

Wydział Informatyki

Artur Jaworski

**Analiza wydajności wczytywania oraz walidacji dużych plików CSV i czasu ich przetwarzania w języku Java.**

Praca: magisterska

Kierunek: Informatyka

Specjalność: Programowanie

Nr albumu: 7896

Praca wykonana pod kierunkiem:

dr hab. Grzegorz Kosior

Wrocław 2025

Spis treści

[1. Wstęp 3](#__RefHeading___Toc341_1335475158_kopia_9)

[1.1. Cel pracy 4](#__RefHeading___Toc341_1335475158)

[1.2. Motywacja podjęcia tematu 5](#__RefHeading___Toc779_1335475158)

[1.3. Metodyka badań 6](#__RefHeading___Toc781_1335475158)

[2. Metodologia 10](#__RefHeading___Toc343_1335475158)

[Pytania badawcze 11](#__RefHeading___Toc2456_2221582084)

[2.1. Model danych 12](#__RefHeading___Toc693_2502114822)

[2.2. Generowanie danych 14](#__RefHeading___Toc38015_2221582084)

[2.3. Walidacja 15](#__RefHeading___Toc695_2502114822)

[2.4. Wczytywanie danych 19](#__RefHeading___Toc1049_2579040556)

[3. Badania 24](#__RefHeading___Toc345_1335475158)

[3.1. Testy statystyczne 24](#__RefHeading___Toc2458_2221582084)

[3.2. Wyniki badań 28](#__RefHeading___Toc2460_2221582084)

[3.3. Wykresy 34](#__RefHeading___Toc38017_2221582084)

[3.4. Wnioski 43](#__RefHeading___Toc38085_2221582084)

[4. Podsumowanie 44](#__RefHeading___Toc347_1335475158)

[5. Dyskusja 45](#__RefHeading___Toc349_1335475158)

[6. Bibliografia 46](#__RefHeading___Toc1471_1721380191)

# Wstęp

Powszechnym problemem w systemach informatycznych jest przesyłania danych pomiędzy aplikacjami. Jest kilka sposobów na rozwiązanie tego dylematu. Jedną z metod jest użycie plików o określonej strukturze tj. plików w formacie CSV. Jak podają Mayur Ramgir oraz Nick Samoylov „**Comma Separated Values** (**CSV**) to idealna metoda przechowywania danych w formie tabelarycznej. Pliki CSV można łatwo przenosić do i z kilku narzędzi do przetwarzania danych, takich jak Microsoft Excel. Jest prosty i idealny do szybkiego transferu w wielu środowiskach.”[[1]](#footnote-2).

Używanie takich plików wiąże się jednak z pewnymi ograniczeniami:

* konieczność sprawdzania formatu danych.
* konieczność walidowania pól.
* konieczność wczytywania plików do pamięci – co może prowadzić do przepełnień pamięci.

Tu rodzi się pytania jaki jest najszybszy sposób na wczytanie oraz walidowanie plików CSV z wykorzystaniem języka programowania Java w wersji 24[[2]](#footnote-3).  
W tej pracy zostały sprawdzone 5 różnych sposobów na wczytywanie danych z plików, z wykorzystaniem:

* *BufferedReader*
* *Scanner*
* *FileReader*
* *Files.lines()*
* *CSVReader*

Cztery pierwsze odnoszą się do biblioteki standardowej języka Java, natomiast ostania pozycja pochodzi z biblioteki *opencsv*.  
Dodatkowo każda z metod zostanie sprawdzona w dwóch przypadkach tj. z walidacją przeprowadzaną w trakcie wczytywania kolejnych linii z pliku oraz z walidacją przeprowadzaną po wczytaniu całego pliku.

# Cel pracy

Celem pracy było sprawdzenie wydajności kilku klas do wczytywania plików w formacie CSV. Wybrane klasy to *BufferedReader*, *Scanner*, *FileReader*, *CSVReader* oraz metoda statyczna *lines*() w klasie *Files*. Wszystkie te sposoby wczytywania plików zostały przetestowane w dwóch przypadkach: z walidacją podczas wczytywania kolnych linii z pliku oraz z walidacją po wczytaniu całej treści dokumentu tekstowego.

W każdym z przypadków walidacja była niezmienna i odnosiła się zarówno do ilości pól, jak i ich typów oraz unikalności danych.

Danych testowe były podzielone na 2 sekcje: dane podstawowe oraz dane rozliczeniowe.  
W danych podstawowych znajdowało się:

* 7 pól typu *String* w tym klucz główny,
* 3 pola typu *Integer*,
* 3 pola typu *Data*,
* 3 pola typu *Boolean*,
* 3 pola typu *Enum*.

Natomiast w danych rozliczeniowych znajdowało się:

* 1 pole typu *String* będące kluczem obcym,
* 3 pola typu *Integer*,
* 1 pole typu *Enum*,
* 2 pola typu *BigDecimal*.

Klucz złożony obejmował pola typu *String*, 3 pola typu *Integer* oraz pola typu *Enum*.

# Motywacja podjęcia tematu

Temat został wybrany z powodów ciekawości autora do najszybszego podejścia do wczytywania oraz walidacji plików CSV. Autor w przeszłości zajmował się w pracy zarobkowej tematem optymalizacji procesów między innymi rzeczonego wczytywania plików. Natomiast w większej części skupiał się na zbędnych operacjach i uproszczeniach systemów, a nie samym wczytywaniu per se.

Wyniki badań przeprowadzonych w pracy mogą podnieść jakość kodu oraz jego wydajność w znaczącym stopniu, co przełoży się na skrócenie czasu oczekiwania na przetwarzanie danych. Wiąże się to również z obniżenie kosztów serwerów aplikacji, a także pośrednio zmniejszenia emisji gazów cieplarnianych, tak ważnym czynniku dzisiejszego, jaki i przyszłego świata.

# Metodyka badań

Badania zostały przeprowadzana na platformie testowej wyposażoną w:

* procesor *AMD* Ryzen *7 3700X* – 8 rdzeni, 16 wątków, 3,60 GHz
* 32 GB RAM, 3200MHz
* Chipset X570
* Dysk SSD 512 GB połączony interfejsem PCIe NVMe 3.0 x4, o deklarowanym odczycie 3350 MB/s
* Windows 11 w wersji 24H2, z kompilacji 26100.4652
* JAVA 24 z dystrybucji Oracle 24.0.1, z budowania 24.0.1+9-30

Dla ograniczenia wpływu zapełniania pamięci operacyjnej program uruchamiany w Java’ie miał nałożony limit 16GB RAM, poprzez ustawienie odpowiadającej flagi -*Xmx16g.*

Testy były przeprowadzana przy minimalnym zużyciu zasobów komputera.  
Testy kilkukrotnie wykonywały wczytanie oraz walidację dla każdego z przypadków testowych oraz dla kilku wielkości plików CSV.   
Ze wszystkich wywołań została wyciągnięta średnia arytmetyczna, wariancja oraz została potwierdzona statystycznie hipoteza, mówiąca o tym, że dane są sobie równe.

Przypadków testowych było łącznie 10 podzielonych na 2 grupy to jest:

Grupa 1. - Walidacja podczas wczytywania plików CSV:

* BufferedReader,
* Scanner,
* FileReader,
* Files.lines,
* CSVReader,

Grupa 2. - Walidacja po wczytaniu całego pliku do programu:

* BufferedReader,
* Scanner,
* FileReader,
* Files.lines,
* CSVReader.

Pliki jakie zostały użyte do badań zostały wygenerowane osobnym programem opisanym dokładniej w punkcie 2.X //TODO

Dane w plikach odnoszą się dziedzinowo do zakładów energetycznych oraz ich danych rozliczeniowych.  
Plik podzielony został na 2 sekcje:

* Dane podstawowe,
* Dane rozliczeniowe.

Dane podstawowe odnoszą się do informacji o samych zakładach produkcyjnych. Dane te posiadają klucz głównych będący 29 znakowym identyfikatorem każdego przedsiębiorstwa. Kolejne dane odnoszą się do danych adresowych oraz danych związanych z działaniem. Wszystkie dane generowane na cale badań nie są rzeczywiste. Łącznie w danych podstawowych jest 19 kolumn z danymi i 1 kolumna określająca sekcję. Jest to pierwsza kolumna, której wartość zawsze jest równa „M”.

Dane rozliczeniowe są to dane odnoszące się do danych podstawowych. Do tego służy klucz obcy będący identyfikatorem przedsiębiorstwa z sekcji numer 1. Dane zawierają ilość wyprodukowanej energii w jednostkach pracy oraz kwotę należną do zapłaty za produkcję. Dane te zawierają ponadto informacje o roku, miesiącu oraz dniu rozliczenia oraz typ raportu. Dzień nie jest polem obowiązkowym.   
Dla każdego identyfikatora zakładu możliwe jest wiele wpisów w sekcji rozliczeniowej. Natomiast nie możliwe jest zduplikowanie danych rozliczeniowych dla takiego samego identyfikatora zakładu, typu raportu i roku, miesiąca oraz dnia. Mówiąc krótko jest to klucz złożony dla sekcji rozliczeniowej.  
Łącznie w danych rozliczeniowych jest 7 kolumn z danymi oraz podobnie jest w przypadku danych podstawowych 1 – pierwsza kolumna określająca sekcję w pliku. Wartość pierwszej kolumny w tej sekcji zawsze przyjmuje wartość „A”.

Dane w sekcjach zostały dobrane tak, aby wykorzystać najpopularniejszą cześć dostępnych typów danych.  
Typy jakie występują w pierwszej sekcji to:

* 7 pól typu *String* w tym klucz główny,
* 3 pola typu *Integer*,
* 3 pola typu *Data*,
* 3 pola typu *Boolean*,
* 3 pola typu *Enum*.

Natomiast w drugiej sekcji znajdowały się:

* 1 pole typu *String* będące kluczem obcym,
* 3 pola typu *Integer*,
* 1 pole typu *Enum*,
* 2 pola typu *BigDecimal*.

Opis poszczególnych pól w pliku w sekcji pierwszej:

1. oznaczenie sekcji
2. identyfikator zakładu – klucz główny
3. miasto w jakim znajdował się zakład
4. ulica z numerem
5. kod pocztowy – 5 cyfr bez myślnika
6. nazwa zakładu
7. region
8. dwu literowe oznaczenie paliwa stosowanego w zakładzie
9. data rozpoczęcia działania zakładu
10. data zakończenia działania zakładu
11. data ostatniej modernizacji zakładu
12. moc produkcyjna zakładu
13. typ osiąganej mocy
14. flaga oznaczająca możliwość zdalnego sterowania produkcją
15. oznaczenie poziomu napięcia przy produkcji
16. identyfikator jednostki pomiarowej
17. typ zakładu
18. liczba określająca poziom ochrony przed hałasem
19. flaga oznaczająca bonus dla biogazowni
20. flaga oznaczająca dodatkowy bonus dla bio-ciepłowni.

Opis poszczególnych pól w pliku w sekcji drugiej:

1. oznaczenie sekcji
2. identyfikator zakładu – klucz obcy
3. typ raportu
4. rok produkcji
5. miesiąc produkcji
6. dzień produkcji – może być pusty dla raportów miesięcznych
7. ilości wyprodukowanej energii
8. kwota do zapłaty

Przykład wygenerowanych danych:

M;EC1048EN00P646607858092143504;vqxhwkd;yzzicigrpymjaoqxzxf 16;45917;nnzjecoee;do;68;03-01-1976;12-09-2042;17-11-1990;4770;TYPE4;0;VLE05;lkkftyupvs3974293498082073;WIND;1;0;0  
M;EC1048EN00P235256287333986541;wxbhlia;qv 24;44405;luch;kf;87;29-01-1992;18-12-2021;11-10-1998;2700;TYPE0;0;VLE01;pnfuzipojv7243862790855085;WATER;2;0;0  
A;EC1048EN00P646607858092143504;TYPE\_5;2020;2;10;9928074977.040;87372.76  
A;EC1048EN00P235256287333986541;TYPE\_5;2015;5;1;8739507934.297;66367.46  
A;EC1048EN00P646607858092143504;TYPE\_5;2015;8;29;1457685930.415;54022.13  
A;EC1048EN00P235256287333986541;TYPE\_2;2020;4;4;6031882117.421;39985.32

Rys. 1. Przedstawiający przykład wygenerowanych danych użytych w badaniu.

