

---

# Sprawozdanie

---

PROBLEM GENEROWANIA STOŁÓWKI

ALGORYTMY I STRUKTURY DANYCH

JAVA

MICHAŁ KOLENDO

NR INDEKSU 286771

26 STYCZEŃ, 2017 BARTEK KRÓLAK

NR INDEKSU 284922

26 STYCZEŃ, 2018

## 1 Diagram Klas

✓	CanteenTest	530ms
✓	GetAlgorithmSetting	505ms
✓	SetWalls	13ms
✓	ValidateWallsPerimeters	0ms
✓	GetCost	12ms

Rysunek 1: *Diagram Klas programu Time2Dine*

Z powodu wielkości diagramu obejmującego cały projekt wraz z metodami oraz polami, został on zamieszczony w załączniku do projektu pod nazwą **Pełny Diagram UML programu Eggcelent**

## 2 Opis zmian w klasach

### Pakiet Model

#### 2.1 Chromosome

Moduł `Chromosome` uległ średnim zmianom. Zrezygnowaliśmy z przechowywania informacji binarnej na temat rozłożenia mebli w `Chromosome`, gdyż została zmieniona metoda mutacji oraz krzyżowania się chromosomów ze sobą. Funkcję generowania chromosomu z `Algorithm` przenieśliśmy również do klasy `Chromosome`.

### Pakiet Count

#### 2.2 Algorithm

W tej klasie zrezygnowaliśmy z funkcji `generateChromosome(Canteen canteen)` na rzecz klasy `Chromosome`. Argumentami funkcji krzyżującej oraz mutującej `Chromosome` stała się `ArrayList<Chromosome> chromosomes`.

#### 2.3 INAlgorithm

Prototypy funkcji `void mutate(ArrayList<Chromosome> chromosomes)` oraz `chromosome crossBreed(ArrayList<Chromosome> chromosomes)` zmieniły argument jaki przyjmują. W funkcji krzyżującej postanowiliśmy zwracać chromosom powstały z krzyżowania i mutacji.

## Nowe Klasy

#### 2.4 FurnitureEnum

Zdecydowaliśmy się na stworzenie klasy `Enum`, która będzie przechowywała informacje o tym jakiego rodzaju jest mebel. Stworzyliśmy w niej funkcje `getWidth(FurnitureEnum furEnum)` oraz `getHeight(FurnitureEnum furEnum)` zwracające kolejno szerokość i wysokość obrazka poszczególnego mebla.

#### 2.5 Model

Została stworzona klasa `Model` usprawniająca wspomaganie realizacji funkcji programu poprzez podmetody:

```
Canteen createCanteen(double bWall, double tWall, double rWall, double lWall)
ArrayList<Chromosomes> createPopulation(Canteen canteen)
ArrayList<Chromosomes> nextGeneration(Canteen canteen, ArrayList<Chromosomes> chromosomes)
```

#### 2.6 Controller

Został stworzony `Controller` zarządzający całym przebiegiem programu, przetwarzających najważniejsze dane algorytmu: `Chromosome getBestChromosome()` `double getIterNumber()` `void nextGeneration()` `void setCanteenCosts(FurnitureEnum key, int cost)` `void setAlgorithmSettings(FurnitureEnum key, int cost)`

### 3 Opis algorytmu

Założenia dotyczące algorytmu genetycznego zostały przez nas zmienione. Zrezygnowaliśmy z reprezentacji bitowej stołówki, gdyż musieliśmy zmierzyć się z problemem nieregularnego rozłożenia rodzajów mebli z Chromosome, przez co obligatoryjnym byłoby ustalenie poprawnej długości ciągu bitów reprezentujących odpowiednio: krzesła, stoły, ławy itd. Mutacja zachodzi poprzez zmianę mebla na jego inny odpowiednik. Stół cztero-osobowy zostaje zamieniony na sześćcio-osobowy, mała ława na dużą ławę. Krzyżowanie polega na znajdowaniu w dwóch chromosomach tych samych mebli oraz wyznaczenie nowej pozycji dla tego mebla na podstawie pozycji jego rodziców.

#### Komplikacje

Niestety, nie przeanalizowawszy dogłębnie całej sytuacji, nie zauważyłem możliwości niezapełnienia wszystkich sklepów. Pojawia się ona w przypadku ,gdy wykorzystamy najwygodniejsze ścieżki z  $n-1$  (gdzie  $n$  to liczba hodowli) ferm , a pozostała nie ma połączenia dostatecznie zaopatrzonego pozostały niepełny sklep.

#### Solucja

Niech  $x$  oznacza liczbę brakujących jaj w sklepie.

Po zdiagnozowaniu problemu,przemyślałem jak należy poprawić sytuację, aby sklepy wszystkie były zapełnione. Tworzyłem liste dróg z fermy w której zostały jajka oraz usuwałem z niej te ,którymi nie prześlę  $x$  jaj. Z pozostałych dróg wybierałem najtańszą. Sprawdzałem do jakiej fermy prowadzi. Następnie z pustych ferm, analizowałem "zużyte" połączenia w których zostało przesłanych  $\geq x$  jaj. W przypadku istnienia takiego połączenia, sprawdzałem czy istnieje droga między daną fermą do sklepu z brakiem, którą mamy możliwość przesłać  $x$  jaj. Jeśli wszystkie warunki zostały spełnione, znaleźliśmy rozwiązanie naszego problemu. Wiemy z którego sklepu zabrać jajka, aby można było je przesłać do sklepu z brakiem. Dodatkowo uzupełniamy sklep, z którego zabraliśmy połączeniem wybranym na początku.

