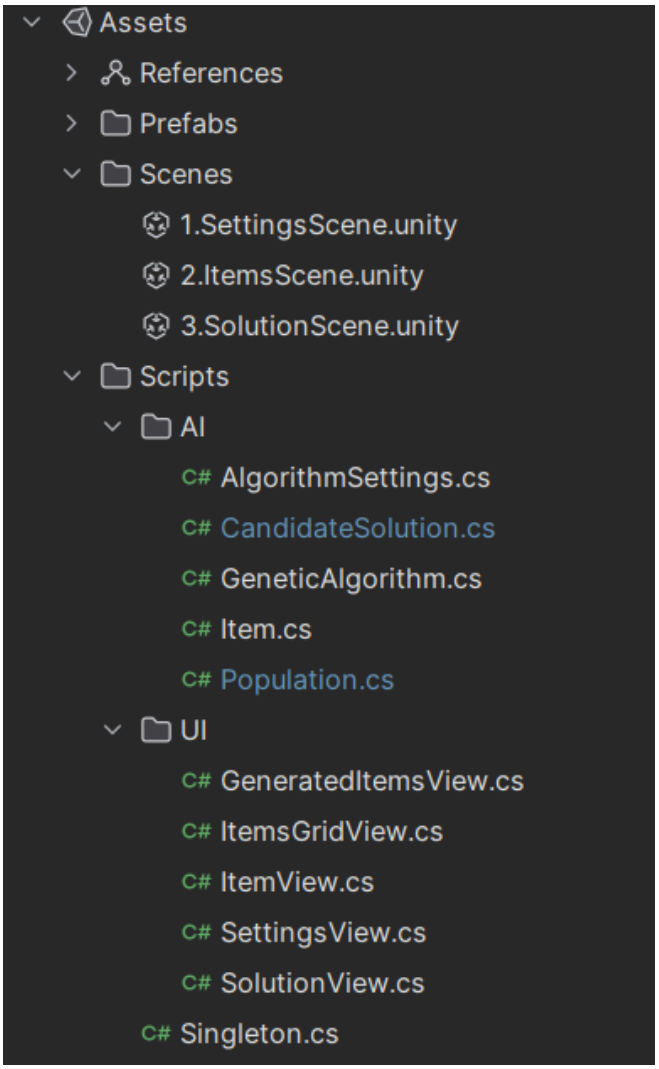
Specyfikacja projektu

Repozytorium: https://github.com/swrczk/GeneticAlgorithm-KnapsackProblem
Demo: https://swrczk.itch.io/knapsack-problem

Projekt został zaimplementowany w języku C# na silniku Unity. Kod źródłowy oraz demo są dostępne w wyżej wymienionych linkach.

W implementacji całkowicie odizolowałam logike UIową od logiki algorytmu, przez co w raporcie skupie się głównie na implementacji algorytmu genetycznego. Na poniższym screenie widać podział po samej strukturze projektu.



