



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria oprogramowania, procesów biznesowych i baz danych

Zofia Katarzyna Zakrzewska

Nr albumu s24365

**Nowoczesne interfejsy WebAPI w tradycyjnych i współczesnych językach
programowania: analiza COBOL vs Java**

Praca magisterska

Promotor

dr inż. Mariusz Trzaska, prof. PJATK

Warszawa, lipiec 2026

Streszczenie

Celem niniejszej pracy magisterskiej jest dokonanie kompleksowego porównania dwóch języków programowania: COBOL i Java.

Pomimo podejmowania inicjatyw wyparcia zarówno Mainframe jak i języka COBOL przez firmy korzystające z tych technologii, znaczna część transakcji krytycznych dla funkcjonowania społeczeństwa w dalszym ciągu odbywa się na platformie IBM.

Z uwagi na swój wiek, programy pisane w języku COBOL są w większości ogromne i monolityczne; trudne zarówno w utrzymaniu, jak i w zrozumieniu. Język ten pozwala jednak na zaimplementowanie bardziej nowoczesnej struktury mikroservisowej.

Na potrzeby pracy zaimplementowany został prosty system umożliwiający zapisywanie zamówień w obu wyżej wymienionych językach. Do implementacji w Javie wybrany został framework Spring Boot. Ponieważ nie istnieją publicznie dostępne frameworki do języka COBOL, część napisana w tym języku używa jedynie preprocesor Ocesql, umożliwiający komunikację z nowoczesną bazą danych PostgreSQL.

Obie implementacje wykorzystują architekturę mikroservisową, do zastosowania docelowo w interfejsach WebAPI.

Słowa kluczowe

COBOL, Java, Spring Boot, Mikroservisy, Interfejsy WebAPI

SPIS TREŚCI

1. WSTĘP	5
1.1. Cel pracy	5
1.2. Uzasadnienie wyboru tematu.....	5
1.3. Kontekst i założenia pracy.....	6
2. OPIS ZASTOSOWANYCH TECHNOLOGII I NARZĘDZI	7
2.1. COBOL	7
2.1.1. <i>Historia Mainframe</i>	7
2.1.2. <i>Komunikacja z Mainframe w praktyce</i>	9
2.1.3. <i>Język programowania COBOL</i>	10
2.1.4. <i>Zastosowane narzędzia</i>	15
2.2. Java.....	16
2.2.1. <i>Historia</i>	16
2.2.2. <i>Założenia, składnia i struktura</i>	16
2.2.3. <i>Zastosowane frameworki, biblioteki, narzędzia</i>	17
2.3. PostgreSQL	19
2.3.1. <i>Uzasadnienie wyboru silnika bazy danych</i>	19
2.4. Postman	19
3. PODJĘTY PROBLEM BIZNESOWY	20
3.1. Podstawowe założenia systemu.....	20
3.2. Wymagania biznesowe	20
3.3. Projekt bazy danych	21
4. ARCHITEKTURA ZAPROJEKTOWANYCH SYSTEMÓW	22
4.1. COBOL	22
4.1.1. <i>Wysokopoziomowy opis rozwiązania</i>	22
4.1.2. <i>Konwencja nazewnictwa</i>	22
4.1.3. <i>Uzasadnienie wyboru architektury projektu</i>	23
4.1.4. <i>Komunikacja poprzez WebAPI</i>	23
4.2. Java.....	25
4.2.1. <i>Wysokopoziomowy opis rozwiązania</i>	25
4.2.2. <i>REST API</i>	25
5. IMPLEMENTACJA SYSTEMÓW	26
5.1. COBOL	26
5.1.1. <i>Lista programów</i>	26
5.1.2. <i>Diagram wywołań funkcjonalności biznesowych</i>	26
5.1.3. <i>Interfejs - ORD000</i>	30
5.1.4. <i>Logika biznesowa - ORD001 - ORD006</i>	39
5.1.5. <i>Wyszukiwanie danych w bazie - ORD010 – ORD070</i>	50
5.1.6. <i>Przetwarzanie pliku CSV - ORD100</i>	57
5.1.7. <i>Modyfikacja danych w bazie - ORD110, ORD150</i>	65
5.1.8. <i>Kompilacja projektu</i>	68
5.2. Java.....	71
5.2.1. <i>Lista endpointów</i>	71
5.2.2. <i>Diagram wywołań funkcjonalności biznesowych</i>	71
5.2.3. <i>Kontroler</i>	75
5.2.4. <i>Serwis</i>	77
5.2.5. <i>Repozytorium</i>	78
5.2.6. <i>Model</i>	79
5.2.7. <i>Klasy pomocnicze (utility)</i>	81
5.2.8. <i>Mapery</i>	81

5.2.9.	<i>Kompilacja projektu</i>	82
5.3.	Warstwa WebAPI dla systemu COBOL-owego w języku Java	82
5.3.1.	<i>Dostępne endpointy i powiązanie z warstwą COBOL</i>	82
5.3.2.	<i>Metoda komunikacji między programami COBOL a warstwą Java</i>	82
5.3.3.	<i>Przetwarzanie struktur Copybook z użyciem Java Reflection API</i>	85
5.4.	Porównanie implementacji	89
5.4.1.	<i>Łatwość implementacji</i>	89
5.4.2.	<i>Obszerność implementacji</i>	90
5.4.3.	<i>Rozwój aplikacji</i>	91
6.	TESTY ZAIMPLEMENTOWANYCH ROZWIĄZAŃ	92
6.1.	Testy wydajnościowe – czas wykonania	92
6.2.	Testy wydajnościowe – zużycie procesora	94
6.3.	Testy złożoności pamięciowej	95
6.4.	Kontekst przeprowadzonych testów	96
7.	PODSUMOWANIE	97
7.1.	Porównanie zaimplementowanych rozwiązań	97
7.2.	Zastosowanie projektów w praktyce	97
7.2.1.	<i>COBOL</i>	97
7.2.2.	<i>Java</i>	98
7.3.	Możliwości rozwoju projektu	98
7.3.1.	<i>COBOL</i>	98
7.3.2.	<i>Java</i>	98
8.	BIBLIOGRAFIA	99
9.	DODATKI	102
9.1.	Spis rysunków	102
9.2.	Spis tabel	102
9.3.	Spis listingów	103

1. Wstęp

Poniższe rozdziały omawiają cel pracy, uzasadniają wybór tematu oraz przybliżają jego kontekst. Omówione są najważniejsze założenia oraz przedstawiona jest sytuacja Mainframe na rynku.

1.1. Cel pracy

Celem niniejszej pracy jest dokonanie analizy dwóch języków programowania – COBOL i Java pod kątem nowoczesnej architektury mikroservisowej, porównanie wad i zalet obydwu języków szczególnie w kontekście istniejących już biznesów.

Efektom pracy są wnioski oparte na zaimplementowanym systemie zarządzania zamówieniami w obu technologiach, działającego na tych samych zasadach biznesowych – oceniona została praktyczność zastosowania każdej z nich, łatwość tworzenia nowego systemu jak i utrzymywania już istniejącego. Omówione i zarekomendowane zostały odpowiednie zastosowania dla obu języków.

1.2. Uzasadnienie wyboru tematu

Mainframe obecnie odnosi się do komputerów produkowanych przez IBM. Są to serwery z ogromną ilością mocy obliczeniowej, pamięci podręcznej oraz przestrzeni dyskowej, zaprojektowane z myślą o wysokiej dostępności, niezawodności oraz ciągłości działania. W dalszym ciągu są z powodzeniem wykorzystywane do wykonywania krytycznych dla społeczeństwa zadań.

COBOL, który jest głównym językiem wykorzystywanym na środowiskach Mainframe, obarczony jest negatywną opinią - szczególnie wśród młodych programistów. Uznawany jest on za bardzo ograniczony, przestarzały język, który znają tylko najstarsi informatycy. Zdanie to ma w sobie wiele prawdy; w 2019 roku, średnim wiekiem programisty COBOL było 58 lat [1].

Pomimo dużej stygmatyzacji tej technologii, znacząca ilość biznesów prowadzących działalność krytyczną dla funkcjonowania społeczeństwa, czyli branże takie jak bankowość, ubezpieczenia, opieka zdrowia czy lotnictwo, stosuje ją w programach niezbędnych do poprawnego działania. Wiele firm podejmuje próby zastąpienia Mainframe bardziej nowoczesnymi technologiami, jednak często okazuje się to niemalże niemożliwe. Przykładem takiej nieudanej transformacji technologicznej jest Zakład Ubezpieczeń Społecznych, który w 2018 roku zapowiedział ograniczenie udziału Mainframe w codziennych operacjach [2], a następnie w 2019 roku, dokupił więcej serwerów [3].

Rozwiązanie problemu mnogości i krytyczności programów napisanych w języków COBOL, które wkrótce będą jeszcze trudniejsze do utrzymania, może stać się jednym z ważniejszych tematów informatyki w nadchodzących latach.

W celu zmniejszenia złożoności programów i tym samym uczynienia ich łatwiejszymi w obsłudze, w ostatnich latach podjęta została inicjatywa modernizacji Mainframe [4]. Temat pracy został wybrany w celu zaprezentowania, w jaki sposób może wyglądać nowoczesny projekt w języku COBOL na praktycznym przykładzie. Architektura zbliżona do zaprezentowanej w niniejszej pracy może stanowić dobrą bazę do modernizacji programów monolitycznych obecnych w wielu firmach, szczególnie w przypadkach, gdzie całkowita zmiana języka programowania nie jest możliwa.

1.3. Kontekst i założenia pracy

Język COBOL nieprzypadkowo jest mało popularnym wyborem. Poza swoją składnią przypominającą języki niższego poziomu, wynikającą głównie z jego prostoty, nauka i zastosowanie go w praktyce nie są prostym zadaniem. IBM nie udostępnia publicznie oficjalnego kompilatora, a zdobycie licencji jest możliwe tylko w przypadku zakupu komputera Mainframe. Ponadto, w przeciwieństwie do bardziej nowoczesnych języków, publicznie dostępna jest bardzo niewielka ilość kodu źródłowego, większość pozostaje objęta prawami autorskimi w dużych instytucjach finansowych, medycznych czy ubezpieczeniowych.

Wszystkie firmy posiadające Mainframe mają na nich moduły, funkcjonujące w praktyce w podobny sposób do bibliotek. Napisane w języku COBOL lub *HLASM (IBM High Level Assembler)* [5], często zawierają one krytyczną logikę biznesową, ale mogą służyć także między innymi do ułatwienia komunikacji między komputerami Mainframe, a światem zewnętrznym, czy do logowania informacji z przebiegu programu. Również przez te moduły całkowita migracja na bardziej nowoczesną technologię jest nierealistyczna – krytyczne dla działania firmy fragmenty kodu niejednokrotnie pisane były bardzo dawno, a ich autorzy zazwyczaj już nie pracują zawodowo.

Przepisanie całej bazy kodu źródłowego w danej firmie jest wykonalne – jednym z przykładów sukcesu w transformacji technologicznej jest mBank [6]. W 2025 roku firma ta zakończyła proces modernizacji, tym samym eliminując Mainframe z dziennych operacji. Przykład ten jest jednak dość specyficzny – mBank jest dosyć młodym bankiem, powstałym w roku 2000 [7]. W porównaniu do konkurencyjnych banków - na przykład Bank PKO założony w 1919 roku [8] - rezultatem krótszego czasu działania prawdopodobnie była mniejsza baza kodu *legacy* do migracji.

Im więcej takiego kodu, tym bardziej nieopłacalne staje się przepisanie systemów. Moduły biznesowe są długie, a logika zawarta w nich jest skomplikowana. Przełożenie ich na nowszą technologię nie jest łatwym zagadnieniem, ponadto przetestowanie ich działania i zweryfikowanie, że wszystko zachowuje się poprawnie, jest równie trudnym, jeśli nie trudniejszym zadaniem. Odpowiedzialność wynikająca z decyzji, czy wszystko działa dobrze, jest bardzo duża.

68% transakcji informatycznych na świecie odbywa się na Mainframe [9]. Mimo tak ważnej pozycji na rynku i mnogości linii kodu napisanych w języku COBOL, przez brak łatwej możliwości jego kompilacji, projekty open source nie mają praktycznego zastosowania. Dodatkowo publicznie nie są dostępne żadne biblioteki, co wymusza na programiście pisanie całego kodu.

W związku z powyżej wspomnianymi problemami z dostępnością oficjalnego kompilatora do języka COBOL, na potrzeby pracy wykorzystany został projekt GnuCOBOL [10]. Jest to darmowy transpiler, dostępny na licencji GNU General Public License [11], stworzony w 2002 roku i w dalszym ciągu rozwijany.

Został utworzony głównie z uwagi na fakt, że pomimo ogromnej ilości krytycznej infrastruktury w świecie informatyki, kod źródłowy w tym języku jest objęty w przytłaczającej większości tajemnicą organizacji, co zdecydowanie zmniejsza jego przystępność [12]. Duża część świata informatycznego nie ma więc pojęcia, jak wygląda program napisany w tym języku. Otwarta natura GnuCOBOL ma na celu jego demistyfikację, pozwalając na wykorzystanie języka w praktyce.

Wykorzystywany transpiler tłumaczy język COBOL na C, a następnie kompiluje program używając najczęściej *GCC* [13]. Zapewnia to wysoki poziom przenoszalności kodu – wspiera on wiele platform, takich jak GNU/Linux, MacOS, Microsoft Windows, a także bliższe realnemu zastosowaniu Mainframe z/OS [14].

2. Opis zastosowanych technologii i narzędzi

Poniższe podrozdziały skupiają się na omówieniu języków COBOL i Java, włącznie z krótkim opisem ich historii w celu dokładnego uzasadnienia ich potencjalnych zastosowań w systemach informatycznych. Opisanie są również ich najważniejsze zasady i założenia.

2.1. COBOL

Następujące podpunkty skupiają się na historii zarówno samego języka COBOL jak i komputerów Mainframe, które stanowią główne środowisko jego wykorzystywania. Opisują jego podstawowe założenia oraz składnię, przedstawiając równocześnie kontekst tych zasad oraz ograniczeń.

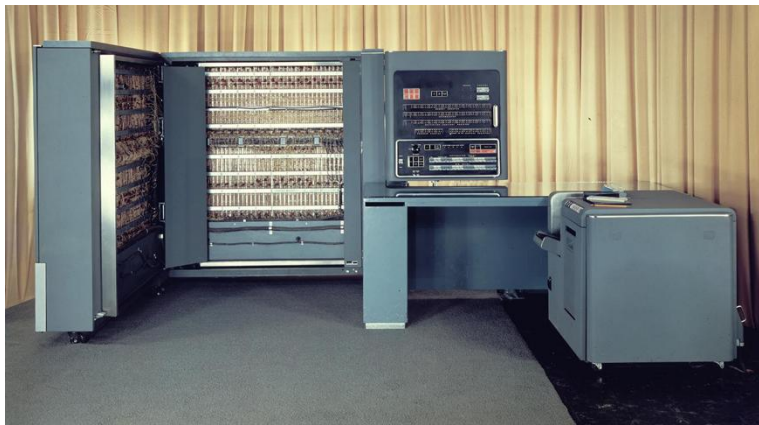
2.1.1. Historia Mainframe

Programy pisane w COBOL mają swoje miejsce przede wszystkim na komputerach Mainframe. Znaczenie tej nazwy jest bardzo dosłowne – odnosi się ono do głównego serwera - *main*, który zamontowany był na metalowej ramie – *frame*. Obecnie słowo *Mainframe* odnosi się przede wszystkim do komputerów produkowanych przez IBM. Zasadniczo są to ogromne serwery, które mają potężną moc obliczeniową i bardzo dużą ilość pamięci. Ich dużym atutem jest możliwość przetwarzania wielu prostych transakcji jednocześnie, co czyni je dobrym wyborem dla instytucji takich jak banki – przykładowo 95% operacji wykonywanych w bankomatach pod spodem odbywa się na właśnie takim komputerze [15]. Historycznie jednak termin ten nie zamykał się jedynie na sprzęcie IBM.

Za pierwszy taki komputer uznaje się *Mark I* [16]. Był on używany podczas drugiej wojny światowej do wykonywania skomplikowanych obliczeń, do której wcześniej niezbędna była duża ilość „komputerów” – wtedy słowo to odnosiło się do ludzi, którzy zajmowali się właśnie wykonywaniem obliczeń.

Pierwszym komercyjnym komputerem Mainframe był *UNIVAC I* [17]. Ukończony w 1951 roku, oparty był na komputerze *ENIAC*, który uznawany jest za pierwszy na świecie programowalny, uniwersalny komputer [18]. *UNIVAC* został z powodzeniem zastosowany przez amerykański Urząd Spisu Ludności – w 1954 roku cały spis gospodarczy powstał z użyciem jego możliwości. W sumie na świecie sprzedano 46 takich komputerów [19].

W 1953 roku, IBM zadebiutował z własną pierwszą odsłoną Mainframe – był nim *Model 701*, widoczny na rysunku 1. Nazywany także „kalkulatorem obronnym” był odpowiedzią IBM na rosnące zapotrzebowanie na moc obliczeniową rządu Stanów Zjednoczonych [20]. Wyprodukowane zostało zaledwie 19 sztuk, a wynajęcie Modelu 701 kosztowało aż 16 tysięcy dolarów miesięcznie [21].



Rysunek 1. IBM Model 701. Źródło: [22]

Za pierwszy nowoczesny Mainframe przyjmuje się wydany w 1964 roku IBM System/360. Komputer ten miał ogromny wpływ na to, jak wygląda współczesny świat informatyki. Był on pierwszym modelem na świecie umożliwiającym oddzielenie oprogramowania od konkretnego procesora, wprowadzając kompatybilność międzyplatformową. Przed wprowadzeniem tego modelu, każdy program mógł działać tylko na jednym, konkretnym procesorze – wymiana komputera oznaczała więc przepisywanie całego kodu. Model S/360 był rewolucyjny; wywołał ogromną zmianę w podejściu całego świata do komputerów – to pierwszy model umożliwiający swoje rozbudowywanie - wcześniej do zwiększenia mocy obliczeniowej firmy musiały albo dokupić kolejną, osobną jednostkę, albo całkiem wymienić sprzęt, co prowadziło do wcześniej wspomnianej konieczności przepisania całego oprogramowania [23].

Sprzedano ponad 33 tysiące sztuk Modelu System/360, a wliczając następne jego iteracje, IBM zarobił ponad 100 miliardów dolarów na rodzinie S/360. W latach 90 komputery tej firmy posiadały 65-procentowy udział w rynku [24].

Na rysunku 2 widoczny jest wspomniany wyżej model – na środku komputer, natomiast po lewej stronie terminal służący do jego obsługi.



Rysunek 2. Model System/360. Źródło: [25]

Po rewolucyjnym modelu System/360 IBM wprowadzał regularnie nowe modele, które przynosiły coraz większe udogodnienia. Mimo powszechnej opinii Mainframe jako systemu *legacy*, regularnie wydawane są kolejne iteracje. Najnowszą z nich jest komputer z/16 widoczny na rysunku 3. Jest dostępny w wielu konfiguracjach, zawiera on nawet akcelerator AI, umożliwiający bardziej efektywne wykonywanie operacji związanych ze sztuczną inteligencją [26].



Rysunek 3. IBM z/16. Źródło: [27]

IBM Model 700 mógł wykonać około 16,000 operacji na sekundę. System/360 w najmniejszej konfiguracji to już 33,00 operacji na sekundę; największej – 750,000 operacji na sekundę [28]. Model z/16 może wykonać do nawet 215,000,000,000 (215 miliardów) operacji na sekundę [29].

2.1.2. Komunikacja z Mainframe w praktyce

Praca na Mainframe wygląda zdecydowanie inaczej niż praca z nowoczesnymi technologiami. Aby wykonać jakąkolwiek operację, każdy użytkownik łączy się po sieci do jednego komputera Mainframe – w praktyce oznacza to, że wszyscy programiści COBOL w danej firmie korzystają w tym samym czasie z tych samych fizycznych zasobów, a nierzadko dzielone są one także ze środowiskiem produkcyjnym. Momenty większego obciążenia są odczuwalne w codziennej pracy.

W interakcję z Mainframe wchodzi się za pomocą tak zwanych terminali. Kiedyś były to monitory podłączone bezpośrednio do maszyny poprzez kable – obecnie komunikacja odbywa się za pomocą emulatorów dostępnych na nowoczesne systemy operacyjne (Windows, Linux), np. IBM Personal Communications. Połączenie odbywa się poprzez TCP/IP, protokół TN3270.

Po zalogowaniu się do Mainframe poprzez TSO – oprogramowanie służące do tworzenia sesji użytkownika oraz udostępniające podstawowe funkcje systemu z/OS [30] - w większości przypadków pojawi się ISPF [31]. Jest to środowisko służące do nawigacji oraz wielu zadań, które użytkownik może podjąć, takie jak przeglądanie zbiorów danych, tworzenie i edycja zbiorów danych (*dataset*), kompilacja programów czy zlecanie zadań do systemu *JES* i monitorowanie ich statusu w *SDSF*. Często ISPF to jedyne, z czym użytkownik będzie wchodził w interakcję.

Rysunek 4 przedstawia typowy wygląd ISPF. Na górze panelu dostępna jest podstawowa lista opcji, poniżej widoczna jest linia poleceń. Środek ekranu zawiera dostępne dla użytkownika programy.

```

Menu Utilities Compilers Options Status Help
-----
ISPF Primary Option Menu

Option ==>
More: +

0 Settings      Terminal and user parameters    < Calendar >
1 View          Display source data or listings    July 2020
2 Edit          Create or change source data      Su Mo Tu We Th Fr Sa
3 Utilities     Perform utility functions        1 2 3 4
4 Foreground    Interactive language processing   5 6 7 8 9 10 11
5 Batch         Submit job for language processing 12 13 14 15 16 17 18
6 Command       Enter TSO or Workstation commands 19 20 21 22 23 24 25
7 Dialog Test   Perform dialog testing           26 27 28 29 30 31
8 LM Facility   Library administrator functions
9 IBM Products  IBM program development products  Time . . . . : 03:50
10 SCLM         SW Configuration Library Manager  Day of year. : 193
11 Workplace    ISPF Object/Action Workplace

----- Other Functions -----

SD SDSF        View output
U Unix         Unix Shell Prompt
F1=Help        F2=Split      F3=Exit      F7=Backward F8=Forward F9=Swap
F10=Actions    F12=Cancel

```

Rysunek 4. ISPF Primary Option Menu. Źródło: [32]

ISPF obsługuje się jedynie za pomocą klawiatury – całe środowisko emulatora jest bardzo zbliżone do pierwszych terminali, które nie posiadały myszek.

2.1.3. Język programowania COBOL

Poniższe podrozdziały opisują historię samego języka programowania, jak również przedstawiają podstawowe założenia i strukturę programów w nim pisanych.

Historia

COBOL, czyli *Common Business-Oriented Language* [33], to wysokopoziomowy język programowania, zaprojektowany pierwotnie z myślą o byciu prostym do zrozumienia, i zgodnie z nazwą, nastawionym na zastosowania biznesowe.

Opublikowany w 1960 roku, był wynikiem spotkań programistów, pracowników uniwersyteckich oraz reprezentantów rządu Stanów Zjednoczonych, których celem było stworzenie języka zarówno prostego w użyciu, jak i kompatybilnego z wieloma procesorami i systemami operacyjnymi. Efektem spotkań były pierwsze wersje języka COBOL, silnie zainspirowane językami FLOW-MATIC oraz IBM COMTRAN [34].

Z uwagi na swoją prostotę w użyciu, nakierowanie na biznes i możliwość kompilacji programów na więcej niż tylko jednym procesorze, projekt okazał się ogromnym sukcesem.

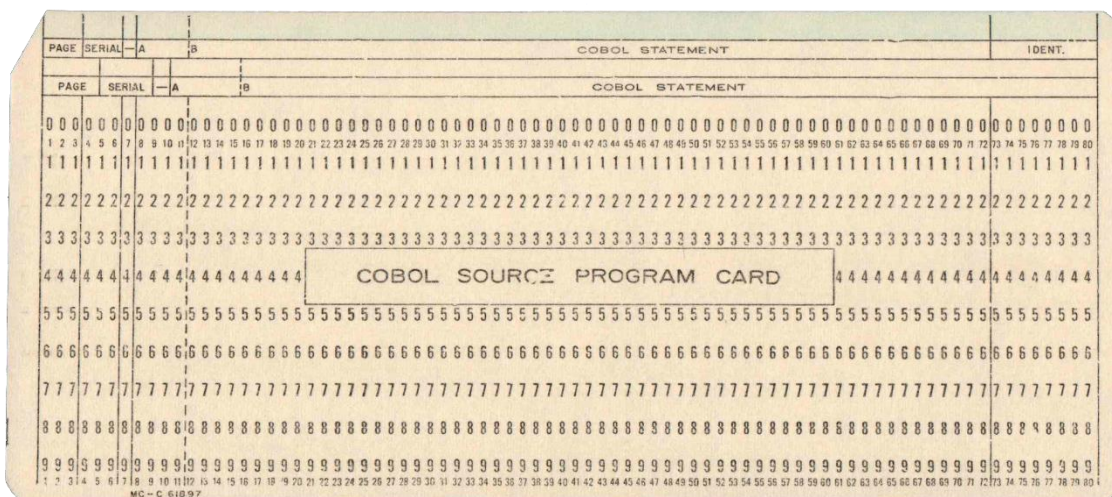
Założenia, składnia i struktura

COBOL fundamentalnie jest językiem proceduralnym, mimo późniejszego dodania do niego elementów programowania obiektowego. Przez swoją składnię zbliżoną do języka angielskiego, czytanie programów napisanych w tym języku jest dość proste.

Wymaga bardzo konkretnej, hierarchicznej struktury i formatu każdego programu. Istnieje w nim wiele bardzo konkretnych ograniczeń, jeśli chodzi o długość zmiennych, nazw sekcji czy znaków w linii kodu.

Zmienne oraz nazwy sekcji mogą mieć maksymalnie 30 znaków, zawierać mogą znaki od A do Z, liczby od 0 do 9 i myślnik („-”). Linia kodu może zawierać nie więcej niż 80 znaków.

Ograniczenia te wynikają z faktu, że programy pisane w COBOLu były pierwotnie wybijane na kartach perforowanych (Rysunek 5), gdzie dostępna była ograniczona ilość kolumn.



Rysunek 5. Karta perforowana COBOL. Źródło: [35]

Kolumny 1-6 zarezerwowane są dla numeru linii, kolumna 7 jest kolumną znacznikową, kolumny 8-11 zarezerwowane są dla obszaru A, w którym znajdują się deklaracje sekcji programu, kolumny 12-72 to obszar B, który zawiera logikę programu, natomiast kolumny 73-80 to obszar zarezerwowany dla komentarzy [36].

Struktura programu w języku COBOL jest także ściśle określona. Każdy program dzieli się na konkretne działy, które muszą następować w określonej kolejności [37]:

- *Identification division*, zawierająca dane na temat programu, takie jak jego nazwa, autor, data utworzenia;
- *Environment division*, opisująca środowisko, w którym program będzie wykonywany, a także interakcje z plikami;
- *Data division*, definiująca wszystkie zmienne, które program będzie wykorzystywał;
- *Procedure division*, zamierzająca całą logikę programów.

W skład każdego z działów wchodzi pomniejszych podsekcje [38]:

- *Configuration section*, część *Environment division*, zawierająca dane na temat komputera i procesora, na którym program będzie działać. Obecnie zbędna, ponieważ język zyskał możliwość zebrania tych danych automatycznie;
- *Input-Output section*, część *Environment division*, definiująca pliki i inne urządzenia, które będą stanowić wejście i wyjście programu;
- *File section*, część *Data division*, opisująca każdy plik, z którym program wejdzie w jakąkolwiek interakcję;
- *Working-storage section*, część *Data division*, definiująca zmienne alokowane tylko raz w trakcie kompilacji programu; ich wartość jest zachowywana nawet między jego oddzielnymi wywołaniami;
- *Local-storage section*, część *Data division*, definiująca zmienne, które alokowane są w momencie zainicjalizowania działania programu i dealokowane po jego zakończeniu;

- *Linkage section*, część *Data division*, definiująca struktury, które będą przekazywane między różnymi programami.

W *Procedure division*, gdzie znajduje się cała logika programu, możliwy jest podział kodu na pomniejsze, definiowane przez programistę, sekcje. Zazwyczaj pogrupowane są z uwagi na wykonywane zadanie. Można je wywołać w dowolnym miejscu *Procedure division* za pomocą słowa kluczowego *PERFORM*. Zastosowanie sekcji nie jest wymagane, ale pozytywnie wpływa na czytelność programu z uwagi na jasny podział odpowiedzialności na pomniejsze fragmenty kodu.

Język COBOL zawiera niewiele typów danych. Każda zmienna klasyfikowana jest poprzez klauzulę *PICTURE / PIC*. Podstawowe rodzaje zmiennych to:

- PIC X, zmienna alfanumeryczna – może zawierać dowolne znaki;
- PIC 9, zmienna numeryczna – może zawierać tylko numery;
- PIC A, zmienna alfabetyczna – może zawierać tylko litery.

Ponadto istnieją dodatkowe rozszerzenia do typów zmiennych:

- PIC S9, zmienna numeryczna uwzględniająca znak (+/-);
- PIC S9V9, zmienna zawierająca liczbę zmiennoprzecinkową;
- PIC 9 COMP, zmienna służąca do wykonywania obliczeń.

Każda zmienna w języku COBOL musi posiadać określoną długość, ponieważ nie istnieje możliwość dynamicznej alokacji pamięci po skompilowaniu programu.

W celu uczynienia języka jak najbardziej przystępnym, zawiera on ponad 500 zarezerwowanych słów kluczowych [39]. Podejście to pozwala na pisanie programów w sposób bardziej zbliżony do naturalnego języka angielskiego, a większość funkcjonalnie identycznego kodu można napisać na wiele różnych sposobów.

Listing 1. Wariant A warunku IF. Źródło: opracowanie własne

```
IF WS-AMOUNT = 0
    DISPLAY "WS-AMOUNT jest równe 0"
ELSE
    DISPLAY "WS-AMOUNT nie jest równe 0"
```

Listing 2. Wariant B warunku IF. Źródło: opracowanie własne

```
IF WS-AMOUNT EQUAL TO 0
    DISPLAY "WS-AMOUNT jest równe 0"
ELSE
    DISPLAY "WS-AMOUNT nie jest równe 0"
```

Przykłady widoczne na listingach 1 i 2, pomimo zastosowania zupełnie innej składni, zwracają identyczny rezultat.

Z uwagi na ilość synonimów, w firmach zazwyczaj funkcjonują konkretne założenia dotyczące kodu pisanego w języku COBOL w celu ujednolicenia programów.

Oprócz hierarchiczności samego programu, dane, które są przez ten program używane, także posiadają hierarchiczną strukturę. Do komunikacji między programami stosowane są tak zwane *copybook*'i, które definiują format przesyłanych danych. Listing 3 przedstawia przykładowy plik copybook.

Listing 3. Struktura danych w copybooku. Źródło: opracowanie własne

```

01 :PFX:-RECORD.
03 :PFX:-FUNCTION          PIC 9(01).
   88 :PFX:-NEW-ORDER      VALUE 01.

03 :PFX:-INPUT.
   05 :PFX:-I-ORDER-DATE   PIC X(50).
   05 :PFX:-I-ORDER-TYPE-ID PIC 9(10).
   05 :PFX:-I-EMPLOYEE-ID  PIC 9(10).
   05 :PFX:-I-SHOP-ID      PIC 9(10).
   05 :PFX:-I-CLIENT-ID   PIC 9(10).

03 :PFX:-OUTPUT.
   05 :PFX:-O-ORDER-ID     PIC 9(10).

   05 :PFX:-RETURN-CODE    PIC 9(03).
   88 :PFX:-OK              VALUE 200.
   88 :PFX:-INVALID-INPUT  VALUE 400.
   88 :PFX:-UNKNOWN-ERROR  VALUE 500.
   88 :PFX:-SQL-ERROR      VALUE 901.

05 :PFX:-MESSAGE          PIC X(500).

```

Plik *copybook* jest porównywalny do formatu XSD – nie zawiera danych, a jedynie definiuje to, jak powinny one wyglądać. Program może odwoływać się do zdefiniowanych w nim pól dołączając taką strukturę poprzez słowo-klucz *COPY*.

Każdy copybook można zawrzeć w programie dowolną ilość razy, pod warunkiem, że zastosowane są w nim *prefixy*, które można zastąpić dowolnym ciągiem znaków. Listing 4 przedstawia dwie sytuacje – standardowe dołączenie copybook'a wraz z zmianą nazwy przedrostka oraz dwukrotne dodanie tego samego copybook'a poprzez nadanie dwóch unikalnych nazw każdej instancji.

Listing 4. Dołączanie struktury copybook w programie COBOL. Źródło: opracowanie własne

```

COPY ORD000AR REPLACING ==:PFX:== BY ==ORD000==.
COPY ORD001AR REPLACING ==:PFX:== BY ==ORD001==.

COPY ORD002AR REPLACING ==:PFX:== BY ==ORD002==.
COPY ORD002AR REPLACING ==:PFX:== BY ==WK0002==.

```

Struktura pliku *copybook* jest ściśle zdefiniowana. Każda zmienna musi zawierać co najmniej poziom, nazwę oraz format danych. Ponadto można zawrzeć również wcześniej wspomniany *prefix* w celu umożliwienia wielokrotnego zawarcia tej struktury w jednym programie.

Na listingu 5 przedstawiony jest przykładowy plik copybook. Widoczna jest jego struktura hierarchiczna. Poszczególne elementy zostały oznaczone kolorami odpowiadającymi legendzie na listingu 6.