Pliki miały rozmiary od 100MB do 4GB. Większe pliki ze względu na zajętość pamięci nie były testowane.

Rozmiary testowanych plików:

* 100MB
* 250MB
* 500MB
* 1GB
* 2GB
* 4GB

# Metodologia

Testy zostały podzielone na 2 grupy:

Pierwsza – testy automatyczne – wykonywana w samym programie oraz druga – testy manualne – wykonywana za pomocą narzędzia *Profiler* z programu *IntelliJ IDEA* w wersji 2025.1.2 dostarczanego przez firmę JetBrains.  
Oba testy wykonywały pomiar czasu jak i zużycia pamięci.

Testy czasu wykonania przeprowadzane w samym programie wykorzystywały metodę *currentMiliseconds()* z klasy *Thread*. Po wywołaniu każdego algorytmu zapamiętywana jest ilość elementów poszczególnych zbiorów danych oraz ilość zwróconych błędów. Po każdym teście wątek był wstrzymany na 5 sekund, wywołując metodę statyczną *sleep(5000)* z klasy *Thread*, aby zminimalizować wpływ nagrzewania się komponentów na wyniki.  
W badaniach uwzględniony został tylko czas pracy algorytmu bez zbierania wyników oraz czyszczenia pamięci.

Test pamięci mierzył różnicę zużycia pamięci bezpośrednio po wykonaniu zadania i przed jego wykonaniem. Aby tego dokonać pobierana jest instancja *MemoryMXBean* za pomocą fabryki *ManagementFactory.getMemoryMXBean()*. Następnie pobierana jest klasa *MemoryUsage* za pomocą *getHeapMemoryUsage()*. Wartość zużywanej akuratnie pamięci można pobrać za pomocą metody *getHeapMemoryUsage()*.  
Przed wykonaniem każdego kolejnego testu wywoływane było wykonanie czyszczenia pamięci za pomocą metody statycznej *gc()* z klasy *System*. Dodatkowo został wstrzymany wątek na 5 sekund, wywołując metodę statyczną *sleep(5000)* z klasy *Thread*, aby zminimalizować wpływ *Garbage Collector’a* na wyniki oraz nagrzewania się komponentów na wyniki.

Testy przeprowadzane przy użyciu *Profiler’a* w narzędziu *IntelliJ IEAD* były wykonywane pojedynczo dla każdego wywołania i ręcznie zbierano dane. Po wykonaniu programu została pobrana wartość całkowitej pamięci zarezerwowanej przez poszczególne algorytmy, jak i czas ich wykonywania. Dodatkowo możliwy był odczyt zajętości pamięci przez poszczególne metody takie jak: *readFile()* czy *validate()*. Podobna możliwość istniała dla czasu wymaganego przez procesor.

Wszystkie testy zostały przeprowadzone 10-krotnie. Z uzyskanych wyników wyciągnięto średnią, odchylenie standardowe oraz potwierdzono hipotezę o braku różnić między wywołaniami.

## Pytania badawcze

Celem badań jest odpowiedź na następujące pytania:  
\* jaki jest najszybszy sposób na wczytanie i przetwarzanie dużych plików CSV spośród testowanych algorytmów?

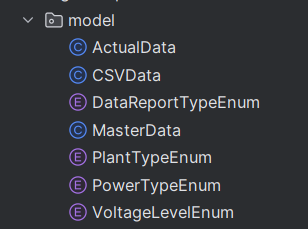
\* jaki sposób wczytanie i przetwarzanie dużych plików CSV zużywa najmniej pamięci spośród testowanych algorytmów?

\* czy moment wykonywania walidacji wpływa na czas oraz pamięć potrzebną do wykonania zadania?

# Model danych

W tym podrozdziale został opisany moduł modelu danych znajdujących się w pliku CSV.

Moduł modelu zawiera następujące klasy:



Rys. X, przedstawiający klasy w module modelu.

Klasa MasterData jest odwzorowaniem sekcji pierwszej z pliku CSV, a klasa ActualData jest odwzorowaniem sekcji drugiej.  
Klasa *CSVData* jest nadklasą dla *MasterData* oraz *ActualData*. Zawiera ono jedno pole *masterKey*, które jest identyfikatorem zakładu.

Pola w klasie MaserData:

* *String* *masterKey*
* *String* *street*
* *String* *postCode*
* *String* *name*
* *String* *region*
* *Integer* *fuel*
* *Date* *startDate*
* *Date* *endDate*
* *Date* *modificationDate*
* *Integer* *power*
* *PowerTypeEnum* *powerType*
* *Boolean* *controllability*
* *VoltageLevelEnum* *voltageLevel*
* *String* *measurementLocationId*
* *PlantTypeEnum* *plantType*
* *Integer* *soundOptimization*
* *Boolean* *isBiomassBonus*
* *Boolean* *isBiomassTechnologyBonus*

Pola w klasie ActualData:

* *DataReportTypeEnum* *dataReportType*
* *Integer* *year*
* *Integer* *month*
* *Integer* *day*
* *BigDecimal* *quantity*
* *BigDecimal* *amount*

Wartości enumów:

* *DataReportTypeEnum*:  
  *TYPE\_1*, *TYPE\_2*, *TYPE\_3*, *TYPE\_4*, *TYPE\_5*, *TYPE\_6*
* *PlantTypeEnum*:  
  *SOLAR*, *BIOMASS*, *COAL*, *WIND*, *ATOM*, *WATER*, *GAS*, *GEOTHERMAL*
* *PowerTypeEnum*:  
  *TYPE0*, *TYPE1*, *TYPE2*, *TYPE3*, *TYPE4*, *TYPE5*
* *VoltageLevelEnum*:  
  *VLE00*, *VLE01*, *VLE02*, *VLE03*, *VLE04*, *VLE05.*

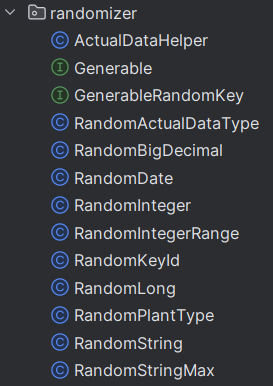
W każdej klasie wszystkie pola są prywatne a dostęp do nich zapewniany jest za pomocą getterów oraz seterów. Oprócz tego w klasie zostały nadpisane metody *equal()*, *hashcode()* oraz *toString()*.

# Generowanie danych

Dane do plików CSV zostały wygenerowane przez za pomocą napisanego programu. Program ten generował każdą linie osobno, dokonując konkatenacji za pomocą średników, a następnie otrzymaną linie zapisywał w pliku o rozszerzeniu .csv. Plik składał się z 2 części, przy czym pierwsza część stanowiła 10% wierszy.

Generowane pliki miały różne rozmiary w zależności od ilości wierszy. Przyjmuje się że 12000 wierszy wygeneruje ok. 1MB plik. Każdy wygenerowany plik może mieć nieco różniący się od siebie rozmiar przy zachowaniu tej samej liczby wierszy.

Generator zawiera pakiet randomizer, który zawiera w sobie: 2 interfejsy z definicją metody generowania danych, klasę pomocniczą oraz 10 implementacji metod interfejsów w zależności od typu danych:



Rys. 1 przedstawiający strukturę pakietu randomizer dla generatora plików.

Interfejs Generable<T> zawiera jedną metodę o sygnaturze T generate(String… args).  
Jej implementacja generuje wartości dla argumentów podanych jako String.

Drugi z interfejsów GenerableRandomKey<T> rozszerza poprzedni interfejs o dodatkową metodę String getExistRandomKeyId(). Służy ona do pobrania wygenerowanych już kluczy głównych, aby druga sekcja miała połączenie z pierwszą.

Klasa pomocnicza ActualDataHelper posiada jedną metodę checkUniqueness, która dla parametrów masterKey, dataReportType, rok, miesiąc i dzień sprawdzi czy dana sekwecja klucza złożonego już została wygenerowana. Jeśli tak się nie stało zostanie nowy rekord zostanie zapisany w pamięci.

**Omówienie implementacji w klasach:**

* RandomInteger – przyjmuje jeden argument i generuje losową liczbę całkowitą nie większą niż podany parametr.
* RandomIntegerRange – przyjmuje 2 argumenty i generuje losową liczbę całkowitą w przedziale między pierwszą a drugą wartością parametru.
* RandomLong - przyjmuje 2 argumenty i generuje losową liczbę zmiennoprzecinkową z przedziału między pierwszą a drugą wartością parametru.
* RandomBigDecimal – przyjmuje 2 argumenty, gdzie pierwszy z nich to precyzja wygenerowanej liczby stałoprzecinkowej, a drugi argument to maksymalna wartość BigDecimal.
* RandomString - przyjmuje 1 argument i generuje losowy ciąg znaków o podanej w parametrze długości.
* RandomStringMax - przyjmuje 1 argument i generuje losowy ciąg znaków o maksymalnej długości podanej w parametrze.
* RandomDate - przyjmuje 2 argumenty i generuje losową datę z przedziału podanych w parametrach dat.
* RandomActualDataType – nie przyjmuje argumentów, generuje ciąg znaków o prefiksie „TYPE\_” oraz sufiksie będącym losową cyfrą z przedziału 1-6
* RandomPlantType – nie przyjmuje argumentów, losuje liczbę z przedziału 0-8 i dla wylosowanej liczby wybiera kolejno SOLAR, BIOMASS, COAL, WIND, ATOM, WATER, GAS, GEOTHERMAL.
* RandomActualDataType -
* RandomKeyId – jako jedyny implementuje drugi interfejs GenerableRandomKey. W metodzie generate() nie przyjmuje żadnych argumentów i generuje ciąg znaków o stałym prefiksie „EC1048EN00P” oraz sufiksie będącym losową 18 cyfrową liczbą. Wygenerowany klucz zostaje następnie zapisany na liście wszystkich kluczy. Druga metoda getExistRandomKeyId() pobiera z listy klucz o losowy indeksie.

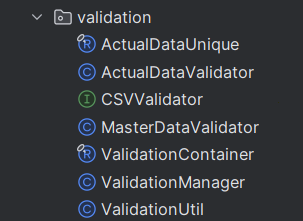
Klasa GenerateLine służy do generowania w postaci String’a poszczególnych linii pierwszej, bądź drugiej sekcji pliku CSV, zgodnie z modelem danych z poprzedniego rozdziału. W swoim działaniu wywołuje ona klasy z pakietu randomizer.

Klasa GenerateCSVFile służy do utworzenia nowego pliku o określonej nazwie oraz odpowiedniej ilości wierszy. Wywołuje ona klasę GenerateLine.

# Walidacja

Ten podrozdział opisuje założenia odnoszące się do walidacji plików oraz zależności między poszczególnymi polami.

Moduł walidacji zawierał w sobie klasy:



Rys. X, przedstawiający klasy w module walidacji.

Klasą wywoływaną do przeprowadzenia walidacji była klasa *ValidationManager* i metoda *isValid(ValidationContainer, String)*. Przyjmuje ona za parametry instancję *ValidationContainer’a* oraz ciąg znaków pierwszego tokenu, aby rozpoznać sekcję i dobrać odpowiednią klasę do walidacji: *MasterDataValidator* lub *ActualDataValidator*.

*ValidationContainer* zawiera w sobie następujące pola:

* *String[]* - jako tablicę tokenów, czyli pojedynczych elementów rekordu z pliku CSV,
* *CSVData* – jako obiekt zmapowania tokenów na docelowy typ danych,
* *Set<String>* zbiór identyfikatorów zakładów, dodawany po każdej kolejnej walidacji sekcji pierwszej oraz sprawdzany w sekcji drugiej,
* *Set<ActualDataUnique>* zbiór kluczy złożonych, sprawdzany i dodawany po każdej kolejnej walidacji sekcji drugiej,
* *List<String>* - jako lista wszystkich zwróconych błędów walidacji,
* *int* – jako numer obecnie walidowanej linii z pliku – informacja dla użytkownika przy błędach walidacji.

