Inżyniera systemów informatycznych

Sprawozdanie

inż. Dávid Ali
inż. Julia Winiarska
Państwowa Akademia Nauk Stosowanych w Nysie
Wydział Nauk Technicznych

2026

Streszczenie

Niniejszy dokument stanowi kompleksowe sprawozdanie z realizacji projektu inżynierskiego "OptimaConverterApp", którego celem było zaprojektowanie i implementacja rozwiązania informatycznego klasy middleware, umożliwiającego integrację heterogenicznych źródeł danych z systemem klasy ERP – Comarch ERP Optima. Projekt koncentruje się na automatyzacji procesu importu dokumentów handlowych oraz kartotek kontrahentów, wykorzystując paradygmat ETL (Extract, Transform, Load). Sprawozdanie obejmuje szczegółową analizę teoretyczną zastosowanych wzorców projektowych, weryfikację mechanizmów walidacji danych w kontekście systemów finansowo-księgowych oraz krytyczną analizę kodu źródłowego pod kątem wydajności i bezpieczeństwa. W pracy przedstawiono również zagadnienia związane z obsługą specyficznych formatów plików (CSV, XML), problematyką kodowania znaków oraz architekturą aplikacji okienkowych w środowisku Java Swing. Dokument ten pełni rolę dokumentacji technicznej, raportu z testów oraz analizy badawczej nad efektywnością metod przetwarzania danych tekstowych.

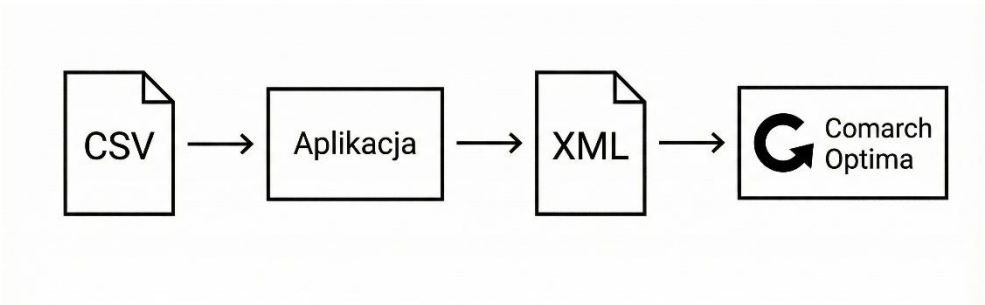
Spis treści

Rozdział 1
Wstęp i kontekst teoretyczny integracji systemów ERP

1.1. Definicja problemu i uzasadnienie biznesowe

Współczesne ekosystemy informatyczne przedsiębiorstw charakteryzują się wysokim stopniem fragmentacji. Organizacje często wykorzystują wyspecjalizowane systemy dziedzinowe (np. platformy e-commerce, systemy CRM, dedykowane aplikacje sprzedażowe), które funkcjonują niezależnie od centralnego systemu zarządzania zasobami przedsiębiorstwa (ERP). Brak automatycznej wymiany danych między tymi systemami prowadzi do konieczności ręcznego przepisywania informacji, co jest procesem czasochłonnym i podatnym na błędy ludzkie.

System Comarch ERP Optima, będący jednym z najpopularniejszych rozwiązań na polskim rynku, oferuje mechanizm "Pracy Rozproszonej" jako standardowy interfejs wymiany danych. Mechanizm ten opiera się na asynchronicznej wymianie plików w formacie XML, co pozwala na integrację systemów nieposiadających bezpośredniego połączenia sieciowego lub pracujących w trybie offline. Projekt 'OptimaConverterApp' odpowiada na zapotrzebowanie rynkowe na narzędzie typu "most", które konwertuje płaskie struktury danych (CSV), generowane przez systemy zewnętrzne, na hierarchiczny format XML wymagany przez system ERP.



Rysunek 1. Schemat ideowy przepływu danych w procesie integracji.

1.2. Teoretyczne podstawy procesu ETL

Architektura aplikacji 'OptimaConverterApp' została oparta na wzorcu ETL (Extract-Transform-Load), który jest standardem w procesach integracji danych i hurtowniach danych:

1. **Extract (Ekstrakcja):** Proces ten w kontekście projektu polega na odczycie danych z plików tekstowych CSV (Comma-Separated Values). Jest to faza krytyczna ze względu na konieczność obsługi różnych standardów kodowania (UTF-8, Windows-1250) oraz wariantów formatu CSV. W analizowanym kodzie odpowiada za to metoda *readCsvFile* wykorzystująca bibliotekę OpenCSV.
2. **Transform (Transformacja):** Etap ten obejmuje konwersję typów danych (np. String do Double lub Date), walidację reguł biznesowych (np. poprawność NIP, format daty) oraz mapowanie struktur płaskich na obiekty domenowe. W projekcie zastosowano szereg walidatorów (np. *validateDate*, *validateSplitNumeric*), które realizują strategię fail-safe lub fail-fast w zależności od krytyczności błędu.
3. **Load (Ładowanie):** Ostatnia faza to serializacja przetworzonych danych do docelowego formatu XML. Wymaga to ścisłego przestrzegania schematu XSD (XML Schema Definition) narzuconego przez producenta systemu ERP. W projekcie wykorzystano model DOM (Document Object Model) do budowy drzewa XML w pamięci, a następnie jego zapisu do pliku wynikowego.

1.3. Cel pracy

Celem niniejszego sprawozdania jest nie tylko dokumentacja wytworzonego oprogramowania, ale również pogłębiona analiza wyzwań inżynierskich napotkanych podczas integracji z systemem Comarch ERP Optima. Szczególny nacisk położono na aspekty walidacji danych, które są kluczowe dla zapewnienia spójności ksiąg rachunkowych. Analiza obejmuje również ocenę wydajności zastosowanych algorytmów parsowania i generowania danych w kontekście dużych wolumenów informacji.