Listing 6. Struktura copybook'a. Źródło: opracowanie własne

```

01 :PFX:-RECORD.
03 :PFX:-FUNCTION          PIC 9(01).
    88 :PFX:-NEW-ORDER      VALUE 01.

03 :PFX:-INPUT.
    05 :PFX:-I-ORDER-DATE   PIC X(50).
    05 :PFX:-I-ORDER-TYPE-ID PIC 9(10).
    05 :PFX:-I-EMPLOYEE-ID  PIC 9(10).
    05 :PFX:-I-SHOP-ID      PIC 9(10).
    05 :PFX:-I-CLIENT-ID    PIC 9(10).

03 :PFX:-OUTPUT.
    05 :PFX:-O-ORDER-ID     PIC 9(10).

    05 :PFX:-RETURN-CODE    PIC 9(03).
        88 :PFX:-OK          VALUE 200.
        88 :PFX:-INVALID-INPUT VALUE 400.
        88 :PFX:-UNKNOWN-ERROR VALUE 500.
        88 :PFX:-SQL-ERROR   VALUE 901.

    05 :PFX:-MESSAGE        PIC X(500).
    
```

Listing 5. Struktura copybook'a - legenda. Źródło: opracowanie własne

```

Poziom zmiennej
Prefix
Nazwa zmiennej
Rodzaj zmiennej
Długość zmiennej
Wartość zmiennej
    
```

Zmienne najwyższego poziomu 01 są zazwyczaj stosowane jako „kontenery” na dane. Użycie ich w tej formie pomaga rozwiązać konflikty nazewnictwa, zarówno w obrębie jednego *copybook'a*, jak i kilku różnych. Kontener nie definiuje rozmiaru zmiennej – nie zawiera klauzuli *PICTURE*, posiada tylko nazwę i poziom w hierarchii.

Aby odwołać się do zmiennej z konkretnego kontenera, można zastosować klauzulę *OF*, która eliminuje niejasności. Przykład jej zastosowania jest widoczny na listingu 7.

Listing 7. Zastosowanie klauzuli *OF*. Źródło: opracowanie własne

```

MOVE ORD000-I-DATE-FROM OF ORD000-I-GET-ALL
                           TO ORD002-I-DATE-FROM
MOVE ORD000-I-DATE-TO   OF ORD000-I-GET-ALL
                           TO ORD002-I-DATE-TO
MOVE ORD000-I-ORDER-TYPE OF ORD000-I-GET-ALL
                           TO ORD002-I-ORDER-TYPE
MOVE ORD000-I-PAGE      OF ORD000-I-GET-ALL
                           TO ORD002-I-PAGE
    
```

Zmienne poziomów 02-49 są kolejnymi poziomami zagnieżdżenia w strukturze. Im wyższa liczba, tym niższy poziom w hierarchii.

Ponadto, istnieją specjalne poziomy – 66, 77 oraz 88.

66 służy do redefiniowania nazwy zmiennej za pomocą słowa klucz *RENAMES*. Na pierwszy rzut oka klauzula ta ma podobne zastosowanie do *REDEFINES*, jednak ta druga służy do „nadpisania” struktury w pamięci, natomiast pierwsza używana jest w celu nadania aliasu innym zmiennym.

Listing 8. Zastosowanie zmiennej poziomu 66 *RENAMES*. Źródło: opracowanie własne

```
01 :PFX:-PERSON.  
03 :PFX:-FIRST-NAME          PIC 9(20).  
03 :PFX:-LAST-NAME          PIC 9(20).  
03 :PFX:-PESEL              PIC 9(20).  
  
66 :PFX:-FULL-NAME RENAMES :PFX:-FIRST-NAME THRU  
   :PFX:-LAST-NAME.
```

Za pomocą tak zdefiniowanego *RENAMES*, możemy odwołać się do obu zmiennych równocześnie, zwiększając czytelność programu.

Zmienne 77 to zmienne samodzielne, które nie kwalifikują się do struktury hierarchicznej. Są zbliżone do poziomu 01, z wyjątkiem faktu, że nie można definiować ich podzmiennych. Z uwagi na ich odcięcie od wszechobecnej struktury hierarchicznej, stosowanie ich zazwyczaj nie jest zalecane.

Listing 9. Zastosowanie zmiennych poziomu 88. Źródło: opracowanie własne

```
01 :PFX:-RECORD.  
03 :PFX:-FUNCTION          PIC 9(01).  
    88 :PFX:-BATCH-PROCESS VALUE 01.  
    88 :PFX:-GET-ALL       VALUE 02.  
88 :PFX:-GET-BY-ID        VALUE 03.  
    88 :PFX:-ADD-NEW-ORDER  VALUE 04.  
    88 :PFX:-ADD-TO-ORDER  VALUE 05.  
    88 :PFX:-REMOVE-FROM-ORDER VALUE 06.
```

Poziom 88 używany jest do zdefiniowania konkretnej listy wartości, którą może przybierać zmienna. Zachowanie zmiennych tego poziomu jest podobne do obiektów *Enum* znanych z innych, nowoczesnych języków programowania. Na listingu 9 przedstawiony jest przykład wykorzystania tego poziomu – służy on za parametr decydujący o funkcji, którą wykonać ma program.

2.1.4. Zastosowane narzędzia

Poniższe podrozdziały krótko omawiają narzędzia, które były niezbędne do ukończenia systemu w języku COBOL.

Narzędzia kompilacji

W celu skompilowania programów napisanych na potrzeby pracy magisterskiej, zastosowany został wspomniany wcześniej kompilator GnuCOBOL.

Udostępnia on wiele opcji kompilacji, w zależności od zamierzonego działania programu. Kompilacja „głównego” modułu, który wchodzi w interakcję ze światem zewnętrznym, odbywa się za pomocą opcji *-x*. Podmoduły, używane przez główne moduły, stosują opcję *-m*. Natomiast programy, które

wchodzą w bezpośrednią interakcję z bazą danych wymagają wstępnego przetworzenia ich – w projekcie wykorzystany został preprocesor Open COBOL ESQL [40].

Visual Studio Code

Do napisania kodu wykorzystane zostało oprogramowanie Visual Studio Code. Wybór dokonany został poprzez dostępność wtyczek obsługujących formatowanie języka COBOL.

2.2. Java

Następujące podpunkty skupiają się na historii języka Java oraz kontekście jego zastosowań. Opisują jego podstawowe założenia oraz składnię.

2.2.1. Historia

Język Java powstał znacznie później niż wcześniej wspomniany COBOL – projekt stworzony przez zespół pod nadzorem Jamesa Goslinga rozpoczął się w 1991 roku, a pierwsza publicznie dostępna wersja została wydana w 1995 [41].

W przeciwieństwie do COBOLa, który był przełomowym i jednym z pierwszych języków programowania zaprojektowanym z myślą o byciu przystępnym oraz łatwym w zrozumieniu dla biznesu, Java nie była rewolucyjna – większość funkcji dostarczana przez ten język była już dostępna na rynku, jednak stanowiła ona pakiet zawierający wiele ważnych elementów, takich jak między innymi maszyna wirtualna w celu ułatwienia przenośności kodu, automatyczna alokacja i dealokacja pamięci poprzez Garbage Collector czy pełna obiektowość. Dzięki dostarczeniu wszystkich tych funkcjonalności w jednym języku programowania stał się on popularnym wyborem.

W dzisiejszych czasach, popularność Javy jest ogromna. Według wielu źródeł należy on do jednego z trzech najbardziej powszechnie używanych języków programowania na świecie [42], [43], [44].

2.2.2. Założenia, składnia i struktura

Java jest fundamentalnie językiem obiektowym. Oznacza to, że cała struktura kodu opiera się na obiektach i przestrzega czterech fundamentalnych zasad:

- Abstrakcja – szczegóły implementacji są ukryte, a programista operuje na wysokopoziomowych interfejsach i klasach abstrakcyjnych,
- Dziedziczenie – klasy występują w określonej hierarchii, mogą dziedziczyć po sobie pola i metody, co umożliwia ponowne wykorzystywanie kodu,
- Enkapsulacja – dostęp do pól obiektu jest chroniony i umożliwiony przez ściśle zdefiniowany interfejs,
- Polimorfizm – ten sam interfejs, metoda czy klasa mogą być implementowane wielokrotnie i na wiele różnych sposobów, a Java decyduje, który wariant wykorzystać w trakcie wykonania lub kompilacji.

W związku z powyższym kod źródłowy w języku Java wygląda zupełnie inaczej niż we wcześniej omówionym COBOLu. Projekt nie obraca się według pojedynczego pliku jako programu, a składa się z wielu klas. Punktem startowym każdego systemu jest metoda *public static void main(String[] args)* i to od niej rozpoczyna się wykonanie instrukcji.

W niniejszym projekcie każdy aspekt zaimplementowanego systemu znacząco różni się od kodu, który można zaobserwować dla systemu korzystającego z „czystej” Javy. Wynika to z zastosowania frameworku Spring Boot, który znacząco modyfikuje sposób interakcji programisty z kodem poprzez zastosowanie mechanizmu wstrzykiwania zależności. Listing 10 przedstawia główną metodę w systemie zarządzania zamówieniami, jednak z uwagi na wspomniane wykorzystywanie frameworka Spring Boot zawiera ona tylko jedną linię, przekazując odpowiedzialność do kontenera aplikacji Spring.

Listing 10. Metoda main w aplikacji Java. Źródło: opracowanie własne

```
package zz.java;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class JavaApplication {

    public static void main(String[] args) {
        SpringApplication.run(JavaApplication.class, args);
    }

}
```

Java Reflection API

W celu dostarczenia funkcjonalności zbliżonej do systemu implementowanego w języku Java dla modułów COBOLowych, niezbędne było napisanie aplikacji, która udostępnia interfejs Web API, a jako implementację usługi stosowany jest system w nadmienionym języku.

Ponieważ komunikacja z programami COBOL odbywa się przy pomocy plików zgodnych z definicją dostarczoną w pliku copybook – a więc plikach tekstowych sformatowanych ze ściśle określoną, odpowiednią ilością znaków, praca z nim znacząco różniłaby się od użycia powszechnie wykorzystywanego formatu JSON. Chcąc dostarczyć do programu wartość tekstową „TEST” do zmiennej, która w copybooku zdefiniowana jest jako X(10), użytkownik musiałby dopilnować, żeby poza zasadniczą zawartością dopisać na koniec 6 spacji. W przeciwnym wypadku wejście zostanie nieprawidłowo zinterpretowane.

Z uwagi na ten sposób działania drugim elementem tego systemu jest wrapper udostępniający endpointy jednolite z projektem w Javie, wykorzystujący do komunikacji z klientem format JSON. Jego główną funkcjonalnością jest tłumaczenie formatu JSON na plik tekstowy zgodny z dowolnym copybook’iem i odwrotnie.

Do zaimplementowania tego systemu wykorzystany został mechanizm refleksji dostępny w języku Java. Dzięki możliwości odczytania struktury klasy w czasie wykonywania programu napisany został procesor, który wymaga jedynie zastosowania odpowiednich adnotacji w celu automatycznej konwersji obiektu klasy na plik tekstowy odpowiadający strukturze copybook. Taka implementacja umożliwia wygodną komunikację z programami COBOL, która dla aplikacji klienckich niczym nie różni się od standardowego REST API.

2.2.3. Zastosowane frameworki, biblioteki, narzędzia

Poniższe rozdziały opisują podstawowe założenia i sposób działania najważniejszych zastosowanych w projekcie frameworków, bibliotek oraz narzędzi.

Spring Boot

Najważniejszym i równocześnie najbardziej fundamentalnie wpływającym na strukturę projektu zastosowanym narzędziem jest framework Spring Boot [45] – oparty na Spring, odwraca kontrolę nad zarządzaniem obiektami zgodnie z mechanizmem *IOC* (Inversion of Control). Oznacza to, że cykl życia każdego obiektu jest zarządzany przez kontener Springa, a nie przez programistę. Wymagane przez klasę obiekty są wstrzykiwane automatycznie poprzez *DI* (Dependency Injection), co eliminuje konieczność ręcznego inicjalizowania obiektów i pozwala skupić się na kodzie biznesowym, pozwalając kontenerowi Spring dostarczyć pasujący do wymagań obiekt.

Zasadniczą różnicą między frameworkiem Spring, a opartym na nim Spring Boot, jest podejście „konwencja nad konfiguracją”. Oznacza to, że w trakcie tworzenia projektu wykorzystywane jest wiele założeń i domyślnych wartości – w przeciwieństwie do swojego prekursora Spring Boot zmniejsza wymagany udział programisty w konfiguracji projektu, dzięki czemu staje się ona znacznie prostsza, a modyfikacje trzeba wprowadzać jedynie w razie określonej potrzeby. Przez łatwość jego zastosowania i znacznie szybsze przystosowanie projektu do pracy, framework ten stał się zdecydowanie bardziej popularny.

Spring Boot umożliwia dodanie do projektu wielu zestawów bibliotek w zależności od potrzeb. Ponieważ na potrzeby niniejszej pracy stworzony został projekt zgodny ze wzorcem REST API, wykorzystane zostały startery takie jak Spring Web, Spring Data JPA czy Spring Validation. Takie podejście zdecydowanie przyspiesza i ułatwia pracę, ponieważ większość przydatnych narzędzi jest dostępna od momentu inicjalizacji projektu.

Maven

Maven to popularne narzędzie do zarządzania i budowy projektu. Umożliwia on między innymi dołączanie bibliotek zewnętrznych jako zależności, korzystanie z wtyczek do automatycznej generacji kodu i kompilację oraz przygotowanie pakietu gotowego do wdrożenia.

Dzięki umieszczeniu wszystkich informacji niezbędnych do zbudowania projektu w pojedynczym pliku *POM* (Project Object Model) praca przy użyciu Maven jest bardzo wygodna. Na listingu 11 przedstawiony jest fragment wspomnianego pliku dla zaimplementowanego systemu.

Listing 11. Plik *pom.xml*. Źródło: opracowanie własne

```
<properties>
  <java.version>21</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

2.3. PostgreSQL

PostgreSQL [46] to jeden z najbardziej popularnych [47] silników relacyjnych bazy danych. Wyróżnia go fakt, że jako jeden z nielicznych jest on open-source. Dzięki swojej otwartej naturze, dostępne jest dla niego wiele rozszerzeń i narzędzi.

2.3.1. Uzasadnienie wyboru silnika bazy danych

Z uwagi na swoją otwartą naturę, autorzy projektu Open COBOL ESQL zdecydowali się na wykorzystanie PostgreSQL jako obsługiwany silnik. Wybór tego narzędzia jako preprocesor SQL oznaczał więc również wykorzystanie niniejszej bazy danych.

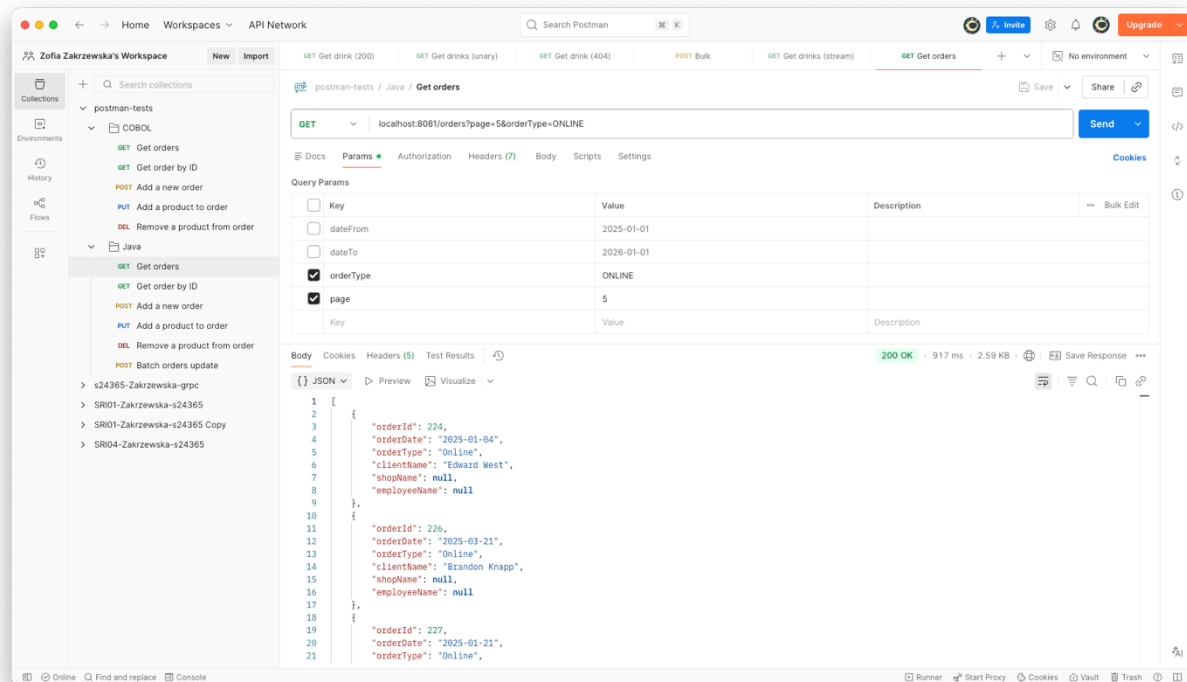
Java natomiast zapewnia wsparcie dla wielu silników – w celu ujednolicenia wykorzystywanych narzędzi w konfiguracji code-first również użyty został PostgreSQL.

W praktyce na Mainframe najczęściej wykorzystywany jest silnik od IBM, Db2 – jednak dzięki zastosowaniu preprocesora dostosowanie programów do użycia innej bazy danych nie byłoby procesem skomplikowanym z uwagi na ograniczoną ilość czystych fragmentów kodu SQL.

Zmiana silnika w systemie zaimplementowanym w Java byłaby jeszcze prostsza i nie wymaga żadnych modyfikacji kodu – wszystkie wykorzystane elementy są uniwersalne i umożliwiają natychmiastowe jego przełączenie.

2.4. Postman

Do przetestowania wszystkich zaimplementowanych systemów wykorzystany został program Postman [48]. Umożliwia on wygodne tworzenie zapytań HTTP i umieszczanie ich w kolekcjach, które można udostępnić poprzez ich wyeksportowanie. Na rysunku 7 został przedstawiony interfejs aplikacji wraz z przykładowym żądaniem wysłanym do zaimplementowanego systemu przetwarzania zamówień.



Rysunek 7. Interfejs aplikacji Postman z przykładowym żądaniem HTTP. Źródło: opracowanie własne

3. Podjęty problem biznesowy

Poniższe rozdziały omawiają wymagania biznesowe zadane dla obu zaimplementowanych systemów. Poza zdefiniowaniem podstawowych niezbędnych funkcjonalności, opisana jest struktura bazy danych, z której korzystają obydwie implementacje.

3.1. Podstawowe założenia systemu

Na potrzeby projektu zaprojektowany został prosty system przetwarzania zamówień. Zakłada on możliwość wykonywania podstawowych operacji na zamówieniu, a także przetworzenia dużej ilości zamówień z dostarczonych plików CSV.

System ten mógłby znaleźć zastosowanie w dowolnej firmie zajmującej się sprzedażą – możliwość szybkiego przetworzenia grupy zamówień umożliwiłoby szybkie uzupełnienie danych oraz łatwą integrację z innymi, obecnymi już systemami.

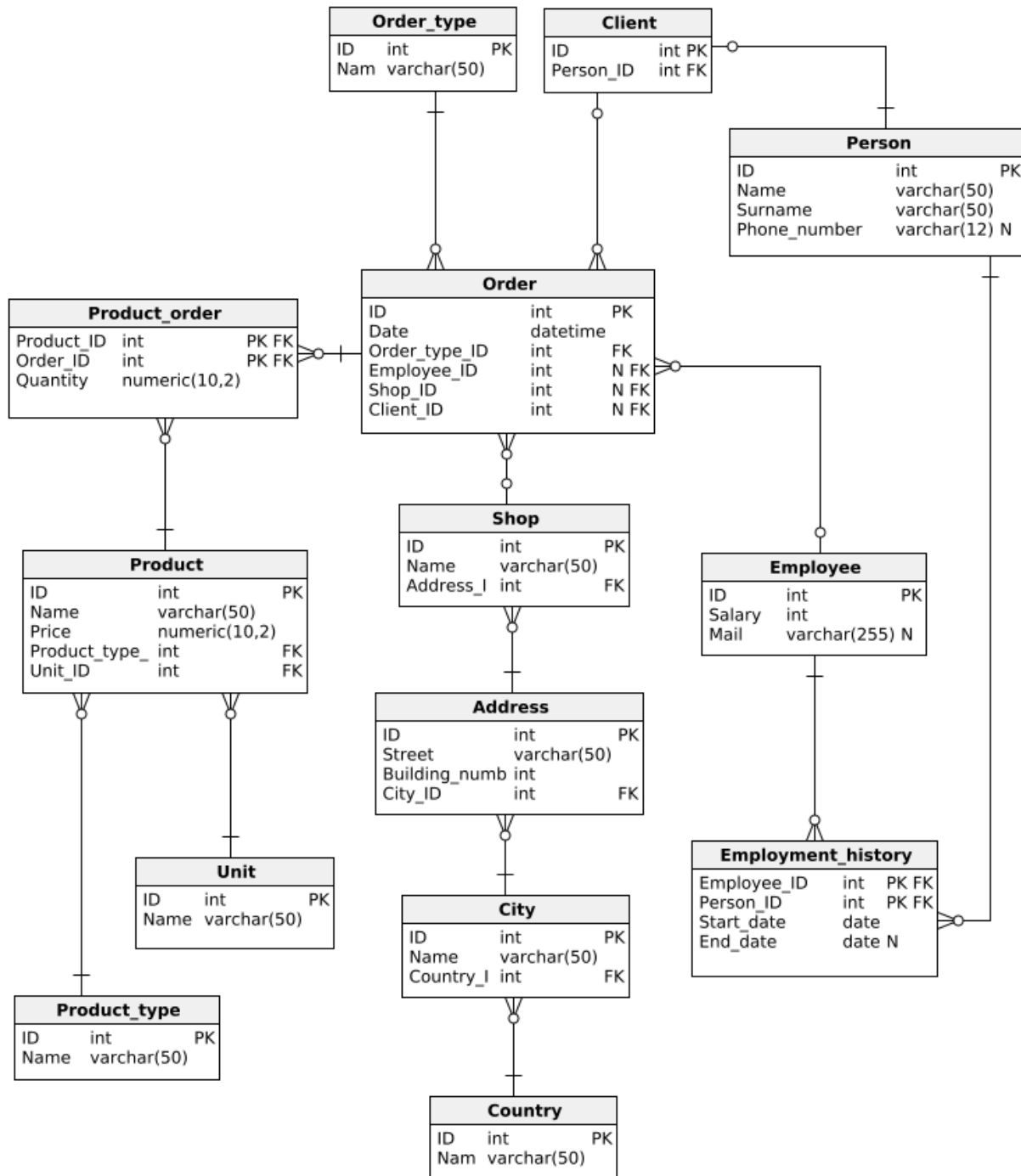
3.2. Wymagania biznesowe

Oba systemy spełniają następujące wymagania biznesowe:

- Udostępniają możliwość masowej aktualizacji zamówień zgodnie z zapewnionymi plikami CSV,
- Pozwalają na pobranie listy zamówień, która wspiera paginację oraz filtry,
- Zwracają detale zamówienia według podanego ID, łącznie z produktami, które zostały zamówione,
- Umożliwiają dodanie nowego zamówienia, łącznie z danymi na temat klienta, sklepu, w którym dokonano zamówienia oraz sprzedawcy,
- Pozwalają na dodanie lub usunięcie produktów z istniejącego już zamówienia.

3.3. Projekt bazy danych

Diagram związków encji zaprojektowany na potrzeby systemu zarządzania zamówieniami jest widoczny na rysunku 8.



4. Architektura zaprojektowanych systemów

Poniższe rozdziały szczegółowo opisują architekturę rozwiązań zaproponowanych na potrzeby pracy. Zawierają wysokopoziomowy opis obydwu implementacji, jak również przyjęte założenia i konwencje.

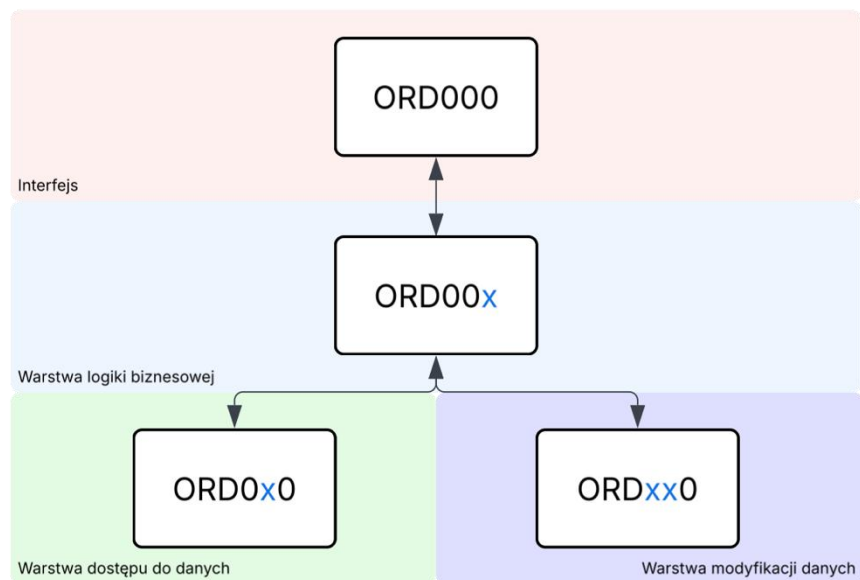
4.1. COBOL

Poniższy rozdział omawia założenia, strukturę i przyjęte konwencje dla systemu w języku COBOL.

4.1.1. Wysokopoziomowy opis rozwiązania

W celu implementacji projektu w stylu mikroserwisowym, podział na moduły dokonany został według funkcjonalności. Każdy program ma swoją ściśle określoną rolę zgodnie z zasadą *Single-Responsibility*.

Dodatkowo, projekt został podzielony na 3 warstwy: interfejs, który służy do komunikowania się z całym systemem, warstwę logiki biznesowej, w której dokonywane jest mapowanie wyjścia i wejścia i walidacja danych, oraz warstwę dostępu do danych, w której programy wchodzi w bezpośrednią interakcję z bazą w celu ich odczytu, zapisu, modyfikacji czy usunięcia.



Rysunek 9. Warstwowa architektura systemu w języku COBOL. Źródło: opracowanie własne

Podział widoczny na rysunku 9 oparty był na standardowej formie projektu WebAPI w nowoczesnych językach programowania. Warstwa interfejsu przypomina kontrolery, warstwa logiki biznesowej – serwisy, natomiast warstwa dostępu do danych jest przybliżona do repozytoriów.

Wybrana architektura posiada niewątpliwą zaletę: podział na warstwy oraz z uwagi na funkcjonalność powoduje, że powstały kod jest bardzo modułowy, i można wykorzystywać go wielokrotnie.

4.1.2. Konwencja nazewnictwa

W celu ustandaryzowania nazw programów wchodzących w skład projektu, została wybrana jednolita konwencja nazewnictwa.

W związku z podjętym tematem dotyczącym tabeli *Order*, dla każdego programu w projekcie wykorzystany został prefix *ORD*. Dla programów wchodzących w interakcję z innymi elementami danych, można wykorzystać inny prefix.

Z uwagi na warstwową architekturę, konkretne numery występujące po prefix'ie zostały zarezerwowane dla programów o określonej funkcjonalności. Tabela 1 zawiera przyjętą numerację i przynależność nazwy do warstwy.

Tabela 1. Konwencja nazewnictwa w projekcie. Źródło: opracowanie własne

Nazwa programu	Warstwa	Funkcjonalność
ORD000	Interfejs	Komunikacja aplikacji klienckich z całym systemem
ORD001 – ORD009	Logiki biznesowej	Walidacja danych, mapowanie wejścia i wyjścia
ORD010 – ORD090	Dostępu do danych	Odczyt danych z bazy
ORD100	Logiki biznesowej	Przetwarzanie danych z pliku CSV
ORD101 – ORD900	Dostępu do danych	Modyfikacja danych w bazie

4.1.3. Uzasadnienie wyboru architektury projektu

Architektura została wybrana z uwagi na kontekst, w jakim zazwyczaj funkcjonuje podobny system. Język COBOL stosowany jest w dużych firmach, gdzie prawdopodobnie dostęp do konkretnej tabeli w bazie danych potrzebny jest wielokrotnie, w różnych kontekstach biznesowych.

Dzięki obudowaniu każdej tabeli w program, który pozwala na odczyt danych, program modyfikujący dane i walidujący wejście, i program walidujący, który ostatecznie będzie używany przez inne systemy, udostępniamy im możliwość skorzystania z naszych tabel, równocześnie zapobiegając „partyzanckiemu” dostępowi do danych poprzez wymuszanie walidacji danych wejściowych zgodnej z zasadami biznesowymi.

Projekt pisany w języku COBOL na Mainframe wygląda zupełnie inaczej niż nowoczesne języki. Z zasady każdy program może odwołać się do innego, dowolnego programu – oczywiście da wprowadzić się pewne ograniczenia z uwagi na autoryzację i środowisko, ale zazwyczaj istnieje duża dowolność.

Program powinien być niewielką jednostką, z ograniczonym zakresem obowiązków – porównywalny do pojedynczej klasy w obiektowych językach programowania. W praktyce więc każdy z programów może funkcjonować jako mała biblioteka.

Tak prosty mechanizm wołania programów przez inne programy pozwala potencjalnie napisać wielokrotnie używalne, standaryzowane moduły, które zawierają niezbędną walidację danych i stanowią dobry poziom izolacji między tabelą w bazie, a programami, które potrzebują dostępu do zawartych w niej danych.

4.1.4. Komunikacja poprzez WebAPI

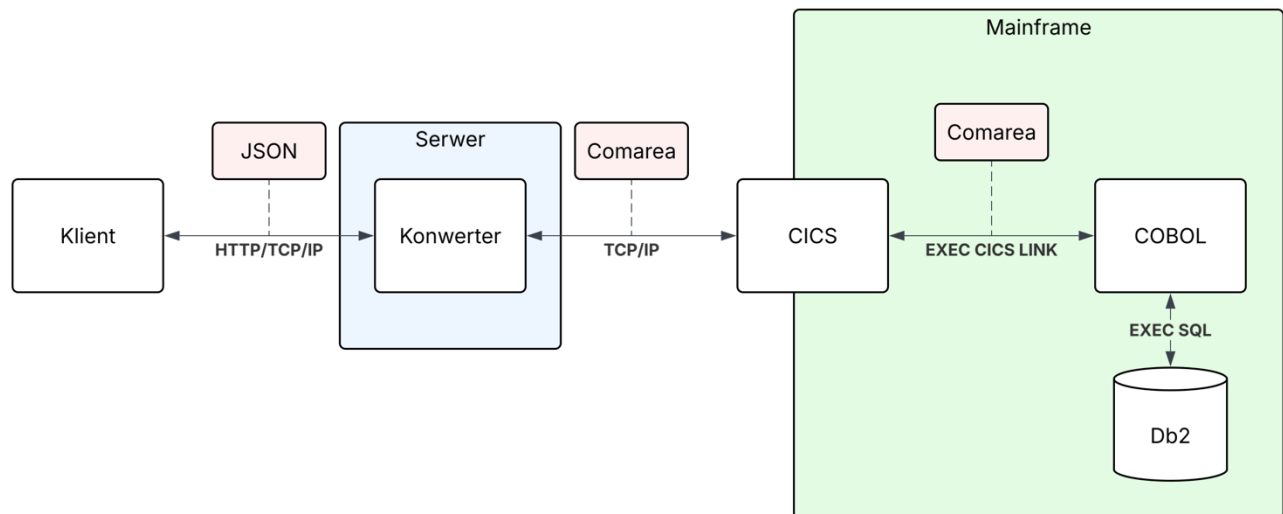
Na prawdziwym Mainframe działanie programu możliwe jest do zainicjalizowania za pomocą CICS. *Customer Information Control System* to produkt IBM, który umożliwia komunikację z Mainframe na kilka sposobów, między innymi za pomocą protokołu HTTP/HTTPS [49]. Programy działające na tych komputerach nie nasłuchują zapytań stale, czym różnią się od standardowych rozwiązań WebAPI.

Z uwagi na problemy z uzyskaniem licencji od IBM, nie istniała możliwość wykorzystania CICS. W celu umożliwienia komunikacji z systemem w języku COBOL poprzez żądania HTTP z sposób zbliżony do standardowo wykorzystywanego rozwiązania, została utworzona usługa w języku Java, wywołująca programy COBOL w odpowiedzi na zapytanie REST.

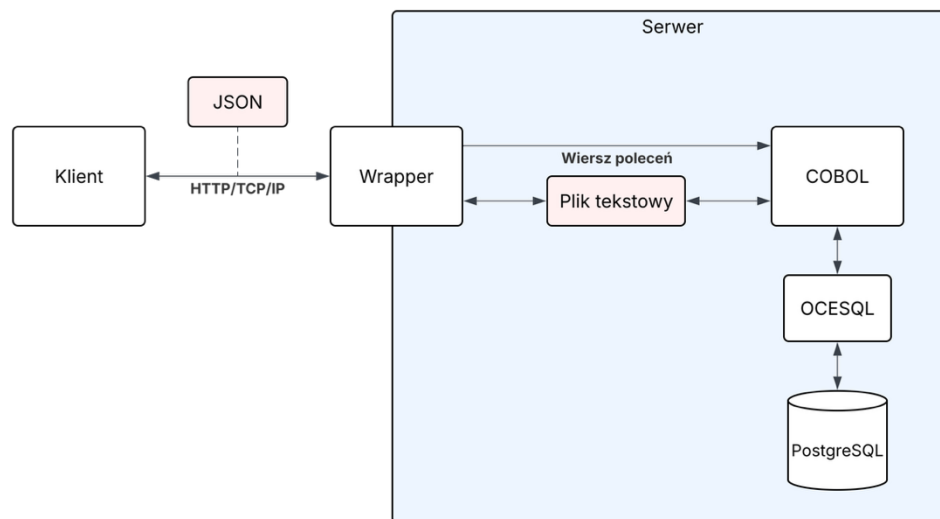
W przeciwieństwie do standardowego funkcjonowania REST API pisanego w nowoczesnym języku programowania, które jest stale włączone oraz nasłuchuje zapytań, programy COBOL dostępne na komputerach Mainframe są odpalane jedynie na żądanie, nie są aktywne cały czas – to CICS nasłuchuje żądań.

Rozwiązanie dostarczone na potrzeby projektu działa więc na bardzo podobnej zasadzie. Wrapper napisany w języku Java zastępuje rolę CICS, wywołując odpowiedni program zgodnie z wywołanym endpointem REST.

Sposób przetwarzania zapytań poprzez narzędzia IBM oraz podejście wykorzystane w niniejszym projekcie została zaprezentowana na rysunkach 10 i 11.



Rysunek 10. Przetwarzanie żądania przez Mainframe. Źródło: opracowanie własne



Rysunek 11. Przetwarzanie żądania przez zaimplementowany system. Źródło: opracowanie własne

W przypadku rozwiązania oficjalnego, komunikacja odbywa się poprzez *Comarea* – jest to surowy, binarny blok danych o stałej długości, sformatowany zgodnie ze zdefiniowanym plikiem copybook. Każdy element zajmuje ściśle określoną pozycję, a w zależności od rodzaju zmiennej, używane są różne formaty binarne, co wymaga odpowiedniej interpretacji przez aplikację kliencką. Zazwyczaj więc stosowany jest pośrednik, tłumaczący zapytania klienta wysyłane w formacie JSON na format binarny zgodny ze strukturą

copybook'a, który będzie poprawnie przetworzony przez Mainframe, oraz z formatu binarnego na JSON, który jest stosowany przez klienta. Rozwiązanie to jest zbliżone do niniejszej implementacji, z różnicą w sposobie wysyłania danych – w tym wypadku zastosowane zostały pliki tekstowe do komunikacji obustronnej.

4.2. Java

Poniższy rozdział omawia założenia, strukturę i przyjęte konwencje dla systemu w języku Java.

4.2.1. Wysokopoziomowy opis rozwiązania

System w języku Java, podobnie jak ten dostarczony w języku COBOL, również dzieli się na warstwy odpowiedzialne za pojedyncze zadania:

- Kontrolery – odpowiadają za komunikację z aplikacjami klienckimi, przyjmują żądania HTTP i odsyłają je do odpowiedniego serwisu,
- Serwisy – zawierają całą logikę biznesową, przetwarzają żądania,
- Repozytoria – odpowiadają za komunikację z bazą danych,
- Modele – reprezentują obiekty biznesowe, tabele w bazie i dane dostarczane jako argumenty zapytań.

Podział w praktyce jest bardzo zbliżony do omówionego wcześniej systemu w COBOL – kontrolery odpowiadają warstwie interfejsu, serwisy – logiki biznesowej, a repozytoria zapewniają tą samą funkcjonalność, co programy warstwy dostępu i modyfikacji danych.

4.2.2. REST API

Aplikacja zrealizowana jest zgodnie z konwencją Representational State Transfer (REST). Jest to określony zestaw zasad [50], [51], które definiują sposób projektowania interfejsów API w architekturze klient-serwer:

- Jednolity interfejs – zasoby są identyfikowane przez URI; klienci używają tych samych metod w ten sam sposób,
- Separacja klient-serwer – odpowiedzialność jest rozdzielona – klient odpowiada za interfejs użytkownika, a serwer za przechowywanie danych, bezpieczeństwo i zarządzanie zasobami,
- Bezstanowość – każde żądanie zawiera wszystkie informacje, które są niezbędne do jego przetworzenia; serwer nie przechowuje stanu sesji aplikacji klienckich,
- Buforowanie – odpowiedzi serwera jasno mówią o tym, czy mogą zostać zachowane w buforze,
- Warstwy – aplikacje pośredniczące, takie jak np. API Gateway, mogą znajdować się między klientem a serwerem, nie zmieniając przy tym kontraktu,
- Kod na żądanie – opcjonalnie – serwer może przysyłać skrypty w celu rozszerzenia funkcjonalności klienta.

5. Implementacja systemów

Poniższe rozdziały omawiają dokładnie obydwa zaimplementowane systemy, jak i również zawierają instrukcję, jak je skompilować oraz włączyć. W przypadku języka COBOL opisane są wszystkie programy wchodzące w skład projektu, w przypadku Javy przedstawiona jest ogólna struktura kodu. Ponadto omówione są diagramy aktywności wszystkich metod biznesowych.

5.1. COBOL

Niniejszy rozdział zawiera diagramy wywołań każdej funkcjonalności biznesowej. Przechodzi również przez każdy z sześciu głównych typów programów i szczegółowo omawia ich działanie na przykładzie konkretnej wykonywanej przez niego funkcji.

5.1.1. Lista programów

System składa się w sumie z 17 programów. Każdy z nich ma swoją konkretną odpowiedzialność, zgodnie z opisaną wcześniej konwencją nazewnictwa:

- ORD000 – interfejs, odpowiada za walidację danych i odpowiednie pokierowanie żądania,
- ORD001 – logika biznesowa dla masowej aktualizacji zamówień,
- ORD002 – logika biznesowa dla pobrania listy zamówień,
- ORD003 – logika biznesowa dla pobrania szczegółów zamówienia,
- ORD004 – logika biznesowa dla dodania nowego zamówienia,
- ORD005 – logika biznesowa dla dodania nowego produktu do zamówienia,
- ORD006 – logika biznesowa dla usunięcia produktu z zamówienia,
- ORD010 – pobieranie danych z tabeli *ORDERS*,
- ORD020 – pobieranie danych z tabeli *ORDER_TYPE*,
- ORD030 – pobieranie danych z tabeli *CLIENT*,
- ORD040 – pobieranie danych z tabeli *SHOP*,
- ORD050 – pobieranie danych z tabeli *PRODUCT_ORDER*,
- ORD060 – pobieranie danych z tabeli *EMPLOYEE*,
- ORD070 – pobieranie danych z tabeli *PRODUCT*,
- ORD100 – przetwarzanie pliku CSV,
- ORD110 – modyfikacja danych w tabeli *ORDERS*,
- ORD150 – modyfikacja danych w tabeli *PRODUCT_ORDER*.

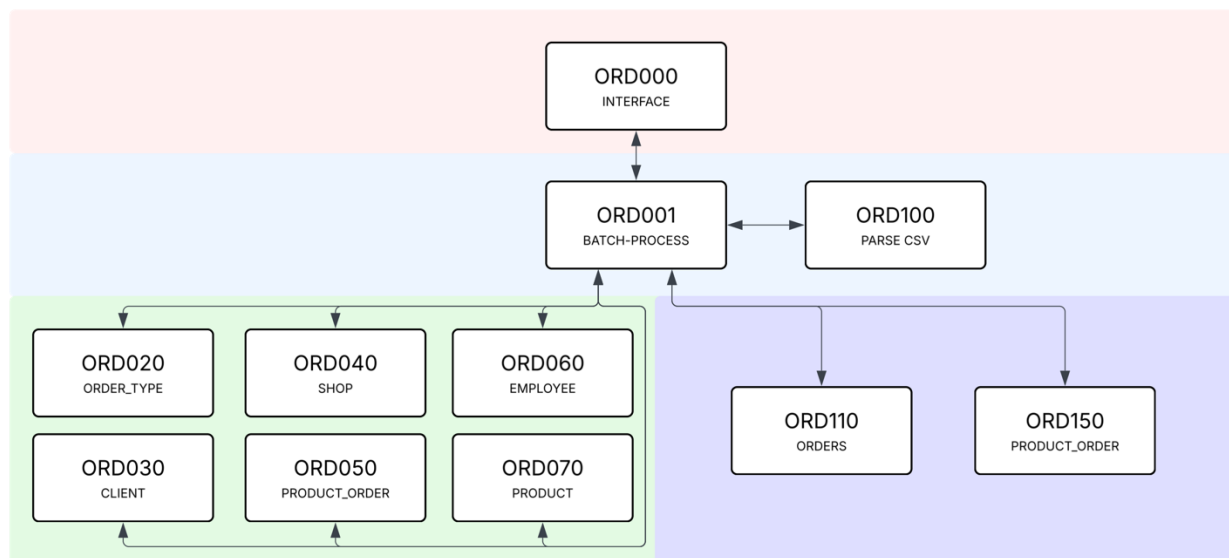
5.1.2. Diagram wywołań funkcjonalności biznesowych

W celu ulokowania programów w kontekście systemu, w następujących punktach przedstawiony zostanie diagram wywołań każdej obsługiwanej funkcjonalności biznesowej.

Masowa aktualizacja zamówień

Proces przetwarzania masowego zamówień jest zdecydowanie najbardziej złożoną funkcją obsługiwaną przez system. Przetwarza ona zamówienia dostarczone do programu za pomocą pliku CSV, w którym w jednej linii znajdują się dane do umieszczenia docelowo w kilku tabelach. Dodatkowo niezbędna jest walidacja każdego rzędu i zweryfikowanie, że rekordy, z którymi powiązane będzie zamówienie, takie jak np. klient, sklep czy pracownik, istnieją w bazie danych.

Na rysunku 10 przedstawiona jest kolejność wywoływania programów przez system w toku działania funkcji *BATCH-PROCESS*.

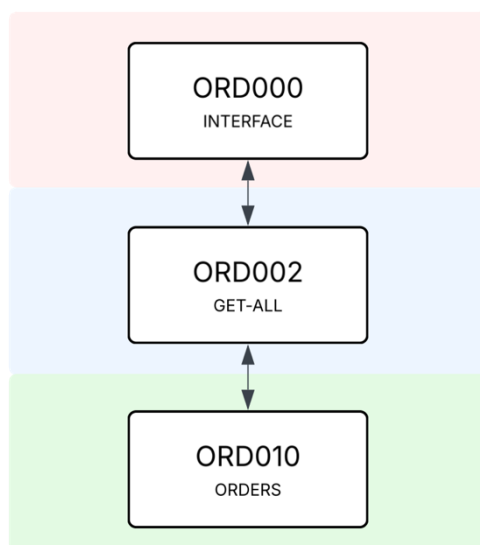


Rysunek 12. Diagram wywołań funkcji BATCH-PROCESS. Źródło: opracowanie własne

Po włączeniu programu ORD000 dokonywana jest wstępna walidacja. Następnie żądanie przekazywane jest do ORD001, który dla każdego rzędu w pliku CSV wywołuje ORD100. Ten przetwarza rekord CSV na strukturę zgodną z copybook’iem ORDER001. Dla każdej instancji tej struktury dokonywana jest walidacja jej elementów poprzez wyszukanie ich w bazie – odpowiedzialne za ten krok są programy ORD020 – ORD070. Po poprawnym przejściu procesu walidacji, dane wprowadzane są do bazy za pośrednictwem programów ORD110 dla zamówień i ORD150 dla elementów zamówienia.

Pobranie listy zamówień

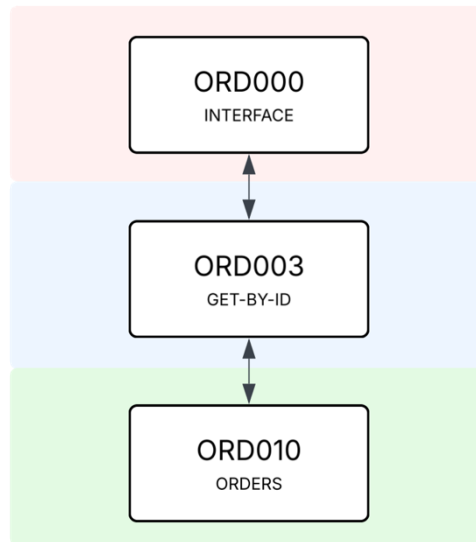
W przeciwieństwie do masowego przetwarzania zamówień, proces pobierania listy zamówień przedstawiony na rysunku 11 jest mniej złożony. W toku działania funkcji wywoływane są jedynie trzy programy – ORD000 jako interfejs, ORD002 przygotowuje dane dla ORD010, który ostatecznie komunikuje się z bazą danych i pobiera zadane informacje.



Rysunek 13. Diagram wywołań funkcji GET-ALL. Źródło: opracowanie własne

Pobranie szczegółów zamówienia

Pobranie szczegółów zamówienia jest procesem bardzo podobnym do listy – korzysta z tych samych programów. Diagram wywołań przedstawiony jest na rysunku 12.

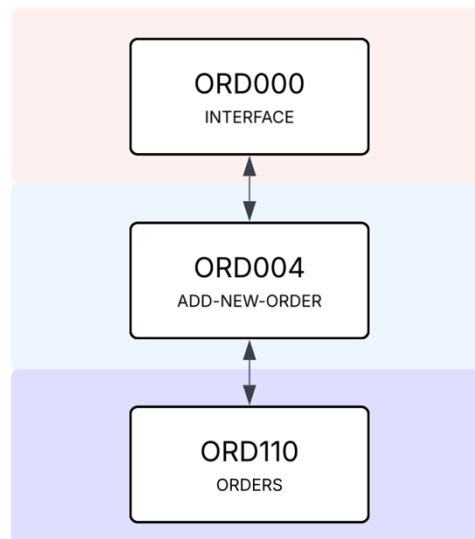


Rysunek 14. Diagram wywołań funkcji GET-BY-ID. Źródło: opracowanie własne

ORD010 jest jedynym programem w systemie, który zawiera bezpośrednie zapytania *SELECT* na tabeli *ORDERS*. Udostępnia dwie funkcje – jedna do pobrania listy, druga do detali.

Dodanie nowego zamówienia

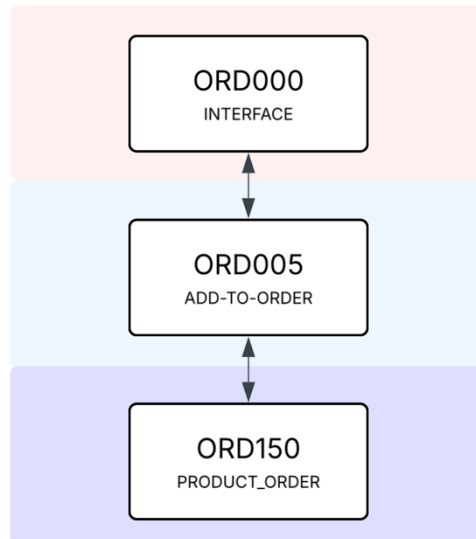
Proces dodawania zamówienia również obejmuje jedynie trzy programy. Tym razem jednak ostatecznie dociera do warstwy modyfikacji danych – ORD110 odpowiadający za modyfikację danych w tabeli *ORDERS*. Schemat przedstawiony jest na rysunku 13.



Rysunek 15. Diagram wywołań funkcji ADD-NEW-ORDER. Źródło: opracowanie własne

Dodanie nowego produktu do zamówienia

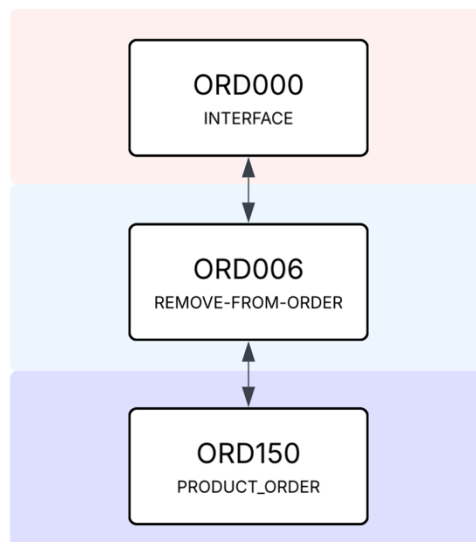
Podobnie jak w przypadku dodawania nowego zamówienia, dodawanie nowego produktu do zamówienia wymaga wywołania trzech programów – interfejsu ORD000, programu logiki biznesowej ORD005 oraz programu odpowiadającego za modyfikację rekordów w *PRODUCT_ORDER* – ORD150. Przebieg przedstawiony jest na rysunku 16.



Rysunek 16. Diagram wywołań funkcji ADD-TO-ORDER. Źródło: opracowanie własne

Usunięcie produktu z zamówienia

Ponieważ funkcja *REMOVE-FROM-ORDER* również modyfikuje tabelę *PRODUCT_ORDER*, jej wykonanie jest bardzo zbliżone do funkcji *ADD-TO-ORDER*. Rysunek 17 obrazuje kroki jej działania.



Rysunek 17. Diagram wywołań funkcji REMOVE-FROM-ORDER. Źródło: opracowanie własne

5.1.3. Interfejs - ORD000

ORD000 jest programem, który służy do komunikacji aplikacji klienckich z serwisem. Można włączyć go za pomocą wiersza poleceń z dwoma argumentami wejściowymi – ścieżką do plików wejściowego oraz wyjściowego. W pliku wejściowym do programu dostarczony musi być argument decydujący o zażądanej funkcji, a także wszystkie dane potrzebne do jej wykonania.

Na potrzeby omówienia działania ORD000 przedstawiony zostanie przepływ funkcji wyszukiwania listy zamówień. Fragment struktury, jaką musi on posiadać, widoczny jest na listingu 12.

Listing 12. Copybook ORD000AR - fragment. Źródło: opracowanie własne