Klasy *MasterDataValidator* i *ActualDataValidator* implementują interfejs *CSVValidator*, który zawiera sygnaturę metody *validate(ValidationContainer)*.

Implementacja tej metody w klasie *MasterDataValidator* sprawdza na początku poprawność ilości tokenów w tablicy. Powinno ich być 20. Jeśli w tym lub innym przypadku nastąpi niezgodność z walidacją do kolekcji błędów zostanie dodany odpowiedni wpis.  
Następnie każdy kolejny token po kolei:

1. section – sprawdzony wcześniej w *ValidatorManager*.
2. MasterKey – sprawdzane jest czy token nie jest pusty oraz czy jego długość jest równa 29 znaków oraz czy w *ValidationContainer* nie istnieje już taki identyfikator zakładu. Na koniec do *ValidationContainer* dodawany jest walidowany identyfikator oraz w obiekcie *MasterData* ustawiana jest wartość pola *masterKey*.
3. City - sprawdzane jest czy token nie jest pusty oraz czy jego długość nie jest większa niż 256 znaków. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *city*.
4. Street - sprawdzane jest czy token nie jest pusty oraz czy jego długość nie jest większa niż 256 znaków. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *street*.
5. PostCode - sprawdzane jest czy token nie jest pusty oraz czy jego długość jest równa 5 znaków. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *postCode*.
6. Name - sprawdzane jest czy token nie jest pusty oraz czy jego długość nie jest większa niż 256 znaków. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *name*.
7. Region - sprawdzane jest czy token nie jest pusty oraz czy długość tokenu jest równa 2 znakom. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *region*.
8. Fuel - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą mniejszą od 100. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *fuel*.
9. StartDate - sprawdzane jest czy token nie jest pusty oraz czy jest poprawną datą. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *startDate*.
10. EndDate - sprawdzane jest czy token jest pusty lub jest poprawną datą. Data ta musi być po dacie *startDate*. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola end*Date*.
11. ModificationDate - sprawdzane jest czy token jest pusty lub jest poprawną datą. Data ta musi być pomiędzy datą *startDate*, a *endDate.* Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *modificationDate*.
12. Power - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą nie ujemną. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *power*.
13. PowerType - sprawdzane jest czy token nie jest pusty oraz czy jego długość jest równa 5 znaków. Następnie sprawdzane jest czy istnieje enum *powerTypeEnum* o wartości podanej w tokenie. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *powerType*.
14. Controllability - sprawdzane jest czy token nie jest pusty oraz jeśli wartość jest równa 0, to wartość pola *Controllability* w obiekcie *MasterData* jest ustawiana na *False*, natomiast jeśli wartość tokenu jest równa 1, wtedy pole *Controllability* ustawiane jest na *True*.
15. VlotageLevel - sprawdzane jest czy token nie jest pusty oraz czy jego długość jest równa 5 znaków. Następnie sprawdzane jest czy istnieje enum *VoltageLevelEnum* o wartości podanej w tokenie. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *powerType*.
16. MeasurementLocationId - sprawdzane jest czy token nie jest pusty oraz czy jego długość jest równa 26 znaków. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *MeasurementLocationId*.
17. PlantType - sprawdzane jest czy token nie jest pusty. Następnie sprawdzane jest czy istnieje enum *PlantTypeEnum* o wartości podanej w tokenie. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *plantType*.
18. SoundOptimization - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą z przedziału od 0 do 2 łącznie. Na koniec w obiekcie *MasterData* ustawiana jest wartość pola *SoundOptimization*.
19. IsBiomassBonus - sprawdzane jest czy token nie jest pusty oraz jeśli wartość jest równa 0, to wartość pola *IsBiomassBonus* w obiekcie *MasterData* jest ustawiana na *False*, natomiast jeśli wartość tokenu jest równa 1, wtedy pole *IsBiomassBonus* ustawiane jest na *True*.
20. IsBiomassTechnologyBonus - sprawdzane jest czy token nie jest pusty oraz jeśli wartość jest równa 0, to wartość pola *IsBiomassTechnologyBonus* w obiekcie *MasterData* jest ustawiana na *False*, natomiast jeśli wartość tokenu jest równa 1, wtedy pole *IsBiomassTechnologyBonus* ustawiane jest na *True*.

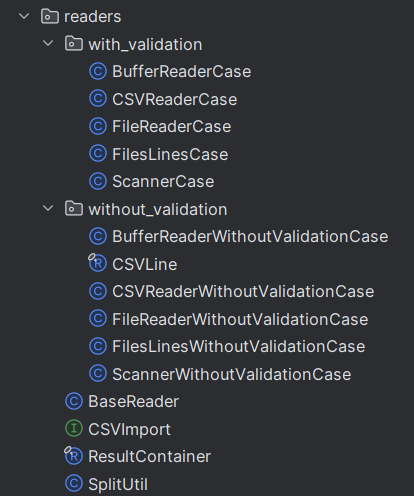
Podobnie jak implementacja w klasie *MasterDataValidator*, tak w klasie Actual*DataValidator* sprawdzana jest na początku poprawność ilości tokenów w tablicy. Powinno ich być 8. Jeśli w tym lub innym przypadku nastąpi niezgodność z walidacją do kolekcji błędów zostanie dodany odpowiedni wpis.  
Następnie każdy kolejny token po kolei:

1. section - sprawdzony wcześniej w *ValidatorManager*.
2. MasterKey – sprawdzane jest czy token nie jest pusty oraz czy długość tokenu jest równa 29 znaków oraz czy w *ValidationContainer* istnieje taki identyfikator zakładu. Na koniec w obiekcie *ActualData* ustawiana jest wartość pola *masterKey*.
3. DataReportType - sprawdzane jest czy token nie jest pusty. Następnie sprawdzane jest czy istnieje enum *DataReportTypeEnum* o wartości podanej w tokenie. Na koniec w obiekcie *ActualData* ustawiana jest wartość pola *dataReportType*.
4. Year - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą z przedziału od 1900 do 2100 łącznie. Na koniec w obiekcie Actual*Data* ustawiana jest wartość pola *year*.
5. Month - sprawdzane jest czy token jest lub oraz czy jest liczbą z przedziału od 0 do 12 łącznie. Na koniec w obiekcie Actual*Data* ustawiana jest wartość pola *month*.
6. Day - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą z przedziału od 0 do 31 łącznie. Na koniec w obiekcie Actual*Data* ustawiana jest wartość pola *day*.
7. Quantity - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą dodatnią z dokładnością mniejszą niż 4 miejsca po przecinku. Na koniec w obiekcie Actual*Data* ustawiana jest wartość pola *quantity*.
8. Amount - sprawdzane jest czy token nie jest pusty oraz czy jest liczbą dodatnią z dokładnością mniejszą niż 3 miejsca po przecinku. Na koniec w obiekcie Actual*Data* ustawiana jest wartość pola *amount*.

Dodatkowo po walidacji tokenów, sprawdzane jest czy podana w 3 kolumnach data jest poprawna. W przypadku braku dnia sprawdzana jest data z pierwszym dniem miesiąca.  
Ostatnią walidacją jest sprawdzenie czy zachowany jest klucz złożony z pól *masterKey*, *dataReportType*, *year*, *month*, *day*. Aby tego dokonać tworzone są obiekty pomocnicze *ActualDataUnique*, o wymienionych wcześniej polach. Następnie sprawdzane jest czy taki obiekt nie istnieje już w kolekcji *actualDataUniques* w *ValidationContainer*.

# Wczytywanie danych

W tym podrozdziale zostały opisane metody używane do odczytu danych z plików CSV oraz ich użycie.

Moduł wczytywania:

Rys. X, pokazujący klasy w module wczytywania.

Klasa *ResultContainer* zawiera w sobie 3 listy danych takich jak:

* maserDataList – kolekcja zawierająca wszystkie obiekty MasertData z pliku CSV,
* actualDataList – kolekcja zawierająca wszystkie obiekty ActualData z pliku CSV,
* errors – kolekcja zawierająca wszystkie błędy zebrane podczas walidacji pliku CSV.

Klasa SplitUtil jest statyczną klasą narzędziową//TODO (potrzebny przypis). Zawiera ona jedną metodę *splitLine()*, która z parametru *String*, będącego linią z pliku zwraca tablicę tokenów. Działa ona na zasadzie przeszukiwania kolejnych wystąpień znaku separatora CSV i tworzenia podciągów od ostatniego do obecnego wystąpienia znaku *;*. Kolejne tworzone w ten sposób podciągi są dodawane do listy wiązanej, a na końcu za pomocą metody *toArray()* zwracane do postaci tablicy tokenów o typie *String*.

Klasa *BaseReader* zawiera metodę validate(), która odpowiada za wywołanie walidacji danej tablicy tokenów oraz tworzenie obiektów pomocniczych takich jak *ValidationContainer*. Każdy z przypadków testowych korzysta z tej metody walidacji dzięki czemu można założyć, że walidacja obiektów dla każdego z przypadków jest taka sama.

W tej części zostanie opisane 5 klas do wczytywania plików:

* + 1. FileReader

**Opis teoretyczny**

FileReader jest to klasa z pakietu java.io i rozszerza klasę InputStreamReader. Klasa ta służy do wczytywania danych z plików jako strumienie znaków.[[3]](#footnote-4)

W klasie tej znajdują się 3 konstruktory, każdy z nich jednoargumentowy. Przyjmują one parametry: File, FileDescriptor lub String. Konstruktor może zgłosić wyjątek FileNotFoundException w momencie, gdy plik, lub nazwa nie istnieją, bądź są folderami.

Metoda read() odczytuje i zwraca z pliku kolejne znaki. Gdy skończą się dane w pliku metoda zwraca wartość -1.[[4]](#footnote-5) Co istotne, zależności od kodowania znak może zawierać różną ilość bajtów. Metoda ta jest mało efektywna, ponieważ za każdym razem wczytuje tylko jeden znak.

Sposobem na ten problem może być użycie metoda read(char[] cbuf), której implementacja znajduje się w klasie Reader. Metoda ta wczytuje znaki do podanej tablicy. Będzie tak dopóki: dane wejściowe są dostępne lub zakończono czytanie znaków lub wystąpi błąd wejścia/wyjścia. Metoda zwraca liczbę znaków jakie zostały wczytane do bufora. Może się zdarzyć, że metoda zgłosi wyjątek IOExcepotion.[[5]](#footnote-6)

Odczytywania znaków za pomocą bufora może z łatwością zwiększyć wydajność kilkunastokrotnie. Jednak ma on swoje limity. Jak podaje J. Jenkov „Jeśli rozmiar tablicy będzie większy niż możliwości systemu operacyjnego bądź sprzętu, nie będzie to skutkowało wzrostem prędkości z tytułu większej tablicy znaków.”[[6]](#footnote-7)

Klasa ta implementuje interfejsy Closeable oraz AutoCloseable. Oznacza to, że po zakończeniu działań na pliku należy wywołać metoda close(), aby zwolnić pamięć oraz zasoby systemowe. Aby ułatwić proces można użyć mechanizmu try-with-resources, który automatycznie wywoła metodę close().[[7]](#footnote-8)  
Według specjalistów z Oracle aż 60% kodu, który korzystał z zasobów w JDK 6 było nieprawidłowo obsłużone. Wniosek jest prosty, jeśli nawet inżynierowie towrzący język miewają problemy z obsługa zasobów, należy oddelegować tę akcję używając Try-with-resources.[[8]](#footnote-9)

**Opis algorytmu z walidacją podczas wczytywania**

Algorytm ten używa metody *fileReader.read(buffer)* do pobrania kolejnych znaków do 256 znakowego buforu.   
Następnie przechodząc po każdym znaku sprawdza czy jest on komentarzem *#*, znakiem separatora *;*, czy znakiem końca linii \n.  
W przypadku komentarza przechodzi przez bufor do czasu znalezienia znaku końca linii.  
W przypadku znaku separatora dodaje nowy token do listy tokenów przez utworzenie nowego obiektu String z bufora od wartości ostatniego indeksu przez długość czytania bufora zanim znaleziono separator. Następnie jako nowy ostatni indeks ustawiany jest kolejny znak z buforu.

