Platformy programistyczne .Net i Java

Projekt aplikacji okienkowej Java na przykładzie prostego problemu optymalizacyjnego

Mateusz Marciak 272599

Opis Projektu

Projekt opierał się na rozwiązaniu nieograniczonego problemu plecakowego w języku Java. Nieograniczony problem plecakowy różni się od zwykłego tym, że algorytm może zapełnić zadaną pojemność przedmiotami o dowolnej krotności np. wybrać jeden przedmiot 10 razy.

Następnie do zaimplementowanego rozwiązania należało utworzyć testy jednstokwe.

Implementacja

Pierwszym korkiem było zdefiniowanie obiektu **Item,** w którym to należało zdefiniować wartość oraz wagę, przedmiotu. Następnie zdefiniowane metodę, która wylicza stosunek tych dwóch wartości i go zwraca. Stosunek ten będzie wykorzystywany jako wyznacznik opłacalności danego przedmiotu.

```
public class Item {  6 usages
    public int value;  6 usages
    public int weight;  8 usages

    public Item(int value, int weight) {  1 usage
        this.value = value;
        this.weight = weight;
    }

    public double getRatio() {  2 usages
        return (double) value / weight;
    }

    @Override
    public String toString() {
        return String.format("Item[v=%d, w=%d]", value, weight);
    }
}
```

Rysunek 1 Implementacja przedmiotu

Kolejnym krokiem, była definicja klasy **Problem.** Klasa ta jest odpowiedzialna za wygenerowanie instancji przedmiotów, w sposób losowy.

Konstruktor klasy przyjmuje ilość przedmiotów, seed losowania oraz ograniczniki dolny i górny dla losowanych wartości. Wartości i wagi przedmiotów są losowane przez metodę **nexInt()**, która to przyjmuję ograniczniki jako argumenty – i zwraca losową liczbę typu int z przyjętego zakresu.

```java
public class Problem {  10 usages
    public List<Item> items;  8 usages

    public Problem(int n, int seed, int lower, int upper) {  5 usages
        Random rand = new Random(seed);
        items = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            int value = lower + rand.nextInt( bound: upper - lower + 1);
            int weight = lower + rand.nextInt( bound: upper - lower + 1);
            items.add(new Item(value, weight));
        }
    }
```

Rysunek 2 Implementacja problemu

Ostatecznie napisano klasę **Result**, której metoda **solve** zwraca rozwiązanie. Działanie tej metody opiera się na posortowaniu obliczonych wcześniej stosunków i wybraniu tego co jest. Wybrany przedmiot jest umieszczany w pojemności tyle razy ile się zmieści, w momencie gdy już nie ma dla niego więcej miejsca, algorytm zachłanny wybiera następny optymalny przedmiot.

```java
public Result solve(int capacity) {  3 usages
    items.sort(( Item a,  Item b) -> Double.compare(b.getRatio(), a.getRatio()));
    Result result = new Result();
    for (Item item : items) {
        int count = capacity / item.weight;
        if (count > 0) {
            result.solution.put(item, count);
            result.totalWeight += item.weight * count;
            result.totalValue += item.value * count;
            capacity -= item.weight * count;
        }
    }
    return result;
}
```

Rysunek 3 Implementacja wyniku

Rezultaty

Po uruchomieniu działanie aplikacji wygląda tak:

```
Podaj liczbę przedmiotów: 5
Podaj ziarno losowania (seed): 42
Podaj pojemność plecaka: 25
Wygenerowane przedmioty:
  1. Item[v=1, w=4]
  2. Item[v=9, w=5]
  3. Item[v=1, w=6]
  4. Item[v=6, w=9]
  5. Item[v=10, w=4]

Wybrane przedmioty:
  Item[v=10, w=4] - wybrany 6 razy
Total weight: 24
Total value: 60
```

Rysunek 4 Rezultat końcowy

Testy jednostkowe

Dla naszego problemu napisano 4 testy:

- Czy liczba przedmiotów się zgadza z wartością oczekiwaną
- Czy wartości i wagi mieszczą się w podanym przedziale
- Czy plecak nie jest pusty
- Czy wartości i wagi będą zerowe, kiedy plecak jest pusty

```java
public class ProblemTest {
    @Test
    public void testItemGenerationCount() {
        Problem p = new Problem( n: 10,  seed: 42,  lower: 1,  upper: 10);
        assertEquals( expected: 10, p.items.size());
    }

    @Test
    public void testValueRange() {
        Problem p = new Problem( n: 10,  seed: 42,  lower: 1,  upper: 10);
        for (Item i : p.items) {
            assertTrue( condition: i.value >= 1 && i.value <= 10);
            assertTrue( condition: i.weight >= 1 && i.weight <= 10);
        }
    }

    @Test
    public void testSolutionNotEmpty() {
        Problem p = new Problem( n: 5,  seed: 42,  lower: 1,  upper: 10);
        Result r = p.solve( capacity: 20);
        assertFalse(r.solution.isEmpty());
    }

    @Test
    public void testSolutionEmptyForZeroCapacity() {
        Problem p = new Problem( n: 5,  seed: 42,  lower: 1,  upper: 10);
        Result r = p.solve( capacity: 0);
        assertEquals( expected: 0, r.totalWeight);
        assertEquals( expected: 0, r.totalValue);
    }
}
```

Rysunek 5 Implementacja testów