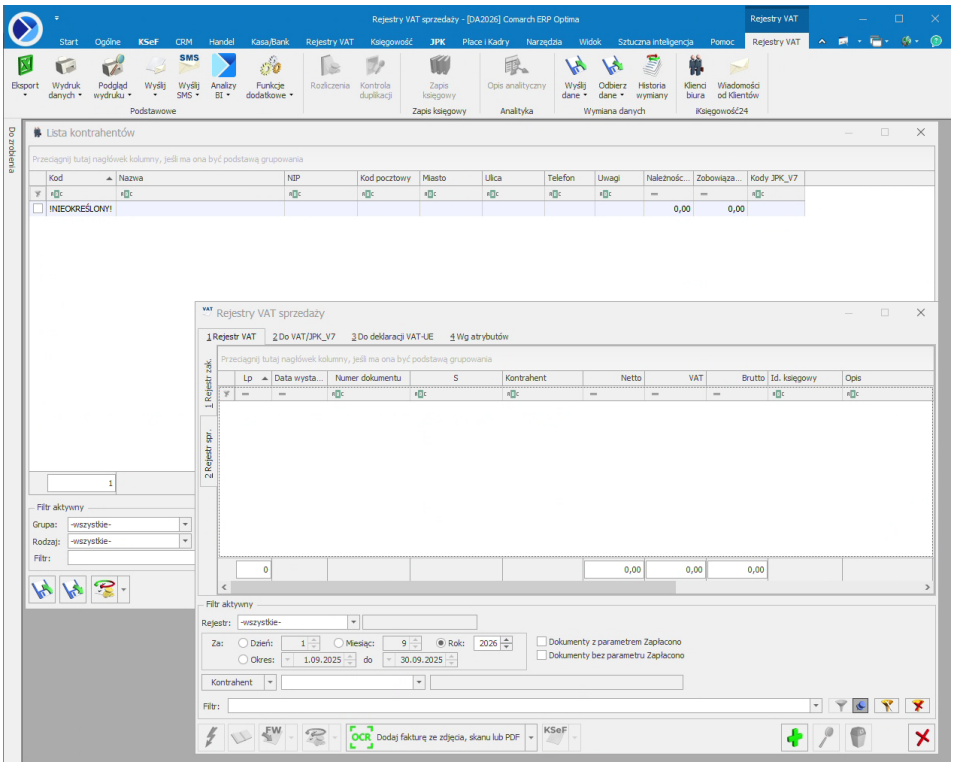
Rozdział 2

Specyfikacja techniczna formatu wymiany danych

2.1. Mechanizm pracy rozproszonej w Comarch ERP Optima

"Praca Rozproszona" to unikalna funkcjonalność systemu Comarch, umożliwiająca wymianę danych między różnymi bazami (np. oddział-centrala lub klient-biuro rachunkowe). Wymiana ta odbywa się za pośrednictwem plików XML o ściśle zdefiniowanej strukturze.

Kluczowym elementem konfiguracji jest "Identyfikator księgowości". Jest to unikalny ciąg znaków (np. "K1", "SKLEP"), który identyfikuje źródło pochodzenia danych. Zgodnie z dokumentacją, aby import powiódł się, identyfikator zawarty w pliku XML (w tagu <BAZA_ZRD_ID>) musi być zgodny z identyfikatorem zdefiniowanym w konfiguracji systemu docelowego (System -> Konfiguracja -> Firma -> Ogólne -> Praca rozproszona). W analizowanym kodzie aplikacji, stała BAZA_ZRD_ID = "CSV_IMPORT" pełni tę funkcję, co wymusza na użytkowniku odpowiednią konfigurację po stronie ERP.



Rysunek 2. Konfiguracja systemu Comarch ERP Optima

2.2. Struktura pliku XML

Format pliku XML wykorzystywany w "Pracy Rozproszonej" jest strukturą hierarchiczną, której korzeniem jest element <ROOT>. Dokumentacja techniczna definiuje szereg węzłów potomnych, z których w projekcie zaimplementowano dwa kluczowe:

1. **<KONTRAHENCI>**: Węzeł ten zawiera listę partnerów biznesowych. Każdy element <KONTRAHENT> musi posiadać unikalny identyfikator (często ID_ZRODLA lub AKRONIM). System Optima wykorzystuje te dane do synchronizacji kartotek, jeśli kontrahent o danym numerze NIP już istnieje, jego dane mogą zostać zaktualizowane lub pominięte w zależności od konfiguracji importu.

2. **<REJESTRY_SPRZEDAZY_VAT>**: Jest to węzeł zawierający dokumenty sprzedaży. Struktura pojedynczego dokumentu (<REJESTR>) jest złożona i obejmuje:
    - **Nagłówek:** Data wystawienia, numer dokumentu, termin płatności.
    - **Podmiot:** Dane nabywcy (referencja do kontrahenta).
    - **Pozycje (<POZYCJE>):** Szczegółowe informacje o stawkach VAT i kwotach.
    - **Płatności (<PLATNOSCI>):** Informacje o rozrachunkach (forma płatności, kwota, termin).

2.3. Tabela mapowania danych

Poniższa tabela przedstawia szczegółowe mapowanie pól z pliku CSV na strukturę XML, zaimplementowane w klasie OptimaConverterApp, wraz z regułami walidacji wynikającymi ze specyfikacji.