W przypadku znaku końca linii do listy tokenów dodawany jest ostatni ciąg znaków, tak jak wyżej. Następnie z listy tokenów tworzona jest tablica tokenów poprzez użycie metody *toArray()*. W kolejnym kroku następuje wywołanie metody validate() z parametrami: lista tokenów, lista identyfikatorów zakładów, lista kluczy złożonych z sekcji drugiej, numer linii oraz kontener rezultatu. Po walidacji lista tokenów jest czyszczona.   
Gdy pętla przejdzie przez cały bufor sprawdzone jest czy ostatni indeks separatora jest ostatnim znakiem pobranym z bufora. Jeśli nie to pozostałość zostanie dodane do następnego tokenu.  
Na koniec wczytywana jest kolejny ciąg znaków do buforu.

**Opis algorytmu z walidacją po wczytaniu wszystkich danych:**

W porównaniu do wersji poprzedniej tutaj zamiast tworzyć listę tokenów, tworzona jest pełna linia z pliku CSV, a zamiast metodę walidacji, dodawany jest nowy obiekt *CSVLine* do listy wszystkich linii.  
*CSVLine*  zawiera 2 pola i są to: *String* z linią z pliku oraz *int* numer wiersza potrzebny do wskazywania błędów podczas walidacji.

Po wczytaniu wszystkich linii do listy *csvLines*, następuje iterowanie po każdym elemencie tej kolekcji oraz parsowanie linii z *CSVLine* do tablicy *String* z wykorzystaniem *SplitUtil.splitLine()*. Następnie wykonywana jest metoda *validate()* z odpowiednimi parametrami. Po walidacji element na liście *csvLines* jest ustawiany na *null*, aby umożliwić szybsze zwolnienie pamięci.

* + 1. BufferedReader

**Opis teoretyczny**

Klasa BufferedReader jest klasą z pakietu java.io i rozszerza klasę Reader. Zgodnie z dokumentacją klasa ta „odczytuje tekst ze strumienia znaków wejściowych, buforując znaki w celu zapewnienia wydajnego odczytywania znaków, tablic i wierszy.”[[9]](#footnote-10)

Klasa zawiera 2 konstruktory. Pierwszy przyjmuje parametr Reader oraz int, tworząc tym samym buforowany strumień znaków, dla bufora wejściowego oraz rozmiaru tego bufora. Drugi z konstruktorów przyjmuje tylko jeden argument Reader, a wielkość bufora jest ustawiana na domyślne 8kb.[[10]](#footnote-11)

Metodą najczęściej używaną w tej klasie jest readLines(), która to wczytuje oraz zwraca String z kolejną linią ze strumienia. Jak podaje dokumentacja Oracle „Uznaje się, że wiersz jest zakończony przez dowolny z wysuwu wiersza ('\n'), powrót karetki (''), powrót karetki, po którym następuje wysuw wiersza lub dotarcie do końca pliku (EOF).”[[11]](#footnote-12). Znak rozdzielający kolejne linie nie jest zwracany. Gdy metoda dojdzie do końca pliku zwróci wartość null, stąd bardzo częste wykorzystywanie jej w pętli while.[[12]](#footnote-13) Metoda może zgłosić wyjątek IOException.

BufferedReader poprawia efektywność odczytu strumienia znaków np. FileReadera. Dlatego ich wspólne stosowanie jest bardzo częste. Podczas gdy FileReader łączy się z plikiem, to BufferedReader zapewnia, że odczyty z dysku nie będą zbyt częste.[[13]](#footnote-14)

Podobnie jak w przypadku FileReader klasa ta implementuje interfejsy Closeable oraz AutoCloseable. Oznacza to, że po zakończeniu działań na pliku należy wywołać metoda close(). Zalecane jest użycie mechanizmu try-with-resources, który wystarczy użyć na klasie BufferedReader, a Reader zostanie zamknięty automatycznie.[[14]](#footnote-15)

**Opis algorytmu z walidacją podczas wczytywania:**

Algorytm ten używa metody *bufferedReader.readLine()* do pobrania kolejnych linii z pliku CSV.   
Następnie dopóki linia nie jest pusta sprawdza czy pierwszy znak jest komentarzem *#*. W takim przypadku pobiera kolejną linię z pliku.   
W kolejnym kroku parsuję linię do tablicy *String* z wykorzystaniem *SplitUtil.splitLine()*. Później następuje wywołanie metody *validate()* z odpowiednimi parametrami.   
Na koniec wczytywany jest kolejny wiersz z pliku.

**Opis algorytmu z walidacją po wczytaniu wszystkich danych:**

W porównaniu do wersji powyżej, nie następuję parsowanie oraz walidacja, tylko do listy wierszy csv*Lines* dodawane są kolejne linii z pliku CSV.  
Po wczytaniu całości następuje przejście po całej kolekcji *csvLines* i każda linia jest parsowana do postaci tablicy *String[]* oraz wywoływana jest metoda *validate()* z odpowiednimi parametrami. Po walidacji element na liście *csvLines* jest ustawiany na *null*, aby umożliwić szybsze zwolnienie pamięci.

* + 1. Scanner

**Opis teoretyczny**

Klasa BufferedReader jest klasą z pakietu java.util. Służy do odczytywania kolejnych tokenów ze źródła. Jak podaje Oracle w swojej dokumentacji „Scanner dzieli dane wejściowe na tokeny przy użyciu wzorca ogranicznika, który domyślnie pasuje do białych znaków. Powstałe tokeny można następnie przekształcić w wartości różnych typów przy użyciu różnych kolejnych metod.”[[15]](#footnote-16)

Konstruktorów w klasie jest aż 14. Dzielą się na grupy, które mogą przyjmować: InputStream – najczęściej używany przypadek, często wykorzystywany z System.in do odbierania danych do użytkownika, String lub ReadableByteChannel, do pracy na łańcuchach znaków lub bajtów, File bądź Path, do działa z plikami.   
Konstruktory jako drugi parametr mogą przyjmować kodowanie znaków w przypadku, gdy nie ma zostać użyte inne niż domyślne UTF-8.[[16]](#footnote-17)

Najczęściej używanymi metodami są hasNext() oraz next() i ich odpowiedniki dla różnych typów np. hasNextInt() oraz nextInt(), hasNextLong() oraz nextLong().

Metoda hasNext() zwraca wartość true, gdy istnieje kolejny token na wejściu. Metoda ta blokuje proces, czekając na dane. Sprawdzenie czy istnieje token nie wpływa na iterator scannera. Natomiast metoda next() zwraca wartość kolejnego tokena. Domyślnie obie metody jako ogranicznika używają białych znaków.[[17]](#footnote-18)

W przypadku użycia Scannera do działania z konsolą użytkownika nie ma stosuje się sprawdzenia hasNext(), lecz gdy działamy na innych typach źródeł np. plikach taka informacja jest bardzo pomocna.[[18]](#footnote-19)

Wariancją powyższych metod jest hasNextLine() oraz nextLine(). Co do zasady działają podobnie, jednak jako ogranicznik używają znaków końca linii. Dodatkowo metoda sprawdzająca może przechować w pamięci linie do zwrócenia, dzięki temu metoda nextLine() szybciej zwróci rezultat.

Podobnie jak w przypadku klas Reader, Scanner również implementuje interfejsy Closeable oraz AutoCloseable i również po zakończeniu działań na pliku bądź innym źródle należy zamknąć scanner. Najlepiej do tego celu sprawdzi się mechanizm try-with-resources.   
Należy jednak pamiętać, że zamknięcie Scannera, który korzysta z System.in sprawi, że nie będzie mógł być użyty więcej razy.[[19]](#footnote-20)

**Opis algorytmu z walidacją podczas wczytywania:**

Algorytm ten sprawdza używając scanner*.hasNextLine()* czy w pliku CSV znajdują się kolejne linie i pobiera każdą z nich poprzez *scanner.nextLine()*.   
Następnie dla każdej linii sprawdzane jest czy nie jest pusta oraz czy pierwszy znak jest komentarzem *#*. W takim przypadku pobiera kolejną linię z pliku.   
W kolejnym kroku parsuję linię do tablicy *String[]* z wykorzystaniem *SplitUtil.splitLine()*. Później następuje wywołanie metody *validate()* z odpowiednimi parametrami.

**Opis algorytmu z walidacją po wczytaniu wszystkich danych:**

Algorytm podobnie jak w wersji *BufferedReader*, w porównaniu do wersji z walidacją podczas wczytania, nie następuję parsowanie oraz walidacja, tylko do listy wierszy csv*Lines* dodawane są kolejne linii z pliku CSV.  
Po wczytaniu całości następuje przejście po całej kolekcji *csvLines* i każda linia jest parsowana do postaci tablicy *String[]* oraz wywoływana jest metoda *validate()* z odpowiednimi parametrami. Po walidacji element na liście *csvLines* jest ustawiany na *null*, aby umożliwić szybsze zwolnienie pamięci.

* + 1. Files.lines()

**Opis teoretyczny**

Klasa Files to klasa narzędziowa z pakietu java.nio.file. Dostarcza ona wiele pomocnych metod do pracy z plikami.

Metoda lines(Path path), służy do wczytania wierszy z pliku i zwrócenia ich w postaci strumienia zgodnie ze standardem Stream API. Istnieje przeciążenie tej metody z dodatkowym argumentem Charset, który nadpisuje domyślne kodowanie znaków UTF-8. Metoda lines wykorzystuje lazy loading, czyli wczytuje kolejne linie w momencie kiedy są używane. Metoda ta może zgłosić IOException lub często owinięty błąd UncheckedIOException.

Jak wskazuje dokumentacja Oracle „Ta metoda musi być używana w instrukcji try-with-resources lub podobnej strukturze kontrolnej, aby zapewnić, że otwarty plik strumienia zostanie zamknięty natychmiast po zakończeniu operacji strumienia.”[[20]](#footnote-21)

Stream API zostało wprowadzone, aby uprościć wykonywanie operacji zbiorczych w sposób sekwencyjny lub równoległy. Podstawą tego mechanizmu są dwa elementy: strumień, czyli ciąg danych pochodzących m.in. z kolekcji, tablic czy plików, oraz potok strumieniowy, opisujący kolejne etapy przetwarzania tych danych. Potok składa się ze źródła, operacji pośrednich (np. filtrowania, mapowania) oraz operacji końcowej. Kluczowym działaniem jest **leniwe wykonywanie** – operacje są realizowane dopiero przy wywołaniu operacji końcowej, co umożliwia pracę nawet na strumieniach nieskończonych. API jest również zaprojektowane w stylu płynnym, co pozwala łączyć wiele wywołań w jedną spójną konstrukcję.[[21]](#footnote-22)

**Opis algorytmu z walidacją podczas wczytywania:**

W przypadku tego algorytmu wczytywanie następuje z wykorzystaniem strumieni używając *Files.lines()*.  
Algorytm ten dla każdej linii z pliku CSV sprawdzane jest, czy nie jest ona pusta oraz czy pierwszy znak nie jest komentarzem *#*. W takim przypadku parsuję linię do tablicy *String[]* z wykorzystaniem *SplitUtil.splitLine()*. Później następuje wywołanie metody *validate()* z odpowiednimi parametrami.