```
01 :PFX:-RECORD.
03 :PFX:-FUNCTION          PIC 9(01).
    88 :PFX:-BATCH-PROCESS    VALUE 01.
    88 :PFX:-GET-ALL          VALUE 02.
    88 :PFX:-GET-BY-ID        VALUE 03.
    88 :PFX:-ADD-NEW-ORDER    VALUE 04.
    88 :PFX:-ADD-TO-ORDER     VALUE 05.
    88 :PFX:-REMOVE-FROM-ORDER VALUE 06.

03 :PFX:-INPUT.
    05 :PFX:-INPUT-RAW.
        07 :PFX:-INPUT-RAW-ALL PIC X(1000).

    05 :PFX:-I-BATCH-PROCESS REDEFINES :PFX:-INPUT-RAW.
        07 :PFX:-I-FILE-NAME   PIC X(100).

    05 :PFX:-I-GET-ALL REDEFINES :PFX:-INPUT-RAW.
        07 :PFX:-I-DATE-FROM   PIC X(10).
        07 :PFX:-I-DATE-TO     PIC X(10).
        07 :PFX:-I-ORDER-TYPE  PIC 9(01).
            88 :PFX:-I-ONLINE    VALUE 1.
            88 :PFX:-I-IN-STORE  VALUE 2.
        07 :PFX:-I-PAGE        PIC 9(10).
```

`:PFX:-FUNCTION` powinno zawierać jedną ze zmiennych poziomu 88 – widoczne powyżej, definiują funkcjonalności udostępniane przez system. Poprzez zastosowanie tego rozwiązania, zakres możliwości programu jest jasny. W przypadku omawianej funkcji `GET-ALL` będzie to wartość 1.

Następnym elementem pliku wejściowego są argumenty potrzebne do wykonania funkcji. Kontener `:PFX:-INPUT-RAW`, zdefiniowany jako zmienna alfanumeryczna o długości 1000 znaków to maksymalna sumaryczna długość podanych argumentów.

Za pomocą klauzuli `REDEFINES` wspomniany wyżej kontener `:PFX:-INPUT-RAW` zamieniany jest na ustrukturyzowane dane, przygotowane pod konkretną funkcjonalność. Zastosowanie tego rozwiązania gwarantuje, że dane, które są przekazane do programu, będą dotyczyć jedynie zadanej opcji oraz umożliwia ono odwoływanie się do konkretnych pól w programie. Z uwagi na narzucaną statyczną długość każdego pliku wejściowego, w przypadku utworzenia definicji danych do osobnych funkcji bez zastosowania takiego kontenera, wszystkie wcześniejsze pola należałoby wypełnić spacjami w ilości równej do długości wszystkich poprzedzających pól.

Listing 13. ORD000 - PROCEDURE DIVISION. Źródło: opracowanie własne

```
*>*****  
PROCEDURE DIVISION.  
*>*****  
    MOVE LOW-VALUES TO ORD000-RECORD  
    MOVE 500          TO WS-RETURN-CODE  
  
    PERFORM READ-ARGS  
    PERFORM VALIDATE-FILE  
  
    PERFORM MAP-INPUT  
    PERFORM VALIDATE-INPUT  
    PERFORM PROCESS-INPUT  
  
    PERFORM END-PROGRAM  
    .
```

Listing 13 przedstawia *PROCEDURE DIVISION*, które w każdym programie w języku COBOL jest początkiem jego wykonania. Pierwszym krokiem w tej sekcji jest wyzerowanie przestrzeni w pamięci przypisanej do copybook'a ORD000AR, w celu wyeliminowania ewentualnych pozostałości po poprzednim wykonaniu. Zmienna *WS-RETURN-CODE* również ma nadawaną generyczną wartość 500 – zastosowane kody wyjścia podążają za protokołem HTTP.

Po inicjalizacji dokonywany jest odczyt argumentów z wiersza poleceń. Listing 14 zawiera kod, który wycina te argumenty oraz przenosi je do odpowiednich zmiennych: *WS-FILE-IN* dla pliku wejściowego oraz *WS-FILE-OUT* dla pliku wyjściowego.

Listing 14. ORD000 - sekcja READ-ARGS. Źródło: opracowanie własne

```
*>*****  
READ-ARGS SECTION.  
*>*****  
    ACCEPT WS-CMDLINE FROM COMMAND-LINE  
    UNSTRING FUNCTION TRIM(WS-CMDLINE)  
        DELIMITED BY ALL SPACES  
        INTO WS-FILE-IN  
            WS-FILE-OUT  
    END-UNSTRING  
    .  
READ-ARGS-X.  
EXIT.
```

Ponieważ ścieżka do pliku wejściowego jest wymagana, a wartość domyślna nie jest przyjmowana, program sprawdza jego obecność pod zmienną *WS-FILE-IN*.

Po tej weryfikacji program rzutuje plik wejściowy na copybook ORD000AR – z uwagi na tą czynność długość każdej zmiennej w pliku musi zgadzać się z definicją. W przeciwnym wypadku zmienne zostaną nieodpowiednio przeniesione – typowym rezultatem takiego błędu są wartości, które zaczynają się od spacji i są ucięte, a ich brakująca część znajduje się w następnej zmiennej. Proces tego rzutowania przedstawiony jest na listingu 15.

Listing 15. ORD000 - sekcja MAP-INPUT. Źródło: opracowanie własne

```
*>*****
MAP-INPUT SECTION.
*>*****

    OPEN INPUT FILE-IN

    READ FILE-IN
      AT END
        SET WS-EOF TO TRUE
    END-READ

    MOVE FILE-INPUT TO ORD000-RECORD
    CLOSE FILE-IN

    .
MAP-INPUT-X.
EXIT.
```

COBOL umożliwia dosyć prostą obsługę plików – wczytanie zawartości z pliku tekstowego nie wymaga napisania wielu linijek kodu, w przeciwieństwie do dosyć skomplikowanego, analogicznego mechanizmu w Javie. Dużym minusem jednak jest brak walidacji zawartości. Zostanie ona wczytana i następnie rzutowana na copybook ORD000AR niezależnie, czy struktura jest prawidłowa.

Na listingu 16 przedstawiony jest odpowiednio sformatowany plik wejściowy dla funkcji *GET-ALL*. Listing 17 natomiast obrazuje niepoprawną strukturę.

Listing 16. Poprawny format pliku wejściowego do programu ORD000. Źródło: opracowanie własne

```
2          2026-01-01100000000004
```

Listing 17. Niepoprawny format pliku wejściowego do programu ORD000. Źródło: opracowanie własne

```
2          2026-01-01100000000004
```

Wynikiem rzutowania poprawnie umieszczonych wartości z listingu 19 byłyby następujące zmienne:

- :PFX:-FUNCTION: 2 (odpowiada :PFX:-GET-ALL)
- :PFX:-I-DATE-FROM: null
- :PFX:-I-DATE-TO: 2026-01-01
- :PFX:-I-ORDER-TYPE: 1 (odpowiada :PFX:-ONLINE)
- :PFX:-I-PAGE: 4

Natomiast rezultatem podania danych z listingu 20, rzutowanie zakończyłoby się innym wynikiem:

- :PFX:-FUNCTION: 2 (odpowiada :PFX:-GET-ALL)
- :PFX:-I-DATE-FROM: 20
- :PFX:-I-DATE-TO: 26-01-0110
- :PFX:-I-ORDER-TYPE: 0 (nie odpowiada żadnej wartości)
- :PFX:-I-PAGE: 400 (spacje w przypadku rzutowania na wartość numeryczną zamieniane są na zera)

Wszystkie wartości zostały przesunięte o dwie spacje, których brakowało do wypełnienia pustej zmiennej :PFX:-I-DATE-FROM.

Tak wrażliwy na drobne nieścisłości system wprowadzania danych generuje bardzo wiele błędów, które czasem bardzo trudno wyłapać. Szczególnie problematyczne są zmiany w copybook'ach – w przypadku rekompilacji tylko niektórych używających go programów, jeden z nich może oczekiwać wartości pola o długości 10 na offsecie 35, natomiast drugi będzie szukać tej samej wartości o długości 10, ale na offsecie 40. Błąd ten wynika z faktu, że definicje danych z copybooków są statycznie dołączane do programu w czasie jego kompilacji.

W przypadku pomyślnego zakończenia procesu rzutowania danych, wykonywana jest ich walidacja. Na tym etapie program sprawdza, która funkcja została wybrana, i sprawdza odpowiednie dla niej pola – sposób działania przedstawiony jest na listingu 18.

Listing 18. ORD000 - sekcja VALIDATE-INPUT. Źródło: opracowanie własne

```
*>*****
VALIDATE-INPUT SECTION.
*>*****

EVALUATE TRUE
  WHEN ORD000-BATCH-PROCESS
    PERFORM VALIDATE-ORD001
  WHEN ORD000-GET-ALL
    PERFORM VALIDATE-ORD002
  WHEN ORD000-GET-BY-ID
    PERFORM VALIDATE-ORD003
  WHEN ORD000-ADD-NEW-ORDER
    PERFORM VALIDATE-ORD004
  WHEN ORD000-ADD-TO-ORDER
    PERFORM VALIDATE-ORD005
  WHEN ORD000-REMOVE-FROM-ORDER
    PERFORM VALIDATE-ORD006

  WHEN OTHER
    DISPLAY "Unknown function"
    MOVE 400 TO WS-RETURN-CODE
    PERFORM END-PROGRAM
END-EVALUATE

.
VALIDATE-INPUT-X.
EXIT.
```

W przypadku omawianej funkcji *GET-ALL* wywołana zostanie sekcja *VALIDATE-ORD002*, przedstawiona na listingu 19.

Listing 19. *ORD000* - sekcja *VALIDATE-ORD002*. Źródło: opracowanie własne

```
*>*****  
VALIDATE-ORD002 SECTION.  
*>*****  
  IF ORD000-I-PAGE OF ORD000-I-GET-ALL = ZEROES OR LOW-VALUE  
    MOVE 1 TO ORD000-I-PAGE  
  END-IF  
  
  .  
VALIDATE-ORD002-X.  
EXIT.
```

Ponieważ funkcjonalność listowania zamówień nie ma żadnych wymaganych argumentów, sekcja ta zawsze zakończy się powodzeniem. W przypadku braku podania numeru strony przyjmowana jest domyślna wartość 1.

Inne funkcje jednak mają wymagane pola – mechanizm błędu walidacji można zaobserwować na przykładzie sekcji *VALIDATE-ORD004* zaprezentowanej na listingu 20, która stosowana jest dla funkcji *ADD-NEW-ORDER*.

Listing 20. *ORD000* - sekcja *VALIDATE-ORD004*. Źródło: opracowanie własne

```
*>*****  
VALIDATE-ORD004 SECTION.  
*>*****  
  IF ORD000-I-ORDER-TYPE OF ORD000-I-ADD-NEW-ORDER  
    = ZEROES OR LOW-VALUE  
  
    DISPLAY "Order type missing (Function ADD-NEW-ORDER)"  
    MOVE 400 TO WS-RETURN-CODE  
    PERFORM END-PROGRAM  
  
  END-IF  
  
  .  
VALIDATE-ORD004-X.  
EXIT.
```

W przypadku wprowadzenia nieprawidłowego typu zamówienia lub jego całkowitego braku, kod wyjścia ustawiany jest jako 400, a program od razu zakańcza wykonywanie. Walidacja wykonywana jest dla każdej dostępnej funkcji.

Po prawidłowym przebiegu procesu weryfikacji danych, program przechodzi do przetworzenia żądania użytkownika. Na listingu 21 pokazana jest funkcja *PROCESS-INPUT*, decydująca o dalszym przebiegu.

Listing 21. ORD000 - sekcja PROCESS-INPUT. Źródło: opracowanie własne

```
*>*****  
PROCESS-INPUT SECTION.  
*>*****  
  EVALUATE TRUE  
    WHEN ORD000-BATCH-PROCESS  
      PERFORM CALL-ORD001  
    WHEN ORD000-GET-ALL  
      PERFORM CALL-ORD002  
    WHEN ORD000-GET-BY-ID  
      PERFORM CALL-ORD003  
    WHEN ORD000-ADD-NEW-ORDER  
      PERFORM CALL-ORD004  
    WHEN ORD000-ADD-TO-ORDER  
      PERFORM CALL-ORD005  
    WHEN ORD000-REMOVE-FROM-ORDER  
      PERFORM CALL-ORD006  
    WHEN OTHER  
      DISPLAY "Unknown function"  
      MOVE 400 TO WS-RETURN-CODE  
      PERFORM END-PROGRAM  
  END-EVALUATE  
  .  
PROCESS-INPUT-X.  
EXIT.
```

W przypadku omawianej funkcji *GET-ALL* program przechodzi do sekcji *CALL-ORD002* widocznej na listingu 22.

Listing 22. ORD000 - sekcja CALL-ORD002. Źródło: opracowanie własne

```
*>*****  
CALL-ORD002 SECTION.  
*>*****  
  PERFORM MAP-ORD002-INPUT  
  CALL "ORD002" USING ORD002-RECORD  
  IF ORD002-OK  
    PERFORM MAP-ORD002-OUTPUT  
    MOVE 0 TO WS-RETURN-CODE  
    PERFORM END-PROGRAM  
  ELSE  
    DISPLAY "ORD002 finished with return code "  
      ORD002-RETURN-CODE  
    MOVE ORD002-RETURN-CODE TO WS-RETURN-CODE  
  END-IF  
  .  
CALL-ORD002-X.  
EXIT.
```

Pierwszym zadaniem wykonywanym przez ten fragment jest zmapowanie argumentów wejściowych do interfejsu komunikacyjnego z programem ORD002 – copybook ORD002AR. Na listingu 23 przedstawiony jest kod odpowiadający za tą konwersję.

Listing 23. ORD000 - sekcja MAP-ORD002-INPUT. Źródło: opracowanie własne

```
*>*****
MAP-ORD002-INPUT SECTION.
*>*****

    SET  ORD002-GET-ALL      TO TRUE
    MOVE ORD000-I-DATE-FROM OF ORD000-I-GET-ALL
                                TO ORD002-I-DATE-FROM
    MOVE ORD000-I-DATE-TO   OF ORD000-I-GET-ALL
                                TO ORD002-I-DATE-TO
    MOVE ORD000-I-ORDER-TYPE OF ORD000-I-GET-ALL
                                TO ORD002-I-ORDER-TYPE
    MOVE ORD000-I-PAGE      OF ORD000-I-GET-ALL
                                TO ORD002-I-PAGE

    .
MAP-ORD002-INPUT-X.
EXIT.
```

Przenoszenie danych między dwoma strukturami copybook jest prostym i dosyć prymitywnym mechanizmem, ponieważ COBOL nie weryfikuje w żaden sposób typów zmiennych i ich kompatybilności. W przypadku wstawienia zmiennej o typie alfanumerycznym i długości 30 (X(30)) do drugiej alfanumerycznej zmiennej, ale o długości 20 znaków (X(20)) kompilator nie zwróci błędu; wartość zostanie po prostu obcięta i stracimy dane.

Po wykonaniu całej konwersji w sekcji *MAP-ORD002-INPUT*, wywoływany jest program ORD002, który odpowiada za logikę biznesową dla funkcji *GET-ALL*. Następnie po jego wykonaniu ORD000 sprawdza kod wyjścia – jeżeli wszystko przebiegło prawidłowo, przechodzi do ostatniej sekcji – *MAP-ORD-002-OUTPUT*, widocznej na listingu 24.

Listing 24. ORD000 - sekcja MAP-ORD002-OUTPUT. Źródło: opracowanie własne

```
*>*****
MAP-ORD002-OUTPUT SECTION.
*>*****

    OPEN OUTPUT FILE-OUT
    PERFORM VARYING WS-IDX FROM 1 BY 1
        UNTIL WS-IDX > ORD002-O-ORDER-COUNT
        MOVE ORD002-O-ORDER-DATA(WS-IDX) TO FILE-OUTPUT
        WRITE FILE-OUTPUT
    END-PERFORM
    CLOSE FILE-OUT
    MOVE 0 TO WS-IDX