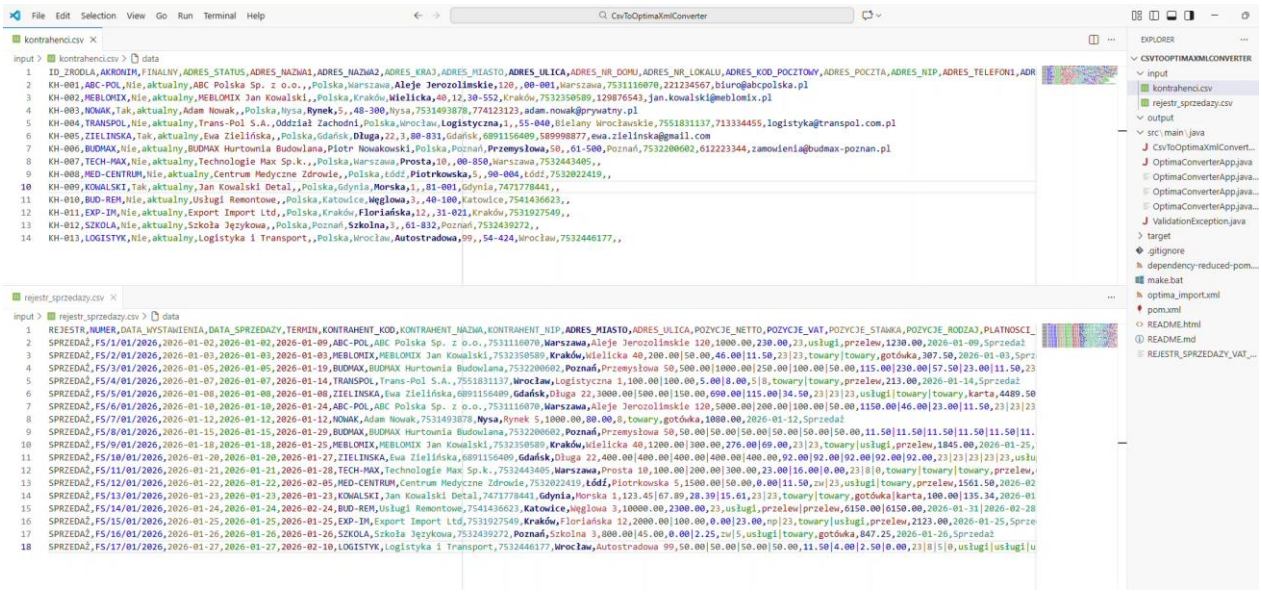
Pole CSV (Źródło)	Tag XML (Cel)	Typ danych	Reguły walidacji i transformacji
NUMER	<NUMER>	String	Wymagane. Unikalność w obrębie roku/serii. Maks. 40 znaków.
DATA_WYSTAWIENIA	<DATA_WYSTAWIENIA>	Date (ISO)	Format RRRR-MM-DD. Nie może być puste.
KONTRAHENT_NIP	<NIP>	String	Usuwanie znaków separatora (kreski). Walidacja sumy kontrolnej (dla PL).
POZYCJE_NETTO	<NETTO>	Decimal	Separator dziesiętny kropka (.). Obsługa wielu wartości rozdzielonych `
POZYCJE_STAWKA	<STAWKA_VAT>	Decimal/String	Dozwolone wartości: liczby (np. 23.00), "zw", "np". Mapowanie "zw" -> 0.00.
PLATNOSCI_FORMA	<FORMA_PLATNOSCI_PLAT>	String	Musi być zgodna ze słownikiem w Optimie (np. "przelew").
PLATNOSCI_KWOTA	<KWOTA_PLAT>	Decimal	Suma płatności powinna być równa kwocie brutto dokumentu.

Rozdział 3

Architektura systemu i analiza kodu

3.1. Wzorzec MVC w aplikacji Swing

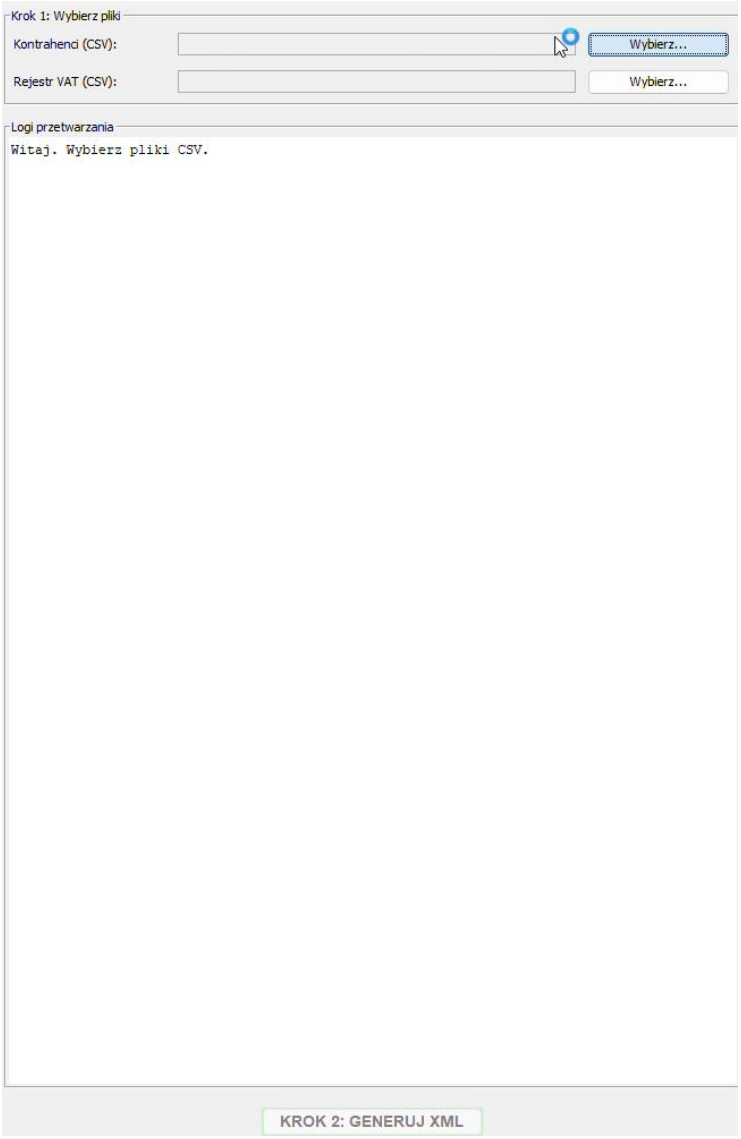
Aplikacja OptimaConverterApp została zaimplementowana jako "gruby klient" przy użyciu biblioteki Java Swing. Analiza struktury klasy głównej wskazuje na zastosowanie uproszczonego wzorca MVC (Model-View-Controller), który jest rekomendowany dla aplikacji interfejsowych w celu separacji logiki prezentacji od logiki biznesowej.