**Opis algorytmu z walidacją po wczytaniu wszystkich danych:**

Podobnie jak w wersjach opartych na BufferedReader i Scanner, w tej implementacji nie wykonuje się parsowania ani walidacji danych podczas odczytu pliku.   
Zamiast tego, kolejne linie z pliku CSV są dodawane do listy csvLines. Dopiero po wczytaniu całego pliku następuje iteracja po elementach tej listy – każda linia jest przekształcana na tablicę typu *String[]*, a następnie przekazywana do metody *validate()* wraz z odpowiednimi argumentami. Po zakończeniu walidacji dana pozycja na liście csvLines zostaje ustawiona na null, co umożliwia wcześniejsze zwolnienie zajmowanej przez nią pamięci.

* + 1. CSVReader

**Opis teoretyczny**

Klasa CSVReader pochodzi z biblioteki com.opencsv. Zawiera ona gotowe do użycia klasy pracujące na plikach CSV. Dzięki niej znacznie czytanie oraz zapis do plików CSV staje się znacznie łatwiejsze. Ponadto jako jedno z niewielu zewnętrznych rozwiązań jest oparte jest na licencji Apache 2.0.[[22]](#footnote-23)

W bibliotece OpenCSV dostępnych jest kilka praktycznych klas:

* CSVReader umożliwia odczyt pliku CSV do postaci łańcuchów znaków.
* CSVWriter służy do zapisywania danych w formacie CSV.
* CsvToBean pozwala mapować zawartość pliku CSV na obiekty typu Java Bean,
* BeanToCsv wspiera proces odwrotny do poprzedniego, czyli eksportowania danych z aplikacji Java do plików CSV.[[23]](#footnote-24)

Klasa CSVReader posiada konstruktor z parametrem Reader, natomiast zalecanym sposobem tworzenia czytnika jest użycie klasy budowniczego CSVReaderBuilder. Dzięki niemu możemy w łatwy sposób ustawić np. znak separatora lub pominąć pewną ilość linii.

Metoda readNext() służy do pobrania kolejnej linii z pliku oraz zwrócenia jej w postaci tablicy stringów. Jeśli danych w pliku już nie będzie metoda ta zwróci null. Może zgłosić IOException oraz csvValidationException, gdy użytkownik zdefiniował reguły.[[24]](#footnote-25)

Podobnie jak w przypadku pozostałych klasa CSVReader również implementuje interfejsy Closeable oraz AutoCloseable. Należy pamiętać o użyciu try-with-resources lub o zamknięciu zasobu po zakończonej pracy.

**Opis algorytmu z walidacją podczas wczytywania:**

Algorytm opiera swoje działanie na tokenach typu *String[]* pobieranych za pomocą csvReader.readNext() z pliku CSV.   
Następnie sprawdza czy tablica tokenów jest różna od wartości *null*, kiedy to plik dobiegnie końca.   
Następnie dla każdej tablicy tokenów sprawdzane jest czy nie jest pusta oraz czy pierwszy token nie zaczyna się od znaku komentarza *#*. W takim przypadku pobiera kolejną linię z pliku.   
W kolejnym kroku następuje wywołanie metody *validate()* z odpowiednimi parametrami, a na koniec pobierany jest kolejna tablica tokenów z pliku CSV.

**Opis algorytmu z walidacją po wczytaniu wszystkich danych:**

Podobnie jak dla wersji wcześniejszych, w implementacji tej nie wykonuje się parsowania ani walidacji danych podczas odczytu pliku.   
Kroki te są rozdzielone na wczytanie kolejnych linii z pliku CSV, które dodawane są do listy csvLines.   
Następnym krokiem jest iteracja po elementach wczytanych wierszy, gzie każda linia jest przekształcana do postaci *String[]* za pomocą metody *SplitUtil.splitLine()*, a następnie wywoływana jest metody *validate()* wraz z odpowiednimi parametrami. Po walidacji wartość elementu na liście *csvLines* jest ustawiana na *null*, aby umożliwić szybsze zwolnienie części pamięci.

**Oczekiwany rezultat:**  
Teoretycznie najszybszym sposobem przetwarzania plików, powinien okazać się BufferedReader oraz przetwarzanie strumieniowe z wykorzystaniem Files.lines. Z drugiej strony najmniej wydajny powinno być wykorzystanie klasy Scanner.  
Pamięć powinna być na podobnym poziomie, jednak Files.lines ze względu na swoją charakterystykę będzie wykorzystywać mniej pamięci operacyjnej.

Z dwóch grup algorytmów z walidacją podczas wczytywania oraz z walidacją po wczytaniu lepszą powinna okazać się pierwsza z grup zarówno pod względem czasu jak i zużytej pamięci.

# Badania

Badania zostały przeprowadzone na 2 zestawach danych:

* dane zebrane z programu – 10 obserwacji,
* dane zebrane z narzędzie Profiler – 3 obserwacje.

W obu zbiorach dane zidentyfikowano oraz zredukowano wpływ wartości odstających obserwacji poprzez ponowienie danej obserwacji.

## Testy statystyczne

Aby sprawdzić czy dana różnica pomiędzy pomiarami jest na tyle duża, aby móc mówić o rzeczywistych efektach, a nie przypadku, posłużono się testami statystycznymi. Sprawdzano występowanie różnic pomiędzy grupami obserwacji za pomocą testu Anova. Aby możliwe było użycie tego testu dane muszą mieć rozkład normalny oraz należy sprawdzić homogeniczność wariancji. Do tego celu użyto odpowiednio testów Shaporo-Wilk oraz Levenes.  
Poniżej opisano testy statystyczne użyte w badaniach.

* + 1. Shaporo-wilka

Jest to jeden z podstawowych testów w statystyce. Test ten sprawdza, czy badane wartość posiadają rozkład normalny. Jeśli dane mają rozkład normalny możliwe jest użycie testu Anova. W przeciwnym przypadku należy użyć innych – nieparametrycznych testów – takich jak: Kruskala–Wallisa, test U Manna–Whitneya lub test Friedmana.

**Schemat działania:**

Pierwszym krokiem jest ustalenie badanej hipotezy. Jak podaje strona pogotowiestatystyczne.pl „Hipoteza zerowa dla tego testu zakłada, że nasza próba badawcza pochodzi z populacji o normalnym rozkładzie.”[[25]](#footnote-26).   
Obliczając za pomocą odpowiedniego wzoru wartość W. Wartość ta wskazuje jak bardzo wartości w próbie odbiegają od wartość z rozkładu normalnego. Im wartość W jest bliższa wartości 1 tym bliżej próbce do przyjęcia hipotezy H0.  
W kolejnym kroku wyznaczana jest wartość p na podstawie wartość *W* oraz ilości prób. Na podstawie wartości *p* można przyjąć, bądź odrzucić hipotezę H0.   
„Jeśli test Shapiro-Wilka osiąga istotność statystyczną (p < 0,05), świadczy to o rozkładzie odbiegającym od krzywej Gaussa.”.[[26]](#footnote-27) W przeciwnym przypadku uznaję się, że rozkład jest normalny.

Wszystkie dane otrzymane w badaniach zarówno z grup zebranych z programu, jak i grup zebranych z narzędzia Profiler mają rozkład normalny.

* + 1. Levene

Test ten ocenia homogeniczność wariancji w próbach.

Jak podaje strona pogotowiestatystyczne.pl „Test Levene’a jest wykonywany standardowo przed wykonaniem obliczeń przy użyciu parametrycznych testów średnich – t Studena dla prób niezależnych oraz jednoczynnikowej analizie wariancji, ponieważ jednorodność wariancji jest ich istotnym założeniem. Jeśli wynik test Levene’a jest istotny statystycznie wykorzystujemy jedną z dostępnych “poprawek” na wynik.”[[27]](#footnote-28)

**Schemat działania:**

Ustalenie hipotezy H0 wskazującą na jednorodność wariancji.   
Następnie wyliczana jest odległość każdej obserwacji od średniej próby.   
W kolejnym kroku obliczana jest wartość statystyki *F.*   
Następnie na podstawie wartości *F* oraz liczności próby wyliczana jest wartość *p*.   
Jeśli jest ona większa niż 0,05, oznacza to, że zachodzi homogeniczność wariancji.

W przypadku badanych danych znaczna większość grup miała statystycznie równe wartość wariancji, natomiast zdarzyły się 2 przypadki (badania pamięci z walidacją po wczytaniu całego pliku o rozmiarach 250MB oraz 4GB), dla których to test Levene’a wykazał różnice w wariancjach. Dla tych dwóch przypadków konieczne było użycie testu Kruskal-Wallis’a.

* + 1. Jednoczynnikowa Anova

Test Anova służy do stwierdzenia czy pomiędzy kilkoma grupami danych zachodzą różnice w wartościach wariancji. Według pogotowiestatystyczne.pl „Polega ona na porównaniu wariancji międzygrupowej do wariancji wewnątrzgrupowej.”[[28]](#footnote-29).  
Istotną cechą tego testu jest „Jeśli statystyka F jest mniejsza od 1 oznacza to, że wariancja niewyjaśniona (wewnątrzgrupowa) jest większa od wariancji wyjaśnionej (międzygrupowej).”[[29]](#footnote-30).

Aby użyć tego testu należy spełnić 4 warunki:

1. Normalność rozkładu,
2. Jednorodność wariancji,
3. Niezależność obserwacji,
4. Zmienne w skali ilościowej.

**Schemat działania:**

Ustalenie hipotezy H0 wskazującą na równość wszystkich sprawdzanych wariancji.  
Następnie oblicza się wariancją wewnątrzgrupową oraz międzygrupową.  
W kolejnym kroku obliczana jest wartość statystyki *F*.  
Następnie na podstawie wartości *F* oraz liczności próby wyliczana jest wartość *p*.  
Jeśli jest ona większa niż 0,05, oznacza to, przynajmniej jedna z grup różni się istotnie od pozostałych.

Aby dowiedzieć się, które grupy są różne stasuje się testy post-hoc. W tym przypadku posłużono się testem Tukey HSD.

Rezultaty tych testów oraz testów post-hoc pokazane będą w rozdziale 3.2.

* + 1. Tukey HSD

Jest to jeden z testów post-hoc, służący do wskazywania grup obserwacji różniących się od siebie. Jak podaje sztos-it.com „Test post-hoc Tukeya, znany również jako Tukey's Honest Significant Difference (HSD), to metoda stosowana w analizie wariancji (ANOVA) w celu porównania średnich różnych grup po uzyskaniu istotnego wyniku testu ANOVA. Pomaga on ustalić, które dokładnie grupy różnią się między sobą, podczas gdy ANOVA sama mówi nam jedynie, czy istnieją jakieś różnice między grupami, ale nie wskazuje, które grupy są różne.”[[30]](#footnote-31)

**Schemat działania:**

Ustalenie hipotezy H0 wskazującą na równość średnich wartości sprawdzanych grup.  
Następnie za pomocą odpowiedniego wzoru oblicza się wartość statystyki *q*.  
W kolejnym kroku obliczoną wartość statystyki *q* porównuje się do wartości granicznej dla danej liczby grup oraz stopni swobody.  
Jeśli wartość *q* jest większa oznacza to, że sprawdzane grupy różnią się istotnie od siebie.

* + 1. Kruskala-Wallisa

Jest to nieparametryczny test, służący do sprawdzania różnic pomiędzy grupami danych, odpowiednik Anova.

Jak podaje pogotowiestatystyczne.pl „test Kruskala-Wallisa jest testem nieparametrycznym, co oznacza, że możemy go wykorzystać wtedy, gdy niespełnione są założenia dotyczące stosowania testów parametrycznych lub gdy nasza zmienna zależna ma charakter porządkowy”.[[31]](#footnote-32) Dzięki temu można użyć tego testu w przypadkach gdy jednorodność wariancji nie jest spełniona.

**Schemat działania:**

Ustalenie hipotezy H0 wskazującą na równość wszystkich sprawdzanych median.  
Na początek wszystkie obserwacje zostają uszeregowane oraz przypisuje się im rangi.  
W kolejnym kroku obliczana jest wartość statystyki *H* oraz porównywana jest do rozkładu chi-kwadrat dla *k*-1 stopni swobody, gdzie *k* to liczność grup.  
Następnie na podstawie wartości *H* oraz liczby stopni swobody wyliczana jest wartość *p*.  
Jeśli jest ona większa niż 0,05, oznacza to, przynajmniej jedna z grup różni się istotnie od pozostałych.