    .
MAP-ORD002-OUTPUT-X.
EXIT.
```

Powyższa sekcja zapisuje cały wynik z działania programu ORD002 do pliku podanego jako argument podczas włączania ORD000 z wiersza poleceń. Struktura pliku wynikowego będzie zgodna z copybook’iem ORD002AR – podobnie jak w przypadku wejścia, będzie to zwykły plik tekstowy, w którym długość poszczególnych pól dokładnie odpowiada zadeklarowanym zmiennym w interfejsie.

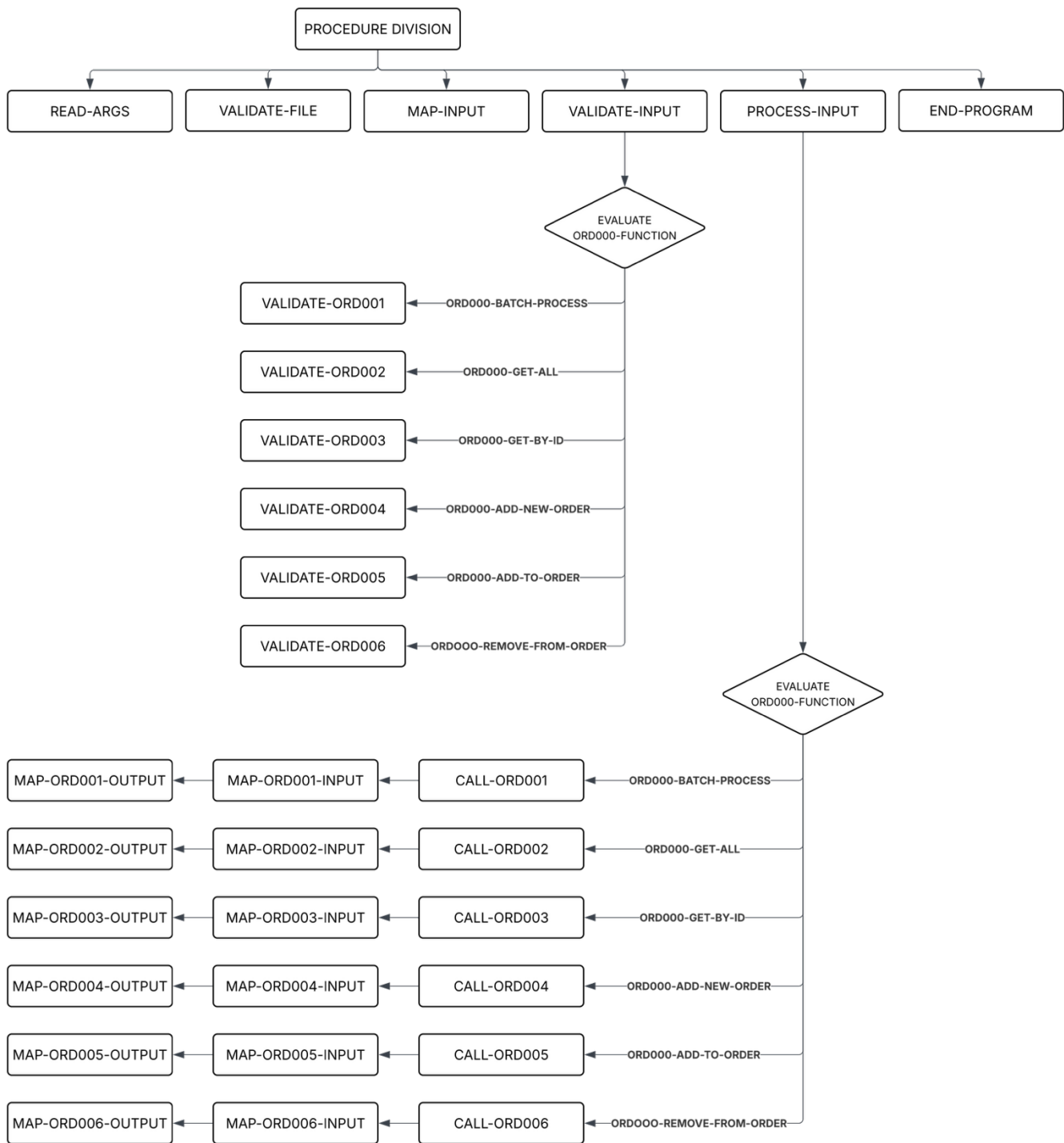
Na listingu 25 przedstawiony jest przykładowy, bezpośredni plik wynikowy po wykonaniu funkcji *GET-ALL* z programu ORD000.

Listing 25. Plik wyjściowy z programu ORD000 po wykonaniu funkcji GET-ALL. Źródło: opracowanie własne

00000027842025-01-29In storeAdam	NovaMart Manchester Dallas
00000027872025-01-17In storeNikolas	NovaMart New York MiEloise
00000027932025-06-10In storeBrynn	NovaMart Chicago LooJennifer
00000027992025-06-03Online Vihaan	
00000028002025-04-07Online Vihaan	
00000028012025-02-10In storeBrandon	NovaMart Birmingham Dior
00000028082025-01-30In storeBentley	NovaMart Los AngelesDior
00000028092025-05-16Online Maria	
00000028102025-01-29In storeAdam	NovaMart Manchester Dallas
00000028132025-01-17In storeNikolas	NovaMart New York MiEloise
00000028192025-06-10In storeBrynn	NovaMart Chicago LooJennifer
00000028252025-06-03Online Vihaan	
00000028262025-04-07Online Vihaan	
00000028272025-02-10In storeBrandon	NovaMart Birmingham Dior
00000028342025-01-30In storeBentley	NovaMart Los AngelesDior
00000028352025-05-16Online Maria	
00000028362025-01-29In storeAdam	NovaMart Manchester Dallas
00000028392025-01-17In storeNikolas	NovaMart New York MiEloise
00000028452025-06-10In storeBrynn	NovaMart Chicago LooJennifer
00000028512025-06-03Online Vihaan	

Wykonanie programu ORD000 w tym momencie zostaje zakończone. Inne funkcje działają w analogiczny sposób - przeprowadzana jest adekwatna walidacja, mapowanie, a następnie wysłane jest żądanie do odpowiedniego podprogramu.

Na diagramie widocznym na rysunku 18 przedstawione są sekcje wywoływane przez wszystkie funkcjonalności dostępne w programie ORD000.



Rysunek 18. Struktura sekcji w programie ORD000. Źródło: opracowanie własne

5.1.4. Logika biznesowa - ORD001 - ORD006

Głównym zadaniem opisanego w poprzednim rozdziale programu ORD000 jest walidacja i mapowanie danych, oraz ostatecznie wywołanie programu, który odpowiada za całą logikę biznesową – czyli w zależności od zadanej funkcji, przechodzi do jednego z programów ORD001 – ORD006.

W przypadku omawianej dla ORD000 funkcji *GET-ALL*, logika biznesowa zawarta w odpowiadającym jej programie ORD002 jest bardzo prosta. Jej głównym zadaniem jest przekazanie danych do programu komunikującego się z bazą danych. Cały program to zaledwie 109 linijek kodu wykonujące bardzo podstawowe zadania, takie jak ponowna walidacja danych, rzutowanie wejścia i wyjścia i przetworzenie kodu błędu.

Mimo prostoty większości programów znajdujących się w tej warstwie, mogą być one bardzo ważnym elementem systemu w języku COBOL. Zakładając dalszy rozwój systemu, pozwalają one na ponowne użycie programów modyfikujących dane w bazie przez inne projekty i systemy, równocześnie zapewniając, że nie będą one naruszać zasad narzuconych na system. Warstwa ta zatem zasadniczo odpowiada głównie za zachowanie integralności danych w bazie.

Jako przykładowy program przyjęty został ORD001 – program ten jest częścią funkcji *BATCH-PROCESS*, która umożliwia zapisanie dużej ilości danych z pliku CSV za pomocą jednego wywołania programu.

Podobnie jak każdy program w języku COBOL, ORD001 również rozpoczyna swoje wykonanie od *PROCEDURE DIVISION*, przedstawionym na listingu 26.

Listing 26. ORD001 - PROCEDURE DIVISION. Źródło: opracowanie własne

```
*>*****
PROCEDURE DIVISION USING ORD001-RECORD.
*>*****

EVALUATE TRUE
WHEN ORD001-BATCH-PROCESS
    PERFORM BATCH-PROCESS-ORDERS
WHEN OTHER
    DISPLAY "Unsupported function"
    MOVE 400 TO WS-RETURN-CODE
END-EVALUATE
.
```

Zmianą w stosunku do poprzednio omawianego programu ORD000 jest klauzula *USING* – mówi ona o fakcie, że przy wywołaniu programu przekazywana jest mu struktura danych zdefiniowana w copybooku w niej zadeklarowanym. W tym przypadku jest to ORD001AR, widoczny na listingu 27.

Listing 27. Copybook ORD001AR. Źródło: opracowanie własne

```
01 :PFX:-RECORD.
03 :PFX:-FUNCTION          PIC 9(01).
08 :PFX:-BATCH-PROCESS    VALUE 01.

03 :PFX:-INPUT.
05 :PFX:-I-FILE-NAME      PIC X(50).
03 :PFX:-OUTPUT.
```

```

05 :PFX:-RETURN-CODE      PIC 9(03).
88 :PFX:-OK                VALUE 200.
88 :PFX:-INVALID-INPUT    VALUE 400.
88 :PFX:-ROW-NOT-FOUND    VALUE 404.
88 :PFX:-UNKNOWN-ERROR    VALUE 500.
88 :PFX:-SQL-ERROR        VALUE 901.

```

Program udostępnia jedynie jedną funkcjonalność – masowe przetwarzanie danych. W przypadku podania jakiegokolwiek innego argumentu, program kończy wykonanie z kodem wyjścia 400. Wprawdzie copybook ORD001AR zawiera tylko jedną zdefiniowaną wartość poziomu 88 dla pola *FUNCTION*, jednak w praktyce można przekazać tam dowolną liczbę. Weryfikacja jest więc koniecznym elementem każdego programu.

Główną i zarazem jedyną funkcją w programie ORD001 jest *BATCH-PROCESS-ORDERS*. Kod tej sekcji jest przedstawiony na listingu 28.

Listing 28. ORD001 - sekcja *BATCH-PROCESS-ORDERS*. Źródło: opracowanie własne

```

*>*****
BATCH-PROCESS-ORDERS SECTION.
*>*****
    MOVE LOW-VALUES TO ORD100-RECORD
    MOVE 500        TO WS-RETURN-CODE

    PERFORM INITIALIZE-VARIABLES
    PERFORM VALIDATE-FILE
    PERFORM PROCESS-ORDERS

    PERFORM END-PROGRAM
.
BATCH-PROCESS-ORDERS-X.
EXIT.

```

Pierwszym krokiem sekcji *BATCH-PROCESS-ORDERS* jest zainicjalizowanie copybook’a ORD100AR, służącego do komunikacji z programem przetwarzającym plik CSV, tak zwanymi *LOW-VALUES* – czyli najmniejszymi możliwymi wartościami, jakie mogą przyjąć pola, oraz wypełnienie zmiennej *WS-RETURN-CODE* kodem 500 odpowiadającym nieznanemu błędowi.

Kolejną sekcją, *INITIALIZE-VARIABLES*, przedstawiona jest na listingu 29.

Listing 29. ORD001 - sekcja *INITIALIZE-VARIABLES*. Źródło: opracowanie własne

```

*>*****
INITIALIZE-VARIABLES SECTION.
*>*****
    MOVE ORD001-I-FILE-NAME TO WS-FILE-IN
.
INITIALIZE-VARIABLES-X.
EXIT.

```

Ponieważ program przyjmuje tylko jedną zmienną – nazwę pliku CSV, w którym znajdują się zamówienia do przetworzenia – przeniesiona jest ona do zmiennej roboczej i sekcja kończy swoje wykonanie.

Następnym krokiem jest walidacja pliku w sekcji *VALIDATE-FILE*, widocznej na listingu 30.

Listing 30. ORD001 - sekcja VALIDATE-FILE. Źródło: opracowanie własne

```
>*****
VALIDATE-FILE SECTION.
>*****

    IF WS-FILE-IN = SPACES OR LOW-VALUE
        DISPLAY "Input file mandatory"
        MOVE 400 TO WS-RETURN-CODE
        PERFORM END-PROGRAM
    END-IF
.
VALIDATE-FILE-X.
EXIT.
```

Niestety przeprowadzana walidacja jest prymitywna – sprawdza jedynie, czy został podany jakikolwiek argument. Nie jest sprawdzane, czy plik istnieje ani czy jego zawartość jest zgodna z oczekiwaną strukturą. Nie ma narzędzi, które umożliwiłyby łatwą, dokładniejszą weryfikację. COBOL został zaprojektowany z założeniem, że plik wejściowy będzie zgodny z oczekiwanym.

Po sprawdzeniu, czy ścieżka została podana, program przechodzi do najważniejszej i zarazem najdłuższej funkcji – *PROCESS-ORDERS*. Przedstawiona została na listingu 31.

Listing 31. ORD001 - sekcja PROCESS-ORDERS. Źródło: opracowanie własne

```
>*****
PROCESS-ORDERS SECTION.
>*****

    PERFORM CALL-ORD100-OPEN
    PERFORM CALL-ORD100-NEXT

    PERFORM UNTIL NOT ORD100-OK
        PERFORM MAP-ORD100-OUTPUT
        IF ORDER1-ORDER-ID = WS-PREV-ORDER-ID
            PERFORM VALIDATE-PRODUCT-ORDER-DATA
            PERFORM INSERT-PRODUCT-ORDER
        ELSE
            PERFORM VALIDATE-ORDER-DATA

        IF WS-VALIDATION-OK
            PERFORM INSERT-NEW-ORDER
            PERFORM VALIDATE-PRODUCT-ORDER-DATA
            PERFORM INSERT-PRODUCT-ORDER
```

```

        ELSE
            DISPLAY "VALIDATION ERROR"
            MOVE 400 TO WS-RETURN-CODE
            PERFORM END-PROGRAM
        END-IF

    END-IF

    PERFORM CALL-ORD100-NEXT
END-PERFORM

PERFORM CALL-ORD100-CLOSE
MOVE 200 TO WS-RETURN-CODE
PERFORM END-PROGRAM

.
PROCESS-ORDERS-X.
EXIT.

```

Cała sekcja *PROCESS-ORDERS* polega na odpowiednio wywołaniu programu ORD100, który przetwarza kolejny wiersz pliku CSV, a następnie sprawdzenie jego zawartości, walidację danych i zweryfikowanie, czy wszystkie potrzebne rekordy istnieją już w bazie danych, i odpowiednio stworzenie nowego zamówienia za pośrednictwem ORD110 lub dodanie nowego produktu do już istniejącego za pomocą ORD150.

Na początku wykonywana jest sekcja *CALL-ORD100-OPEN*, przedstawiona na listingu 32.

Listing 32. ORD001 - sekcja CALL-ORD100-OPEN. Źródło: opracowanie własne

```

*>*****
CALL-ORD100-OPEN SECTION.
*>*****
    MOVE FUNCTION TRIM(WS-FILE-IN) TO ORD100-FILE-IN-NAME
    SET ORD100-OPEN-CSV           TO TRUE

    CALL "ORD100" USING ORD100-RECORD

    IF ORD100-UNKNOWN-ERROR
        DISPLAY "Error while opening input file"
        MOVE 500 TO WS-RETURN-CODE
        PERFORM END-PROGRAM
    END-IF

.
CALL-ORD100-OPEN-X.
EXIT.

```

Ważnym elementem tej sekcji jest fakt, że stosowana w niej jest funkcja *TRIM* na argumencie *WS-FILE-IN*. Zabieg ten jest istotny, ponieważ jak zostało już wspomniane, język COBOL operuje na

stałych długościach zmiennych. Pole *WS-FILE-IN* to pole alfanumeryczne o długości 50. Jeżeli nazwa pliku ma 10 znaków, to pozostałe 40 zostałoby wypełnione spacjami – system próbowałby więc dostać się do błędnej nazwy.

Następnie wywoływany jest program ORD100 z argumentem *ORD100-OPEN-CSV*, który otwiera plik CSV z nazwą podaną jako argument.

Po otwarciu pliku zlecany jest odczyt pierwszego wiersza pliku CSV za pomocą sekcji *ORD100-NEXT*, przedstawionej na listingu 33.

Listing 33. ORD001 - sekcja CALL-ORD100-NEXT. Źródło: opracowanie własne

```
*>*****  
CALL-ORD100-NEXT SECTION.  
*>*****  
MOVE FUNCTION TRIM(WS-FILE-IN) TO ORD100-FILE-IN-NAME  
SET ORD100-NEXT-ROW TO TRUE  
  
CALL "ORD100" USING ORD100-RECORD  
.  
CALL-ORD100-NEXT-X.  
EXIT.
```

Ponownie prawie cała odpowiedzialność w tej sekcji jest przejęta przez program ORD100. Dostarczamy jedynie nazwę pliku i funkcję – w tym wypadku jest to *NEXT-ROW*.

Po wykonaniu tych dwóch sekcji wynikiem jest wiersz z pliku CSV przetworzony na strukturę copybook'a ORDER001, co zdecydowanie ułatwia pracę z zawartymi w niej danymi. Na listingu 34 przedstawione zostały przykładowe linie pliku CSV, natomiast listing 35 to copybook ORDER001, obejmujący wszystkie niezbędne kolumny.

Listing 34. Przykładowy fragment wejściowego pliku CSV. Źródło: opracowanie własne

```
Lp,Order ID,Product,Quantity,Unit,Product type,Price,Order  
date,Order type,Shop ID,Client ID,Employee ID  
1,1,Pasta,10.0,pcs,Food,5,29.01.2025,In store,3.0,24.0,4.0  
2,1,Flour,4.8,kg,Food,4,29.01.2025,In store,3.0,24.0,4.0  
3,2,Webcam,9.0,pcs,Electronics,250,12.08.2023,In store,2.0,,9.0  
4,3,Milk,6.0,pcs,Food,5,04.04.2024,In store,5.0,,5.0  
5,3,Skirt,2.0,pcs,Clothing,140,04.04.2024,In store,5.0,,5.0  
6,3,TV 65 inch,2.0,pcs,Electronics,4500,04.04.2024,In  
store,5.0,,5.0  
7,3,Chocolate,7.0,pcs,Food,7,04.04.2024,In store,5.0,,5.0  
8,3,Sticky Notes,2.0,pcs,Office supplies,8,04.04.2024,In  
store,5.0,,5.0  
9,3,Dress,6.0,pcs,Clothing,220,04.04.2024,In store,5.0,,5.0  
10,3,Lamp,6.0,pcs,Home improvement,120,04.04.2024,In  
store,5.0,,5.0  
11,4,Formal Shoes,2.0,pcs,Clothing,380,17.01.2025,In  
store,4.0,30.0,9.0
```

Listing 35. Copybook ORDER001. Źródło: opracowanie własne

```

08 :PFX:-RECORD.
10 :PFX:-ORDER-ID          PIC 9(10).
10 :PFX:-PRODUCT          PIC X(50).
10 :PFX:-QUANTITY         PIC 9(10)V99.
10 :PFX:-UNIT-NAME        PIC X(10).
10 :PFX:-PRODUCT-TYPE     PIC X(50).
10 :PFX:-ORDER-DATE       PIC X(50).
10 :PFX:-PRICE            PIC 9(10).
10 :PFX:-ORDER-TYPE       PIC X(50).
10 :PFX:-SHOP-ID          PIC 9(10).
10 :PFX:-CLIENT-ID        PIC 9(10).
10 :PFX:-EMPLOYEE-ID      PIC 9(10).

```

Po rzutowaniu wiersza z pliku CSV na strukturę ORDER001 przez program ORD100, ORD001 kopiuje wartości z tego copybook'a do swoich zmiennych roboczych poprzez sekcję *MAP-ORD100-OUTPUT*, widoczną na listingu 36.

Listing 36. ORD001 - sekcja MAP-ORD100-OUTPUT. Źródło: opracowanie własne

```

>*****
MAP-ORD100-OUTPUT SECTION.
>*****
EVALUATE TRUE
  WHEN ORD100-OK
    MOVE ORDER1-ORDER-ID      TO WS-ORDER-ID
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-PRODUCT      TO WS-PRODUCT
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-QUANTITY     TO WS-QUANTITY
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-UNIT-NAME    TO WS-UNIT-NAME
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-PRODUCT-TYPE TO WS-PRODUCT-TYPE
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-ORDER-DATE   TO WS-ORDER-DATE
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-PRICE        TO WS-PRICE
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-ORDER-TYPE   TO WS-ORDER-TYPE
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-SHOP-ID      TO WS-SHOP-ID
                                OF WS-PROCESSED-ORDER
    MOVE ORDER1-CLIENT-ID    TO WS-CLIENT-ID
                                OF WS-PROCESSED-ORDER

```

```

        MOVE ORDER1-EMPLOYEE-ID TO WS-EMPLOYEE-ID
                                OF WS-PROCESSED-ORDER

    WHEN ORD100-INVALID-INPUT
        DISPLAY "ORD100: invalid input"
        MOVE 400 TO WS-RETURN-CODE

    WHEN ORD100-FILE-MISSING
        DISPLAY "ORD100: missing input file"
        MOVE 404 TO WS-RETURN-CODE

    WHEN OTHER
        DISPLAY "ORD100: unknown error"
        MOVE 500 TO WS-RETURN-CODE

END-EVALUATE

.
MAP-ORD100-OUTPUT-X.
EXIT.

```

W przypadku poprawnego zakończenia działania programu ORD100, przeprowadzane jest wyżej wspomniane skopiowanie wartości wynikowych do zmiennych roboczych. Jeśli jednak coś pójdzie nie tak, odpowiada ona również za przekazanie odpowiedniego kodu błędu, zgodnego ze standardem http – 400 w przypadku błędnych argumentów wejściowych, 404 w przypadku nieznalezienia pliku i 500 w przypadku innego, nieznanego błędu.

Gdy dane są już odpowiednio przygotowane, program sprawdza, czy numer poprzednio procesowanego rzędu jest ten sam, co obecny – system zakłada, że elementy zamówienia w pliku następują zawsze tuż po sobie – w przypadku niezgodności, dodawane jest nowe zamówienie, jeżeli ID są takie same, dodawany jest jedynie rekord do już istniejącego zamówienia.

Proces wygląda podobnie w obu przypadkach, jedyną znaczącą różnica jest taka, że w przypadku wstawiania nowego zamówienia do bazy używany jest dodatkowo program ORD110. Omówiony więc zostanie tylko ten scenariusz.

Pierwszym krokiem jest walidacja danych w sekcji *VALIDATE-ORDER-DATA*, przedstawionej na listingu 37.

Listing 37. ORD001 - sekcja VALIDATE-ORDER-DATA. Źródło: opracowanie własne

```

>*****
VALIDATE-ORDER-DATA SECTION.
>*****

    SET WS-VALIDATION-OK TO TRUE

    PERFORM CLEAR-ORDER-TO-INSERT
    PERFORM VALIDATE-ORDER-TYPE
    PERFORM VALIDATE-CLIENT
    PERFORM VALIDATE-SHOP

```

```

PERFORM VALIDATE-EMPLOYEE
PERFORM VALIDATE-ORDER-DATE
.
VALIDATE-ORDER-DATA-X.
EXIT.

```

Pole *WS-VALIDATION-RESULT* służy jako sygnał czy dane są poprawne i może przyjąć dwie wartości: *WS-VALIDATION-OK* i *WS-VALIDATION-NOT-OK*. Ponieważ COBOL jest proceduralny i sekcje nie zwracają żadnej wartości bezpośrednio, zostało one zastosowane w zbliżony sposób – po przebiegu sekcji *VALIDATE-ORDER-DATA* przyjmuje ono odpowiednią wartość. Domyślnie ustawiane jest na *WS-VALIDATION-OK*, a jeśli sekcja napotka problem, jest przestawiane na drugą możliwą wartość.

Druga wywoływana instrukcja to przejście do sekcji *CLEAN-ORDER-TO-INSERT*. Sekcja ta inicjalizuje domyślnymi wartościami obszar roboczy dotyczący zamówienia w celu upewnienia się, że nie zostały w niej dane po poprzednim przejściu pętli.

Następnym krokiem jest zweryfikowanie, czy każdy element wiersza ma swój odpowiednik w bazie danych. W tym celu uruchamiane są kolejno sekcje *VALIDATE-ORDER-TYPE*, *VALIDATE-CLIENT*, *VALIDATE-SHOP* oraz *VALIDATE-EMPLOYEE*. Ponadto *VALIDATE-ORDER-DATE* weryfikuje obecność wymaganej daty zamówienia. Listing 38 przedstawia *VALIDATE-ORDER-TYPE* – reszta sekcji działa w analogiczny sposób, odwołując się do odpowiedniego programu.

Listing 38. ORD001 - sekcja *VALIDATE-ORDER-TYPE*. Źródło: opracowanie własne

```

*>*****
VALIDATE-ORDER-TYPE SECTION.
*>*****

PERFORM MAP-ORD020-INPUT
CALL "ORD020" USING ORD020-RECORD
PERFORM MAP-ORD020-OUTPUT

.
VALIDATE-ORDER-TYPE-X.
EXIT.

```

Dzięki zaprojektowaniu całego systemu zgodnie z zasadą podzielonej odpowiedzialności, sekcje walidacyjne są krótkie i proste. Składają się jedynie z mapowania odpowiednich pól, wywołania programu i następnie przetworzenia wyniku jego wykonania. Na listingu 39 znajduje się sekcja *MAP-ORD020-INPUT*, natomiast listing 40 przedstawia *MAP-ORD020-OUTPUT*.

Listing 39. ORD001 - sekcja *MAP-ORD020-INPUT*. Źródło: opracowanie własne

```

*>*****
MAP-ORD020-INPUT SECTION.
*>*****

MOVE LOW-VALUES TO ORD020-RECORD
SET ORD020-GET-BY-NAME TO TRUE

```

```

MOVE WS-ORDER-TYPE      OF WS-PROCESSED-ORDER
                           TO ORD020-I-ORDER-TYPE-NAME

.
MAP-ORD020-INPUT-X.
EXIT.

```

Listing 40. ORD001 - sekcja MAP-ORD020-OUTPUT. Źródło: opracowanie własne

```

*>*****
MAP-ORD020-OUTPUT SECTION.
*>*****

EVALUATE TRUE
  WHEN ORD020-OK
    MOVE ORD020-O-ORDER-TYPE-ID TO WS-ORDER-TYPE-ID
                                   OF WS-ORDER-TO-INSERT

  WHEN ORD020-INVALID-INPUT
    DISPLAY "ORD020: invalid input"
    SET WS-VALIDATION-NOT-OK TO TRUE
    MOVE 400 TO WS-RETURN-CODE

  WHEN ORD020-ROW-NOT-FOUND
    DISPLAY "ORD020: row not found"
    SET WS-VALIDATION-NOT-OK TO TRUE
    MOVE 404 TO WS-RETURN-CODE

  WHEN ORD020-SQL-ERROR
    DISPLAY "ORD020: SQL error"
    SET WS-VALIDATION-NOT-OK TO TRUE
    MOVE 901 TO WS-RETURN-CODE

  WHEN OTHER
    DISPLAY "ORD020: unknown error"
    SET WS-VALIDATION-NOT-OK TO TRUE
    MOVE 500 TO WS-RETURN-CODE
END-EVALUATE

.
MAP-ORD020-OUTPUT-X.
EXIT.

```

Ponieważ program ORD020 w celu wykonania funkcji *GET-BY-NAME* przyjmuje jedynie jeden argument – nazwę typu zamówienia, sekcja mapująca wejście nie jest obszerna. Natomiast dla wyniku programu niezbędna jest weryfikacja kodu zwrotnego.

W przypadku powodzenia, kod wyjścia będzie równy *ORD020-OK*, więc program przenosi znalezione ID z bazy do odpowiedniej zmiennej i wraca do sekcji *VALIDATE-ORDER-TYPE*. Jeśli jednak wykonanie

ORD020 zakończy się innym kodem, konieczne jest zaznaczenie tego przy pomocy przestawienia flagi *WS-VALIDATION-RESULT* na *WS-VALIDATION-NOT-OK* oraz zmapowanie kodu błędu.

Po prawidłowym przejściu wszystkich sekcji walidujących, następnym działaniem jest umieszczenie zamówienia w bazie danych. W tym celu *PROCESS-ORDERS* wywołuje *INSERT-NEW-ORDER*, przedstawione na listingu 41.

Listing 41. ORD001 - sekcja INSERT-NEW-ORDER. Źródło: opracowanie własne

```
*>*****  
INSERT-NEW-ORDER SECTION.  
*>*****  
PERFORM MAP-ORD110-INPUT  
CALL "ORD110" USING ORD110-RECORD  
  
MOVE ORD110-O-ORDER-ID TO WS-DB-ORDER-ID  
                           OF WS-PROCESSED-ORDER  
MOVE WS-ORDER-ID        OF WS-PROCESSED-ORDER  
                           TO WS-PREV-ORDER-ID  
  
.  
INSERT-NEW-ORDER-X.  
EXIT.
```

Odpowiedzialność za wstawianie rekordów do bazy jest oddelegowana do programu odpowiadającego za modyfikację danej tabeli – w tym wypadku, ponieważ chcemy dodać rekord do *ORDERS*, używany jest program ORD110. Ponownie pierwszym krokiem jest zmapowanie pól wejściowych – w tym celu zastosowana jest funkcja MAP-ORD110-INPUT, przedstawiona na listingu 42.

Listing 42. ORD001 - sekcja MAP-ORD110-INPUT. Źródło: opracowanie własne

```
*>*****  
MAP-ORD110-INPUT SECTION.  
*>*****  
MOVE LOW-VALUES TO ORD110-RECORD  
  
SET ORD110-NEW-ORDER TO TRUE  
MOVE WS-ORDER-DATE   OF WS-ORDER-TO-INSERT  
                       TO ORD110-I-ORDER-DATE  
MOVE WS-ORDER-TYPE-ID OF WS-ORDER-TO-INSERT  
                       TO ORD110-I-ORDER-TYPE-ID  
MOVE WS-SHOP-ID       OF WS-ORDER-TO-INSERT  
                       TO ORD110-I-SHOP-ID  
MOVE WS-CLIENT-ID     OF WS-ORDER-TO-INSERT  
                       TO ORD110-I-CLIENT-ID  
MOVE WS-EMPLOYEE-ID   OF WS-ORDER-TO-INSERT  
                       TO ORD110-I-EMPLOYEE-ID  
  
.  
MAP-ORD110-INPUT-X.  
EXIT.
```

Wszystkie wartości, które używane są jako wejście do ORD110 zostały zweryfikowane lub pozyskane przy wywołaniu odpowiednich programów z warstwy dostępu do danych w omówionej wcześniej sekcji *VALIDATE-ORDER-DATA*. Pewne jest więc, że rekordy istnieją i nie zostanie naruszona integralność danych w bazie.

Po poprawnym zakończeniu wykonania ORD110 ID nowo utworzonego zamówienia przenoszone jest do zmiennej *WS-DB-ORDER-ID* w celu umożliwienia odwołania się do niego podczas wstawiania powiązanych rekordów do tabeli *PRODUCT_ORDER*.

Program ponownie wraca do sekcji *PROCESS-ORDERS*; następną wykonywaną sekcją jest *VALIDATE-PRODUCT-ORDER-DATA*, przedstawiona na listingu 43.

Listing 43. ORD001 - sekcja VALIDATE-PRODUCT-ORDER-DATA. Źródło: opracowanie własne

```
*>*****  
VALIDATE-PRODUCT-ORDER-DATA SECTION.  
*>*****  
SET WS-VALIDATION-OK TO TRUE  
  
PERFORM CLEAR-PRODUCT-ORDER-TO-INSERT  
PERFORM VALIDATE-QUANTITY  
PERFORM VALIDATE-PRODUCT  
PERFORM VALIDATE-ORDER  
  
.  
VALIDATE-PRODUCT-ORDER-DATA-X.  
EXIT.
```

Sposób działania tej sekcji jest analogiczny do *VALIDATE-ORDER-DATA* – przestrzeń w pamięci przeznaczona na zamówiony produkt jest czyszczona, a następnie wszystkie dane jej dotyczące są walidowane przez programy warstwy dostępu do danych w celu zapewnienia istnienia ich w bazie. Do zaznaczenia, czy wszystko jest w porządku, ponownie stosowana jest flaga *WS-VALIDATION-RESULT*.

W celu wstawienia rekordu do bazy wykorzystywana jest sekcja *INSERT-PRODUCT-ORDER*, przedstawiona na listingu 44.

Listing 44. ORD001 - sekcja INSERT-PRODUCT-ORDER. Źródło: opracowanie własne

```
*>*****  
INSERT-PRODUCT-ORDER SECTION.  
*>*****  
PERFORM MAP-ORD150-INPUT  
CALL "ORD150" USING ORD150-RECORD  
  
.  
INSERT-PRODUCT-ORDER-X.  
EXIT.
```

Jedyną zasadniczą różnicą między wstawianiem rekordu do tabeli *ORDERS* a *PRODUCT_ORDER* jest użyty w tym celu program.

Tym sposobem sekcja *PROCESS-ORDERS* przechodzi po kolei po każdej linii pliku CSV i odpowiednio wstawia rekordy do bazy danych. Dane są dokładnie walidowane poprzez zastosowanie

programów z warstwy dostępu do danych, a później aktualizowane przez warstwę modyfikacji danych. Dzięki takiemu podziałowi mimo stosunkowo dużej długości programu ORD001 – 678 linijek – odpowiada on jedynie za odpowiednie mapowanie wejścia i rozsyłanie zapytań po innych częściach systemu.

Program zakańcza wykonanie w sekcji END-PROGRAM widocznej na listingu 45.

Listing 45. ORD001 - sekcja END-PROGRAM. Źródło: opracowanie własne

```
*>*****  
END-PROGRAM SECTION.  
*>*****  
    MOVE WS-RETURN-CODE TO ORD001-RETURN-CODE  
    GOBACK  
    .  
END-PROGRAM-X.  
EXIT.
```

Kod wyjścia uzyskany w toku wykonania programu przenoszony jest do copybook'a służącego do komunikacji z nim, a polecenie *GOBACK* powoduje powrót do programu wywołującego – czyli w każdym przypadku do ORD000.

5.1.5. Wyszukiwanie danych w bazie - ORD010 – ORD070

Każdy program z warstwy logiki biznesowej, a w szczególności omówiony powyżej ORD001, nie komunikuje się bezpośrednio z bazą danych, a wywołuje odpowiednią „nakładkę” na tabelę. W celu zaprezentowania sposobu działania tej warstwy, przedstawiony zostanie ORD010, wybierający dane z *ORDERS*.

Podobnie jak wcześniej omówione programy, ten również rozpoczyna wykonanie od *PROCEDURE DIVISION*, przedstawionym na listingu 46.

Listing 46. ORD010 - PROCEDURE DIVISION. Źródło: opracowanie własne

