

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)

[2.1. Model danych 11](#__RefHeading___Toc693_2502114822)

[2.2. Walidacja 13](#__RefHeading___Toc695_2502114822)

[2.3. Wczytywanie danych 17](#__RefHeading___Toc1049_2579040556)

[3. Badania 22](#__RefHeading___Toc345_1335475158)

[4. Wnioski 23](#__RefHeading___Toc347_1335475158)

[5. Podsumowanie 24](#__RefHeading___Toc349_1335475158)

[6. Bibliografia 25](#__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.

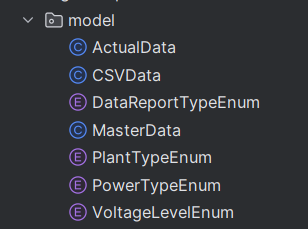
Testy przeprowadzane przy użyciu *Profiler’a* w narzędziu *IntelliJ IEAD* były wykonywane pojedynczo dla każdego wywołania i ręcznie zbierano dane. Po wykonaniu programu została pobrana wartość całkowitej pamięci zarezerwowanej przez poszczególne algorytmy, jak i czas ich wykonywania. Dodatkowo możliwy 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.

# 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:

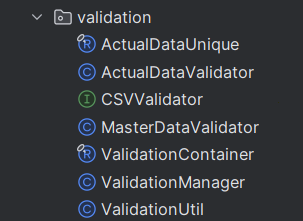
* *DataReportTypeEnum*:  
  *TYPE\_1*, *TYPE\_2*, *TYPE\_3*, *TYPE\_4*, *TYPE\_5*, *TYPE\_6*
* *PlantTypeEnum*:  
  *SOLAR*, *BIOMASS*, *COAL*, *WIND*, *ATOM*, *WATER*, *GAS*, *GEOTHERMAL*
* *PowerTypeEnum*:  
  *TYPE0*, *TYPE1*, *TYPE2*, *TYPE3*, *TYPE4*, *TYPE5*
* *VoltageLevelEnum*:  
  *VLE00*, *VLE01*,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

# 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*.

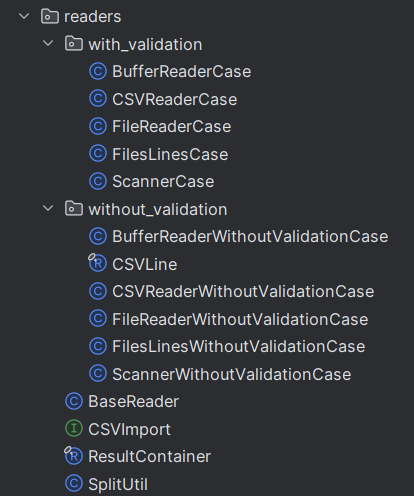
Podobnie jak implementacja w klasie *MasterDataValidator*, tak w klasie Actual*DataValidator* sprawdzana jest na początku poprawność ilości tokenów w tablicy. Powinno ich być 8. Jeśli w tym lub innym przypadku nastąpi niezgodność z walidacją do kolekcji błędów zostanie dodany odpowiedni wpis.  
Następnie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

//TODO opis teoretyczny

**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

//TODO opis teoretyczny + książki 479

**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

//TODO opis teoretyczny

**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()

//TODO opis teoretyczny

**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

//TODO opis teoretyczny

**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.

# Badania

# Wnioski

# Podsumowanie

# Bibliografia

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)