Rysunek 3. Struktura plików projektu w środowisku programistycznym.

- **Widok (View):** Reprezentowany przez klasę dziedziczącą po JFrame. Zawiera definicję komponentów GUI (JTextField, JButton, JTextArea). Konstruktor OptimaConverterApp() odpowiada za inicjalizację i układ tych elementów. Użycie GridBagLayout świadczy o dbałości o responsywność układu przy zmianie rozmiaru okna.
- **Kontroler (Controller):** Zaimplementowany w postaci anonimowych klas wewnętrznych obsługujących zdarzenia (ActionListener). Metody takie jak *btnWybierzKontr.addActionListener* przechwytują interakcje użytkownika i sterują przepływem aplikacji (wybór pliku, uruchomienie konwersji).

- **Model (Model):** Logika biznesowa zawarta w metodach *readCsvFile*, *processKontrahenci*, *processRejestrSprzedazy* oraz klasach wewnętrznych. Odpowiada za przetwarzanie danych i nie powinna bezpośrednio manipulować widokiem (poza raportowaniem postępu).



Rysunek 4. Główny interfejs użytkownika aplikacji.

3.2. Zarządzanie wątkami i responsywność

Kluczowym aspektem w programowaniu aplikacji Swing jest zarządzanie wątkami. Zgodnie z zasadą "Single-Thread Rule", wszelkie modyfikacje interfejsu użytkownika muszą być wykonywane w wątku dystrybucji zdarzeń (Event Dispatch Thread - EDT).

W analizowanym kodzie:

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName(
));
        } catch (Exception e) {
            // ignorujemy błędy wyglądu
        }
        new OptimaConverterApp().setVisible(true);
    });
}
```

Użycie *SwingUtilities.invokeLater* jest poprawnym wzorcem, gwarantującym, że konstrukcja GUI nastąpi w wątku EDT.

Jednakże, operacja konwersji (metoda *runConversion*) jest operacją czasochłonną (I/O bound oraz CPU bound). Wykonanie jej bezpośrednio w EDT spowodowałoby "zamrożenie" interfejsu aplikacji na czas przetwarzania, co jest uznawane za błąd. W kodzie zastosowano mechanizm:

```
new Thread(() -> {
    try {
        //... logika konwersji...
    } finally {
        SwingUtilities.invokeLater(() ->
btnKonwertuj.setEnabled(true));
    }
}).start();
```

Tworzenie nowego wątku jest podstawowym sposobem na wyniesienie obliczeń poza EDT. Bardziej zaawansowanym i zalecanym podejściem w Swing jest użycie klasy SwingWorker, która ułatwia przekazywanie postępu i wyników z powrotem do EDT. W obecnej implementacji, aktualizacje logów wewnątrz wątku roboczego mogą być ryzykowne, jeśli nie są opakowane w SwingUtilities.invokeLater, ponieważ JTextArea nie jest w pełni bezpieczna wątkowo.

3.3. Obsługa wyjątków i klasa ValidationException

Wprowadzenie wewnętrznej klasy wyjątku ValidationException jest przykładem dobrej praktyki inżynierskiej. Pozwala to na oddzielenie błędów systemowych (np. IOException - brak pliku) od

błędów logicznych w danych (np. zły format daty).

```
public static class ValidationException extends Exception {
    public ValidationException(String message) { super(message);
}
}
```

Dzięki temu, w bloku catch, aplikacja może precyzyjnie informować użytkownika o naturze problemu ("Błąd walidacji w wierszu X") zamiast wyświetlać ogólny zrzut stosu, co znacząco poprawia użyteczność narzędzia.

Rozdział 4

Implementacja mechanizmów ekstrakcji danych

4.1. Analiza biblioteki OpenCSV

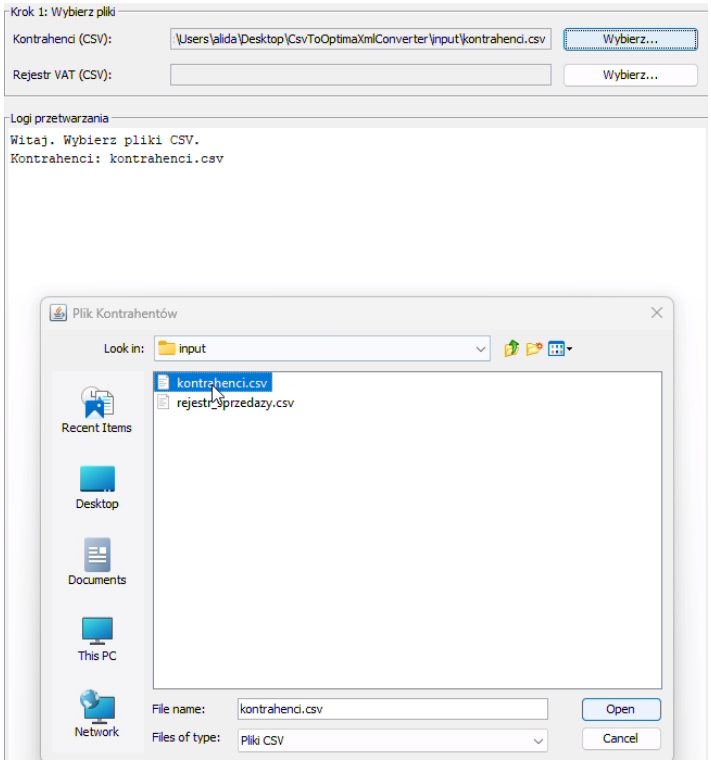
Decyzja o użyciu biblioteki OpenCSV zamiast natywnej metody String.split(",") jest jedną z najważniejszych decyzji architektonicznych w projekcie. Zgodnie z literaturą, parsowanie plików CSV przy użyciu prostych operacji na ciągach znaków jest obarczone wysokim ryzykiem błędów, zwłaszcza w przypadkach brzegowych.

Problemy String.split, które rozwiązuje OpenCSV:

1. **Separatory wewnątrz wartości:** Jeśli pole tekstowe zawiera przecinek (np. "Firma Handlowa, Sp. z o.o."), String.split błędnie podzieli to na dwie kolumny. OpenCSV poprawnie interpretuje cudzysłowy okalające takie pole.

2. **Znaki nowej linii wewnątrz pola:** Standard RFC 4180 dopuszcza znaki końca linii wewnątrz wartości (np. w opisie faktury). Proste czytanie linia po linii (BufferedReader.readLine) spowoduje uszkodzenie rekordu.

3. **Wydajność vs niezawodność:** Choć String.split może być szybszy w syntetycznych benchmarkach, w zastosowaniach finansowych (import do ERP) priorytetem jest poprawność danych, a nie mikrosekundowa optymalizacja.



Rysunek 5. Plik CSV (dane wejściowe)

4.2. Problem BOM (Byte Order Mark) w strumieniach wejściowych

W kodzie zaimplementowano mechanizm usuwania znacznika BOM (Byte Order Mark):