```
*>*****  
PROCEDURE DIVISION USING ORD010-RECORD.  
*>*****  
    PERFORM INITIALIZE-VARIABLES  
    PERFORM CONNECT-TO-DATABASE  
    EVALUATE TRUE  
        WHEN ORD010-GET-BY-ID  
            PERFORM GET-BY-ID  
        WHEN ORD010-GET-LIST  
            PERFORM GET-LIST  
        WHEN OTHER  
            SET ORD010-INVALID-INPUT TO TRUE  
            PERFORM END-PROGRAM  
    END-EVALUATE  
    .
```

Program ORD010 udostępnia dwie funkcje – jedna z nich to *GET-BY-ID*, która zwraca zamówienie wraz z jego detalami takimi jak powiązane z nim produkty, oraz *GET-LIST*, dostarczająca niewielką ilość informacji o większej ilości zamówień.

Zanim jednak program przejdzie do wykonywania zasadniczych instrukcji biznesowych, dokonywana jest inicjalizacja oraz walidacja danych. Obydwie te czynności dokonywane są w sekcji *INITIALIZE-VARIABLES*, widocznej na listingu 47.

Listing 47. ORD010 - sekcja INITIALIZE-VARIABLES. Źródło: opracowanie własne

```
>*****  
INITIALIZE-VARIABLES SECTION.  
>*****  
  EVALUATE TRUE  
    WHEN ORD010-GET-BY-ID  
      IF ORD010-I-ORDER-ID = ZEROES  
        SET ORD010-INVALID-INPUT TO TRUE  
        PERFORM END-PROGRAM  
      END-IF  
      MOVE ORD010-I-ORDER-ID    TO WK-I-ORDER-ID  
  
    WHEN ORD010-GET-LIST  
      IF ORD010-I-PAGE = ZEROES OR LOW-VALUE  
        MOVE 1 TO WK-I-PAGE  
      ELSE  
        MOVE ORD010-I-PAGE TO WK-I-PAGE  
      END-IF  
  
      COMPUTE WK-OFFSET      = (WK-I-PAGE - 1) * 20  
  
      IF ORD010-I-DATE-FROM = SPACES OR LOW-VALUE  
        MOVE '0001-01-01' TO WK-I-DATE-FROM  
      ELSE  
        MOVE ORD010-I-DATE-FROM TO WK-I-DATE-FROM  
      END-IF  
  
      IF ORD010-I-DATE-TO = SPACES OR LOW-VALUE  
        MOVE '9999-12-31' TO WK-I-DATE-TO  
      ELSE  
        MOVE ORD010-I-DATE-TO TO WK-I-DATE-TO  
      END-IF  
  
      IF ORD010-I-ORDER-TYPE = SPACES OR LOW-VALUE  
        MOVE SPACES TO WK-I-ORDER-TYPE  
      ELSE  
        EVALUATE TRUE  
          WHEN ORD010-I-ONLINE  
            SET WK-I-ONLINE TO TRUE  
            MOVE 1 TO WK-USE-ORDER-TYPE  
          WHEN ORD010-I-IN-STORE  
            SET WK-I-IN-STORE TO TRUE  
            MOVE 1 TO WK-USE-ORDER-TYPE
```

```

                WHEN OTHER
                    MOVE SPACES TO WK-I-ORDER-TYPE
                END-EVALUATE
            END-IF

        END-EVALUATE

        MOVE SPACES TO ORD010-O-RAW-ALL
        MOVE 0 TO WS-IDX
        MOVE 0 TO WS-PROD-IDX
        SET ORD010-UNKNOWN-ERROR TO TRUE
    .
    INITIALIZE-VARIABLES-X.
    EXIT.

```

Mimo faktu, że weryfikacja była dokonywana już na wcześniejszych etapach, warto dokonać ponownego sprawdzenia wejścia – dzięki temu unikamy wysłania nieprawidłowego zapytania do bazy danych. Dodatkowo przysyłany jest bardzo sztywny sposób współdziałania języka COBOL z SQL – jest on statycznie osadzony w kodzie i wymaga dostarczenia wartości dla każdej zmiennej. W związku z tym, dla zapytania zakresowego z datami, niemożliwe jest zastosowanie powszechnego w nowoczesnych technologiach dynamicznego pominięcia *WHERE*. Wartość musi zostać dostarczona, więc w razie braku przekazania jej do programu ORD010 jest ona wypełniana w odpowiedni sposób, tak, aby zwrócić wynik jednakowy do braku zastosowania *WHERE*. W tym wypadku, jeżeli klient nie chce uwzględnić minimalnej daty zamówienia *ORD010-I-DATE-FROM*, uzupełniana jest ona wartością '0001-01-01' – w praktyce więc uwzględnione zostaną wszystkie rekordy w bazie, niezależnie od daty.

Drugą zaletą zastosowania tego podejścia jest zapobiegnięcie konieczności zmiany każdego programu w przypadku zaktualizowania wartości słownikowej w bazie danych. Przykładem w tym programie jest *ORD010-I-ORDER-TYPE*, odpowiadająca kolumnie *Order_type_ID*. Dla programów używających ORD010, widoczna jest zmienna poziomu 88 z czytelnymi nazwami. Wartościami tych zmiennych natomiast są dokładne nazwy z bazy danych. Jeżeli dokonana zostanie jakakolwiek zmiana w tabeli *Order_type* i zostanie dodana nowa wartość lub istniejąca się zmieni, jedynym programem, który będzie trzeba zmodyfikować, jest ORD010.

Po uzupełnieniu wszystkich wymaganych wartości każdy program omawianej warstwy łączy się z bazą za pomocą przedstawionej na listingu 48 sekcji *CONNECT-TO-DATABASE*.

Listing 48. ORD010 - sekcja *CONNECT-TO-DATABASE*. Źródło: opracowanie własne

```

*>*****
CONNECT-TO-DATABASE SECTION.
*>*****

    EXEC SQL
        CONNECT :WS-USER
        IDENTIFIED BY :WS-PASS
        USING :WS-DBNAME
    END-EXEC

    IF SQLCODE NOT = 0
        DISPLAY "ORD010: SQLCODE=" SQLCODE

```

```

        DISPLAY "ORD010: SQLSTATE=" SQLSTATE
        DISPLAY "ORD010: SQLERRMC=" SQLERRMC
        SET ORD010-SQL-ERROR TO TRUE
        PERFORM END-PROGRAM
    END-IF

    EXEC SQL
        SET client_encoding TO 'UTF8'
    END-EXEC

    .
CONNECT-TO-DATABASE-X.
EXIT.

```

Każdorazowo przy uruchomieniu programu podejmuje on próbę połączenia się z bazą danych. W przypadku niepowodzenia wyświetlany jest odpowiedni kod błędu SQL umożliwiający zlokalizowanie źródła problemu. Jeżeli natomiast połączenie z bazą zostanie nawiązane poprawnie, ustawiane jest kodowanie dla tego połączenia i program przechodzi do następnego kroku. W tym wypadku omówiona zostanie funkcja *GET-LIST*, więc następnym krokiem jest analogicznie nazywająca się sekcja, przedstawiona na listingu 49.

Sekcja ta jest zasadniczą częścią programu i wykonuje całą logikę biznesową. W języku COBOL, jeżeli wynikiem zapytania SQL jest więcej niż jeden wiersz, przetwarza się je za pomocą kursora – w przeciwnym wypadku zwrócony zostanie błąd. Powszechnym działaniem zapobiegawczym w przypadku zadeklarowania zapytania punktowego jest ograniczenie pobieranych wyników do 1 za pomocą *FETCH FIRST 1 ROWS ONLY*.

Listing 49. ORD010 - sekcja *GET-LIST*. Źródło: opracowanie własne

```

*>*****
GET-LIST SECTION.
*>*****

    PERFORM DECLARE-ORDERS-CURSOR
    PERFORM OPEN-ORDERS-CURSOR
    PERFORM FETCH-ORDERS
    PERFORM CLOSE-ORDERS-CURSOR

    PERFORM END-PROGRAM
    .
GET-LIST-X.
EXIT.

```

W tym przypadku program ORD010 spodziewa się wiele wyników, więc zastosowany jest cursor - deklarowany w sekcji *DECLARE-ORDERS-CURSOR* widocznej na listingu 50 i otwierany w *OPEN-ORDERS-CURSOR* na listingu 51.

Listing 51. ORD010 - sekcja OPEN-ORDERS-CURSOR. Źródło: opracowanie własne

```
*>*****  
DECLARE-ORDERS-CURSOR SECTION.  
*>*****  
  
EXEC SQL  
    DECLARE ORDERS_CSR CURSOR FOR  
    SELECT O.ID  
           ,O.DATE  
           ,OT.NAME  
           ,PE.NAME  
           ,S.NAME  
           ,PC.NAME  
    FROM orders O  
    LEFT JOIN order_type OT  
        ON O.ORDER_TYPE_ID = OT.ID  
    LEFT JOIN employment_history EH  
        ON O.EMPLOYEE_ID = EH.EMPLOYEE_ID  
    LEFT JOIN person PE  
        ON EH.PERSON_ID = PE.ID  
    LEFT JOIN shop S  
        ON O.SHOP_ID = S.ID  
    LEFT JOIN client CL  
        ON O.CLIENT_ID = CL.ID  
    LEFT JOIN person PC  
        ON CL.PERSON_ID = PC.ID  
    WHERE (O.DATE >= CAST(:WK-I-DATE-FROM AS DATE))  
        AND (O.DATE <= CAST(:WK-I-DATE-TO AS DATE))  
        AND (:WK-USE-ORDER-TYPE = 0  
            OR OT.NAME = TRIM(:WK-I-ORDER-TYPE))  
    ORDER BY O.ID  
    LIMIT 20  
    OFFSET :WK-OFFSET  
END-EXEC  
.  
DECLARE-ORDERS-CURSOR-X.  
EXIT.
```

Listing 50. ORD010 - sekcja DECLARE-ORDERS-CURSOR. Źródło: opracowanie własne

```
*>*****  
OPEN-ORDERS-CURSOR SECTION.  
*>*****  
  
EXEC SQL  
    OPEN ORDERS_CSR  
END-EXEC  
  
IF SQLCODE NOT = 0
```

```

        DISPLAY "ORD010 OPEN ORDER CURSOR SQLCODE=" SQLCODE
        SET ORD010-SQL-ERROR TO TRUE
        PERFORM END-PROGRAM
    END-IF
    .
    OPEN-ORDERS-CURSOR-X.
    EXIT.

```

Najważniejszym elementem deklaracji kursora jest fakt, że zmienne używane w warunkach *WHERE* w przypadku dat są rzutowane, a w przypadku tekstowych – przycinane, ponieważ w przeciwnym wypadku podjęta zostanie próba znalezienia tekstu o długości zgodnej z zadeklarowaną w programie zmienną, łącznie ze spacjami wypełniającymi pustą przestrzeń w pamięci.

Otwarcie kursora wygląda podobnie do nawiązywania połączenia z bazą danych – zawiera tylko jedną instrukcję w bloku *EXEC SQL*, a w przypadku niepowodzenia program kończy wykonanie po wyświetleniu odpowiedniego kodu błędu.

Po poprawnym zadeklarowaniu i otwarciu kursora wykonywana jest sekcja *FETCH-ORDERS* widoczna na listingu 52.

Listing 52. ORD010 - sekcja FETCH-ORDERS. Źródło: opracowanie własne

```

*>*****
    FETCH-ORDERS SECTION.
*>*****
        PERFORM UNTIL SQLCODE NOT = 0 OR WS-IDX >= 20
            EXEC SQL
                FETCH ORDERS_CSR
                INTO :WK-0-ORDER-ID
                    ,:WK-0-ORDER-DATE
                    ,:WK-0-ORDER-TYPE
                    ,:WK-0-EMP-NAME
                    ,:WK-0-SHOP-NAME
                    ,:WK-0-CLIENT-NAME
            END-EXEC

            IF SQLCODE = 0
                PERFORM MAP-CURSOR-OUTPUT

            ELSE
                IF SQLCODE NOT = 100
                    DISPLAY "ORD010 FETCH ERROR: SQLCODE=" SQLCODE
                    SET ORD010-SQL-ERROR TO TRUE
                    PERFORM END-PROGRAM
                END-IF
            END-IF

        END-IF
    END-PERFORM

```

```

IF WS-IDX > 0
    MOVE WS-IDX TO ORD010-0-ORDER-COUNT OF ORD010-0-GET-LIST
    SET ORD010-OK TO TRUE
ELSE
    SET ORD010-ROW-NOT-FOUND TO TRUE
END-IF
.
FETCH-ORDERS-X.
EXIT.

```

Rekordy z tabeli *ORDERS* przetwarzane są w pętli tak długo, jak nie zostanie napotkany żaden błąd SQL lub dopóki zmienna w pamięci *WS-IDX* osiągnie wartość 20. Nie wynika to z żadnego odgórnego limitu, a z decyzji dotyczącej ilości elementów na stronie. Możliwe byłoby również zdefiniowanie ilości rekordów pobranych na raz, jednak wtedy copybook ORD010AR musiałby załokować więcej miejsca w pamięci na kolejne zamówienia poprzez zwiększenie liczby *OCCURS*.

Jeżeli po pobraniu danych przez kursor kod SQL jest równy 0, program mapuje wynik zapytania w sekcji MAP-CURSOR-OUTPUT widocznej na listingu 53.

Listing 53. ORD010 - sekcja MAP-CURSOR-OUTPUT. Źródło: opracowanie własne

```

*>*****
MAP-CURSOR-OUTPUT SECTION.
*>*****

    ADD 1 TO WS-IDX
    MOVE WK-0-ORDER-ID      TO ORD010-0-ORDER-ID
                                OF ORD010-0-GET-LIST (WS-IDX)
    MOVE WK-0-ORDER-DATE    TO ORD010-0-ORDER-DATE
                                OF ORD010-0-GET-LIST (WS-IDX)
    MOVE WK-0-ORDER-TYPE    TO ORD010-0-ORDER-TYPE
                                OF ORD010-0-GET-LIST (WS-IDX)
    MOVE WK-0-EMP-NAME       TO ORD010-0-EMP-NAME
                                OF ORD010-0-GET-LIST (WS-IDX)
    MOVE WK-0-SHOP-NAME      TO ORD010-0-SHOP-NAME
                                OF ORD010-0-GET-LIST (WS-IDX)
    MOVE WK-0-CLIENT-NAME   TO ORD010-0-CLIENT-NAME
                                OF ORD010-0-GET-LIST (WS-IDX)
    .
MAP-CURSOR-OUTPUT-X.
EXIT.

```

Najważniejszym elementem powyższego kodu jest indeksowanie – jak już wcześniej zostało wspomniane copybook'i COBOL mają statycznie przypisaną przestrzeń w pamięci. Kolekcje są wspierane w bardzo prymitywny sposób poprzez wielokrotne powtórzenie alokacji określonych pól. Nie ma możliwości sprawdzenia, ile elementów zostało już w copybooku zapisane – program więc musi dopilnować miejsca, w którym należy zapisać dane.

Po przejściu po wszystkich wierszach w tabeli lub osiągnięciu indeksu 20, wykonanie pętli zakańcza się i program przechodzi do zamknięcia kursora w sekcji *CLOSE-ORDERS-CURSOR*, widocznej na listingu 54.

Listing 54. *ORD010* - sekcja *CLOSE-ORDERS-CURSOR*. Źródło: opracowanie własne

```
*>*****  
CLOSE-ORDERS-CURSOR SECTION.  
*>*****  
EXEC SQL  
    CLOSE ORDERS_CSR  
END-EXEC  
  
IF SQLCODE NOT = 0  
    DISPLAY "ORD010 CLOSE ORDERS CURSOR SQLCODE=" SQLCODE  
    SET ORD010-SQL-ERROR TO TRUE  
    PERFORM END-PROGRAM  
END-IF  
.  
CLOSE-ORDERS-CURSOR-X.  
EXIT.
```

Instrukcje SQL w języku COBOL muszą być zawarte w *EXEC SQL*. Dotyczy to nie tylko bezpośrednich zapytań czy instrukcji, ale też wszystkich zmiennych, które będą wykorzystywane w tych blokach. Wynika to z faktu, że niezależnie czy do kompilacji programu używany jest oficjalny kompilator od IBM czy narzędzie takie jak OpenCOBOL, kod musi przejść przez preprocesor – DB2 Coprocessor w przypadku rozwiązania oficjalnego, Open-COBOL-ESQL dla projektu przygotowanego na potrzeby niniejszej pracy. Zasadniczą różnicą między produktem IBM a narzędziami wykorzystanymi w projekcie jest fakt, że w pierwszym z nich preprocessing jest ukryty jako krok kompilacji i nie trzeba przetwarzać kodu źródłowego dodatkowym poleceniem.

Tym sposobem program zakańcza wykonanie w sekcji *END-PROGRAM*, która jest jednolita ze wszystkimi sekcjami o tej nazwie w projekcie.

5.1.6. Przetwarzanie pliku CSV - ORD100

Aby system mógł udostępnić możliwość masowego przetwarzania zamówień z pliku, możliwe było wykorzystanie dwóch podejść.

Jedno z nich to dostarczenie danych w formacie zgodnym z zdefiniowanym plikiem copybook, który byłby bardzo prosty do odczytania przez dowolny program COBOL, tak długo, jak znałby on jego strukturę. To rozwiązanie jednak nie jest w żaden sposób praktyczne – założeniem zapewnienia takiej funkcjonalności było dostarczenie możliwości łatwej synchronizacji danych między zewnętrznymi projektami, a Mainframe. Założenie, że każdy z tych systemów znałby oraz dostosowywałby się do struktury copybook jest bardzo niepraktyczne oraz wywołuje bardzo wiele problemów w razie konieczności zmodyfikowania, usunięcia lub dodania jakiegokolwiek pola w tej strukturze – takie działanie spowodowałoby, że system nie byłby w stanie prawidłowo odczytać rekordów powstałych przed wprowadzeniem tej zmiany.

Z uwagi na bardzo liczne negatywne aspekty zastosowania rozwiązania pierwszego, w projekcie wykorzystane zostało drugie podejście – rekordem wejściowym jest plik CSV, który następnie musi zostać przetworzony na strukturę zgodną z plikiem copybook.

Język COBOL zapewnia dobre wsparcie, jeśli chodzi o przetwarzanie plików tekstowych; większość operacji przeprowadzanych na Mainframe ma charakter wsadowy i korzysta z datasetów porównywalnych

do plików znanych z powszechnych systemów operacyjnych. Jednakże przetwarzanie ich zawartości między popularnymi formatami takimi jak CSV czy JSON wygląda zdecydowanie inaczej niż podobna operacja we współczesnych językach programowania. Jeśli chodzi o format JSON, istnieją polecenia *JSON PARSE* oraz *JSON GENERATE*, które zdecydowanie ułatwiają pracę z tym formatem, jednak nie ma publicznie dostępnego ekwiwalentu dla plików CSV – wymaga to ręcznej implementacji.

Ponieważ w projekcie formatem, który ma być obsługiwany jest plik o formacie CSV, niezbędne było napisanie modułu umożliwiającego szybkie przetłumaczenie go na strukturę copybook.

To podejście to powszechna praktyka - wiele ograniczeń języka COBOL zostaje uzupełnione poprzez napisanie modułów takich jak ORD100, dostępnych w obrębie całego środowiska Mainframe w danej firmie. Ponieważ może skorzystać z nich dowolny program, pełnią one funkcję bibliotek, które uzupełniają ten język o wiele niezbędnych możliwości – również tych specyficznych dla potrzeb biznesowych.

Program ORD100 również rozpoczyna swoje wykonanie w sekcji *PROCEDURE DIVISION*, widocznej na listingu 55.

Listing 55. ORD100 - PROCEDURE DIVISION. Źródło: opracowanie własne

```
*>*****
PROCEDURE DIVISION USING ORD100-RECORD.
*>*****

PERFORM INITIALIZE-VARIABLES

EVALUATE TRUE
    WHEN ORD100-OPEN-CSV
        PERFORM OPEN-CSV
    WHEN ORD100-NEXT-ROW
        PERFORM GET-NEXT-ROW
    WHEN ORD100-CLOSE-CSV
        PERFORM CLOSE-CSV
    WHEN OTHER
        SET ORD100-INVALID-INPUT TO TRUE
        PERFORM END-PROGRAM
END-EVALUATE
.
```

Program ORD100 realizuje swoją funkcjonalność w sposób inny niż poprzednio omawiane programy, które były wywoływane tylko raz i zakańczyły wykonanie dostarczając zadany rezultat. W tym wypadku dostarczane jest kilka funkcji – jednokrotnie wykorzystywana funkcja *OPEN-CSV*, otwierająca zadany plik, wielokrotnie wywoływana *GET-NEXT-ROW*, która przetwarza zawartość pliku po jednym rzędzie oraz *CLOSE-CSV*, która jest wykorzystywana na końcu w celu zamknięcia przetworzonego pliku.

Wykorzystanie takiego podejścia ma duży minus – jeżeli dwa programy równocześnie korzystałyby z ORD100, stan otwartego plik zostałby nadpisany przez ten, który został zażądany przez drugi program, co mogłoby prowadzić do błędów. Jednakże został on zaprojektowany z myślą o wsadowych programach, które odpalane są w oddzielnych przestrzeniach adresowych [52] – dzięki temu unikany jest problem współdzielonej pamięci w programach online, wywoływanych na żądanie.

Zanim jednak program przejdzie do wykonywania którejkolwiek z zadanych funkcji, przeprowadza inicjalizację oraz walidację wejścia w sekcji INITIALIZE-VARIABLES, widocznej na listingu 56.

Listing 56. ORD100 - sekcja INITIALIZE-VARIABLES. Źródło: opracowanie własne

```
*>*****
INITIALIZE-VARIABLES SECTION.
*>*****

EVALUATE TRUE
  WHEN ORD100-OPEN-CSV
    IF ORD100-FILE-IN-NAME = SPACES OR LOW-VALUE
      SET ORD100-FILE-MISSING TO TRUE
      PERFORM END-PROGRAM
    END-IF
    IF WS-FILE-OPEN-Y
      SET ORD100-INVALID-INPUT TO TRUE
      PERFORM END-PROGRAM
    END-IF
    WHEN ORD100-NEXT-ROW
      IF WS-FILE-OPEN-N OR WS-HEADER-DISCARDED-N
        SET ORD100-INVALID-INPUT TO TRUE
        PERFORM END-PROGRAM
      END-IF
    WHEN ORD100-CLOSE-CSV
      IF WS-FILE-OPEN-N
        SET ORD100-INVALID-INPUT TO TRUE
        PERFORM END-PROGRAM
      END-IF
    END-EVALUATE

    MOVE ORD100-FILE-IN-NAME TO WS-FILE-IN
    SET ORD100-N-EOF          TO TRUE
    SET ORD100-UNKNOWN-ERROR TO TRUE
  .
INITIALIZE-VARIABLES-X.
EXIT.
```

Mechanizm weryfikacji jest podobny do wcześniej opisywanych programów – w zależności od zadanej funkcji, sprawdzana jest obecność wymaganych pól oraz w razie jakichkolwiek nieścisłości ustawiany jest odpowiedni kod błędu i wykonanie zakańcza się. Aspektem unikatowym dla programu ORD100 jest sprawdzanie zmiennych z *WORKING-STORAGE* w sekcji walidacji – wszystkie uprzednio omawiane programy sprawdzały jedynie stan pól dostarczonych poprzez copybook jako argument wejściowy. W tym wypadku ważne jest, żeby program znał stan przetwarzanego pliku – wykorzystany został fakt, że przestrzeń adresowa programu jest zachowywana między kolejnymi wywołaniami – tak długo więc jak program pozostaje w pamięci i uruchamiany jest w jednej przestrzeni adresowej, zmienne z pamięci programu będą miały poprawne wartości.

Program wykona operację otwarcia pliku pod warunkiem, że zostanie dostarczona jego nazwa i żaden inny plik nie jest otwarty, natomiast przeczytania linii i zamknięcia pliku, jeśli plik jest otwarty.

Operacja otwarcia pliku jest dosyć prosta i wykonywana w sekcji *OPEN-CSV*, zaprezentowanej na listingu 57.

Listing 57. ORD100 - sekcja OPEN-CSV. Źródło: opracowanie własne

```
*>*****  
OPEN-CSV SECTION.  
*>*****  
    OPEN INPUT CSV-IN  
    SET WS-FILE-OPEN-Y TO TRUE  
  
    PERFORM DISCARD-HEADER-LINE  
  
    SET ORD100-OK TO TRUE  
    PERFORM END-PROGRAM  
.  
OPEN-CSV-X.  
EXIT.
```

Kluczowym fragmentem kodu dla tej sekcji jest jej pierwsza instrukcja – OPEN INPUT. Tym sposobem plik jest już otwarty i można odczytać jego zawartość. Dodatkowo ustawiana jest flaga zaznaczająca, czy operacja otwarcia została już dokonana – jest to niezbędna wartość do podjęcia jakichkolwiek dalszych czynności przez program.

Ostatnią operacją niniejszej sekcji przed ustawieniem poprawnego kodu błędu i wrócenia do programu wywołującego ORD100 jest sekcja *DISCARD-HEADER-LINE* widoczna na listingu 58.

Listing 58. ORD100 - sekcja DISCARD-HEADER-LINE. Źródło: opracowanie własne

```
*>*****  
DISCARD-HEADER-LINE SECTION.  
*>*****  
    PERFORM READ-CSV-LINE  
  
    SET WS-HEADER-DISCARDED-Y      TO TRUE  
  
    IF WS-EOF  
        SET ORD100-EOF              TO TRUE  
        SET ORD100-FILE-EMPTY      TO TRUE  
        PERFORM END-PROGRAM  
    ELSE  
        SET ORD100-N-EOF            TO TRUE  
    END-IF  
.  
DISCARD-HEADER-LINE-X.  
EXIT.
```

Podstawowym założeniem przetwarzania pliku CSV przez program ORD100 jest fakt, że posiada on wiersz nagłówka. W omawianej sekcji dokonywany jest odczyt pierwszej linijki z pliku, ignorując ją

i oznaczając go jako pominięty. Jeżeli natomiast sekcja *READ-CSV-LINE*, widoczna na listingu 59 zakończy swój przebieg ustawiając flagę *WS-EOF* na wartość prawdziwą, program kończy wykonanie z błędem *FILE-EMPTY*, oznaczającym brak nagłówka i jakiegokolwiek zawartości pliku.

Listing 59. ORD100 - sekcja READ-CSV-LINE. Źródło: opracowanie własne

```
*>*****
READ-CSV-LINE SECTION.
*>*****
    READ CSV-IN
      AT END
        SET WS-EOF TO TRUE
    END-READ
    .
READ-CSV-LINE-X.
EXIT.
```

Sekcja *READ-CSV-LINE* składa się z jednej linijki – przeczytania kolejnego fragmentu pliku. O tym, jak wygląda definicja fragmentu decyduje zdefiniowana w *FILE-CONTROL* organizacja pliku – w tym wypadku zastosowana została opcja *LINE SEQUENTIAL*, co w praktyce przekłada się na odczyt linijka po linijce.

Tym samym plik został otwarty i przy następnym wywołaniu może zostać zażądana funkcja *NEXT-ROW*, rozpoczynającą się sekcją *GET-NEXT-ROW* widoczną na listingu 60.

Listing 60. ORD100 - sekcja GET-NEXT-ROW. Źródło: opracowanie własne

```
*>*****
GET-NEXT-ROW SECTION.
*>*****
    PERFORM READ-CSV-LINE

    IF WS-EOF
      SET ORD100-EOF TO TRUE
      SET ORD100-FILE-EMPTY TO TRUE
      PERFORM END-PROGRAM
    END-IF

    PERFORM PARSE-CSV-LINE
    PERFORM BUILD-ORDERS-REC

    MOVE ORDER1-RECORD TO ORD100-ORDER-ROW
    SET ORD100-OK TO TRUE
    PERFORM END-PROGRAM
    .
GET-NEXT-ROW-X.
EXIT.
```

Pierwszym krokiem jest wczytanie kolejnej linii pliku za pomocą wcześniej przedstawionej funkcji *READ-CSV-LINE*. Następnie sprawdzane jest, czy dane się nie skończyły – w przypadku napotkania końca pliku program zakańcza wykonanie. Jeśli jednak dane zostały poprawnie odczytane program przechodzi do sekcji *PARSE-CSV-LINE* przedstawionej na listingu 61.

Listing 61. ORD100 - sekcja PARSE-CSV-LINE. Źródło: opracowanie własne

```
*>*****
PARSE-CSV-LINE SECTION.
*>*****

  MOVE SPACES TO WS-ORDER
  UNSTRING CSV-LINE
    DELIMITED BY ","
    INTO WS-LP
      WS-ORDER-ID
      WS-PRODUCT
      WS-QUANTITY
      WS-UNIT
      WS-PRODUCT-TYPE
      WS-PRICE
      WS-ORDER-DATE
      WS-ORDER-TYPE
      WS-SHOP-ID
      WS-CLIENT-ID
      WS-EMPLOYEE-ID

  END-UNSTRING

  MOVE FUNCTION TRIM(WS-ORDER-ID)      TO WS-ORDER-ID
  MOVE FUNCTION TRIM(WS-PRODUCT)       TO WS-PRODUCT
  MOVE FUNCTION TRIM(WS-QUANTITY)       TO WS-QUANTITY
  MOVE FUNCTION TRIM(WS-UNIT)           TO WS-UNIT
  MOVE FUNCTION TRIM(WS-PRODUCT-TYPE)   TO WS-PRODUCT-TYPE
  MOVE FUNCTION TRIM(WS-PRICE)          TO WS-PRICE
  MOVE FUNCTION TRIM(WS-ORDER-DATE)     TO WS-ORDER-DATE
  MOVE FUNCTION TRIM(WS-ORDER-TYPE)     TO WS-ORDER-TYPE
  MOVE FUNCTION TRIM(WS-SHOP-ID)        TO WS-SHOP-ID
  MOVE FUNCTION TRIM(WS-CLIENT-ID)      TO WS-CLIENT-ID
  MOVE FUNCTION TRIM(WS-EMPLOYEE-ID)    TO WS-EMPLOYEE-ID

  .
  PARSE-CSV-LINE-X.
  EXIT.
```

Przetwarzanie linii jest dość proste i zgodne z filozofią języka COBOL jako docelowo prostego do czytania i zbliżonego do języka angielskiego. Ciąg znaków dzielony jest według zdefiniowanego delimitera, a wynikowe fragmenty umieszczane są po kolei w odpowiednich zmiennych. Następnie każda wartość jest przycinana za pomocą *TRIM* w celu usunięcia nadmiernych spacji. Rząd w odpowiedniej postaci jest już zapisany w pamięci programu; pozostaje więc tylko przenieść go do komunikacyjnego copybook'a w celu

przekazania informacji do programu wywołującego ORD100. Proces ten wykonywany jest w sekcji *BUILD-ORDERS-REC* przedstawionej na listingu 62.

Listing 62. ORD100 - sekcja BUILD-ORDERS-REC. Źródło: opracowanie własne

```
*>*****
BUILD-ORDERS-REC SECTION.
*>*****

  MOVE SPACES TO ORDER1-RECORD
  MOVE ZEROES TO ORDER1-ORDER-ID
                ORDER1-QUANTITY
                ORDER1-PRICE
                ORDER1-SHOP-ID
                ORDER1-CLIENT-ID
                ORDER1-EMPLOYEE-ID

  MOVE WS-PRODUCT      TO ORDER1-PRODUCT
  MOVE WS-UNIT          TO ORDER1-UNIT-NAME
  MOVE WS-PRODUCT-TYPE TO ORDER1-PRODUCT-TYPE
  MOVE WS-ORDER-DATE   TO ORDER1-ORDER-DATE
  MOVE WS-ORDER-TYPE   TO ORDER1-ORDER-TYPE

  MOVE WS-ORDER-ID      TO WS-NUM-IN
  PERFORM CONVERT-TO-INTEGER
  MOVE WS-NUM-OUT       TO ORDER1-ORDER-ID

  MOVE WS-PRICE         TO WS-NUM-IN
  PERFORM CONVERT-TO-INTEGER
  MOVE WS-NUM-OUT       TO ORDER1-PRICE

  MOVE WS-SHOP-ID       TO WS-NUM-IN
  PERFORM CONVERT-TO-INTEGER
  MOVE WS-NUM-OUT       TO ORDER1-SHOP-ID

  MOVE WS-CLIENT-ID    TO WS-NUM-IN
  PERFORM CONVERT-TO-INTEGER
  MOVE WS-NUM-OUT       TO ORDER1-CLIENT-ID

  MOVE WS-EMPLOYEE-ID   TO WS-NUM-IN
  PERFORM CONVERT-TO-INTEGER
  MOVE WS-NUM-OUT       TO ORDER1-EMPLOYEE-ID

  MOVE WS-QUANTITY      TO WS-NUM-IN
  PERFORM CONVERT-TO-FLOATING
  MOVE WS-DEC-OUT       TO ORDER1-QUANTITY
  .
BUILD-ORDERS-REC-X.
EXIT.
```

Podobnie do innych języków programowania, operacje na plikach tekstowych w COBOL również dostarczają zmienną rodzaju alfanumerycznego jako wynik odczytu. Wiele pól z rekordu w CSV jest jednak liczbowa, więc w celu umożliwienia poprawnego przetwarzania danych niezbędna jest ich konwersja ze zmiennej *X* na zmienną *9*. W przypadku liczby zmiennoprzecinkowej dokonywana jest ona w sekcji *CONVERT-TO-FLOATING*, natomiast jeżeli wejściem jest liczba całkowita uruchamiana jest sekcja *CONVERT-TO-INTEGER*. W celu przedstawienia mechanizmu działania takiej zmiany rodzaju, zaprezentowana zostanie sekcja konwertująca wartość tekstową na liczbę całkowitą w sekcji *CONVERT-TO-INTEGER* na listingu 63.

Listing 63. ORD100 - sekcja CONVERT-TO-INTEGER. Źródło: opracowanie własne

```
*>*****
CONVERT-TO-INTEGER SECTION.
*>*****

    MOVE 0 TO WS-NUM-OUT
    MOVE FUNCTION TRIM(WS-NUM-IN) TO WS-NUM-TRIM

    IF WS-NUM-TRIM = SPACES
        CONTINUE
    ELSE
        INSPECT WS-NUM-TRIM
            TALLYING WS-DOT-COUNT FOR ALL "."