#### 3.1 Pliki dodatkowe

Program wykorzystywał dwa pliki dodatkowe będącymi przykładowymi danymi wejścia w programie:

- eggData.txt;
- test.txt.

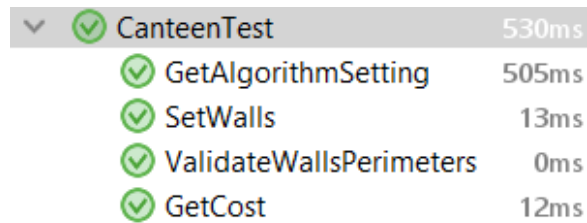
.

## 4 Testy programu

By wystrzec się błędów zrobiłem kilkanaście testów jednostkowych. Ich wynik znajduje się poniżej:

### 4.1 Testy jednostkowe

Do wykonania testów jednostkowych została wykorzystana biblioteka `JUnit4`. Testy jednostkowe zostały stworzone do najważniejszych klas wpływających na sukces wykonania algorytmu. Była to klasa `CheckFile`, klasa `LoadFile`, klasa `Graph` oraz w małym stopniu klasa `Deliver`. Zdjęcia z wykonania testów:



The image shows a screenshot of a JUnit test runner interface. It displays a list of tests under the 'CanteenTest' class. Each test is marked with a green checkmark, indicating it passed. The tests and their execution times are: 'GetAlgorithmSetting' (505ms), 'SetWalls' (13ms), 'ValidateWallsPerimeters' (0ms), and 'GetCost' (12ms). The total time for the 'CanteenTest' class is 530ms.

▼	✓ CanteenTest	530ms
	✓ GetAlgorithmSetting	505ms
	✓ SetWalls	13ms
	✓ ValidateWallsPerimeters	0ms
	✓ GetCost	12ms

Rysunek 2: *Testy jednostkowe*

## 4.2 Testy całościowe

Do wykonania testów całościowych użyłem przede wszystkim przykładowych danych zamieszczonych w treści projektu oraz dla danych wymyślonych przeze mnie.

Przykład testu całościowego bazowego przykładu:

▼	✓ CanteenTest	530ms
	✓ GetAlgorithmSetting	505ms
	✓ SetWalls	13ms
	✓ ValidateWallsPerimeters	0ms
	✓ GetCost	12ms

Rysunek 3: *Test Całościowy*

## 5 Przykładowe użycie programu

Zaprezentowane zostaną tu 2 przykładowe użycia programu z konsoli w systemie Windows:

1. Pierwsze użycie dla danych bazowych: Wywołanie programu:

```
java -jar Eggcelent.jar C:\Users\Michal\IdeaProjects\Eggcelent\src\load\eggData.txt
```

▼	✓ CanteenTest	530ms
	✓ GetAlgorithmSetting	505ms
	✓ SetWalls	13ms
	✓ ValidateWallsPerimeters	0ms
	✓ GetCost	12ms

2. Drugie użycie dla innych danych:

Wywołanie programu:

```
java -jar Eggcelent.jar C:\Users\Michal\IdeaProjects\Eggcelent\src\load\eggData3.txt
```

Z powodu ogromnej liczby danych, pokazana zostanie część wyniku:

▼	✓ CanteenTest	530ms
	✓ GetAlgorithmSetting	505ms
	✓ SetWalls	13ms
	✓ ValidateWallsPerimeters	0ms
	✓ GetCost	12ms

W tym przypadku dla pokazania ogromu transportu jajek, wypisana została liczba jaj.

## 6 Kompilacja

Program jest przeznaczony na dowolony system operacyjny. Ze względu na to, iż wirtualna maszyna Java zajmuje się kompilacją programu, a nie system operacyjny na którym program jest, możemy używać Symulatora gdziekolwiek. Wystarczy stworzyć odpowiedni plik `|.jar|` projektu. Ostatnim krokiem będzie uruchomienie programu z linii poleceń z podaniem argumenty ze ścieżką. W celu bezproblemowego wyświetlania polskich znaków w konsoli, zaleca się użycie komendy `chcp 65001`, dzięki której zmieniamy kodowanie otwartej konsoli na UTF-8.

## 7 Podsumowanie

Pozytywnie zaskoczony byłem tym, iż diagram UML programu **Eggcelent** zgadzał się z moimi założeniami w **Specyfikacja Implementacyjna AiSD**. Zadanie z którym przyszło mi się zmierzyć nie było łatwe. Analizując przebieg mojej pracy nad tym projektem, zauważyłem wiele rzeczy ,które mogłem zrobić inaczej. Jedne z kluczowych jakie zapadną mi w pamięć to:

- Rozpoczęcie implementacji od modułów ,które wydają się najtrudniejsze, a nie od najprostszych;
- Dogłębne przeanalizowanie problemu przed przystąpieniem do implementacji (w moim przypadku nie rozważyłem przypadku, iż nie wszystkie sklepy się napełnią);
- Zwracanie uwagi na kodowanie plików z danymi.

Algorytm zaimplementowany przeze mnie nie jest idealny, lecz udostępnia możliwe dobre rozwiązanie problemu. Jestem w pełni świadom ,iż skutecznym byłoby udoskonalenie go o lepszą poprawę w wariacie napełnianiu sklepów.