Podobnie jak w przypadku Anova, test ten wskazuje tylko różnice w jednej z grup, ale nie wskazuje w której. Aby tego dokonać potrzeby jest test post-hoc np. test Dunna.

Rezultaty tych testów oraz testów post-hoc pokazane będą w rozdziale 3.2.

* + 1. Dunna

Jest Testem post-hoc do znajdowania różnic pomiędzy konkretnymi obserwacjami.

Jak podaje pqstst.pl „Test Dunna zawiera poprawkę na rangi wiązane i jest testem korygowanym ze względu na wielokrotne testowanie. Najczęściej wykorzystuje się tu korektę Bonferroniego lub Sidaka”[[32]](#footnote-33)

**Schemat działania:**Ustalenie hipotezy H0 wskazującą na brak różnic między medianami w sprawdzanych grupach.  
Podobnie jak w teście Kruskala-Wallisa wszystkie obserwacje zostają uszeregowane oraz przypisuje się im rangi.  
Następnie dla każdej z par obliczana jest statystyka *Z.*  
Później z tablic rozkładu normalnego obliczana jest wartość prawdopodobieństwa *p.*W kolejnym kroku stosuje się poprawką Bonferroniego.  
Na koniec jeśli wartość p jest mniejsza od 0,05, oznacza to, że sprawdzana para posiada istotną statystycznie różnicę.

W obu przeprowadzonych badaniach ze względu na wielkość błędu typu pierwszego korekta Bonferroniego wyniosła 0,005.

## Wyniki badań

Dane w zestawianiach zostały przedstawione jako średnie z odchyleniem standardowym próbki. Dane źródłowe zostały dołączone w złączniku do pracy w plikach result.xlsx oraz wyniki\_profiler.xlsx. Wszystkie testy statystyczne zostały przeprowadzone z użyciem narzędzi na stronie https://www.statskingdom.com.

* + 1. Wyniki zebrane za pomocą programu

Dane dla przypadku gdy walidacja jest w trakcie wczytywania danych:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Średni czas [ms] | Odch. std. czasu [ms] | Średnia pamięć [MB] | Odch. std. pamięci [MB] |
| BufferedReader | 100 | 1,21 | 0,01 | 1 364,89 | 249,57 |
| CSVReader | 100 | 1,54 | 0,04 | 1 271,41 | 191,66 |
| FileReader | 100 | 1,26 | 0,03 | 1 343,61 | 213,09 |
| FilesLines | 100 | 1,24 | 0,02 | 1 296,80 | 234,55 |
| Scanner | 100 | 1,58 | 0,02 | 1 877,24 | 345,90 |
| BufferedReader | 250 | 3,39 | 0,06 | 1 567,90 | 387,02 |
| CSVReader | 250 | 4,12 | 0,06 | 1 893,38 | 283,11 |
| FileReader | 250 | 3,39 | 0,07 | 1 778,80 | 286,30 |
| FilesLines | 250 | 3,37 | 0,04 | 1 784,59 | 489,96 |
| Scanner | 250 | 4,37 | 0,06 | 2 268,52 | 148,03 |
| BufferedReader | 500 | 7,04 | 0,09 | 2 824,44 | 433,27 |
| CSVReader | 500 | 8,57 | 0,10 | 2 779,14 | 387,72 |
| FileReader | 500 | 7,16 | 0,13 | 2 859,47 | 431,07 |
| FilesLines | 500 | 7,13 | 0,09 | 3 184,94 | 161,32 |
| Scanner | 500 | 8,89 | 0,08 | 2 908,32 | 547,96 |
| BufferedReader | 1000 | 15,18 | 0,27 | 4 814,11 | 409,51 |
| CSVReader | 1000 | 18,33 | 0,31 | 4 814,24 | 316,65 |
| FileReader | 1000 | 15,52 | 0,23 | 4 870,07 | 400,80 |
| FilesLines | 1000 | 15,36 | 0,36 | 4 817,92 | 388,62 |
| Scanner | 1000 | 19,03 | 0,29 | 6 134,54 | 332,20 |
| BufferedReader | 2000 | 33,53 | 0,65 | 9 150,79 | 555,78 |
| CSVReader | 2000 | 39,96 | 0,76 | 8 873,21 | 340,57 |
| FileReader | 2000 | 33,98 | 0,92 | 8 986,49 | 252,83 |
| FilesLines | 2000 | 33,49 | 0,95 | 9 104,64 | 414,44 |
| Scanner | 2000 | 40,65 | 0,45 | 9 126,64 | 580,67 |
| BufferedReader | 4000 | 71,21 | 0,91 | 16 522,89 | 149,76 |
| CSVReader | 4000 | 85,34 | 1,96 | 16 573,30 | 220,84 |
| FileReader | 4000 | 73,62 | 1,87 | 16 410,09 | 140,44 |
| FilesLines | 4000 | 72,12 | 1,28 | 16 355,65 | 79,58 |
| Scanner | 4000 | 86,71 | 0,88 | 16495,58 | 180,38 |

Tab. 1. Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z napisanego programu dla przypadku, gdy walidacja jest w trakcie wczytywania pliku.

Dane dla przypadku, gdy walidacja była po wczytaniu danych:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Średni czas [ms] | Odch. std czasu [ms] | Średnia Pamięć [MB] | Odch. std pamięci [MB] |
| BufferedReader | 100 | 1,29 | 0,04 | 1 776,08 | 71,76 |
| CSVReader | 100 | 2,09 | 0,02 | 1 678,14 | 304,70 |
| FileReader | 100 | 1,48 | 0,05 | 1 658,36 | 124,51 |
| FilesLines | 100 | 1,33 | 0,07 | 1 739,10 | 99,10 |
| Scanner | 100 | 1,67 | 0,06 | 1 786,08 | 131,96 |
| BufferedReader | 250 | 3,55 | 0,08 | 1 776,08 | 71,76 |
| CSVReader | 250 | 5,44 | 0,07 | 2 272,85 | 490,96 |
| FileReader | 250 | 3,96 | 0,07 | 2 251,01 | 256,00 |
| FilesLines | 250 | 3,57 | 0,09 | 1 920,94 | 281,74 |
| Scanner | 250 | 4,41 | 0,07 | 2 280,80 | 377,18 |
| BufferedReader | 500 | 7,60 | 0,15 | 3 296,17 | 430,49 |
| CSVReader | 500 | 11,35 | 0,14 | 3 186,26 | 342,12 |
| FileReader | 500 | 8,55 | 0,18 | 3 254,55 | 371,65 |
| FilesLines | 500 | 7,64 | 0,16 | 3 332,43 | 503,77 |
| Scanner | 500 | 9,33 | 0,08 | 3 287,51 | 347,56 |
| BufferedReader | 1000 | 16,26 | 0,32 | 5 984,33 | 595,82 |
| CSVReader | 1000 | 23,95 | 0,29 | 5 622,13 | 781,81 |
| FileReader | 1000 | 18,26 | 0,28 | 5 678,65 | 783,84 |
| FilesLines | 1000 | 16,43 | 0,32 | 6 083,24 | 349,28 |
| Scanner | 1000 | 19,88 | 0,15 | 5 576,37 | 569,07 |
| BufferedReader | 2000 | 34,67 | 0,29 | 10 276,34 | 864,32 |
| CSVReader | 2000 | 50,36 | 0,62 | 10 510,77 | 702,20 |
| FileReader | 2000 | 38,28 | 0,58 | 10 279,93 | 502,42 |
| FilesLines | 2000 | 34,91 | 0,31 | 10 366,53 | 707,22 |
| Scanner | 2000 | 41,66 | 0,40 | 10 082,20 | 432,90 |
| BufferedReader | 4000 | 82,05 | 1,02 | 16 255,76 | 352,00 |
| CSVReader | 4000 | 113,39 | 1,39 | 16 269,32 | 476,81 |
| FileReader | 4000 | 89,07 | 1,86 | 16 276,49 | 369,83 |
| FilesLines | 4000 | 82,38 | 1,68 | 16 296,43 | 388,33 |
| Scanner | 4000 | 96,86 | 1,84 | 16 530,40 | 158,68 |

Tab. 2. Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z napisanego programu dla przypadku, gdy walidacja jest po wczytaniu pliku.

* + 1. Dane zebrane za pomocą narzędzia Profiler

Dane dla przypadku gdy walidacja jest w trakcie wczytywania danych:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Czas całkowity [ms] | Odch. std czasu [ms] | Czas wczytywania [ms] | Odch. std czasu [ms] |
| BufferedReader | 100 | 1,46 | 0,04 | 0,44 | 0,15 |
| CSVReader | 100 | 1,82 | 0,10 | 0,70 | 0,03 |
| FileReader | 100 | 1,64 | 0,04 | 0,61 | 0,09 |
| FilesLines | 100 | 1,54 | 0,09 | 0,48 | 0,08 |
| Scanner | 100 | 1,74 | 0,05 | 0,80 | 0,09 |
| BufferedReader | 250 | 3,40 | 0,07 | 1,01 | 0,02 |
| CSVReader | 250 | 4,07 | 0,03 | 1,79 | 0,05 |
| FileReader | 250 | 3,59 | 0,03 | 1,25 | 0,07 |
| FilesLines | 250 | 3,30 | 0,02 | 1,01 | 0,11 |
| Scanner | 250 | 4,29 | 0,04 | 1,85 | 0,04 |
| BufferedReader | 500 | 6,57 | 0,03 | 1,75 | 0,04 |
| CSVReader | 500 | 8,20 | 0,04 | 3,63 | 0,12 |
| FileReader | 500 | 7,66 | 0,46 | 2,85 | 0,31 |
| FilesLines | 500 | 6,68 | 0,13 | 1,98 | 0,23 |
| Scanner | 500 | 8,42 | 0,09 | 3,74 | 0,05 |
| BufferedReader | 1000 | 13,45 | 0,45 | 3,47 | 0,06 |
| CSVReader | 1000 | 17,07 | 0,37 | 7,15 | 0,01 |
| FileReader | 1000 | 14,81 | 0,34 | 4,62 | 0,13 |
| FilesLines | 1000 | 13,79 | 0,44 | 3,96 | 0,37 |
| Scanner | 1000 | 16,88 | 0,31 | 7,09 | 0,12 |
| BufferedReader | 2000 | 29,64 | 0,73 | 7,80 | 0,58 |
| CSVReader | 2000 | 35,94 | 0,53 | 14,63 | 0,40 |
| FileReader | 2000 | 32,25 | 1,13 | 10,26 | 1,61 |
| FilesLines | 2000 | 29,61 | 0,87 | 7,96 | 0,28 |
| Scanner | 2000 | 35,79 | 0,10 | 14,46 | 0,35 |
| BufferedReader | 4000 | 65,92 | 3,05 | 15,98 | 0,51 |
| CSVReader | 4000 | 77,91 | 0,51 | 29,62 | 0,21 |
| FileReader | 4000 | 67,25 | 1,12 | 19,88 | 0,49 |
| FilesLines | 4000 | 59,65 | 2,49 | 15,05 | 0,96 |
| Scanner | 4000 | 72,72 | 2,27 | 27,05 | 1,57 |