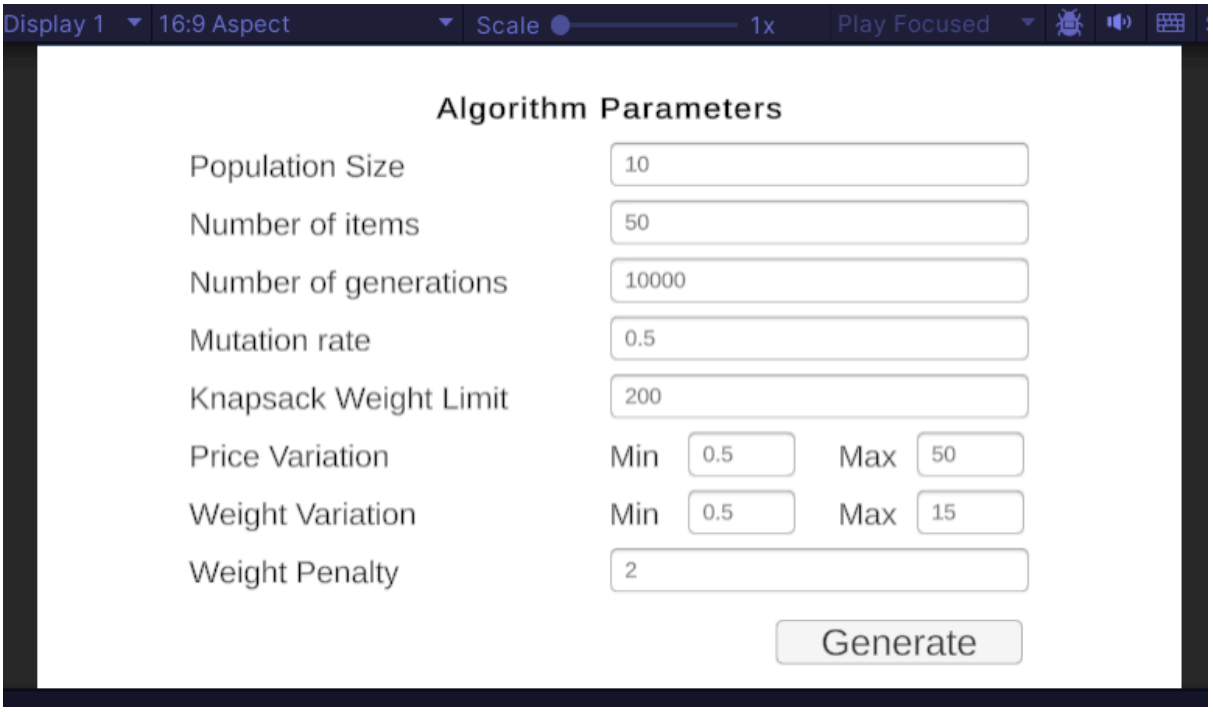
W algorytmie genetycznym operuje na losowo wygenerowanych przedmiotach. Wszystkie nie losowe parametry znajdują się w statycznej klasie *AlgorithmSettings*. Dostęp do wartości można uzyskać z kodu lub za pomocą interfejsu graficznego.

```
public static class AlgorithmSettings
{
    public static int PopulationSize = 10;
    public static int NumberOfItems = 50;
    public static int NumberOfGenerations = 10000;
    public static float MutationRate = 0.5f;
    public static float WeightLimit = 200;
    public static float MinWeight = 0.5f;
    public static float MaxWeight = 15f;
    public static float MinPrice = 0.5f;
    public static float MaxPrice = 50f;
    public static float WeightPenalty = 2f;

    public static readonly List<Item> Items = new List<Item>();
    public static float AllItemsTotalWeight = 0;
    public static float AllItemsTotalPrice = 0;
```

Opis programu

Na początku programu możemy dostosować zmienne dla algorytmu. To tutaj możemy dostosować parametry generowania przedmiotów, limit wagi plecaka, ilość osobników w poouacji, ilosc generacji, itp. Oczywiście są już podabe defaultowe wartości, na których można sprawdzić działanie programu.



Po naciśnięci przycisku *Generate* zostają wygenerowane przedmioty do naszego problemu. Możemy je podejrzeć za pomocą przewijalnej listy. W podsumowaniu w górnej częsci ekranu widzimy sume wartości i wagi wszystkich przedmiotów.

Po przejściu do następnego ekranu wydzimy wynik końcowy działania algorytmu. Za pomocą przycisków w górnej częsci ekranu możemy przełacząc się między trzema najlepszymi rozwiązaniami zostatniej generacji. Widzimy, które przedmioty został zabrane, a które nie (te wykluczone są przekreślone i są zaznaczone fioletowym tłem) oraz podsumowanie rozwiązania, czyli wagę i wartość plecaka. Możemy także użyć pierwszego przycisku, aby podejrzeć wszystkie przedmioty. Po przyciśnięciu *Try Again* zostaniemy przeniesieni do pierwszego ekranu z wyborem parametru, a wynik algorytmu zostaje utracony.