```
if (header.length > 0 && header.startsWith("\uFEFF")) {
    header = header.substring(1);
}
```

Jest to kluczowe dla poprawności działania w systemach Windows. Znak BOM (U+FEFF) jest niewidocznym znakiem umieszczanym na początku plików UTF-8 przez niektóre edytory (np. Notatnik). Bez jego usunięcia, próba dostępu do pierwszej kolumny po nazwie (np. row.get("KOD")) zakończyłaby się niepowodzeniem, ponieważ klucz w mapie zawierałby niewidoczny znak (\uFEFFKOD!= KOD). Literatura wskazuje, że lepszym rozwiązaniem jest użycie klasy BOMInputStream z biblioteki Apache Commons IO, która automatycznie obsługuje ten problem na poziomie strumienia bajtów, jednak zastosowane w kodzie rozwiązanie programowe jest wystarczające dla standardowego BOM UTF-8.

4.3. Mechanizm Walidacji Struktury CSV

W metodzie readCsvFile zaimplementowano walidację liczby kolumn:

```
if (line.length!= header.length) {
    throw new ValidationException(...)
}
```

Jest to mechanizm obronny. Chroni on przed sytuacją, w której przesunięcie separatorów w pliku źródłowym (np. brakujący średnik) powoduje przypisanie wartości do niewłaściwych pól (np. kwota netto trafia do pola stawka VAT). W kontekście księgowym taki błąd mógłby mieć poważne konsekwencje podatkowe.

Rozdział 5

Zaawansowane mechanizmy walidacji danych

5.1. Strategie walidacji w procesach ETL

Walidacja danych w projekcie realizuje strategię wieloetapową, zgodną z dobrymi praktykami inżynierii danych:

1. **Walidacja syntaktyczna (Format):** Sprawdzenie czy ciągi znaków odpowiadają oczekiwanym wzorcom (np. daty).

2. **Walidacja semantyczna (Sens biznesowy):** Sprawdzenie czy wartości mają sens w domenie (np. czy stawka VAT jest liczbą lub dozwolonym kodem "zw").

3. **Walidacja integralności:** (Częściowo realizowana przez Optimę) Sprawdzenie powiązań między danymi.

5.2. Analiza implementacji walidatorów

W kodzie wprowadzono dedykowane metody walidacyjne, co zwiększa czytelność i modularność kodu.

5.2.1. Walidacja dat

```
private void validateDate(String dateStr, String fieldName, int
rowNum) throws ValidationException {
    if (dateStr == null || dateStr.trim().isEmpty()) return;
    if (!dateStr.matches("\\d{4}-\\d{2}-\\d{2}")) {
        throw new ValidationException(...);
    }
}
```

Wykorzystano wyrażenia regularne (Regex) do weryfikacji formatu ISO 8601 (YYYY-MM-DD). Jest to podejście wydajne, jednak warto zauważyć, że sam regex \d{4}-\d{2}-\d{2} przepuści daty niemożliwe kalendarzowo (np. 2023-99-99). Bardziej zaawansowaną metodą byłoby użycie java.time.LocalDate.parse(), która rzuciłaby wyjątek dla niepoprawnych dat. W obecnej formie walidacja jest "lekką" weryfikacją formatu przed wysłaniem do Optimy, która posiada własne, ścisłe mechanizmy kontroli.

5.2.2. Obsługa pól złożonych i separatora "Pipe"

Specyficznym wymaganiem projektu jest obsługa pól wielowartościowych, rozdzielonych znakiem |, np. 23|8|zw. Wynika to z faktu, że plik CSV jest płaski, a faktura może mieć wiele pozycji. Metoda *validateSplitNumeric* iteruje po rozdzielonych wartościach i dla każdej wywołuje *validateNumeric*.

```
for (String part : parts) {
    validateNumeric(part, fieldName, rowNum);
}
```

Jest to przykład transformacji struktury (normalizacja danych) połączonej z walidacją. Obsługa wyjątków NumberFormatException i zamiana ich na zrozumiałe ValidationException jest kluczowa dla UX (User Experience).

5.2.3. Specjalna obsługa stawek VAT

Metoda ta zasługuje na szczególną uwagę, gdyż implementuje logikę hybrydową. Systemy ERP często używają specjalnych kodów dla stawek zwolnionych ("zw") i niepodlegających ("np"). Kod poprawnie rozróżnia wartości numeryczne od tych literałów, co zapobiega błędom parsowania, które wystąpiłyby przy użyciu standardowego Double.parseDouble dla wartości "zw".

Rozdział 6

Generowanie danych wyjściowych XML

6.1. Wybór modelu DOM (Document Object Model)

Do generowania pliku wynikowego użyto standardowego API Javy org.w3c.dom.