Tab. 3 Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z narzędzia Profiler dla przypadku, gdy walidacja jest w trakcie wczytywania pliku (1/2).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Czas walidacji [ms] | Odch. std czasu [ms] | Całkowita zaalokowana pamięć [GB] | Odch. std pamięci [GB] |
| BufferedReader | 100 | 1,01 | 0,16 | 1,77 | 0,01 |
| CSVReader | 100 | 1,12 | 0,13 | 1,68 | 0,00 |
| FileReader | 100 | 1,03 | 0,08 | 1,68 | 0,00 |
| FilesLines | 100 | 1,07 | 0,04 | 1,76 | 0,01 |
| Scanner | 100 | 0,95 | 0,04 | 2,26 | 0,01 |
| BufferedReader | 250 | 2,39 | 0,06 | 4,38 | 0,01 |
| CSVReader | 250 | 2,29 | 0,04 | 4,16 | 0,01 |
| FileReader | 250 | 2,34 | 0,09 | 4,13 | 0,00 |
| FilesLines | 250 | 2,29 | 0,11 | 4,38 | 0,00 |
| Scanner | 250 | 2,43 | 0,01 | 5,58 | 0,01 |
| BufferedReader | 500 | 4,81 | 0,07 | 8,75 | 0,00 |
| CSVReader | 500 | 4,57 | 0,13 | 8,30 | 0,01 |
| FileReader | 500 | 4,81 | 0,16 | 8,24 | 0,00 |
| FilesLines | 500 | 4,69 | 0,10 | 8,75 | 0,00 |
| Scanner | 500 | 4,68 | 0,06 | 11,14 | 0,00 |
| BufferedReader | 1000 | 9,98 | 0,51 | 17,49 | 0,00 |
| CSVReader | 1000 | 9,92 | 0,36 | 16,58 | 0,00 |
| FileReader | 1000 | 10,19 | 0,38 | 16,48 | 0,02 |
| FilesLines | 1000 | 9,83 | 0,21 | 17,46 | 0,02 |
| Scanner | 1000 | 9,79 | 0,41 | 22,28 | 0,03 |
| BufferedReader | 2000 | 21,84 | 0,63 | 34,97 | 0,05 |
| CSVReader | 2000 | 21,31 | 0,36 | 33,19 | 0,00 |
| FileReader | 2000 | 21,99 | 0,67 | 32,92 | 0,01 |
| FilesLines | 2000 | 21,66 | 0,60 | 34,94 | 0,05 |
| Scanner | 2000 | 21,33 | 0,41 | 44,63 | 0,00 |
| BufferedReader | 4000 | 49,94 | 3,18 | 69,73 | 0,11 |
| CSVReader | 4000 | 48,29 | 0,49 | 66,18 | 0,01 |
| FileReader | 4000 | 47,37 | 0,83 | 65,80 | 0,01 |
| FilesLines | 4000 | 44,61 | 1,58 | 69,73 | 0,10 |
| Scanner | 4000 | 45,68 | 0,71 | 89,05 | 0,00 |

Tab. 4 Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z narzędzia Profiler dla przypadku, gdy walidacja jest w trakcie wczytywania pliku (2/2).

Dane dla przypadku gdy walidacja następuje po wczytaniu danych:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Czas całkowity [ms] | Odch. std czasu [ms] | Czas wczytywania [ms] | Odch. std czasu [ms] |
| BufferedReader | 100 | 1,57 | 0,06 | 0,54 | 0,07 |
| CSVReader | 100 | 2,12 | 0,07 | 1,22 | 0,15 |
| FileReader | 100 | 1,71 | 0,07 | 0,66 | 0,06 |
| FilesLines | 100 | 1,58 | 0,03 | 0,47 | 0,08 |
| Scanner | 100 | 1,92 | 0,05 | 0,89 | 0,03 |
| BufferedReader | 250 | 3,37 | 0,07 | 1,19 | 0,06 |
| CSVReader | 250 | 5,34 | 0,09 | 2,98 | 0,13 |
| FileReader | 250 | 3,68 | 0,25 | 1,25 | 0,07 |
| FilesLines | 250 | 3,45 | 0,06 | 1,08 | 0,20 |
| Scanner | 250 | 4,38 | 0,10 | 1,88 | 0,08 |
| BufferedReader | 500 | 6,74 | 0,07 | 2,05 | 0,15 |
| CSVReader | 500 | 10,54 | 0,15 | 5,76 | 0,15 |
| FileReader | 500 | 7,58 | 0,07 | 2,77 | 0,26 |
| FilesLines | 500 | 6,78 | 0,17 | 2,13 | 0,19 |
| Scanner | 500 | 8,54 | 0,27 | 3,77 | 0,08 |
| BufferedReader | 1000 | 14,44 | 0,20 | 4,27 | 0,25 |
| CSVReader | 1000 | 21,55 | 0,31 | 11,55 | 0,22 |
| FileReader | 1000 | 16,28 | 0,34 | 6,08 | 0,48 |
| FilesLines | 1000 | 14,34 | 0,27 | 4,22 | 0,15 |
| Scanner | 1000 | 18,05 | 0,20 | 7,42 | 0,12 |
| BufferedReader | 2000 | 30,90 | 0,63 | 8,60 | 0,18 |
| CSVReader | 2000 | 46,03 | 1,02 | 23,08 | 0,64 |
| FileReader | 2000 | 33,90 | 1,95 | 11,96 | 0,58 |
| FilesLines | 2000 | 30,52 | 1,16 | 8,68 | 0,40 |
| Scanner | 2000 | 37,40 | 0,65 | 15,21 | 0,50 |
| BufferedReader | 4000 | 65,91 | 0,52 | 17,65 | 0,29 |
| CSVReader | 4000 | 96,32 | 0,29 | 47,96 | 0,75 |
| FileReader | 4000 | 72,91 | 0,61 | 24,93 | 0,15 |
| FilesLines | 4000 | 67,67 | 0,65 | 18,42 | 1,14 |
| Scanner | 4000 | 78,27 | 3,39 | 30,54 | 1,01 |

Tab. 5 Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z narzędzia Profiler dla przypadku, gdy walidacja jest po wczytaniu pliku (1/2).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorytm | Rozmiar pliku [MB] | Czas walidacji [ms] | Odch. std czasu [ms] | Całkowita zaalokowana pamięć [GB] | Odch. std pamięci [GB] |
| BufferedReader | 100 | 1,03 | 0,01 | 1,81 | 0,01 |
| CSVReader | 100 | 0,93 | 0,03 | 2,85 | 0,01 |
| FileReader | 100 | 1,05 | 0,02 | 1,89 | 0,01 |
| FilesLines | 100 | 1,12 | 0,05 | 1,81 | 0,01 |
| Scanner | 100 | 1,04 | 0,05 | 2,32 | 0,07 |
| BufferedReader | 250 | 2,18 | 0,03 | 4,50 | 0,01 |
| CSVReader | 250 | 2,37 | 0,10 | 7,10 | 0,01 |
| FileReader | 250 | 2,28 | 0,01 | 4,72 | 0,01 |
| FilesLines | 250 | 2,36 | 0,14 | 4,46 | 0,07 |
| Scanner | 250 | 2,50 | 0,02 | 5,83 | 0,11 |
| BufferedReader | 500 | 4,69 | 0,19 | 8,96 | 0,01 |
| CSVReader | 500 | 4,78 | 0,14 | 14,17 | 0,01 |
| FileReader | 500 | 4,81 | 0,22 | 9,38 | 0,01 |
| FilesLines | 500 | 4,65 | 0,15 | 8,96 | 0,01 |
| Scanner | 500 | 4,78 | 0,20 | 11,50 | 0,23 |
| BufferedReader | 1000 | 10,18 | 0,29 | 17,91 | 0,02 |
| CSVReader | 1000 | 10,01 | 0,09 | 28,34 | 0,03 |
| FileReader | 1000 | 10,20 | 0,16 | 18,74 | 0,02 |
| FilesLines | 1000 | 10,12 | 0,32 | 17,91 | 0,01 |
| Scanner | 1000 | 10,63 | 0,16 | 22,98 | 0,45 |
| BufferedReader | 2000 | 22,40 | 0,41 | 35,92 | 0,05 |
| CSVReader | 2000 | 22,95 | 0,38 | 56,69 | 0,01 |
| FileReader | 2000 | 21,95 | 1,37 | 37,52 | 0,05 |
| FilesLines | 2000 | 21,84 | 0,97 | 35,89 | 0,05 |
| Scanner | 2000 | 22,19 | 0,20 | 46,00 | 0,88 |
| BufferedReader | 4000 | 48,26 | 0,71 | 71,79 | 0,00 |
| CSVReader | 4000 | 48,36 | 0,67 | 113,43 | 0,01 |
| FileReader | 4000 | 47,98 | 0,49 | 74,94 | 0,13 |
| FilesLines | 4000 | 49,25 | 1,68 | 71,59 | 0,34 |
| Scanner | 4000 | 47,70 | 2,44 | 90,92 | 0,12 |

Tab. 6 Przedstawiająca średnie oraz odchylenia standardowe danych pochodzące z narzędzia Profiler dla przypadku, gdy walidacja jest po wczytaniu pliku (2/2).

## Wykresy

W tej części w sposób wizualny zostały przedstawione dane zawarte we wcześniejszym podrozdziale. Na wykresach tych łatwo można dostrzec różnice i wytypować lepsze i gorsze rozwiązania.

* + 1. Wyniki zebrane za pomocą programu

**Wykresy przestawiające czas działania w zależności od rozmiaru plików dla poszczególnych algorytmów:**

Wyk. 1. Przedstawiający zależność czasu od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

Na wykresie pierwszym widać, że najlepszy czas uzyskały algorytmy oparte o BufferedReader oraz FilesLines. Natomiast najgorszy czas uzyskały CSVReader oraz Scanner. W żadnym z przypadków pomiędzy dwoma najszybszymi rozwiązaniami nie występują istotne statystycznie różnice. Co za tym idzie nie można jednoznacznie wskazać, który z tych dwóch podejść jest wydajniejsze pod względem czasu. Efekt istotnie statystycznie pojawiają się pomiędzy dwoma pierwszymi, a kolejnym algorytmem, czyli FileReader. Pomiędzy 2 najwolniejszymi rozwiązaniami istotna różnica jest do pliku o wielkości 1GB. Szybszy w tych przypadkach jest CSVReader, nad Scannerem. Powyżej tej granicy nie można jednoznacznie wskazać najwolniejszego algorytmu.

Wyk. 2. Przedstawiający zależność czasu od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

W przypadku opóźnionej walidacji ponownie najszybszymi algorytmami są BufferedReader oraz FilesLines i ponownie nie istnieje statystycznie istotna różnica pomiędzy tymi algorytmami. Natomiast wszystkie pozostałe przypadki są na tyle zróżnicowane, że każde różnice są potwierdzone analizą statystyczną. Kolejnymi podejścia to: FileReader, Scanner oraz najwolniejszy ze stawki CSVReader.

Wyk. 3. Przedstawiający zależność czasu od rozmiaru dla porównania średnich czasów pomiędzy podejściami do walidacji.

Wykres, który przedstawia średnie z 5 algorytmów z walidacją w trakcie wczytywania oraz z walidacją po wczytaniu całego pliku, aby w lepszy sposób ukazać różnicę w czasie pomiędzy tymi podejściami.

Wykresy przedstawiające zużycie pamięci podczas działania w zależności od rozmiaru pliku dla poszczególnych algorytmów:

Wyk. 4. Przedstawiający zależność pamięci od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

W przypadku zużycia pamięci z walidacją podczas wczytywania, o różnicach możemy mówić tylko w przypadku Scannera i to w przypadkach 100MB, 250MB oraz 1000MB. W pozostałych przypadkach nie stwierdzono statystycznie żadnych różnic między algorytmami.

Wyk. 5. Przedstawiający zależność czasu od rozmiaru dla przypadku, gdy walidacja odbywa się po wczytywaniu pliku.

Zużycie pamięci z walidacją uruchomioną po zrealizowaniu operacji wejścia/wyjścia, było we wszystkich przypadkach równe pod względem statystycznym.

Porównując pamięć względem sposobu walidacji, okazuje się, że dla każdej próbki poniżej 4GB walidacja w trakcie wczytywania jest bardziej wydajna pamięciowo. W przypadku 4GB algorytmy z późniejszą walidacją są mocno organiczne przez limit pamięci operacyjnej dla programu, przez co wyniki są porównywalne.