        IF WS-DOT-COUNT > 0
            MOVE 0 TO WS-DOTPOS
            INSPECT WS-NUM-TRIM TALLYING WS-DOTPOS
                FOR CHARACTERS BEFORE INITIAL "."
            IF WS-DOTPOS > 0
                MOVE WS-NUM-TRIM(1:WS-DOTPOS) TO WS-NUM-TRIM
            ELSE
                MOVE SPACES TO WS-NUM-TRIM
            END-IF
        END-IF

        IF WS-NUM-TRIM = SPACES
            CONTINUE
        ELSE
            MOVE FUNCTION NUMVAL(WS-NUM-TRIM) TO WS-NUM-OUT
        END-IF

    END-IF

    .
CONVERT-TO-INTEGER-X.
EXIT.
```

Konwersja jest dosyć skomplikowana, ponieważ w przypadku przeniesienia zmiennej o wartości 5.0 do pola zdefiniowanego jako *PIC 9(02)* wynikiem będzie błąd. Należy więc upewnić się, że nawet jeżeli liczba jest całkowita, to nie ma ona żadnych liczb po przecinku – również zer. Sekcja sprawdza więc czy

dane istnieją, jeżeli tak, zlicza ilość zawartych w nich kropek. W przypadku braku separatorów i istnienia wartości używana stosowana jest funkcja *NUMVAL* w celu konwersji tekstu na liczbę. Jeżeli jednak separator istnieje, zliczana jest ilość znaków przed kropką a następnie do „czystej”, przyciętej zmiennej przenoszona jest jedynie część całkowita. Wtedy również wywoływana jest funkcja *NUMVAL*, podobnie jak w przypadku zupełnego braku separatora.

Po takiej konwersji danych są one już przygotowane do przeniesienia ich do pól odpowiadających wartościom w copybooku i przekazania ich do programu wywołującego ORD100. Po przetworzeniu wszystkich rzędów pliku pozostało już jedynie go zamknąć przy użyciu funkcji *CLOSE-CSV*, wywołującej sekcję o tej samej nazwie, przedstawionej na listingu 64.

Listing 64. ORD100 - sekcja *CLOSE-CSV*. Źródło: opracowanie własne

```
*>*****
CLOSE-CSV SECTION.
*>*****
CLOSE CSV-IN

SET WS-FILE-OPEN-N      TO TRUE
SET WS-N-EOF            TO TRUE
SET WS-HEADER-DISCARDED-N TO TRUE

SET ORD100-OK TO TRUE
PERFORM END-PROGRAM
.
CLOSE-CSV-X.
EXIT.
```

Uruchomienie funkcji *CLOSE-CSV* jest bardzo ważne z uwagi na wcześniej wspomniane zachowanie wartości zmiennych z *WORKING-STORAGE* między wywołaniami. Przed zakończeniem wykonania należy je więc zresetować do stanu początkowego. Po tym działaniu proces przetwarzania pliku można uznać za zakończony.

Przesunięcie odpowiedzialności za poprawne zamknięcie pliku i zresetowanie zmiennych roboczych na program wywołujący wiąże się z ryzykiem, że programista korzystający z tego modułu nie zrobi tego poprawnie i tym samym spowoduje błędy w innych, niepowiązanych funkcjonalnościach. Z uwagi na ten fakt nie powinno stosować się takiego rozwiązania w środowisku innym niż wsadowe – w takim wypadku zmienne dotyczące stanu pliku oraz wszelkie odnośniki należałoby umieścić w pliku copybook zamiast w pamięci programu.

5.1.7. Modyfikacja danych w bazie - ORD110, ORD150

Ostatnią kategorią programów, które zostały napisane na potrzebę zaimplementowania systemu zarządzania zamówieniami są programy modyfikujące dane w bazie. Zgodnie z założeniami projektu jedynie ta warstwa może wywoływać polecenia *UPDATE*, *INSERT* czy *DELETE*. Jest to ograniczenie wynikające jedynie z zaproponowanej architektury projektu, w praktyce każdy program może odwoływać się oraz modyfikować tabele – jednak rozdzielenie odpowiedzialności między różne warstwy zdecydowanie ułatwia i systematyzuje późniejszy rozwój całego projektu.

W celu przedstawienia mechanizmu działania programów w tej warstwie omówiony zostanie ORD110 – program służący do modyfikowania danych w tabeli *ORDERS*.

Ponownie pierwszym krokiem w wykonaniu omawianego programu jest PROCEDURE DIVISION, przedstawione na listingu 65.

Listing 65. ORD110 - PROCEDURE DIVISION. Źródło: opracowanie własne

```
*>*****
PROCEDURE DIVISION USING ORD110-RECORD.
*>*****

PERFORM INITIALIZE-VARIABLES
PERFORM CONNECT-TO-DATABASE

EVALUATE TRUE
    WHEN ORD110-NEW-ORDER
        PERFORM ADD-NEW-ORDER
    WHEN OTHER
        SET ORD110-INVALID-INPUT TO TRUE
        PERFORM END-PROGRAM
END-EVALUATE
.
```

Struktura programu wygląda bardzo podobnie do warstwy odczytu danych – rozpoczyna się inicjalizacją, następnie podejmowana jest próba połączenia się z bazą i w przypadku powodzenia przechodzi on do zasadniczej funkcjonalności biznesowej.

W sekcji *INITIALIZE-VARIABLES* sprawdzana jest obecność wymaganych zmiennych – są przenoszone do odpowiednich zmiennych w pamięci programu i następnie weryfikowane, natomiast *CONNECT-TO-DATABASE* wygląda analogicznie do sekcji omawianej już wcześniej i widocznej na listingu 43, więc zostaną one pominięte.

Sekcja *ADD-NEW-ORDER*, zaprezentowana na listingu 66, odpowiedzialna jest za dodanie nowego rekordu do tabeli *ORDERS*.

Listing 66. ORD110 - sekcja ADD-NEW-ORDER. Źródło: opracowanie własne

```
*>*****
ADD-NEW-ORDER SECTION.
*>*****

PERFORM INSERT-NEW-ORDER
PERFORM MAP-OUTPUT
.
ADD-NEW-ORDER-X.
EXIT.
```

Ponieważ dane są wielokrotnie walidowane – zarówno przez ORD110 w sekcji *INITIALIZE-VARIABLES*, jak i programy wywoływane wcześniej w przebiegu funkcji – w tym te z warstwy logiki biznesowej, są one przystosowane do umieszczenia w bazie danych. Funkcja więc od razu przechodzi do *INSERT-NEW-ORDER* widocznej na listingu 67.

```

*>*****
INSERT-NEW-ORDER SECTION.
*>*****

EXEC SQL
    INSERT INTO orders (
        "date"
        ,ORDER_TYPE_ID
        ,EMPLOYEE_ID
        ,SHOP_ID
        ,CLIENT_ID
    ) VALUES (
        to_date(RTRIM(:WK-I-ORDER-DATE), 'DD.MM.YYYY')
        ,CAST(:WK-I-ORDER-TYPE-ID AS INTEGER)
        ,NULLIF(CAST(:WK-I-EMPLOYEE-ID AS INTEGER), 0)
        ,NULLIF(CAST(:WK-I-SHOP-ID AS INTEGER), 0)
        ,NULLIF(CAST(:WK-I-CLIENT-ID AS INTEGER), 0)
    )
END-EXEC

EXEC SQL
    SELECT
        currval(pg_get_serial_sequence('orders','id'))
    INTO :WK-O-ORDER-ID
END-EXEC
.
SELECT-BY-ID-X.
EXIT.

```

W powyższej sekcji wyraźnie widoczne są naleciałości wynikające z wybranego silnika bazy danych – PostgreSQL. Ponieważ COBOL obsługuje zapytania przede wszystkim w formie embedded, ewentualna zmiana silnika wymagałaby zmian programowych. W praktyce jednak rzadko jest to problem, ponieważ zdecydowana większość programów w tym języku funkcjonuje w środowisku Mainframe – co zazwyczaj wiąże się również z korzystaniem z silnika Db2 zapewnionego przez IBM.

Sekcja zawiera dwa bloki *EXEC SQL*, pierwszy z nich to faktyczne umieszczenie danych w tabeli *ORDERS* wraz z odpowiednim dostosowaniem pól. Szczególnie istotny jest mechanizm zastosowany w przypadku niewymaganych kluczy obcych – *EMPLOYEE_ID*, *SHOP_ID* oraz *CLIENT_ID*. COBOL nie obsługuje domyślnie wartości *NULL*, więc wartość 0 jest konwertowana na jej całkowity brak. Dzięki temu nienaruszane są więzy integralności. Pola tekstowe muszą zostać poddane przycięciu – podobnie jak we wcześniej omawianych programach, jeżeli funkcja *TRIM* nie zostałaby zastosowana, w bazie danych umieszczona zostałaby wartość wypełniona spacjami od końca tekstu do długości zmiennej zdefiniowanej w *WORKING-STORAGE*.

Drugi blok pobiera obecną wartość sekwencji, aby mogła ją następnie zwrócić do programu wywołującego.

Tym samym program przechodzi do ostatniej sekcji – MAP-OUTPUT, widocznej na listingu 68.

```
*>*****
MAP-OUTPUT SECTION.
*>*****

    IF SQLCODE = 0
        MOVE WK-0-ORDER-ID      TO ORD110-0-ORDER-ID
        SET  ORD110-OK           TO TRUE

        PERFORM COMMIT-SQL
        PERFORM END-PROGRAM
    ELSE
        MOVE SQLCODE TO WS-SQLCODE
        STRING
            "ORD110 SQLCODE=" WS-SQLCODE
            " SQLSTATE=" SQLSTATE
            " SQLERRMC=" SQLERRMC
        INTO ORD110-MESSAGE
        END-STRING

        SET  ORD110-SQL-ERROR     TO TRUE
        PERFORM ROLLBACK-SQL
        PERFORM END-PROGRAM
    END-IF
.
MAP-OUTPUT-X.
EXIT.
```

Jeżeli wykonanie wcześniejszych instrukcji SQL przebiegło prawidłowo i kod błędu to 0, program dokonuje operacji *COMMIT*. W przeciwnym wypadku informacje niezbędne do zlokalizowania problemu wyświetlane są w terminalu, dokonywany jest *ROLLBACK* a ORD110 zakańcza wykonanie z odpowiednim kodem.

We współczesnych językach programowania błędy raportowane są zazwyczaj za pomocą logów, które później można umieścić w systemach takich jak Elastic w celu łatwego ich przeszukiwania. Jednakże integracja takich usług na Mainframe jest zdecydowanie trudniejsza – w celu logowania danych w środowisku online stosowany jest zwykle CICS log, który następnie można przejrzeć na przykład w SDSF. Proces ten jest zdecydowanie bardziej zbliżony do oglądania bezpośredniego outputu z kontenerów aplikacji niż do bardziej przyjaznej pracy z Kibaną.

5.1.8. Kompilacja projektu

Do skompilowania projektu niezbędny jest system operacyjny oparty na UNIX oraz następujące biblioteki:

- GnuCOBOL 2.3.0 [53],
- Open-COBOL-ESQL 1.4.0 [54],
- GCC 15.2 [55].

Po spełnieniu powyższych wymagań, kompilacja przebiega w ściśle określony sposób.

W celu przygotowania środowiska pod kompilację programów, w wierszu poleceń otwartym w folderze, w którym znajduje się kod źródłowy projektu, należy wkleić polecenia zawarte na listingu 69.

Listing 69. Polecenia przygotowujące środowisko do kompilacji. Źródło: opracowanie własne

```
export PGHOST=[Adres IP serwera bazy danych]
export PGPORT=[Port serwera bazy danych]
export PGCONNECT_TIMEOUT=5
export OCESQL_CPY=/opt/homebrew/share/open-cobol-esql/copy
export COB_LIBRARY_PATH=$PWD/bin
```

Następnym krokiem jest dokonanie wstępnego przetworzenia programów wchodzących w interakcję z bazą danych za pomocą Open-COBOL-ESQL, zgodnie z poleceniami z listingu 70.

Listing 70. Polecenia do przetworzenia wstępnego programów komunikujących się z bazą danych. Źródło: opracowanie własne

```
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD010.ecb
preprocessed/ORD010.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD020.ecb
preprocessed/ORD020.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD030.ecb
preprocessed/ORD030.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD040.ecb
preprocessed/ORD040.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD050.ecb
preprocessed/ORD050.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD060.ecb
preprocessed/ORD060.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD070.ecb
preprocessed/ORD070.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD110.ecb
preprocessed/ORD110.cob
ocesql --inc="$OCESQL_CPY" --inc="$APP_CPY" ORD150.ecb
preprocessed/ORD150.cob
```

Po poprawnym przebiegu wstępnego przetworzenia, należy skompilować utworzone w poprzednim kroku moduły poprzez uruchomienie poleceń zawartych na listingu 71.

Listing 71. Polecenia do kompilacji programów komunikujących się z bazą danych. Źródło: opracowanie własne

```
cobc -m -o bin/ORD010 preprocessed/ORD010.cob -I cpy -I "$OCESQL_CPY" -
-L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD020 preprocessed/ORD020.cob -I cpy -I "$OCESQL_CPY" -
-L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD030 preprocessed/ORD030.cob -I cpy -I "$OCESQL_CPY" -
-L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD040 preprocessed/ORD040.cob -I cpy -I "$OCESQL_CPY" -
-L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD050 preprocessed/ORD050.cob -I cpy -I "$OCESQL_CPY" -
-L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
```

```
cobc -m -o bin/ORD060 preprocessed/ORD060.cob -I cpy -I "$OCESQL_CPY" -
L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD070 preprocessed/ORD070.cob -I cpy -I "$OCESQL_CPY" -
L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD110 preprocessed/ORD110.cob -I cpy -I "$OCESQL_CPY" -
L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
cobc -m -o bin/ORD150 preprocessed/ORD150.cob -I cpy -I "$OCESQL_CPY" -
L/opt/homebrew/lib -L/opt/homebrew/opt/libpq/lib -Q "-Wl,-
rpath,/opt/homebrew/opt/libpq/lib" -locesql -lpq
```

Przygotowanie programów, które nie wchodzi w interakcję z bazą danych, składa się tylko z jednego kroku – ich kompilacji, do której polecenia podane są na listingu 72.

Listing 72. Polecenia do kompilacji programów niekomunikujących się z bazą danych. Źródło: opracowanie własne

```
cobc -x -I cpy ORD000.cbl -o bin/ORD000
cobc -m -I cpy ORD001.cbl -o bin/ORD001
cobc -m -I cpy ORD002.cbl -o bin/ORD002
cobc -m -I cpy ORD003.cbl -o bin/ORD003
cobc -m -I cpy ORD004.cbl -o bin/ORD004
cobc -m -I cpy ORD005.cbl -o bin/ORD005
cobc -m -I cpy ORD006.cbl -o bin/ORD006
cobc -m -I cpy ORD100.cbl -o bin/ORD100
```

Po pomyślnym zakończeniu procesu kompilacji, można uruchomić program z poziomu wiersza poleceń za pomocą komendy na listingu 73.

Listing 73. Polecenie służące do uruchomienia programu. Źródło: opracowanie własne

```
./bin/ORD000 data/input.txt data/output.txt
```

Pierwszy argument to plik, który będzie traktowany jako wejście do programu. Drugi argument definiuje lokalizację, do której zapisany zostanie wynik jego działania.

5.2. Java

Niniejszy rozdział zawiera diagramy wywołań każdej funkcjonalności biznesowej. Zawiera również opis każdej z sześciu rodzajów zaimplementowanych klas na przykładzie jednej, należącej do danej warstwy, klasy.

5.2.1. Lista endpointów

System przetwarzania zamówień zaimplementowany w Javie udostępnia następujące endpointy:

- *GET /orders* – pobranie listy zamówień,
- *GET /orders/{id}* – pobranie szczegółów zamówienia określonego numerem ID,
- *POST /orders* – dodanie nowego zamówienia,
- *PUT /orders/{id}/products* – dodanie produktu do istniejącego zamówienia,
- *DELETE /orders/{id}/products* – usunięcie produktu z istniejącego zamówienia,
- *POST /orders/batch* – masowa aktualizacja zamówień z pliku CSV,
- *GET /products* – pobranie listy produktów,
- *GET /products/{id}* – pobranie szczegółów produktu określonego numerem ID,
- *POST /products* – dodanie nowego produktu.

Poza zadanymi przez zdefiniowane wymagania biznesowe funkcjonalnościami związanymi z obsługą zamówień, system został rozszerzony o możliwości związane z produktami. Dzięki dostępnym narzędziom i bibliotekom dla języka Java implementacja dodatkowych metod biznesowych w istniejącym już projekcie jest dużo łatwiejsza i wymaga zdecydowanie mniejszej ilości kodu w porównaniu do implementacji w języku COBOL.

5.2.2. Diagram wywołań funkcjonalności biznesowych

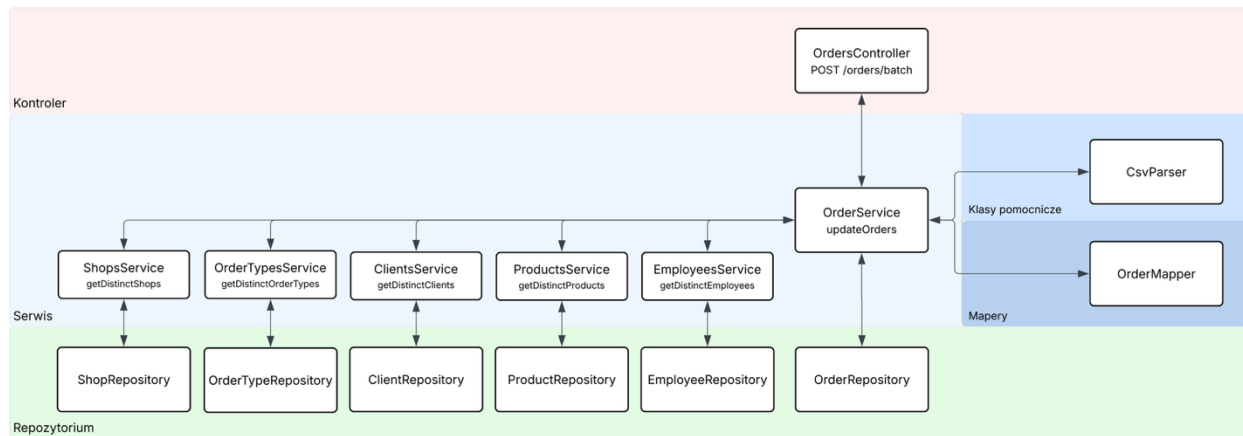
W celu ulokowania klas w kontekście systemu, w następujących punktach przedstawiony zostanie diagram wywołań każdej obsługiwanej wymaganej funkcjonalności biznesowej.

Masowa aktualizacja zamówień

Funkcjonalność dotycząca masowego przetwarzania zamówień w praktyce byłaby wywoływana automatycznie w skonfigurowanych odstępach czasu. W celu umożliwienia łatwego przetestowania funkcjonalności na potrzeby niniejszej pracy, odpowiedzialność za uruchomienie przetwarzania została przeniesiona do standardowego endpointu, dostępnego pod adresem URL *POST /orders/batch*.

W celu upewnienia się, że implementacje w obydwu językach będą do siebie jak najbardziej zbliżone, funkcja masowego przetwarzania zamówień nie przyjmuje argumentów wejściowych. Ponieważ system w języku COBOL korzysta z plików umieszczonych na dysku w podanej lokalizacji, w omawianej aplikacji zostało wykorzystane podobne rozwiązanie. Zamiast dostarczać plik CSV jako część żądania program spodziewa się plików wejściowych w folderze *resources*. Takie rozwiązanie odbiega formą od typowej implementacji REST API, jednak zostało przyjęte założenie, że omawiana funkcjonalność ma być zbliżona do wsadowych operacji znanych ze środowiska Mainframe.

Na rysunku 19 został przedstawiony diagram wywołań funkcjonalności masowego aktualizowania zamówień.



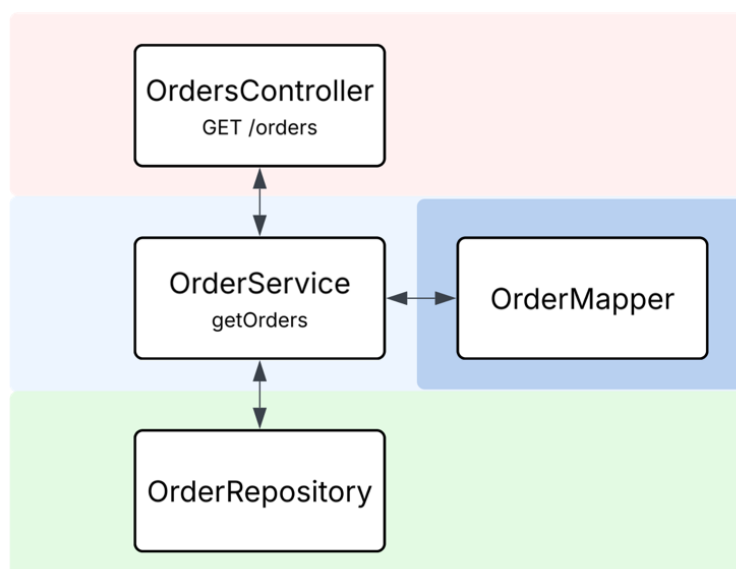
Rysunek 19. Diagram wywołań funkcji masowej aktualizacji zamówień. Źródło: opracowanie własne

Podobnie jak w przypadku systemu w COBOL, niniejsza funkcja jest zdecydowanie najbardziej złożonym elementem systemu. W celu zweryfikowania poprawności danych stosowany jest bardzo podobny mechanizm – w trakcie przetwarzania zamówień *OrderService* odwołuje się do większości programów z warstwy serwisów. Wynika to z faktu, że bezpośredni dostęp do tabeli ma tylko jeden konkretny obiekt, którego nazwa odpowiada nazwie encji z bazy.

Kolejnym zbliżonym elementem jest oddelegowanie przetwarzania zawartości pliku tekstowego do warstwy klas pomocniczych – odpowiednikiem ORD100 z poprzednio omówionego systemu jest tutaj statyczna klasa *CsvParser*, dostarczająca zbliżoną funkcjonalność do nadmienionego programu. Odpowiedzialność za przetwarzanie pliku tekstowego jest więc odseparowana od kodu biznesowego, i można w łatwy sposób wykorzystać tą funkcjonalność ponownie.

Pobranie listy zamówień

Adresem URL, pod jakim dostępna jest lista zamówień to *GET /orders*. Na rysunku 20 przedstawiony jest diagram wywołań tego żądania.



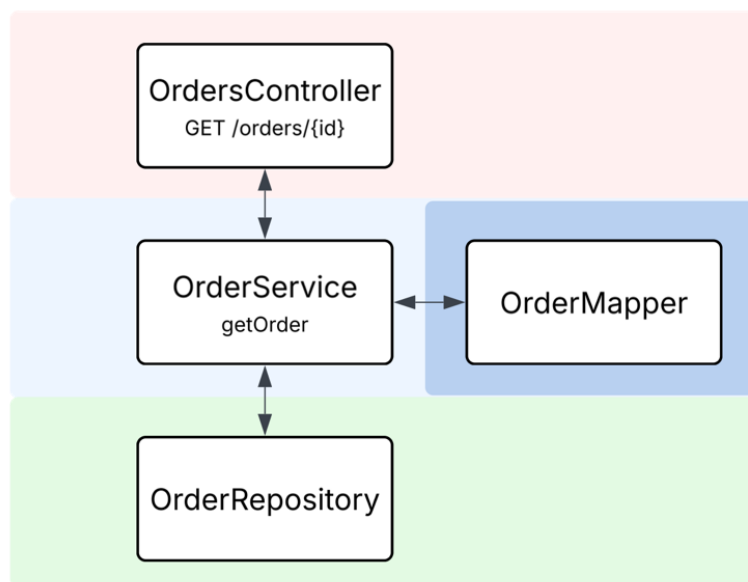
Rysunek 20. Diagram wywołań funkcji pobrania listy zamówień. Źródło: opracowanie własne

Ponownie można zauważyć duże podobieństwo procesu pobierania listy zamówień do omówionego wcześniej systemu w języku COBOL. Punktem wejścia jest kontroler, który wysyła je do serwisu obsługującego zamówienia. Serwis ma dostęp do repozytorium, które komunikuje się z bazą danych.

Największą różnicą między sposobem działania obu systemów jest obecność mapera *OrderMapper*. Dobrą praktyką jest oddzielić model reprezentujący encję z bazy danych od modelu, który aplikacja wykorzystuje do komunikacji z klientami. Warto więc wydzielić metody konwertujące obydwa obiekty do osobnego serwisu. Pozwala to na zachowanie zasady pojedynczej odpowiedzialności oraz ułatwia ponowne wykorzystanie kodu.

Pobranie szczegółów zamówienia

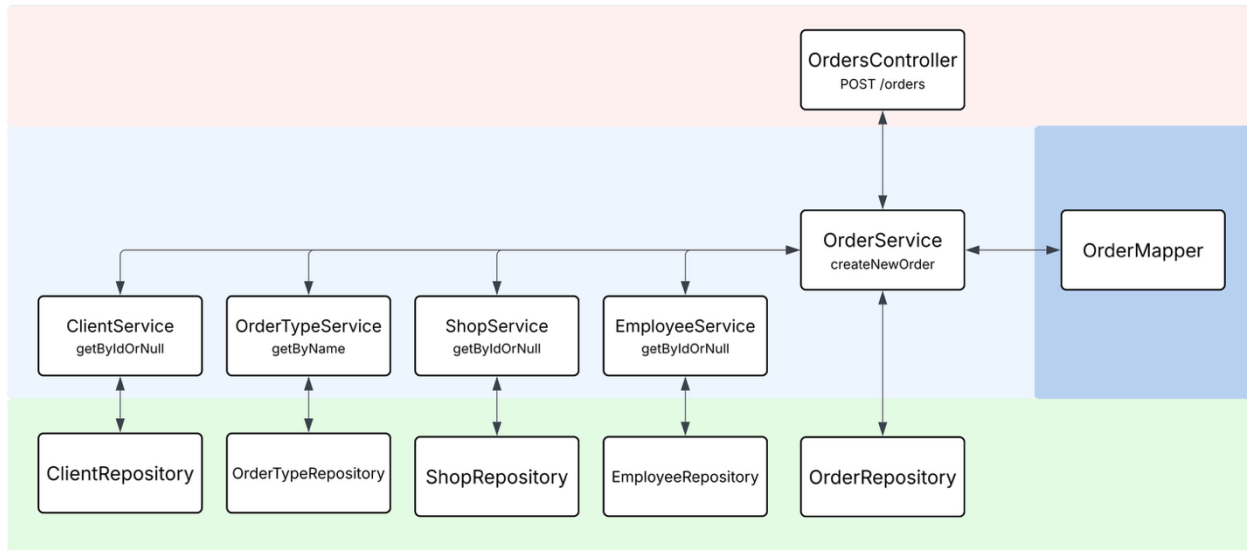
Aby pobrać szczegóły zamówienia, należy wysłać zapytanie zawierające numer ID żadanego zasobu na adres URL *GET /orders/{id}*. Rysunek 21 przedstawia klasy wywoływane podczas procesu przetwarzania zapytania.



Rysunek 21. Diagram wywołań funkcji pobrania szczegółów zamówienia. Źródło: opracowanie własne

Dodanie nowego zamówienia

Funkcja umożliwiająca dodanie nowego zamówienia jest dostępna pod adresem URL *POST /orders*. Jej diagram wywołań zaprezentowany jest na rysunku 22.

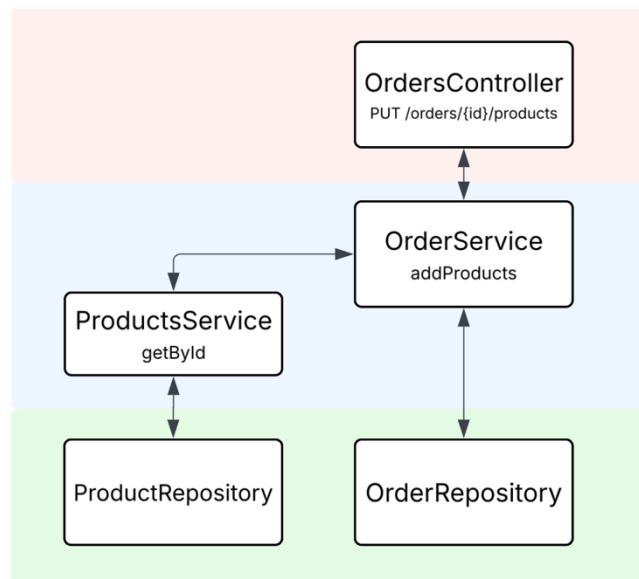


Rysunek 22. Diagram wywołań funkcji dodania nowego zamówienia. Źródło: opracowanie własne

W celu upewnienia się, że dane, które dodawane są do bazy danych są prawidłowe i nie zostaną naruszone więzy integralności, oprócz samego zapisu nowego rekordu do tabeli *Orders* przeprowadzana jest także weryfikacja poprzez inne serwisy.

Dodanie nowego produktu do zamówienia

Aby dodać nowy produkt do istniejącego już zamówienia, należy wysłać żądanie na adres URL *PUT /orders/{id}/products*. ID oznacza numer identyfikacyjny zamówienia, które należy zaktualizować. Rysunek 23 przedstawia jego przebieg.

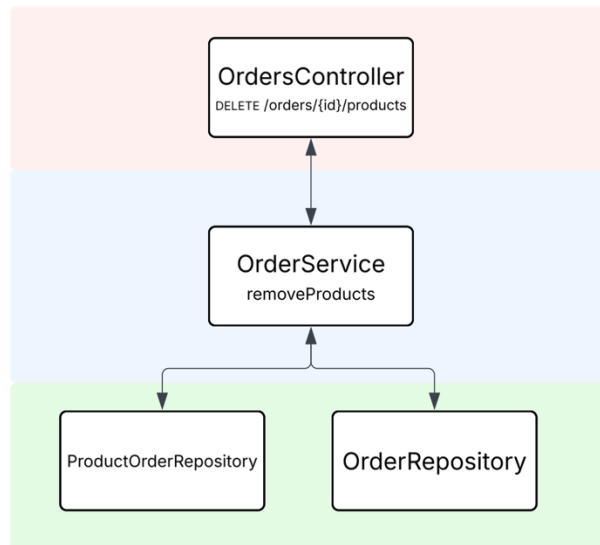


Rysunek 23. Diagram wywołań funkcji dodania produktu do zamówienia. Źródło: opracowanie własne

Ponownie weryfikowana jest zawartość zapytania – sprawdzane jest zarówno zamówienie jak i lista wszystkich produktów. Jeżeli wysłane dane są poprawne, dokonywana jest aktualizacja i zapisanie nowych rekordów poprzez *OrderRepository*.

Usunięcie produktu z zamówienia

Usuwanie produktu z zamówienia działa w sposób zbliżony do dodawania – należy wysłać inne żądanie HTTP na ten sam adres URL – *DELETE /orders/{id}/products*. Rysunek 24 obrazuje, które klasy używane są podczas przebiegu tej funkcjonalności.



Rysunek 24. Diagram wywołań funkcji usunięcia produktu z zamówienia. Źródło: opracowanie własne

Największą różnicą pomiędzy usuwaniem produktu, a wszystkimi uprzednio omówionymi funkcjonalnościami, jest odwołanie się bezpośrednio przez *OrderService* do *ProductOrderRepository*. Aby w pełni utrzymać konwencję separacji tabel między obsługującymi je serwisami, należałoby wydzielić usuwanie wierszy z tabeli *Product_Order* do osobnego serwisu. Ponieważ jest to tabela asocjacyjna, w niniejszym projekcie dozwolone zostało odwołanie się do niej bezpośrednio z serwisu dotyczącego tabeli *Orders*.

5.2.3. Kontroler

W celu omówienia struktury warstwy kontrolerów, przedstawiona zostanie klasa *OrdersController*. Jej fragment widoczny jest na listingu 74.

Listing 74. *OrdersController* - fragment. Źródło: opracowanie własne

```
@Slf4j
@RestController
@RequestMapping(value = "/orders")
@RequiredArgsConstructor
public class OrdersController {

    private final OrderService orderService;

    @GetMapping
    @Operation(
        summary = "Get a summary of orders saved in the database",
        description = """"
            Returns a list of orders based on given criteria.
            """"
    )
}
```

```

        responses = {
            @ApiResponse(responseCode = "200", description =
                "Returns an order summary"),
            @ApiResponse(responseCode = "204", description =
                "No orders with given search criteria found"),
            @ApiResponse(responseCode = "400", description =
                "Provided data is incorrect"),
        })
    public ResponseEntity<List<OrderDataDto>> getOrders(
        @RequestParam(value = "orderType", required = false)
        List<OrderTypes> orderTypes,
        @RequestParam(value = "dateFrom", required = false)
        LocalDate dateFrom,
        @RequestParam(value = "dateTo", required = false)
        LocalDate dateTo,
        @RequestParam(value = "page", defaultValue = "0")
        Integer page
    ) {
        List<OrderDataDto> orders = orderService
            .getOrders(orderTypes, dateFrom, dateTo, page);