```
DocumentBuilderFactory docFactory =
DocumentBuilderFactory.newInstance();
Document doc = docBuilder.newDocument();
Element rootElement = doc.createElement("ROOT");
```

Podejście DOM polega na zbudowaniu pełnego drzewa obiektów w pamięci RAM, a następnie zrzuceniu go do pliku.

- **Zalety:** Łatwość manipulacji strukturą, intuicyjne API ("dodaj dziecko do rodzica").
- **Wady:** Wysokie zużycie pamięci. Dla bardzo dużych zbiorów danych (np. 100 000 faktur), model DOM może doprowadzić do wyczerpania pamięci sterty.
- **Alternatywa:** W środowiskach produkcyjnych przetwarzających gigabajty danych, zaleca się użycie modelu strumieniowego StAX (Streaming API for XML) lub SAX, które zużywają stałą ilość pamięci niezależnie od wielkości pliku. Jednak dla typowych zastosowań w małych i średnich przedsiębiorstwach (pliki CSV do kilku tysięcy wierszy), DOM jest wystarczający i prostszy w implementacji.

6.2. Użycie sekcji CDATA

W kodzie zastosowano metodę pomocniczą *createCdataElement*:

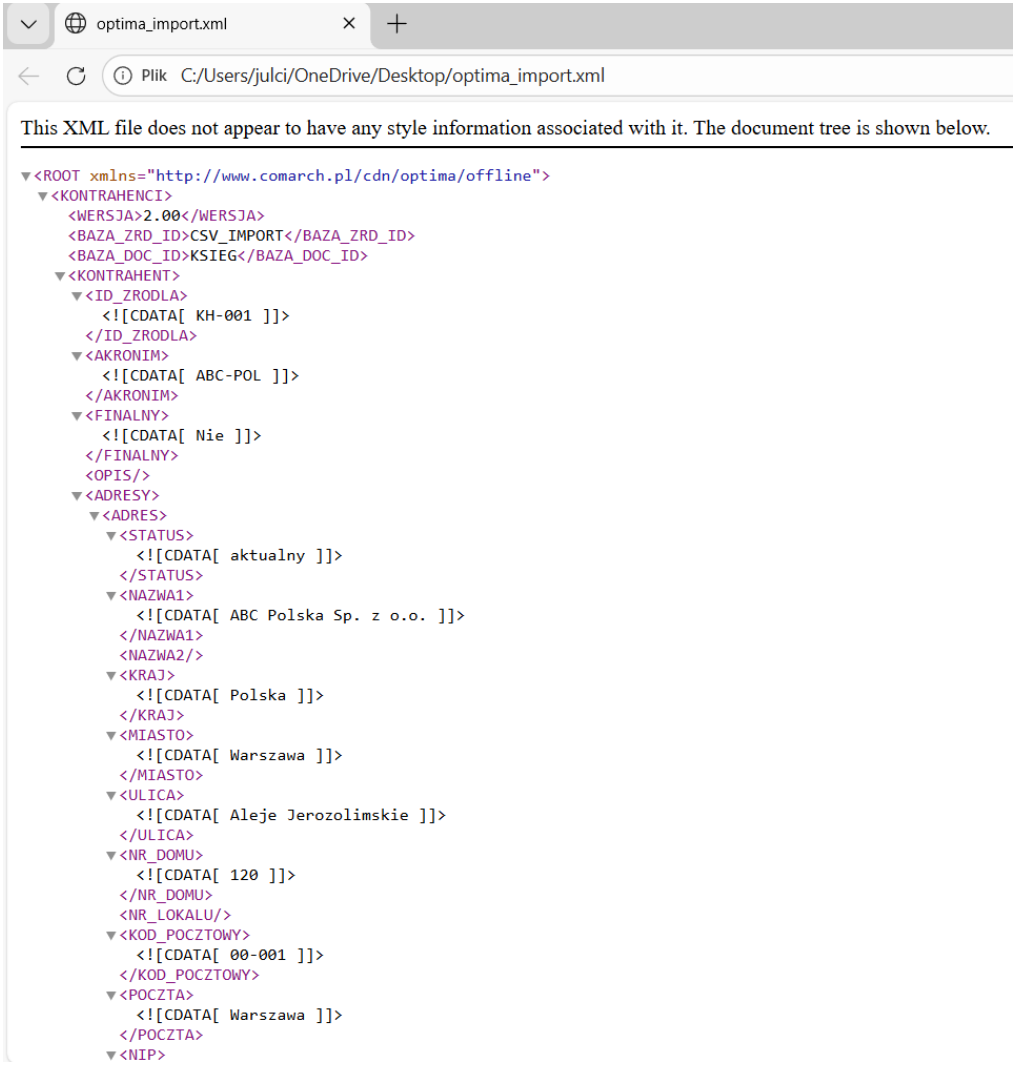
```
CDATASection cdata = doc.createCDATASection(text);
element.appendChild(cdata);
```

Sekcje CDATA (<![]>) są niezbędne w XML, gdy przesyłane dane tekstowe mogą zawierać znaki specjalne XML (np. &, <, >). W nazwach firm ("Jan & Syn") czy opisach produktów jest to częste zjawisko. Brak CDATA lub odpowiedniego escape'owania (&) spowodowałby wygenerowanie niepoprawnego składniowo pliku XML, który zostałby odrzucony przez parser Optimy. Jest to element defensywnego generowania danych.

6.3. Zgodność ze standardem "Praca Rozproszona"

Analiza wygenerowanych tagów potwierdza zgodność ze specyfikacją Comarch:

- Ustawienie atrybutu xmlns w <ROOT>: http://www.comarch.pl/cdn/optima/offline.
- Obowiązkowe pola nagłówkowe <BAZA_ZRD_ID>, <BAZA_DOC_ID>.