* + 1. Wyniki zebrane za pomocą narzędzia Profiler

**Wykresy przestawiające poszczególne czasy działania w zależności od rozmiaru plików dla poszczególnych algorytmów:**

Wyk. 6 Przedstawiający zależność całkowitego czasu działania programu od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

Na wykresie widać, że najmniej czasu na działanie potrzebował algorytm FilesLines oraz BufferedReader. Dla pliku 4GB istnieje statystycznie istotna różnica pomiędzy tymi algorytmami i można wskazać, że FilesLines przetworzył dane szybciej niż BufferedReader. Jednak dla pozostałych przypadków różnica była zbyt mała, aby jednoznacznie wskazać zwycięzce.  
FileReader we wszystkich przypadkach był w środku stawki, czasem współdzieląc miejsce z BufferedReader dla pliku 4GB lub z CSVReader oraz Scannerem dla pliku 100MB.   
Wymieniona dwójka najmniej wydajnych algorytmów tylko w przypadku pliku 1000MB miała istotną statystycznie różnicę w czasie działania. W tym przypadku lepszy okazał się Scanner nad CSVReaderem.

Wyk. 7. Przedstawiający zależność całkowitego czasu działania programu od rozmiaru dla przypadku, gdy walidacja następuję po wczytaniu pliku.

Najszybsza dwójka algorytmów tj. BufferedReader oraz FilesLines nie miała wystarczająco dużej różnicy między sobą, aby była statystycznie istotna.   
Z kolei FileReader w przypadku 100MB oraz 250MB pliku również znajdował się na miejscu pierwszym. W każdym większym pliku był na miejscy 3 z istotną statystycznie różnicą między innymi algorytmami.  
Kolejnym algorytmem jest Scanner w każdym przypadku zajmujący 4. miejsce oraz zawsze mający statystycznie istotne różnice do sąsiednich rozwiązań.  
Najwolniejszym algorytmem potwierdzonym analizą dla każdego z plików okazał się CSVReader.

Wyk. 8 Przedstawiający zależność czasu trwania procesu samego wczytywania i obrabiania danych w programie od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

Najistotniejsze i najdokładniejsze wyniki pokazuje właśnie wykres samego czasu wczytywania.  
Najlepszymi algorytmami okazały się BufferedReader oraz FilesLines bez statystycznie istotnej różnicy między nimi, co oznacza, że nie można jednoznacznie wskazać jednego najszybszego algorytmu.  
Dla pliku 100MB różnice były na tyle małe, że żadna para algorytmów nie ma istotnej statystycznie różnicy.   
Środek stawki zajmuje FileReader w każdym przypadku (oprócz 100MB) mając istotną różnicę między sąsiednimi miejscami.  
Natomiast różnica pomiędzy Scannerem a CSVReaderem tylko dla pliku 4000MB zachodzi efekt istotny statystycznie. W pozostałych przypadkach nie można stwierdzić dokładnej kolejności tych algorytmów.

Wyk. 9 Przedstawiający zależność czasu trwania procesu samego wczytywania i obrabiania danych w programie od rozmiaru dla przypadku, gdy walidacja następuje po wczytaniu pliku.

Ponownie najlepszymi algorytmami okazały się BufferedReader oraz FilesLines bez statystycznie istotnej różnicy między nimi, co oznacza, że nie można jednoznacznie wskazać jednego najszybszego algorytmu.  
Dla plików 100MB oraz 250MB na pierwszym miejscy znalazł się także FileReader. Dla większych plików różnica wzrosła na tyle, że jest ona potwierdzona analizą.  
Kolejnym algorytmem w zestawieniu jest Scanner, a zdecydowanie najgorszym okazał się CSVReader. Oba algorytmy mają istotne statystycznie różnice pomiędzy sobą jak i pozostałymi rozwiązaniami.

Wyk. 10 Przedstawiający zależność czasu samego wczytywania i obrabiania danych w programie od rozmiaru, dla porównania średnich czasów pomiędzy podejściami do walidacji.

Porównując oba podejścia do momentu walidacji, ewidentnie widać, że walidacja w trakcie wczytywania jest znacznie efektywniejsza pod względem czasu niżeli walidacja przeprowadzona po załadowaniu całego pliku do pamięci.

Wyk. 11 Przedstawiający zależność czasu trwania procesu walidacji w programie od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania.

Zgodnie z oczekiwaniami czas samej walidacji plików jest pod względem statystycznym identyczny dla każdego rozmiaru pliku.

Wyk. 12 Przedstawiający zależność czasu trwania procesu walidacji w programie od rozmiaru dla przypadku, gdy walidacja następuje po wczytaniu pliku.

Podobnie jak w przypadku walidacji w trakcie wczytywania tutaj również czas walidacji jest równy pod względem statystycznym.

**Wykresy przedstawiające całkowitą zaalokowaną pamięci podczas działania programu w zależności od rozmiaru pliku dla poszczególnych algorytmów:**

Wyk. 13 Przedstawiający zależność całkowitej zaalokowanej pamięci od rozmiaru dla przypadku, gdy walidacja odbywa się w trakcie wczytywania pliku.

Najmniej pamięci zaalokował algorytm FileReader, dzieląc pierwsze miejsce tylko w przypadku 100MB z CSVReaderem. W pozostałych przypadkach różnica między tą parą była istotna statystycznie.  
W środku stawki znalazły się 2 algorytmy BufferedReader oraz CSVReader, które między sobą nie mają istotnej statystycznie różnicy, ale mają takowa pomiędzy sąsiednimi miejscami.  
Niechlubne ostatnie miejsce i najwięcej zaalokowanej pamięci zajął Scanner.

Wyk. 14 Przedstawiający zależność całkowitej zaalokowanej pamięci od rozmiaru dla przypadku, gdy walidacja następuje po wczytywaniu pliku.

Najmniej zaalokowanej pamięci przypadło dwóm algorytmom: BufferedReader oraz FilesLines. Różnica między nimi jest zbyt mała, aby można było wskazać faworyta. Dodatkowo dla pliku 100MB trzeci algorytm, czyli FileReader uzyskał taki sam wynik. Dla większych plików FileReader zajmował miejsce trzecie.   
Czwarte miejsce przypadło podejściu Scannera.  
Algorytmem, który zaalokował najwięcej pamięci został CSVReader.   
Wszystkie algorytmy z miejsc 3-5 miały między sąsiadującymi miejscami istotne statystycznie różnice.

## Wnioski

# Podsumowanie

# Spis table i wykresów

# Bibliografia

* TODO alfabetycznie
* https://pogotowiestatystyczne.pl/slowniki/test-shapiro-wilka/

1. M. Ramgir i N. Samoylov, *Java 9 High Performance.* Wyd. Packt, tł. włane, Birmingham 2017 [↑](#footnote-ref-2)
2. Na czas pisania tej pracy najnowszą wersja języka Java w wersji LTS była właśnie wersja 24. [↑](#footnote-ref-3)
3. J. Jenkov, *Java FileReader*, 2021, https://jenkov.com/tutorials/java-io/filereader.html [dostęp. 09.09.2025]. [↑](#footnote-ref-4)
4. Venniladeenan, *Java FileReader Class* ,2025, https://www.geeksforgeeks.org/java/java-io-filereader-class/ [dostęp. 09.09.2025]. [↑](#footnote-ref-5)
5. Oracle, *Class InputStreamReader*, https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html#read-- [dostęp. 09.09.2025] [↑](#footnote-ref-6)
6. J. Jenkov, *Java FileReader*, 2021, https://jenkov.com/tutorials/java-io/filereader.html [dostęp. 09.09.2025]. [↑](#footnote-ref-7)
7. Oracle, *Interface AutoCloseable*, https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html[dostęp. 09.09.2025]. [↑](#footnote-ref-8)
8. Flanagan D. *Java in a Nutshell*, O'Reilly Media, 2019, str. 452. [↑](#footnote-ref-9)
9. Oracle, *Class BufferedReader*, https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html[dostęp. 09.09.2025]. [↑](#footnote-ref-10)
10. Oracle *Class BufferedReader*, https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html [dostęp. 09.09.2025]. [↑](#footnote-ref-11)
11. Oracle *Class BufferedReader*, https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/io/BufferedReader.html [dostęp. 09.09.2025]. [↑](#footnote-ref-12)
12. Sierra K., Bates B., Gee T. Java. Rusz głową!, Helion 2010 wyd III. str. 478. [↑](#footnote-ref-13)
13. Sierra K., Bates B., Gee T. Java. Rusz głową!, Helion 2010 wyd III. str. 477-481. [↑](#footnote-ref-14)
14. Jenkov J. *Java BufferedReader*, https://jenkov.com/tutorials/java-io/bufferedreader.html[dostęp. 09.09.2025]. [↑](#footnote-ref-15)
15. Oracle *Class Scanner*, https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html [dostęp: 09.09.2025] [↑](#footnote-ref-16)
16. Kumar P. *Scanner Class in Java*, https://www.digitalocean.com/community/tutorials/scanner-class-in-java [dostęp: 09.09.2025] [↑](#footnote-ref-17)
17. W3Schools *Java Scanner Methods*, https://www.w3schools.com/java/java\_ref\_scanner.asp [dostęp: 09.09.2025] [↑](#footnote-ref-18)
18. Morelli R., Walde R. *Java, Java, Java Object-Oriented Problem Solving,* Trinity College 2016 str. 183-187 [↑](#footnote-ref-19)
19. W3Schools *Java Scanner close() Method*, https://www.w3schools.com/java/ref\_scanner\_close.asp [dostęp: 09.09.2025] [↑](#footnote-ref-20)
20. Oracle *Class Files*, https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/nio/file/Files.html [dostęp: 09.09.2025] [↑](#footnote-ref-21)
21. Bloch J. *Effective Java*, Addison Wesley 2018 str. 203-209. [↑](#footnote-ref-22)
22. Opencsv *Opencsv Users Guide*, 2025, https://opencsv.sourceforge.net/ [dostęp: 10.09.2025] [↑](#footnote-ref-23)
23. Amansingh2210 *Reading a CSV file in Java using OpenCSV,* 2025, https://www.geeksforgeeks.org/java/reading-csv-file-java-using-opencsv/ [dostęp: 10.09.2025] [↑](#footnote-ref-24)
24. Opencsv *Class CSVReader*, https://opencsv.sourceforge.net/apidocs/com/opencsv/CSVReader.html#hasNext [dostęp: 10.09.2025] [↑](#footnote-ref-25)
25. P. Iwankowski, *Test Shapiro-Wilka.* https://pogotowiestatystyczne.pl/slowniki/test-shapiro-wilka/ [dostęp: 06.09.2025]. [↑](#footnote-ref-26)
26. P. Iwankowski, *Test Shapiro-Wilka.* https://pogotowiestatystyczne.pl/slowniki/test-shapiro-wilka/ [dostęp: 06.09.2025]. [↑](#footnote-ref-27)
27. P. Iwankowski, *Test Levene’a.* https://pogotowiestatystyczne.pl/slowniki/test-levenea/ [dostęp: 06.09.2025]. [↑](#footnote-ref-28)
28. P. Iwankowski, *Test ANOVA.* https://pogotowiestatystyczne.pl/slowniki/anova/ [dostęp: 06.09.2025]. [↑](#footnote-ref-29)
29. P. Iwankowski, *Test ANOVA.* https://pogotowiestatystyczne.pl/slowniki/anova/ [dostęp: 06.09.2025]. [↑](#footnote-ref-30)
30. Sztos IT, Post Hoc Tukeya, https://sztos-it.com/wzory\_statystyczne\_analiza\_post\_hoc\_tukeya.html [dostęp: 06.09.2025]. [↑](#footnote-ref-31)
31. P. Iwankowski, *Test Kruskala-Wallisa.* https://pogotowiestatystyczne.pl/slowniki/test-kruskala-wallisa/ [dostęp: 06.09.2025]. [↑](#footnote-ref-32)
32. Admin, *Testy nieparametryczne*, 2014. https://manuals.pqstat.pl/statpqpl:porown3grpl:nparpl [dostęp: 06.09.2025]. [↑](#footnote-ref-33)