        HttpStatus status = orders.isEmpty() ?
            HttpStatus.NO_CONTENT :
            HttpStatus.OK;

        return new ResponseEntity<>(
            orders,
            status
        );
    }
}

```

Każdy kontroler oznaczony jest co najmniej dwoma adnotacjami: *@RestController*, która informuje kontener Springa o jej roli w strukturze projektu, oraz *@RequestMapping*, definiująca początkowy fragment URL, pod którym będą znajdować się udostępnione metody. Adnotacja *@RequiredArgsConstructor* pochodzi z biblioteki Lombok i ułatwia pracę poprzez automatyczną generację konstruktora inicjalizującego niezbędne dla klasy pola oznaczone jako *final*. *@Slf4j* pochodzi z tej samej biblioteki i pełni podobną rolę – zapewnia dostęp do loggera, mimo braku jego jawnego zadeklarowania.

Ponieważ *OrdersController* komunikuje się jedynie z serwisem *OrderService*, jest on zadeklarowany jako zależność, aby kontener Springa mógł ją odpowiednio wstrzyknąć poprzez konstruktor (generowany poprzez uprzednio wspomnianą adnotację *@RequiredArgsConstructor*).

W ten sposób kontroler zawiera wszystkie niezbędne do poprawnego działania narzędzia, więc reszta klasy składa się z definicji endpointów. Każdy z nich musi zawierać adnotację deklarującą metodę HTTP, której będzie używać. W przypadku omawianego przykładu pobierania listy zamówień jest to *@GetMapping*, ale w projekcie używane są też adnotacje *@PostMapping*, *@PutMapping* oraz *@DeleteMapping*.

Zawartość adnotacji *@Operation* to dokumentacja, która będzie dostępna w automatycznie generowanym kontrakcie OpenAPI – dzięki jej zastosowaniu będzie ona bardziej przystępna w użyciu dzięki szerszym opisaniu sposobu zachowania aplikacji w różnych przypadkach.

Każdy kontroler zdefiniowany w ramach niniejszego projektu podąża za opisaną strukturą – zawierają jednakowe adnotacje, definiują wymagane obiekty oraz udostępniają różne endpointy, z których każdy posiada rozszerzoną definicję w celu poprawy jakości dokumentacji technicznej.

W przykładzie widocznym na listingu 69 funkcja *getOrders* definiuje cztery niewymagane argumenty wejściowe, służące za filtry do zwracanej listy zamówień.

5.2.4. Serwis

Klasy serwisowe zawierają cały kod dotyczący logiki biznesowej. W projekcie zdefiniowane zostało 8 takich serwisów. Omówiony zostanie fragment najważniejszego z nich – *OrdersService*, który widoczny jest na listingu 75.

Listing 75. *OrdersService* - fragment. Źródło: opracowanie własne

```
@Service
@RequiredArgsConstructor
public class OrdersService {

    private final OrderMapper orderMapper;

    private final ProductsService productsService;
    private final ShopService shopsService;
    private final ClientService clientsService;
    private final EmployeeService employeesService;
    private final OrderTypeService orderTypeService;

    private final OrdersRepository ordersRepository;
    private final ProductOrderRepository productOrderRepository;

    public List<OrderDataDto> getOrders(
        List<OrderTypes> orderTypes,
        LocalDate dateFrom,
        LocalDate dateTo,
        Integer page
    ) {
        List<String> orderTypesStrings;

        if (orderTypes != null && !orderTypes.isEmpty()) {
            orderTypesStrings =
                orderTypes.stream().map(OrderTypes::getFullName).toList();
        } else {
            orderTypesStrings = null;
        }

        return ordersRepository
            .getOrders(orderTypesStrings, dateFrom, dateTo,
                PageRequest.of(page, 20))
            .stream()
            .map(orderMapper::mapOrder)
            .toList();
    }
}
```

W przypadku klas serwisowych, Spring również wymaga zastosowania odpowiedniej adnotacji – w tym wypadku jest to *@Service*. Podobnie jak w przypadku kontrolera, zastosowana została też adnotacja *@RequiredArgsConstructor* generująca konstruktor inicjalizujący wszystkie wymagane pola. Omawiana klasa posiada znacznie więcej zależności – wstrzykiwane jest 5 innych serwisów, dwa repozytoria oraz klasa pomocnicza mapper. Ilość ta wynika z faktu, że podczas procesu masowej aktualizacji zamówień oraz dodawania nowych zamówień przeprowadzana jest szeroka walidacja danych, która wymaga dostępu do wierszy w bazie. Z przyjętych założeń wynika, że dostęp do nich może mieć jedynie odpowiedni serwis.

Metoda zwracająca listę zamówień sprawdza, czy klient zastosował filtry dotyczące rodzaju zamówienia. W celu zapewnienia stałego kontraktu API nawet w przypadku modyfikacji danych w bazie, słownikowym wartościom zapewnione zostały odpowiadające im klasy Enum. Zastosowanie ich umożliwia łatwe dostosowanie aplikacji do zaktualizowanych wartości bez konieczności modyfikowania kodu aplikacji klienckich. Enum *OrderTypes*, odpowiadający tabeli *Order_Type* widoczny jest na listingu 76.

Listing 76. *OrderTypes* - enumeracja odpowiadająca tabeli *Order_Type*. Źródło: opracowanie własne

```
package zz.java.controller.model;

import lombok.Getter;

@Getter
public enum OrderTypes {
    ONLINE("Online"),
    IN_STORE("In store");

    private final String fullName;

    OrderTypes(String fullName) {
        this.fullName = fullName;
    }
}
```

Ponieważ API oczekuje, że klient wyśle wartość „ONLINE”, jedyna niezbędna zmiana przy aktualizacji w bazie danych to zaktualizowanie wartości *fullName*.

Po odpowiedniej konwersji obiektów *OrderTypes* na wartość z bazy danych wywoływana jest metoda *getOrders* w repozytorium, która zwraca listę zamówień zgodnie z podanymi argumentami. Następnie obiekty wynikowe wysyłane są do mapera, który dokonuje konwersji między reprezentacją encji a obiektem DTO, gotowym do przekazania do aplikacji klienckich.

5.2.5. Repozytorium

Repozytoria zawierają funkcjonalność zbliżoną do programów warstwy dostępu i modyfikacji do danych omówionych dla implementacji w COBOL. Jako przykład przedstawiona zostanie klasa *OrdersRepository*, której fragment widoczny jest na listingu 77.

Listing 77. *OrdersRepository* - fragment. Źródło: opracowanie własne

```
import java.time.LocalDate;
import java.util.List;

public interface OrdersRepository extends JpaRepository<Order, Integer> {

    @Query("""
        SELECT o
        FROM Order o
        WHERE (:orderTypes IS NULL OR o.orderType.name IN
            :orderTypes)
            AND o.date BETWEEN COALESCE(:dateFrom, o.date) AND
            COALESCE(:dateTo, o.date)
        """)
    Page<Order> getOrders(@Param("orderTypes") List<String> orderTypes,
        @Param("dateFrom") LocalDate dateFrom,
        @Param("dateTo") LocalDate dateTo,
        Pageable page
    );
}
```

Zdecydowana większość funkcjonalności, które mogą okazać się użyteczne w przypadku pracy z bazami danych, są dostępne poprzez zdefiniowanie repozytorium jako *extends JpaRepository*. Jest to interfejs pochodzący z biblioteki Spring Data JPA, a zastosowanie go zdecydowanie zmniejsza ilość kodu, którą trzeba napisać ręcznie.

W przypadku *OrdersRepository* wystąpiła potrzeba zadeklarowania tylko jednej metody, wykorzystywanej dla pobrania listy zamówień z filtrami. Alternatywnie istnieje możliwość skorzystania z dostępnej w interfejsie *JpaRepository* *getAll()* – jednak jej użycie oznaczałoby pobranie całej zawartości tabeli, na której następnie trzeba byłoby zastosować odpowiednie filtry. Takie podejście zdecydowanie może prowadzić do problemów z wydajnością i lepiej zdecydować się na zdefiniowanie własnego zapytania. Lepszym podejściem od stosowania JPQL jest JPA Criteria Query, jednak ze względu na prostotę i czytelność pierwszego z rozwiązań zostało one wykorzystane w tym przypadku.

5.2.6. Model

W niniejszym projekcie obiekty modelowe dzielą się na dwa rodzaje – reprezentacje związków encji wchodzące w interakcję z bazą danych oraz *DTO* (Data Transfer Object), które są stosowane do komunikacji z aplikacjami klienckimi. Podział ten pozwala na kontrolowanie, które informacje przesyłane są do i z bazy.

Na listingu 78 zaprezentowany jest fragment klasy *Order*, która stanowi odpowiednik dla tabeli *Orders*.

Listing 78. Klasa *Order* – fragment. Źródło: opracowanie własne

```
@Entity
@Getter
@Table(name = "Orders")
@NoArgsConstructor
public class Order {

    @Id
    @Column(name = "ID")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Integer id;

    @NotNull
    @Column(name = "Date", nullable = false)
    LocalDateTime date;

    @ManyToOne(optional = false)
    @JoinColumn(name = "Order_Type_ID")
    OrderType orderType;

    @ManyToOne
    @JoinColumn(name = "Employee_ID")
    Employee employee;

    @ManyToOne
    @JoinColumn(name = "Client_ID")
    Client client;

    @ManyToOne
    @JoinColumn(name = "Shop_ID")
    Shop shop;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
        orphanRemoval = true)
    Set<ProductOrder> productOrders;
```

Order zawiera obowiązkową adnotację *@Entity*, która informuje o jej roli jako reprezentacja encji. *@Table* jest adnotacją definiującą, do której tabeli odnosi się klasa. Jest ona nieobowiązkowa, jeżeli nazwa klasy dokładnie odpowiada nazwie z bazy – jednak w tym wypadku należy ją dostarczyć. *Order* to zarezerwowane słowo w wielu popularnych silnikach baz danych; co prawda można go użyć jako nazwę tabeli, ale komplikuje to później na przykład automatyczną generację struktury bazy przy wykorzystaniu podejścia code-first. Z uwagi na ten fakt została użyta nazwa *Orders*.

Poza polami definiującymi kolumny w bazie danych, klasa posiada również metody wymuszające podążanie za ustalonymi zasadami biznesowymi i upewniające się, że zostanie zachowana integralność więzów. Przykładem takiej metody jest *addProductOrders*, widoczna na listingu 79.

Listing 79. *Order* - metoda *addProductOrders*. Źródło: opracowanie własne

```
public void addProductOrders(List<ProductOrder> productOrder) {
    if (productOrder == null || productOrder.isEmpty()) {
        throw new IllegalArgumentException("Product orders cannot be null
                                         or empty");
    }

    List<ProductOrder> filteredProductOrders = productOrder
        .stream()
        .filter(Objects::nonNull)
        .toList();

    filteredProductOrders.forEach(po -> po.linkOrder(this));
    this.productOrders.addAll(filteredProductOrders);
}
```

W celu upewnienia się, że do bazy nie zostaną dodane żadne błędne wartości, metoda upewnia się, że zostały dostarczone jakiekolwiek dane, a następnie lista produktów w zamówieniu jest filtrowana oraz do każdego z nich dodawana jest referencja zwrotna.

Listing 80 przedstawia *OrderDataDto*, czyli drugi rodzaj wykorzystywanych modeli, przesyłany jako odpowiedź na omawiane zapytanie o listę zamówień.

Listing 80. Rekord *OrderDataDto*. Źródło: opracowanie własne

```
package zz.java.controller.model;

import java.time.LocalDate;

public record OrderDataDto(
    Integer orderId,
    LocalDate orderDate,
    String orderType,
    String clientName,
    String shopName,
    String employeeName
) {
}
```

Klasy DTO w niniejszym projekcie zdefiniowane są jako obiekty typu *Record*. Jest to rodzaj dostępny w języku Java od wersji 16, który znacząco zmniejsza ilość powtarzalnego kodu w klasach stosowanych do transferu danych. Dzięki zapewnieniu domyślnego konstruktora oraz metod umożliwiających pobranie wartości pól, stanowią one dobry wybór dla DTO.

5.2.7. Klasy pomocnicze (*utility*)

Aby umożliwić jasną separację kodu biznesowego od technicznego, w projekcie zastosowane zostały klasy pomocnicze. Przykładem może być wspomnianą wcześniej klasa *CsvParser*, która jest odpowiednikiem ORD100 w COBOL i umożliwia przetwarzanie plików CSV. Jej fragment został przedstawiony na listingu 81.

Listing 81. CsvParser - fragment. Źródło: opracowanie własne

```
@UtilityClass
public class CsvParser {

    private final String ORDERS_PATH = "classpath*:orders/*.csv";

    public static List<ParseResult<OrderDto>> parseOrderFiles() throws
IOException {
        Resource[] orderFiles = getOrderFiles();
        List<ParseResult<OrderDto>> parseResults = new ArrayList<>();

        for (Resource orderFile : orderFiles) {
            try (BufferedReader orderReader = new BufferedReader(new
InputStreamReader(orderFile.getInputStream(), StandardCharsets.UTF_8))) {
                parseResults.add(processFile(orderFile.getFilename(),
orderReader));
            }
        }

        return parseResults;
    }
}
```

W celu oznaczenia klasy jako pomocniczej, zastosowana została adnotacja *@UtilityClass* – pochodząca z biblioteki Lombok zapobiega tworzeniu instancji klasy oraz oznacza ją jako *final*. Nie jest ona więc wstrzykiwana jako zależność – zdefiniowane w niej metody są statyczne, więc można odwołać się do nich bezpośrednio bez posiadania jej instancji.

5.2.8. Mapery

Poprzez zastosowanie w projekcie dwóch rodzajów modeli, występuje konieczność konwersji pomiędzy obiektami bazodanowymi oraz DTO. Ponieważ jest to powszechna operacja, kod odpowiadający za przekazanie danych między modelami został wydzielony do oddzielnych klas. Przykładem jest *OrderMapper*, którego fragment został przedstawiony na listingu 82.

Listing 82. OrderMapper - fragment. Źródło: opracowanie własne

```
@Component
@RequiredArgsConstructor
public class OrderMapper {

    public OrderDataDto mapOrder(Order order) {
        return new OrderDataDto(
            order.getId(),
            order.getDate().toLocalDate(),
            order.getOrderType().getName(),
            extractClientName(order),
            extractShopName(order),
            extractEmployeeName(order)
        );
    }
}
```

W celu umożliwienia kontenerowi Springa wstrzykiwania instancji mapera do klas, zastosowana została adnotacja *@Component*.

5.2.9. Kompilacja projektu

W celu skompilowania projektu należy zapewnić następujące narzędzia:

- JDK 21 [56],
- Maven 3.9.12 [57].

Po spełnieniu tych wymagań można dokonać procesu kompilacji za pomocą polecenia *mvn install*.

5.3. Warstwa WebAPI dla systemu COBOL-owego w języku Java

Z uwagi na zastosowane na potrzeby projektu narzędzia, nie istniała możliwość zapewnienia interfejsu WebAPI dla aplikacji zaimplementowanej w języku COBOL. Zamiast zastosowania standardowego dla systemów na środowisku Mainframe IBM CICS, utworzona została aplikacja w języku Java udostępniająca taki interfejs, która służy jako pośrednik między programami w COBOL a aplikacjami klienckimi.

Rozwiązanie to nie jest całkowicie oddalone od rzeczywistości; w przypadku prawdziwego komputera Mainframe programy byłyby uruchamiane na żądanie za pomocą na przykład protokołu TCP, jednak nie wspierają one domyślnie formatu JSON ani standardowych metod HTTP – w związku z tym stosuje się rozwiązania zbliżone do omawianego, gdzie współczesny język programowania służy jako interfejs między klientem a CICS.

5.3.1. Dostępne endpointy i powiązanie z warstwą COBOL

Każda z dostępnych w systemie funkcjonalności biznesowych otrzymała odpowiadający endpoint:

- *POST /orders/batch* - Masowa aktualizacja zamówień,
- *GET /orders* – Pobranie listy zamówień,
- *GET /orders/{id}* - Pobranie szczegółów zamówienia,
- *POST /orders* - Dodanie nowego zamówienia,
- *PUT /orders/{id}/products* - Dodanie nowego produktu do zamówienia,
- *DELETE /orders/{id}/products* - Usunięcie produktu z zamówienia.

Adresy URL dokładnie odpowiadają adresom przypisanym systemowi zaimplementowanemu całkowicie w Java, a przyjmowane i zwracane dane są jednolite dla obu aplikacji – umożliwia to wykorzystywanie tego samego klienta oraz zapytań HTTP, niezależnie od tego, która implementacja jest uruchomiona.

5.3.2. Metoda komunikacji między programami COBOL a warstwą Java

Zasadniczą różnicą między zastosowaniami praktycznymi a implementacją na potrzeby projektu jest sposób uruchamiania modułów – dla środowiska Mainframe odpowiedzialność leży po stronie CICS, natomiast w niniejszej implementacji skompilowane programy uruchamiane są za pośrednictwem wiersza poleceń.

W aplikacji pośredniczącej za wywołanie programu odpowiedzialna jest klasa *CobolExecutor*. Jej fragment został przedstawiony na listingu 83.

```

@Slf4j
@Component
public class CobolExecutor {

    private static String SERVER_IP_ADDRESS;
    private static String COBOL_BIN_PATH;
    private static String COBOL_IO_PATH;

    public <REQ extends CopybookRepresentation, RES extends
CopybookRepresentation> RES callOrderCobolProgram(
        OrderFunction function,
        REQ copybookRequestRepresentation,
        Class<RES> copybookResponseRepresentation
    ) {
        Path inputFile = Path.of(COBOL_IO_PATH + "input.txt");
        Path outputFile = Path.of(COBOL_IO_PATH + "output.txt");
        String result;

        try {
            String parsedInput = CopybookParser.parseToCopybook(
                function.getValue(), copybookRequestRepresentation);
            writeOutput(parsedInput, inputFile);

            long beforeExecution = System.currentTimeMillis();
            log.debug("Calling ORD000 using input: {}", parsedInput);
            log.info("Timestamp before COBOL execution: {}", beforeExecution);

            result = executeCobolProgram(
                COBOL_BIN_PATH + "ORD000", inputFile, outputFile);

            long afterExecution = System.currentTimeMillis();
            log.info("Timestamp after COBOL execution: {}", afterExecution);
            log.debug("ORD000 returned output: {}", result);
            log.info("Total COBOL execution time (ms): {}", afterExecution -
                beforeExecution);

            return CopybookParser.parseFromCopybook(
                result, copybookResponseRepresentation);
        } catch (IOException | InterruptedException e) {
            throw new CobolExecutorException("Error when calling ORD000", 500);
        }
    }
}

```

Wartości zmiennych wymaganych do odpowiedniego uruchomienia modułów, takie jak adres IP serwera bazy danych, lokalizacja skompilowanych programów oraz folder, za pośrednictwem którego będzie odbywać się komunikacja, muszą zostać podane w profilu uruchomienia warstwy WebAPI.

Ponieważ w zaimplementowanym systemie istnieje tylko jeden program warstwy interfejsu, ORD000, jego nazwa jest ustawiona na stałe w kodzie. Alternatywnym rozwiązaniem byłoby podanie nazwy modułu jako argument do funkcji.

Z uwagi na potrzebę zbadania wydajności systemu, informacje na temat czasu rozpoczęcia i zakończenia wykonania zostały umieszczone tuż przed i po wywołaniu programu. W przypadku logowania go w kontrolerze, dodany zostałby czas spędzony w interfejsie WebAPI – zastosowane rozwiązanie minimalizuje więc czynniki zewnętrzne, pozwalając na dokładniejszy pomiar wydajności programów w COBOL.

Faktyczne uruchomienie procesu oraz wywołanie modułu odbywają się w metodzie *executeCobolProgram*, która jest widoczna na listingu 84.

Listing 84. *CobolExecutor* - metody *executeCobolProgram* i *getProcess*. Źródło: opracowanie własne

```
private String executeCobolProgram(
    String programName,
    Path inputFile,
    Path outputFile
) throws IOException, InterruptedException {
    Process process = getProcess(programName, inputFile, outputFile);
    String output = new String(
        process.getInputStream().readAllBytes(), StandardCharsets.UTF_8
    );
    int exitCode = process.waitFor();

    if (exitCode != 0) {
        throw new CobolExecutorException(
            "COBOL program exited with code " + exitCode + ": " + output, exitCode
        );
    }

    return Files.readString(outputFile, StandardCharsets.UTF_8);
}

private Process getProcess(String programName, Path inputFile, Path outputFile)
throws IOException {
    ProcessBuilder processBuilder = new ProcessBuilder(
        programName, inputFile.toString(), outputFile.toString()
    );
    processBuilder.directory(Path.of(COBOL_BIN_PATH).getParent().toFile());
    processBuilder.redirectErrorStream(true);

    Map<String, String> env = processBuilder.environment();
    env.put("PGHOST", SERVER_IP_ADDRESS);
    env.put("PGPORT", "5432");
    env.put("PGCONNECT_TIMEOUT", "5");
    env.put("OCESQL_CPY", "/opt/homebrew/share/open-cobol-esql/copy");
    env.put("COB_LIBRARY_PATH", COBOL_BIN_PATH);

    return processBuilder.start();
}
```

W funkcji *getProcess* w środowisku ustawiane są wszystkie potrzebne parametry dostarczone poprzez profil uruchamiania aplikacji. Następnie uruchamiany jest zadany moduł, a jeżeli wykonanie zakończy się sukcesem jednoznaczny z kodem wyjścia 0, odczytywany jest plik wynikowy i przekazywany z powrotem do metody *callOrderCobolProgram*.

Zarówno wejście do programu, jak i wyjście muszą ściśle odpowiadać omówionym we wcześniejszych rozdziałach copybookom – w przeciwnym wypadku wartości nie będą poprawnie interpretowane i funkcjonalność nie zadziała poprawnie. Z uwagi na bardzo wrażliwą naturę komunikacji z programami COBOL zarówno wejście, jak i wyjście z programu jest konwertowane za pomocą klasy *CopybookParser*, która zapewnia odpowiednią konwersję między obiektem Java a plikiem tekstowym zgodnym ze strukturą copybook.

5.3.3. Przetwarzanie struktur Copybook z użyciem Java Reflection API

Z uwagi na omówiony powyżej problem konwersji między obiektami Java a formatem copybook, w celu umożliwienia prostej komunikacji między aplikacjami COBOL a interfejsem WebAPI, zaimplementowany został serwis tłumaczący informacje pomiędzy formatami.

Aby skorzystać z konwersji, należy zdefiniować klasę, która będzie odpowiadała wymaganiom copybookowi. Musi ona zawierać adnotacje posiadające informacje na temat rozmiaru oraz typu pola. W celu zapewnienia zgodności z formatem procesor musi wiedzieć, jakimi wartościami uzupełnić pustą przestrzeń – oraz ile tej przestrzeni jest.

Większość funkcjonalności udostępniana przez system w COBOL wymaga zdefiniowania dwóch klas, które będą odzwierciedlać format danych wejściowych oraz wyjściowych. Wyjątkiem w przypadku omawianego systemu jest funkcja masowego przetwarzania zamówień – ponieważ została zaprojektowana jako program wsadowy, nie zwraca ona danych – w takim wypadku definiowanie klasy wyjściowej jest zbędne.

Listing 85 przedstawia klasę *GetOrderResponseCpy*, która jest odzwierciedleniem copybook'a ORD003AR, widocznego na listingu 86.

Listing 85. GetOrderResponseCpy - fragment. Źródło: opracowanie własne

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class GetOrderResponseCpy extends CopybookRepresentation {

    @CobolField(length = 10, fieldType = FieldType.NUMERIC)
    Integer orderId;

    @CobolField(length = 10, fieldType = FieldType.ALPHANUMERIC)
    String orderDate;

    @CobolField(length = 8, fieldType = FieldType.ALPHANUMERIC)
    String orderType;

    @CobolField(length = 20, fieldType = FieldType.ALPHANUMERIC)
    String employeeName;

    @CobolField(length = 20, fieldType = FieldType.ALPHANUMERIC)
    String shopName;

    @CobolField(length = 20, fieldType = FieldType.ALPHANUMERIC)
    String clientName;

    @CobolOccursCount(forField = "productData")
    @CobolField(length = 2, fieldType = FieldType.NUMERIC)
    Integer productCount;

    @CobolOccurs(count = 20)
    List<ProductDataResponseCpy> productData;
}
```

Listing 86. ORD003AR - fragment. Źródło: opracowanie własne

```
03 :PFX:-OUTPUT.
  05 :PFX:-O-ORDER-DATA.
    07 :PFX:-O-ORDER-ID          PIC 9(10).
    07 :PFX:-O-ORDER-DATE        PIC X(10).
    07 :PFX:-O-ORDER-TYPE        PIC X(08).
    07 :PFX:-O-EMP-NAME          PIC X(20).
    07 :PFX:-O-SHOP-NAME         PIC X(20).
    07 :PFX:-O-CLIENT-NAME       PIC X(20).

    05 :PFX:-O-PRODUCT-COUNT     PIC 9(02).
    05 :PFX:-O-PRODUCT-DATA      OCCURS 20.
      07 :PFX:-O-PRODUCT-ID      PIC 9(10).
      07 :PFX:-O-PRODUCT-NAME    PIC X(50).
      07 :PFX:-O-QUANTITY         PIC 9(10)V99.
      07 :PFX:-O-UNIT             PIC X(50).
      07 :PFX:-O-PRICE            PIC 9(10)V99.
```

Porównując zawartość klasy *GetOrderResponseCpy* z wskazanym na listingu 81 fragmentem copybook'a ORD003AR, można zauważyć, że nazwy pól oraz zdefiniowane w adnotacji *@CobolField* typy oraz długości są zgodne z ich COBOLowym odpowiednikiem.

Warto zaznaczyć, że każdy copybook może posiadać dowolnie zagnieżdżone pola *OCCURS*, które odpowiadają kolekcjom. Ponieważ ich liczba również musi zostać statycznie określona i na etapie kompilacji przypisywana jest im stała ilość przestrzeni w pamięci, to nawet w przypadku umieszczenia tam jedynie jednego elementu puste miejsca wypełnione są domyślnymi znakami. W związku z tym implementacja systemu konwersji wymagała dodania sposobu przetwarzania takich pól. Przykładem na powyższych listingach jest pole *PRODUCT-DATA*, które w Javie zdefiniowane jest jako lista elementów klasy *ProductDataResponseCpy* oznaczona adnotacją *@CobolOccurs*. Klasa ta przedstawiona jest na listingu 87, a jej pola odpowiadają niższemu w hierarchii elementom poziomemu 07 należącym do kontenera *PRODUCT-DATA*.

Listing 87. ProductDataResponseCpy - fragment. Źródło: opracowanie własne

```
@CobolField(length = 10, fieldType = FieldType.NUMERIC)
Integer productId;

@CobolField(length = 50, fieldType = FieldType.ALPHANUMERIC)
String productName;

@CobolField(length = 12, fieldType = FieldType.FLOATING_POINT,
            decimalPlaces = 2)
Double quantity;

@CobolField(length = 50, fieldType = FieldType.ALPHANUMERIC)
String unit;

@CobolField(length = 12, fieldType = FieldType.FLOATING_POINT,
            decimalPlaces = 2)
Double price;
```

Tak zadeklarowany model jest przystosowany do poddania konwersji za pośrednictwem klasy *CopybookParser*. Fragment jej kodu jest widoczny na listingu 88.

Listing 88. *CopybookParser* - fragment. Źródło: opracowanie własne

```
public static <C extends CopybookRepresentation> C parseFromCopybook(String
copybookRepresentation, Class<C> copybookClass) {
    Map<String, CobolFieldMetadata> fieldsMetadata = extractCobolFieldMetadata(
        copybookClass);

    String[] lines = copybookRepresentation.split("\\r?\\n");
    int lineIndex = 0;
    int offset = 0;

    try {
        C instance = copybookClass.getDeclaredConstructor().newInstance();

        for (CobolFieldMetadata fieldMetadata : fieldsMetadata.values()) {
            Field field = copybookClass.getDeclaredField(fieldMetadata.fieldName());
            field.setAccessible(true);

            if (field.isAnnotationPresent(CobolOccursCount.class)) {
                continue;
            }

            if (fieldMetadata.subFields() != null && !fieldMetadata.subFields().isEmpty()) {
                lineIndex++;
                offset = 0;
                int slotLength = fieldMetadata
                    .subFields()
                    .stream()
                    .mapToInt(CobolSubFieldMetadata::length)
                    .sum();

                List<Object> list = new ArrayList<>();

                while (lineIndex < lines.length && !lines[lineIndex].isBlank()) {
                    String line = String.format("%-" + slotLength + "s", lines[lineIndex]);
                    list.add(parseSubFields(
                        line,
                        fieldMetadata.subFields(),
                        getListElementClass(field)
                    ));

                    lineIndex++;
                }

                field.set(instance, list);
            } else {
                String currentLine = lineIndex < lines.length ? lines[lineIndex] : "";
                String paddedLine = String.format(
                    "%-" + (offset + fieldMetadata.length()) + "s", currentLine);
                String value = paddedLine.substring(offset, offset +
fieldMetadata.length());
                field.set(instance, convertField(fieldMetadata, value));
                offset += fieldMetadata.length();
            }
        }

        return instance;
    } catch (Exception e) {
```

Aby umożliwić poprawne przetworzenie struktury zadanej klasy reprezentującej copybook'a, metoda pobiera jej metadane za pomocą refleksji w metodzie *extractCobolFieldMetadata*, widocznej na listingu 89.

Listing 89. Zastosowanie refleksji w metodzie *extractCobolFieldMetadata*. Źródło: opracownie własne

```
private static Map<String, CobolFieldMetadata> extractCobolFieldMetadata(Class<?
extends CopybookRepresentation> copybookRepresentation) {
    Map<String, CobolFieldMetadata> annotationMap = new LinkedHashMap<>();

    for (Field field : copybookRepresentation.getDeclaredFields()) {
        CobolField cobolFieldAnnotation = field.getAnnotation(CobolField.class);

        if (cobolFieldAnnotation == null) {
            CobolOccurs cobolOccursAnnotation = field.getAnnotation(CobolOccurs.class);

            if (cobolOccursAnnotation != null &&
                List.class.isAssignableFrom(field.getType())) {

                if
                (CopybookRepresentation.class.isAssignableFrom(getListElementClass(field))) {
                    List<CobolSubFieldMetadata> subFields =
                    extractCobolSubFieldMetadata(getListElementClass(field));
                    annotationMap.put(field.getName(), new CobolFieldMetadata(
                        field.getName(),
                        null,
                        null,
                        cobolOccursAnnotation.count(),
                        null,
                        subFields
                    ));
                }
            }
        } else {
            annotationMap.put(field.getName(), new CobolFieldMetadata(
                field.getName(),
                cobolFieldAnnotation.length(),
                cobolFieldAnnotation.fieldType(),
                cobolFieldAnnotation.decimalPlaces(),
                1,
                null
            ));
        }
    }

    return annotationMap;
}
```

ExtractCobolFieldMetadata przechodzi przez wszystkie pola zadeklarowane w klasie poprzez wywołanie *getDeclaredFields*. Następnie sprawdzane są adnotacje za pomocą *getAnnotation* – jeżeli znalezione zostało *@CobolField*, oznacza to, że to standardowe pole i można przenieść informacje na jego temat takie jak długość, typ i nazwa do wynikowej mapy. W przeciwnym wypadku sprawdzana jest obecność *@CobolOccurs*, która oznacza, że pole reprezentuje zagnieżdżoną strukturę *OCCURS*. Wówczas weryfikowane jest, czy jego typ to lista oraz czy jej elementy implementują *CopybookRepresentation* – jeżeli obydwa warunki są spełnione, metoda rekurencyjnie przetwarza klasę reprezentującą zagnieżdżenie i zapisuje metadane jako subpola w wynikowej mapie.

Po ekstrakcji adnotacji i odpowiednim przygotowaniu mapy, metoda *parseFromCopybook* może przejść do konwersji formatów.

Jeżeli pole oznaczone jest jako *@CobolOccursCount*, jest ono ignorowane, ponieważ jest to tylko licznik informujący o ilości elementów w polu *OCCURS*, niezbędny dla programów w COBOL, natomiast niepotrzebny w Javie – *CopybookParser* sprawdza, czy w kolejnym bloku z takiego elementu są obecne dane, a jeżeli nie – pomija je.

Pola oznaczone jako *@CobolOccurs* obsługiwane są za pomocą sprawdzenia, czy zebrane metadane zawierają submetadane. Wartości przetwarzane są zgodnie z informacjami podanymi w subklasie. Metoda dodaje element do listy tylko w przypadku, kiedy obecne są dane.

W przypadku standardowego pola *@CobolField* sprawdzana jest długość wartości. W celu zabezpieczenia kodu przed ewentualnym wystąpieniem wyjątku *IndexOutOfBoundsException* związanego z próbą obcięcia wartości, która jest w rzeczywistości krótsza, dodawane są domyślne znaki. Następnie obcinany jest nadmiar danych wypełnionych domyślnymi wartościami. Metoda *convertField* pozwala na dokonanie konwersji odpowiedniej dla typu danych zdefiniowanego w adnotacji.

Dostarczona implementacja ma dwa negatywne aspekty – po pierwsze, kolejność pól w klasie musi być całkowicie zgodna z reprezentowanym copybook’iem. W przeciwnym wypadku konwersja nie zadziała prawidłowo. Drugim problemem jest konieczność ręcznego definiowania klas reprezentacyjnych łącznie z wymaganymi adnotacjami – mimo, że jest to proces zdecydowanie łatwiejszy, niż ręczne uzupełnianie pliku wejściowego o odpowiednią ilość znaków, w dalszym ciągu łatwo błędnie zdefiniować długość i tym samym uniemożliwić poprawną zamianę formatu.