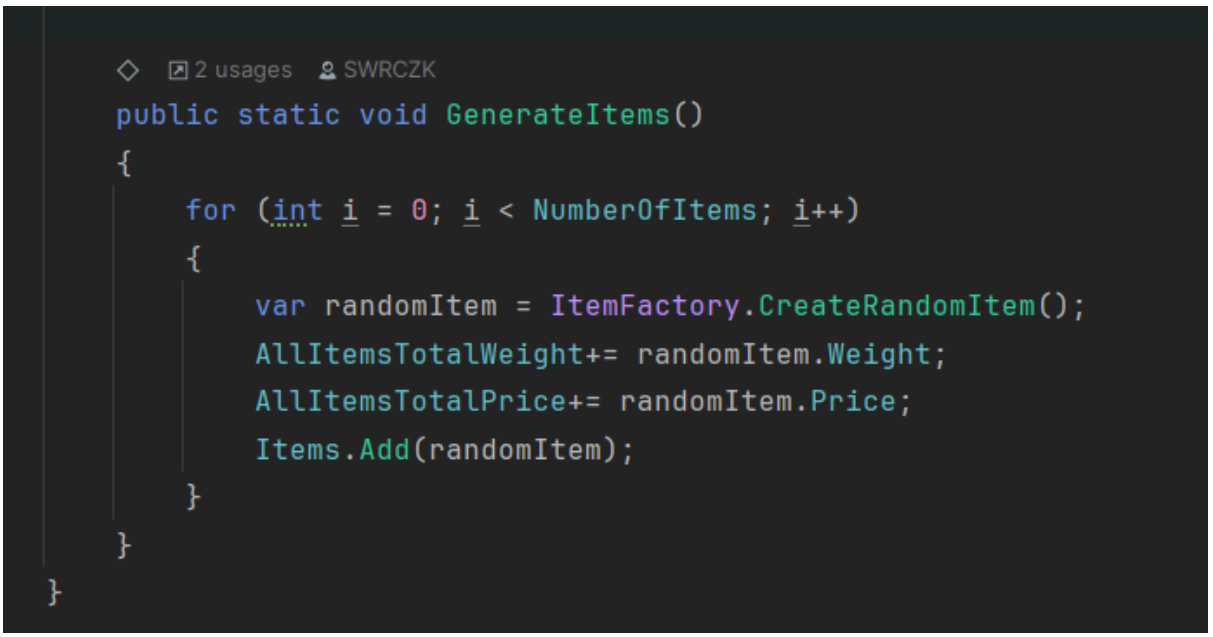
Implementacja Algorytmu

Reprezentacja rozwiązania:

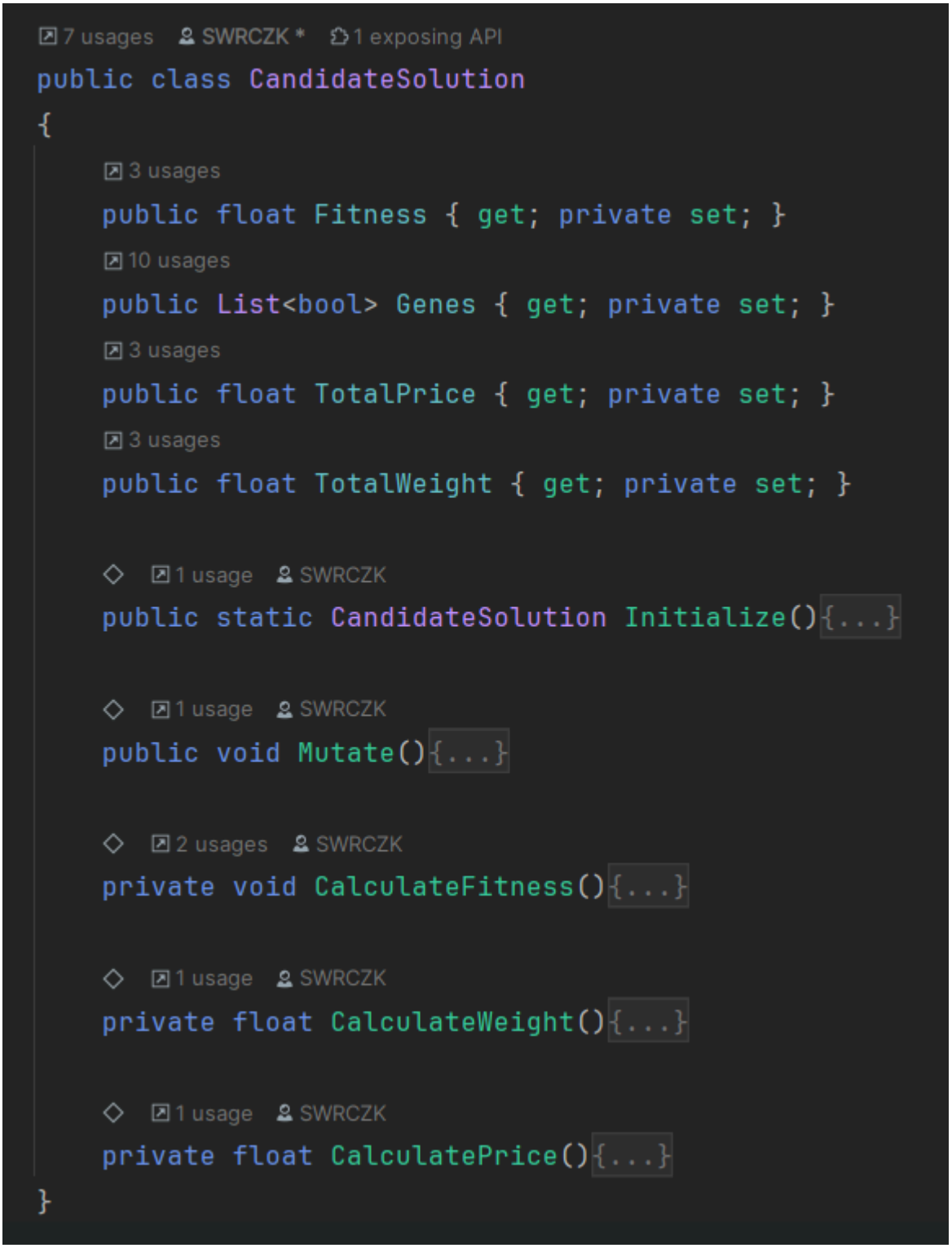
Na potrzeby algorytmu operuje na klasie *Item*, która przechowuje informacje dotyczące wagi i wartości przedmiotu:

```
public class Item
{
    public float Weight { get; set; }
    public float Price { get; set; }
}
```

Natomiast informacje dotyczące ogólnych założeń, tj. zakres możliwych wag, zakres wartości i limit wagowy plecaka zostają zdefiniowane w klasie *AlgorithmSettings*. Do rozwiązania problemu potrzebujemy liste przedmiotów, którą generuje w sposób losowy. Każdy obiekt klasy *Item* jest tworzony za pomocą *ItemFactory*, który przypisuje mu właściwości z zakresu wagi i wartości zdefiniowanych w klasie statycznej *AlgorithmSettings*.



Klasa *CandidateSolution* zawiera opis rozwiązania, czyli przedmiotów wybranych do plecaka. Lista wartości logicznych *Genes* określa, czy przedmiot o danym indeksie zostal zabrany do plecaka. Klasa zawiera też podstawowe informacje potrzebne do algorytmu, tj. wartość fitness, suma wartości, suma wag.



Populacja jest reprezentowana przez klasę *Population*, która zawiera zbiór rozwiązań (*CandidateSolution*) oraz podstawowe funkcje ułatwiające iterowanie po rozwiązaniach.

```
4 usages  SWRCZK *  2 exposing APIs
public class Population
{
    5 usages
    public List<CandidateSolution> Solutions { get; private set; } = new List<CandidateSolution>();

    1 usage  SWRCZK
    public void Mutate()
    {
        Solutions.ForEach(solution => solution.Mutate());
    }

    2 usages  SWRCZK
    public void SortByFitness(){...}

    1 usage  SWRCZK
    public static Population Initialize()
    {
        var population = new Population();
        population.Solutions = new List<CandidateSolution>(AlgorithmSettings.PopulationSize);
        for (int i = 0; i < AlgorithmSettings.PopulationSize; i++)
        {
            population.Solutions.Add( item: CandidateSolution.Initialize());
        }

        return population;
    }
}
```