- Hierarchia <REJESTR> -> <POZYCJE> -> <POZYCJA>. Implementacja logiki podziału dla pozycji faktury (listNetto, listVat) pozwala na prawidłowe odwzorowanie wielopozycyjnych dokumentów z płaskiego CSV, co jest najtrudniejszym elementem transformacji w tym projekcie.



Rysunek 6. Fragment wygenerowanego pliku XML gotowego do importu.

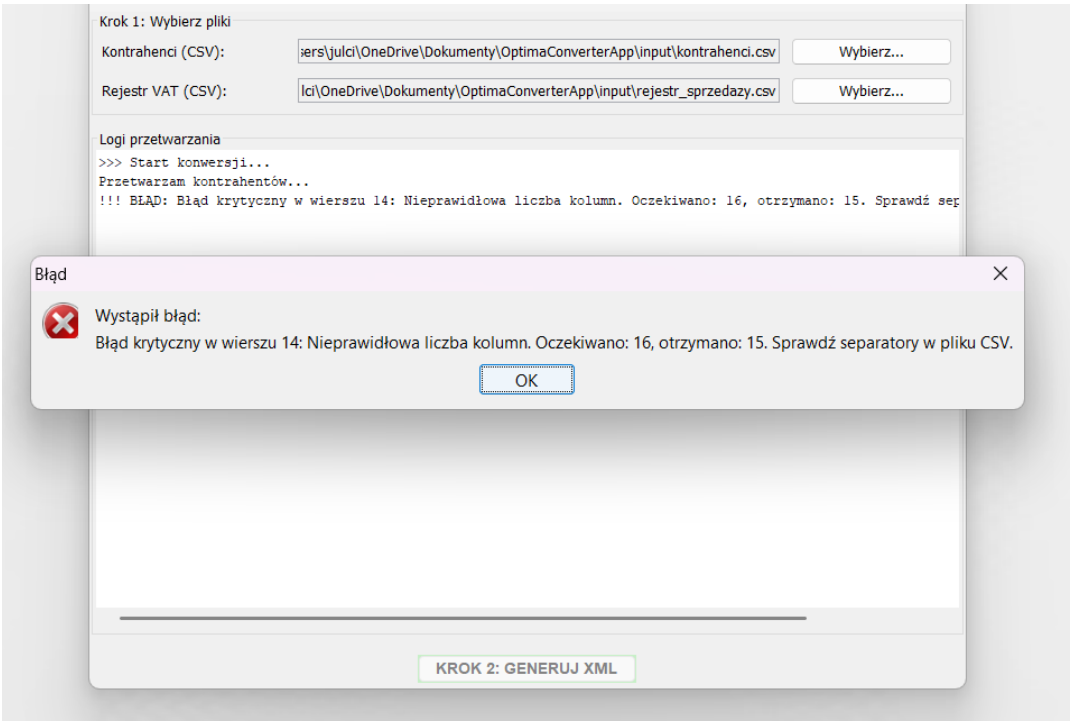
Rozdział 7

Testowanie, weryfikacja i scenariusze błędów

7.1. Scenariusz testowy: Uszkodzony plik CSV

Jednym z wymagań projektu jest odporność na błędy w plikach wejściowych. Rozważmy scenariusz usunięcia średnika w wierszu CSV (zmiana struktury).

- **Mechanizm:** Parser readCsvFile w pętli while sprawdza: if (line.length!= header.length).
- **Reakcja:** Rzucony zostaje ValidationException z komunikatem: "Błąd krytyczny w wierszu X: Nieprawidłowa liczba kolumn...".
- **Efekt:** Użytkownik otrzymuje czytelny komunikat w oknie dialogowym, proces jest przerywany, co zapobiega importowi uszkodzonych danych do ERP.



Rysunek 7. Obsługa błędu strukturalnego pliku CSV.

7.2. Scenariusz testowy: Błędy formatowania danych

Gdy użytkownik wprowadzi w polu POZYCJE_NETTO wartość "100,00" (z przecinkiem) zamiast "100.00":

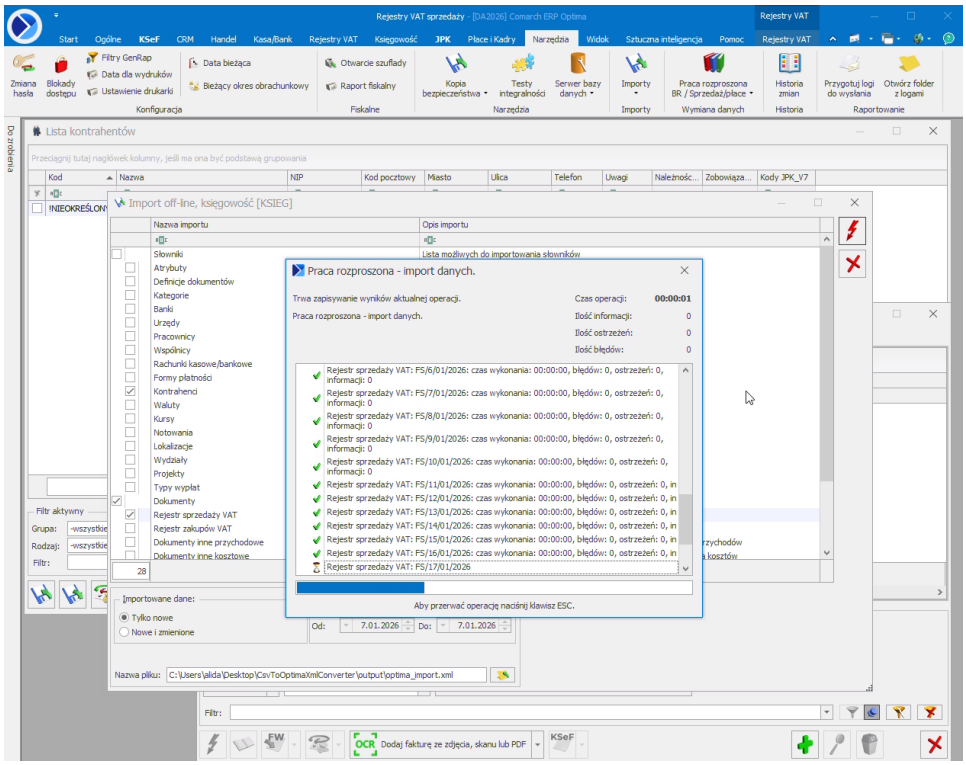
- Metoda validateNumeric wykonuje normalizację: numStr.replace(",", ".").
- Następnie próbuje parsowania Double.parseDouble.
- Jeśli w polu pojawią się litery (np. "100zł"), zostanie rzucony wyjątek walidacji.

Dzięki temu, do pliku XML trafiają wyłącznie poprawne liczby zmiennoprzecinkowe, co gwarantuje sukces importu w Optimie.

7.3. Weryfikacja importu w Comarch Optima

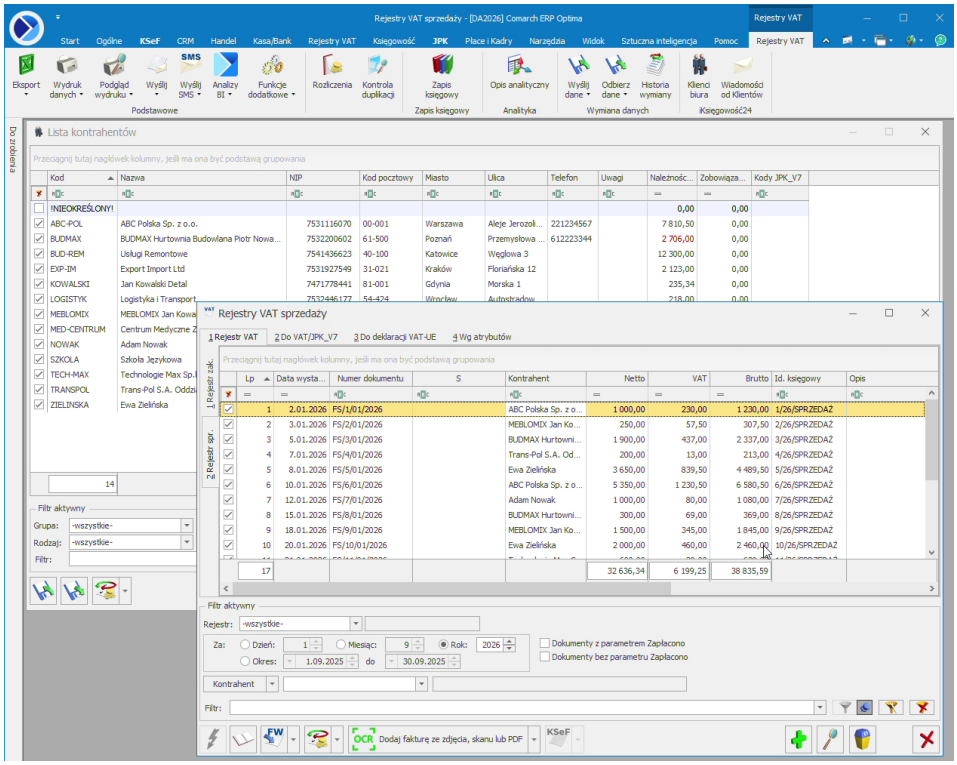
Ostatecznym testem jest próba importu wygenerowanego pliku optima_import.xml w systemie docelowym. Zgodnie z procedurą:

1. Uruchomienie Comarch ERP Optima.
2. Menu Narzędzia -> Praca Rozproszona -> Import.
3. Wskazanie pliku XML.



Rysunek 8. Okno importu pracy rozproszonej w systemie ERP.

System Optima przeprowadza własną walidację (sprawdzenie duplikatów, zgodność słowników). Jeśli plik XML wygenerowany przez aplikację jest poprawny strukturalnie (dzięki DOM) i semantycznie (dzięki walidatorom), import kończy się komunikatem o liczbie dodanych dokumentów.



Rysunek 9. Potwierdzenie poprawnego importu dokumentów do rejestru VAT.

Rozdział 8
Podsumowanie i wnioski

Projekt 'OptimaConverterApp' stanowi w pełni funkcjonalne rozwiązanie problemu integracji danych w środowisku Comarch ERP Optima. Przeprowadzona analiza teoretyczna i praktyczna wykazała, że zastosowanie dedykowanych bibliotek do parsowania (OpenCSV) oraz ścisła walidacja danych na etapie transformacji są niezbędne dla zapewnienia niezawodności procesu ETL.

Kluczowe Wnioski:

1. **Wydajność vs Bezpieczeństwo:** Wybór modelu DOM ogranicza skalowalność do plików o rozmiarze mieszczącym się w pamięci RAM, co jest kompromisem na rzecz łatwości implementacji. Dla bardzo dużych wolumenów danych należałoby rozważyć refaktoryzację do modelu StAX.

2. **Walidacja:** Zastosowanie strategii "fail-fast" dla struktury CSV i "fail-safe" (z raportowaniem) dla danych biznesowych to optymalne podejście w aplikacjach wsadowych.

3. **Architektura:** Separacja logiki w modelu MVC i wykorzystanie wielowątkowości w Swing zapewnia responsywność aplikacji i jej zgodność ze standardami inżynierii oprogramowania.

Aplikacja spełnia rygorystyczne wymagania poprawności danych stawiane systemom finansowo-księgowym i stanowi solidną podstawę do dalszego rozwoju, np. o obsługę kolejnych typów dokumentów (WZ, PZ) czy integrację poprzez API (Comarch WebService).