Inicjalizacja populacji:

Po zainicjowaniu losowych przedmiotów tworzona jest pierwsza populacja, która zawiera zbiór rozwiązań w ilosci zdefiniowanych w *AlgorithmSettings.PopulationSize*. Każde rozwiązanie decyduje, czy przedmiot zostanie zabrany z prawdopodobiestwime *AlgorithmSettings.MutationRate*. Dopuszczalne jest przekroczenie wagi plecaka (*AlgorithmSettings.WeightLimit*). Dla kadego rozwiązania przeliczana jest fukncja *Fitness*, a następnie są sortowane malejąco według otrzymanych wartości. W ten sposób otrzymujemy pierwszą populacje.

```
public void SortByFitness(){...}

1 usage  SWRCZK
public static Population Initialize()
{
    var population = new Population();
    population.Solutions = new List<CandidateSolution>(AlgorithmSettings.PopulationSize);
    for (int i = 0; i < AlgorithmSettings.PopulationSize; i++)
    {
        population.Solutions.Add(item: CandidateSolution.Initialize());
    }

    return population;
}
```

Funkcja oceny (fitness):

Funkcja fitness obliczana jest w klasie *CandidateSolution*. Jest to suma wartości wszystkich zabranych przedmiotów, pomniejszona przez przekroczoną wagę plecaka (*TotalWeight-AlgorithmSettings.WeightLimit*) razy wartość karty za przekroczenie (*AlgorithmSettings.WeightPenalthy*). W ten sposób wraz z przyrostem nadmiarowej wagi algorytm będzię w sposób przyrastajacym karany za przekroczenie limitu.

```
2 usages  SWRCZK
private void CalculateFitness()
{
    TotalWeight = CalculateWeight();
    TotalPrice = CalculatePrice();
    var overweight :float = Mathf.Min( a: 0,  b: TotalWeight - AlgorithmSettings.WeightLimit);
    Fitness = TotalPrice - AlgorithmSettings.WeightPenalty * overweight;
}
```

Mutacja:

Losowo wprowadź mutacje w osobnikach. Mutacje polegają na zmianie losowych bitów w genotypie z prawdopodobieństwem *AlgorithmSettings.MutationRate*. Po każdej operacji mutacji przeliczana jest wartośc funkcji fitness.

```
1 usage  SWRCZK
public void Mutate()
{
    for (int i = 0; i < Genes.Count; i++)
    {
        if (Random.value < AlgorithmSettings.MutationRate)
        {
            Genes[i] = !Genes[i];
        }
    }

    CalculateFitness();
}
```

Warunek zakończenia:

Po inicjalizacji populacji algorytm iteruje po niej określoną ilosc zary zdefiniowaną w *AlgorithmSettings.NumberOfGenerations*. Każda iteracja polega na przeprowadzeniu opracji mutacji na wszystkich rozwiązaniach, przeliczeniu fukcji oceniającej i posortowania rozwiązań.

```
2 usages  SWRCZK
public static class GeneticAlgorithm
{
    5 usages
    public static Population CurrentPopulation { get; set; }

    1 usage  SWRCZK
    public static void Run()
    {
        if (AlgorithmSettings.Items.Count == 0)
            AlgorithmSettings.GenerateItems();
        CurrentPopulation = Population.Initialize();
        CurrentPopulation.SortByFitness();
        for (int i = 0; i < AlgorithmSettings.NumberOfGenerations; i++)
        {
            CurrentPopulation.Mutate();
            CurrentPopulation.SortByFitness();
        }
    }
}
```

Wybór najlepszego rozwiązania:

Ponieważ rozwiązania są posortowane według wartości funkcji fitness, ich kolejność decyduje o jakości rozwiązania. Na interfejsie graficznym przedstawione są szczegóły trzech najlepszych rozwiązań.