5.4. Porównanie implementacji

Poniższe rozdziały omawiają podobieństwa i różnice w pracy z oboma językami programowania, zarówno w przypadku tworzenia nowego projektu jak i rozwijania już istniejącego systemu.

5.4.1. Łatwość implementacji

Z uwagi na ilość narzędzi, bibliotek oraz obszerną dokumentację dostępną dla języka Java, implementacja systemu w tym języku jest nieporównywalnie łatwiejsza niż ta w COBOL. W przypadku chęci nauczania się starszego z języków, bardzo ciężko znaleźć jakiekolwiek informacje na jego temat. Wynika to z wspomnianego wcześniej kontekstu jego zastosowania – większość kodu napisanego w nim jest ściśle chronioną własnością firm.

Ponadto Java zapewnia możliwość automatycznego wygenerowania struktury bazy danych za pośrednictwem biblioteki *Hibernate*. Na podstawie odpowiednio zdefiniowanych klas reprezentujących tabele, eliminowana jest konieczność inicjalizowania struktury za pomocą SQL. COBOL nie zapewnia żadnego zbliżonego rozwiązania; większość komunikacji z bazą odbywa się poprzez embedded SQL i konieczne jest modyfikowanie kodu źródłowego w przypadku zmiany silnika.

Większość problemów związanych ze starszym z języków w praktyce jest mało odczuwalna. Jeżeli firma posiada dużą bazę kodu w COBOLu, prawdopodobnie korzysta z wielu innych narzędzi IBM, w tym ich silnika bazy danych Db2. Ze względu na ten fakt wiele firm musiałoby ponieść ogromne koszty, aby móc skorzystać z rozwiązań zapewnianych przez innych dostawców, więc nie jest to popularna praktyka.

W porównaniu Java zapewnia dużą elastyczność, a zmiana silnika bazy danych jest bardzo prosta pod warunkiem ograniczenia zastosowania specyficznych dla konkretnego dialektu SQL zapytań. Projekt zaimplementowany na potrzeby niniejszej pracy w fazie implementacji uruchamiany był zarówno przy użyciu bazy PostgreSQL w podejściu code first, jak i Microsoft SQL Server w podejściu database first. Nie wymagało to żadnych zmian w kodzie – wystarczyło uruchomić aplikację przy użyciu innego profilu.

5.4.2. Obszerność implementacji

Mimo faktu, że projekt w języku Java dostarcza więcej funkcjonalności, ilość kodu, której wymaga ich implementacja jest znacznie mniejsza. Wynika to z faktu, że dostępne jest wiele bibliotek i narzędzi, które w zdecydowanym stopniu ułatwiają pracę i dostarczają funkcje, które w języku COBOL trzeba napisać od zera.

W związku z powyższym implementacja w COBOL zawiera 3809 linii kodu:

- ORD000 – 537 linii,
- ORD001 – 678 linii,
- ORD002 – 109 linii,
- ORD003 – 103 linie,
- ORD004 – 74 linie,
- ORD005 – 80 linii,
- ORD006 – 81 linii,
- ORD010 – 464 linie,
- ORD020 – 172 linie,
- ORD030 – 140 linii,
- ORD040 – 140 linii,
- ORD050 – 149 linii,
- ORD060 – 135 linii,
- ORD070 – 172 linie,
- ORD100 – 330 linii,
- ORD110 – 208 linii,
- ORD150 – 237 linii.

Oczywiście ilość linii kodu nie jest najbardziej istotną metryką, jeśli chodzi o złożoność projektu; nie każda z nich jest wymagana i wiele z nich służy jedynie większej estetyce oraz czytelności kodu. Jednakże mimo swojej relatywnej prostoty można zauważyć, że projekt jest obszerny – COBOL wymaga samodzielnej implementacji wielu funkcji, które uznawane są za podstawowe w nowoczesnych językach programowania. W praktyce w każdej firmie posiadającej Mainframe moduły takie jak ORD100 są już dostępne w ramach powszechnych, współdzielonych funkcji. Ta specyfika środowiska również ma pozytywne aspekty – cały kod jest pod kontrolą i nie są stosowane żadne zewnętrzne biblioteki, co może być szczególnie ważne dla krytycznej infrastruktury biznesowej, w której niniejszy język znajduje najbardziej powszechne zastosowanie. Ponadto całkowicie unikane jest ryzyko związane z publicznie znanymi podatnościami obecnych w publicznych bibliotekach – nie jest wymagane stosowanie narzędzi kontrolujących ich wersje.

Zupełnie inaczej wygląda implementacja systemu w języku Java. Zawiera ona 2135 linii kodu, natomiast jeżeli brane pod uwagę powinny być również wszystkie biblioteki zastosowane w projekcie, faktyczny rozmiar całego rozwiązania znacznie rośnie. Nie ma też żadnej kontroli nad kodem bibliotek, które nie należą do firmy (czyli w praktyce - ich zdecydowana większość) – jeżeli ze względu na podatności trzeba zaktualizować wersję do takiej, w której została wprowadzona zmiana źle lub wcale nie działająca z zaimplementowanym systemem, konieczne jest wprowadzanie modyfikacji do kodu. Często wymaga to również rygorystycznego testowania takich zmian, szczególnie, jeśli dotyczą one jego krytycznych części – nierzadko błędy nie są proste do wykrycia i ukryte w gotowych, dostarczonych funkcjach.

W związku z powyższym, dla programisty współczesne języki takie jak Java są znacznie przyjaźniejsze w użyciu i umożliwiają zastosowanie wielu zewnętrznych narzędzi oraz udogodnień. W porównaniu język COBOL może wydawać się bardzo złym wyborem – jednak w praktyce bardzo duża ilość powszechnie potrzebnego kodu została już kiedyś napisana w firmie i może zostać wykorzystana ponownie w nowych projektach. Dodatkowo takie podejście daje gwarancję, że kod jest bezpieczny

i przystosowany dokładnie do jego przypadku użycia. Żadna firma nie zdecydowałaby się na zakup Mainframe, jeżeli nie posiada już ogromnej bazy kodu w COBOLu – jednak, jeżeli od wielu lat było to stosowane rozwiązanie i istnieje duża baza dostępnych modułów, całkowita zmiana technologii często nie ma finansowego i praktycznego uzasadnienia.

5.4.3. Rozwój aplikacji

Rozpoczynając nowy projekt, wybór języka COBOL można nazwać irracjonalnym. Jest on niewygodny, mało elastyczny, przestarzały – mimo faktu, że jest stale aktualizowany, IBM nie może wprowadzić żadnych gruntownych zmian z uwagi na ilość krytycznego kodu, który w dalszym ciągu jest intensywnie eksploatowany przez największe i najważniejsze firmy na świecie. Implementacja systemu na potrzeby niniejszej pracy trwała bardzo długo, zarówno z uwagi na konieczność samodzielnego dostarczenia większości funkcjonalności, jak i również przez brak ogólnodostępnych w internecie informacji na temat języka.

W praktyce, COBOL nie jest brany pod uwagę jako dostępna opcja w nowych projektach. Istnieje w starszych, działających od dawna systemach, takich jak między innymi aplikacje bankowe, ubezpieczeniowe czy medyczne. Większość kodu w nich napisana została w momencie, kiedy COBOL był wyborem wygodnym i rewolucyjnym; fakt, że w dalszym ciągu można trafić na moduł z lat 70, który z powodzeniem działa do dziś, pozytywnie świadczy o jego implementacji.

Utrzymywanie istniejącego już systemu zdecydowanie różni się od pisania całego kodu źródłowego od zera – brak dostępnych źródeł czy bibliotek jest znacznie mniejszym problemem, ponieważ osoba wprowadzająca do niego poprawki ma dostęp do zasobów rozwijanych w firmie od bardzo długiego czasu. W praktyce praca z COBOLEM wygląda dosyć podobnie do współczesnych języków; z uwagi na ogromną bazę kodu z większością zadań można sobie poradzić sprawdzając, jak inny programista rozwiązał zbliżony do naszego problem.

Środowisko Mainframe można porównać do „modułowego monolitu” – każdy program może mieć dostęp do całego systemu, nie jest wymagana komunikacja sieciowa, ale równocześnie aktualizowanie jednej części projektu nie wymaga przebudowywania i rekompilacji wszystkich jego części – jeżeli nie został zmieniony copybook komunikacyjny, wystarczy zaktualizować i wymienić tylko objęty zmianami moduł. Jeżeli jednak interfejs się zmienił, wymagana jest rekompilacja wszystkich programów, które z niego korzystają – nierzadko jest to duża ilość; mimo to sam proces wdrożenia pojedynczego modułu jest już relatywnie prostszy i nie wymaga zasadniczo żadnych zasobów, w przeciwieństwie do kosztownego i często zajmującego bardzo dużo czasu procesu przebudowywania dużych, monolitycznych aplikacji.

Z uwagi na powyższe, obydwa języki w dalszym ciągu mają swoje pozytywne strony we współczesnym środowisku informatycznym. Całkowite wyeliminowanie COBOLa byłoby bardzo kosztowne i pracochłonne; prawdopodobnie mogłoby również prowadzić do spadku stabilności krytycznych w społeczeństwie systemów – Mainframe jest platformą o bardzo wysokiej niezawodności, a problemy występują rzadziej niż w nowoczesnej infrastrukturze chmurowej. Aby zapewnić najwyższą jakość rozwiązania, firmy stosują podejście hybrydowe – środowisko Mainframe dalej jest obecne i zawiera krytyczne elementy systemu, a w nowych projektach wybierane są współczesne języki programowania, takie jak Java. Według badań IBM, 71% liderów działów technologicznych popiera tę perspektywę – Mainframe w dalszym ciągu ma swoje miejsce w biznesie [58].

6. Testy zaimplementowanych rozwiązań

Niniejsze rozdziały omawiają testy, które zostały przeprowadzone na obydwu implementacjach systemu obsługi zamówień.

6.1. Testy wydajnościowe – czas wykonania

W celu porównania obydwu implementacji, przeprowadzone zostały pomiary wydajności wszystkich funkcjonalności zapewnionych przez systemy. Należy podkreślić, że z uwagi na wykorzystanie projektu GnuCOBOL do kompilacji, wyniki uzyskane przez ten system są w praktyce wynikiem wykonania programów przetłumaczonych z COBOL na C. To rozwiązanie nie jest jednak całkowicie odizolowane od rzeczywistości – firmy, które posiadają dużą bazę kodu źródłowego w COBOLu, ale chcą zaprzestać użytkowania Mainframe'ów, wybierają ten projekt jako moment przejściowy [59].

Tabela 2 przedstawia wyniki pomiaru czasu wykonania każdej z wymaganych funkcjonalności obydwu zaimplementowanych systemów, przy uruchamianiu ich jako pierwsza czynność po starcie aplikacji.

*Tabela 2. Wyniki przeprowadzonych testów wydajnościowych – pierwsze wywołanie po uruchomieniu aplikacji.
Źródło: opracowanie własne*

Testowana funkcjonalność	COBOL [ms]	Java [ms]
Masowa aktualizacja zamówień (1 000 rzędów)	23 318	1 081
Masowa aktualizacja zamówień (10 000 rzędów)	230 026	53 197
Masowa aktualizacja zamówień (100 000 rzędów)	2 288 201	520 291
Dodanie nowego zamówienia	1 138	457
Pobranie szczegółów zamówienia	724	76
Pobranie listy zamówień (brak filtrów)	509	283
Pobranie listy zamówień (filtr zakresowy)	254	390
Pobranie listy zamówień (wszystkie filtry)	302	182
Dodanie produktu do zamówienia	966	173
Dodanie 10 produktów do zamówienia	228	343
Usunięcie produktu z zamówienia	689	155

Jak widać na tabeli 7, system zaimplementowany w języku COBOL w większości testów wypada znacząco gorzej niż jego ekwiwalent w Java.

Najbardziej odczuwalną różnicą jest przetwarzanie wsadowe – aby przetworzyć 100 000 rzędów, pierwszy system potrzebował ponad 38 minut, natomiast drugi – niecałe 9 minut. Do umieszczenia 1 000 rzędów różnica była 23-krotna. Pobranie szczegółów zamówienia implementacji w COBOL zajęło dziesięciokrotnie więcej czasu w porównaniu do Javy.

Podsumowując wyniki testów pierwszego wywołania, system w COBOL wypadł 5 razy gorzej niż system w Javie.

Tabela 3 zawiera średni wynik pomiaru czasu wykonania żądania na przestrzeni 10 zapytań, gdzie pierwsze z nich było pierwszą czynnością wykonaną przez system po jego starcie.

*Tabela 3. Wyniki przeprowadzonych testów wydajnościowych – średnia z dziesięciu wywołań po uruchomieniu aplikacji.
Źródło: opracowanie własne*

Testowana funkcjonalność	COBOL [ms]	Java [ms]
Dodanie nowego zamówienia	117,5	48,3
Pobranie szczegółów zamówienia	143,3	29,9
Pobranie listy zamówień (brak filtrów)	298,7	158,7
Pobranie listy zamówień (filtr zakresowy)	208,2	190,6
Pobranie listy zamówień (wszystkie filtry)	192,7	93,9
Dodanie produktu do zamówienia	138,2	64,5
Usunięcie produktu z zamówienia	136,7	47,4

Z uwagi na długi czas wykonania, pominięta została masowa aktualizacja zamówień.

W niniejszym zestawieniu wyraźnie widać, że obydwa systemy w większości przypadków wykonują zadania szybciej po odpaleniu ich wielokrotnie. Więcej korzyści zyskał system w Java – zapytania przyspieszyły 3,4-krotnie w porównaniu do 2,9-krotnego zwiększenia wydajności implementacji w COBOL.

Przy pominięciu testu masowej aktualizacji zamówień, dla zimnego startu system w COBOL jest 2,9-krotnie wolniejszy niż Java; w przypadku średniej z 10 uruchomień współczynnik ten spada do 2,5 raza. Ponadto mimo szybszego pierwszego wykonania pobrania listy zamówień przez implementację w starszym języku przy zimnym starcie, po wielokrotnym wysłaniu tego zapytania Java uzyskała znacząco lepszy wynik.

6.2. Testy wydajnościowe – zużycie procesora

Oprócz pomiaru czasu wykonania żądań, zmierzone zostało maksymalne zużycie procesora podczas wykonania kolejno wszystkich funkcjonalności biznesowych zapewnianych przez oba systemy.

Należy podkreślić, że trafniejszym porównaniem wydajności jest pomiar czasu wykonania. Jednak z uwagi na brak wsparcia dla operacji wsadowego tworzenia rzędów dla GnuCOBOL, znacząca część czasu, którego wymagała implementacja w COBOL, została wykorzystana do komunikacji z bazą danych.

Tabela 4 przedstawia maksymalny procent zużycia procesora przy wykonywaniu kolejnych operacji.

Tabela 4. Maksymalny procent zużycia procesora (Apple M1) podczas wykonania żądań. Źródło: opracowanie własne

Testowana funkcjonalność	COBOL [%]			Java [%]
	Proces [%]	Wrapper [%]	Suma [%]	
Pobranie listy zamówień (wszystkie filtry)	0,29	1	1,29	1
Pobranie listy zamówień (brak filtrów)	0,23	1	1,23	1
Pobranie listy zamówień (filtr zakresowy)	0,30	1	1,30	3
Pobranie szczegółów zamówienia	0,29	1	1,29	1
Dodanie nowego zamówienia	0,14	1	1,14	2
Dodanie produktu do zamówienia	0,43	1	1,43	2
Usunięcie produktu z zamówienia	0,39	1	1,39	1
Masowa aktualizacja zamówień (100 rzędów)	0,11	1	1,11	8
Masowa aktualizacja zamówień (1 000 rzędów)	0,35	1	1,35	9
Masowa aktualizacja zamówień (10 000 rzędów)	0,35	1	1,35	13

Zużycie procesora podczas wykonywania prostych operacji jest podobne w przypadku obydwu systemów. Znacząca różnica występuje w przypadku masowego wstawiania rzędów – w tym wypadku implementacja w COBOL potrzebuje podobnej ilości mocy obliczeniowej niezależnie od ilości danych; dla systemu w Java zauważalny jest wzrost zużycia.

Nie oznacza to jednak, że system w COBOL jest bardziej wydajny. Z uwagi na wspomniany wcześniej brak możliwości wsadowego wstawienia danych do bazy przy zastosowaniu GnuCOBOL sposób wykorzystywania zasobów różni się. Niższe zużycie wynika z rozłożenia żądania na wiele niewielkich operacji w czasie, natomiast Java obciąża system mocniej, ale przez znacznie krótszy czas.

6.3. Testy złożoności pamięciowej

Testy złożoności pamięciowej zostały wykonane w takim samym kontekście, jak testy wydajności. Ponownie należy podkreślić, że z uwagi na wykorzystanie projektu GnuCOBOL do kompilacji, wyniki uzyskane przez ten system są w praktyce wynikiem wykonania programów przetłumaczonych z COBOL na C.

Tabela 5 przedstawia wyniki pomiaru zużycia pamięci podczas wykonania każdej z wymaganych funkcjonalności obydwu zaimplementowanych systemów, przy uruchamianiu ich kolejno po starcie aplikacji.

Tabela 5. Maksymalne zużycie pamięci przez zaimplementowane systemy podczas uruchamiania kolejno zapewnionych funkcjonalności biznesowych. Źródło: opracowanie własne

Testowana funkcjonalność	COBOL [MB]			Java [MB]
	Proces [MB]	Wrapper [MB]	Suma [MB]	
Pobranie listy zamówień (wszystkie filtry)	20,5	26	46,5	72
Pobranie listy zamówień (brak filtrów)	20,6	25	45,6	74
Pobranie listy zamówień (filtr zakresowy)	20,6	24	44,6	75
Pobranie szczegółów zamówienia	20,5	28	48,5	76
Dodanie nowego zamówienia	20,6	32	52,6	83
Dodanie produktu do zamówienia	20,4	21	41,4	88
Usunięcie produktu z zamówienia	20,4	21	41,4	89
Masowa aktualizacja zamówień (100 rzędów)	21,4	36	57,4	93
Masowa aktualizacja zamówień (1 000 rzędów)	21,3	36	57,4	93
Masowa aktualizacja zamówień (10 000 rzędów)	21,3	36	57,4	95

Można zauważyć, że każde żądanie w systemie zaimplementowanym w COBOL zużywa stałą ilość pamięci – 20 MB. Wynika to z faktu, że większość napisanych programów została skompilowana w formie bibliotek, które są ładowane podczas wykonania głównego procesu ORD000.

Zarówno system napisany w Javie, jak i aplikacja pomocnicza dla systemu w COBOL obserwuje wzrost w zużytej pamięci wraz z liczbą wykonanych operacji. Ponieważ operacje dealokacji pamięci są wykonywane automatycznie przez *garbage collector*, zasoby nie są zwolnione od razu po zakończeniu przetwarzania żądania, w przeciwieństwie do całkowitego ich zwolnienia po zakończeniu procesu wykonania ORD000.

6.4. Kontekst przeprowadzonych testów

Wyniki testów mogą sugerować, że COBOL jest bardzo niewydajny. Czas wykonania jest zdecydowanie dłuższy niż ekwiwalent w języku Java. Warto jednak zauważyć, że z uwagi na niskie zużycie procesora podczas przetwarzania żądań dla systemu w starszym języku, długi czas wykonania nie zależy jedynie od jego niewydajności, ale również od mniejszej możliwości optymalizacji zapytań do bazy danych. System w Java zbiera listę zamówień do kolekcji w pamięci, następnie za pomocą jednego polecenia zapisuje je wszystkie. Zastosowany na potrzebę projektu transpiler GnuCOBOL nie udostępnia możliwości wsadowego umieszczenia wszystkich zadanych rekordów na raz; wysyłane jest więc wiele żądań do bazy danych, co znacząco wpływa na otrzymane wyniki.

Ponadto należy wziąć pod uwagę, że różnica w architekturze procesora, na którym zostały przeprowadzone testy bardzo mocno odbiega od procesora na Mainframe. Wykorzystany został transpiler, który nie jest narzędziem dostarczonym przez IBM, więc nie gwarantuje on najwyższej wydajności. Narzędzia zastosowane w przypadku kompilacji systemu w Java są standardowe i stosowane w środowisku profesjonalnym – są więc zdecydowanie bardziej ułożone w rzeczywistości.

7. Podsumowanie

W niniejszym rozdziale przedstawione zostały wnioski wyciągnięte w związku z opracowaniem systemów zarządzania zamówieniami. Omówione są różnice między implementacjami, możliwości ich zastosowania oraz rozwoju.

7.1. Porównanie zaimplementowanych rozwiązań

Obydwa zaimplementowane systemy, pomimo dużych różnic w wykorzystanych technologiach, zostały zaprojektowane w zbliżony do siebie sposób. Oba rozwiązania ściśle trzymają się architektury warstwowej zgodnej z konwencją Single Responsibility. Dzięki temu, że każdy element projektów jest odpowiedzialny za tylko jedno zadanie, systemy są modularne i zdecydowanie ułatwiają ewentualne wprowadzanie zmian w dowolnym ich fragmencie.

Implementacja w COBOL jest bardzo długa, proceduralna; każda funkcjonalność wymaga samodzielnego napisania wielu podstawowych funkcjonalności, które są powszechnie dostępne w nowszych językach.

Słabsze wyniki przeprowadzonych testów w przypadku implementacji wykorzystującej GnuCOBOL nie są jednoznaczne z absolutną przewagą systemu Java. Ich zastosowania są znacząco różne; wykorzystanie GnuCOBOL jako narzędzie przejściowe w celu modernizacji istniejących systemów może okazać się niezastąpionym sposobem bezbolesnej migracji między Mainframe a nowszymi technologiami. Mniejsza wydajność widoczna była przede wszystkim w przypadku operacji wsadowych, które działają poza godzinami biznesowymi – najczęściej w nocy. Jeżeli czas ich wykonania okazałby się zbyt długi i przetworzenie danych wykraczałoby poza dostępne okno, dobrym rozwiązaniem jest przeniesienie i konsekwentne przepisywanie funkcjonalności zapewnianych przez moduły online, w przypadku których różnica była znacznie mniejsza.

Java jest językiem wysoce popularnym, szeroko wykorzystywanym w środowisku biznesowym, a implementowanie systemów jest znacznie szybsze i wygodniejsze. Przepisanie starych programów w COBOL na ten język z pewnością jest dobrym celem długoterminowym – systemy są łatwiejsze w utrzymaniu, mogą skorzystać z nowoczesnych metod optymalizacji wydajności takich jak korzystanie z *cache*, elastyczność zaimplementowanych rozwiązań jest większa, a ilość programistów znających tę technologię nie spada z uwagi na zaawansowany średni wiek.

Obydwa języki mają swoje miejsce w środowisku informatycznym; Mainframe i COBOL od lat wykorzystywane są przez najważniejsze instytucje i rzadko zawodzą. Całkowite ich wyparcie z pewnością przyniesie zarówno pozytywne, jak i negatywne skutki.

7.2. Zastosowanie projektów w praktyce

Niniejsze rozdziały omawiają możliwość wykorzystania projektów zaimplementowanych na potrzeby pracy w praktyce.

7.2.1. COBOL

Z uwagi na omówioną wcześniej specyfikę języka COBOL i jego trudność w użyciu, projekt nie stanowi bazy dla nowych systemów. W ostatnich latach jednak powszechną praktyką w firmach stosujących środowisko Mainframe jest modularyzacja i modernizacja istniejących już projektów.

Napisany na potrzeby pracy kod ściśle podąża za filozofią modularności – każdy program posiada tylko jedną odpowiedzialność, kod jest bardzo prosty do ponownego wykorzystania, a podział na konkretnie zdefiniowane warstwy umożliwia bezbolesną aktualizację systemu w przypadku zmiany wymagań.

Przepisanie długowiecznego systemu, w którym programy składają się z tysięcy linii kodu, dokumentacja nie istnieje, a logika biznesowa nie jest znana żadnemu z czynnych zawodowo pracowników, nie jest łatwym zadaniem. Mimo to, niniejszy projekt może służyć jako przykład, jak taki przepisany system mógłby wyglądać, i jakie korzyści może za sobą nieść modernizacja architektury. Najdłuższy z zaimplementowanych programów to około 500 linii kodu; w przypadku bardziej złożonej logiki biznesowej długość byłaby znacznie większa, jednak odnosząc się do własnego doświadczenia zawodowego, niektóre programy w COBOL sięgają dziesiątek tysięcy linii.

Wprowadzenie lepszego podziału zdecydowanie ułatwia czytanie i rozumienie kodu, zlokalizowanie istotnych reguł biznesowych oraz większą pewność we wprowadzanych modyfikacjach.

7.2.2. Java

System zaimplementowany w języku Java jest przykładem nowoczesnego interfejsu WebAPI oraz może stanowić bazę dla nowych systemów. Zachowano zasady podziału odpowiedzialności między różne rodzaje obiektów, zapewniono wysoką jakość automatycznie generowanej dokumentacji poprzez użycie odpowiednich adnotacji w kodzie, a cały projekt podąża za regułami REST API.

W obecnej formie projekt jest gotowy do zastosowania do celu zarządzania zamówieniami oraz umożliwia łatwy rozwój poprzez implementację nowych, lub modyfikację istniejących, funkcjonalności.

7.3. Możliwości rozwoju projektu

Niniejsze rozdziały omawiają możliwości rozwoju projektów zaimplementowanych na potrzeby pracy w praktyce.

7.3.1. COBOL

Z uwagi na modularną architekturę, rozwój projektu w COBOL nie stanowi problemu. Każda nowa funkcjonalność biznesowa oznaczałaby konieczność napisania nowego modułu z logiką biznesową, a w zależności od używanej tabeli, programy z warstwy komunikacji z bazą danych mogą być już zaimplementowane i gotowe do wykorzystania. Jeżeli zasobem byłby obiekt inny niż zamówienie, zalecane byłoby utworzenie nowego programu warstwy interfejsów podobnego do ORD000 – podział zbliżony jest do konwencji znanych z implementacji w języku Java.

Ponadto, dobrym kierunkiem rozwoju całego systemu byłoby napisanie kodu automatycznie generującego klasę reprezentującą copybook na podstawie dostarczonej jego zawartości. Obecne rozwiązanie jest wrażliwe na błędy programisty, a ponieważ ściśle trzymanie się długości w formacie komunikacji z Mainframe jest bardzo istotne, generacja klas wyeliminowałaby możliwość pomyłki i tym samym zapewniła większą niezawodność systemu.

7.3.2. Java

System zaimplementowany w Javie można dowolnie rozszerzać o nowe funkcjonalności biznesowe, może też stanowić pojedynczy serwis udostępniający funkcjonalności związane z zamówieniami w większym systemie mikroserwisów obsługującym firmy zajmujące się sprzedażą.

Dobrym kierunkiem rozwoju byłaby automatyczna generacja raportów dotyczących sprzedaży na podstawie danych w systemie lub dostarczonych plików CSV. Pozwoliłoby to na obserwację zachowania klientów zarówno historycznie, jak i na podstawie konkretnego wycinka danych.

8. Bibliografia

- [1] Open Mainframe Project, Phil Teplitzky, „COBOL: a Demographic Disaster in the Making”, dostępne: <https://lists.openmainframeproject.org/g/wg-cobol/attachment/113/1/systemsjournalcobolsept172020.pdf> [Dostęp 20 stycznia 2026].
- [2] Computer World, Grzegorz Stech, „ZUS ograniczy udział mainframe’ów”, dostępne: <https://www.computerworld.pl/article/2143170/zus-ograniczy-udzial-mainframeow.html> [Dostęp 20 stycznia 2026].
- [3] Zakład Ubezpieczeń Społecznych, „Nowe serwery – szybsza obsługa”, dostępne: <https://www.zus.pl/-/nowe-serwery-szybsza-obsługa> [Dostęp 20 stycznia 2026].
- [4] IBM, Rina Diane Caballar i Cole Stryker, „What is COBOL modernization?”, dostępne: <https://www.ibm.com/think/topics/cobol-modernization> [Dostęp 20 stycznia 2026].
- [5] IBM, High Level Assembler and Toolkit Feature, dostępne: <https://www.ibm.com/products/high-level-assembler-and-toolkit-feature> [Dostęp 19 kwietnia 2026].
- [6] mBank, Joanna Duda, „mBank wyznacza standardy w Europie. Prestiżowa nagroda Forrester’a za przełomową transformację technologiczną”, dostępne: <https://pl.media.mbank.pl/428899-mbank-wyznacza-standardy-w-europie-prestizowa-nagroda-forrester-a-za-przelomowa-transformacje-technologiczna> [Dostęp 20 kwietnia 2026].
- [7] mBank, „Pierwsze urodziny mBanku”, dostępne: <https://pl.media.mbank.pl/47241-pierwsze-urodziny-mbanku> [Dostęp 20 kwietnia 2026].
- [8] PKO Bank Polski, „GRUPA KAPITAŁOWA PKO BANKU POLSKIEGO SA – RYS HISTORYCZNY”, dostępne: <https://raportroczny2015.pkobp.pl/pl/grupa-kapitalowa-pko-banku-polskiego-sa-rys-historyczny.html> [Dostęp 20 kwietnia 2026].
- [9] Rocket Software, Que Mangus, „The Mainframe Turns 60: A Milestone in Computing History”, dostępne: <https://www.rocketsoftware.com/en-us/insights/mainframe-turns-60> [Dostęp: 20 kwietnia 2026].
- [10] Free Software Foundation, GnuCOBOL, dostępne: <https://gnucobol.sourceforge.io> [Dostęp 19 kwietnia 2026].
- [11] Free Software Foundation, GNU General Public License, dostępne: <https://www.gnu.org/licenses/gpl-3.0.html> [Dostęp 19 kwietnia 2026].
- [12] Free Software Foundation, Brian Tiffin, „Will I be amazed by GnuCOBOL?”, dostępne: <https://gnucobol.sourceforge.io/faq/index.html#will-i-be-amazed-by-gnucobol> [Dostęp 19 kwietnia 2026].
- [13] Free Software Foundation, „GCC, the GNU Compiler Collection”, dostępne: <https://gcc.gnu.org> [Dostęp 19 kwietnia 2026].
- [14] Free Software Foundation, Brian Tiffin, „What is GnuCOBOL?”, dostępne: <https://gnucobol.sourceforge.io/faq/index.html#what-is-gnucobol> [Dostęp 19 kwietnia 2026].
- [15] IBM, Chrystal R. China i Michael Goodwin, „What is COBOL?”, dostępne: <https://www.ibm.com/think/topics/cobol> [Dostęp 2 maja 2026].
- [16] Harvard University, „About the Mark I”, dostępne: <https://chsi.harvard.edu/harvard-ibm-mark-1-about> [Dostęp 2 maja 2026].
- [17] United States Census Bureau, „UNIVAC I”, dostępne: <https://www.census.gov/about/history/bureau-history/census-innovations/technology/univac-i.html> [Dostęp 2 maja 2026].

- [18] Britannica, Michael R. Swaine, "ENIAC computer", dostępne: <https://www.britannica.com/technology/ENIAC> [Dostęp 2 maja 2026].
- [19] Computer History Museum, „UNIVAC”, dostępne: <https://www.computerhistory.org/revolution/early-computer-companies/5/100> [Dostęp 2 maja 2026].
- [20] IBM, „The IBM 700 Series”, dostępne: <https://www.ibm.com/history/700> [Dostęp 2 maja 2026].
- [21] Columbia University Computing History, „The IBM 701 Defense Calculator”, dostępne: <https://www.columbia.edu/cu/computinghistory/701.html> [Dostęp 2 maja 2026].
- [22] IBM, „The IBM 700 Series”, dostępne: <https://www.ibm.com/history/700> [Dostęp 19 maja 2026].
- [23] IEEE Spectrum, James W. Cortada, “Building the System/360 Mainframe nearly destroyed IBM”, dostępne: <https://spectrum.ieee.org/building-the-system360-mainframe-nearly-destroyed-ibm> [Dostęp 2 maja 2026].
- [24] This Day in Tech History, Marcel Brown, “IBM System 360 changes the industry forever”, dostępne: <https://thisdayintechhistory.com/04/07/ibm-system-360-changes-the-industry-forever/> [Dostęp 2 maja 2026].
- [25] Wikimedia, Sandstein, dostępne: https://commons.wikimedia.org/wiki/File:IBM_system_360-50_console_-_MfK_Bern.jpg [Dostęp 19 maja 2026].
- [26] IBM, „IBM z/16”, dostępne: <https://www.ibm.com/products/z16> [Dostęp 2 maja 2026].
- [27] IBM, „The new IBM z16: Creating a Bright Future for Hybrid Cloud and AI with Groundbreaking Innovation”, dostępne: <https://www.ibm.com/new/announcements/ibm-z16> [Dostęp 19 maja 2026].
- [28] IBM, Chuck Boyer, “The 360 Revolution”, dostępne: <https://www.ibm.com/support/pages/zvm/history/360rev.pdf> [Dostęp 2 maja 2026].
- [29] IBM, LSPR, dostępne: <https://www.ibm.com/support/pages/ibm-z-lspr-itr-zosv2r4#ibmz16A01> [Dostęp 2 maja 2026].
- [30] IBM, „What is TSO?”, dostępne: <https://www.ibm.com/docs/en/zos-basic-skills?topic=interfaces-what-is-tso> [Dostęp 4 maja 2026].
- [31] IBM, „What is ISPF?”, dostępne: <https://www.ibm.com/docs/en/zos-basic-skills?topic=interfaces-what-is-ispf> [Dostęp 4 maja 2026].
- [32] IBM Mainframer, „ISPF Panel”, dostępne: <https://www.ibmmainframer.com/tso-ispf/ispf-panel/> [Dostęp 19 maja 2026].
- [33] IBM, Chrystal R. China i Michael Goodwin, „What is COBOL?”, dostępne: <https://www.ibm.com/think/topics/cobol> [Dostęp 23 stycznia 2026].
- [34] History of Information, Jeremy Norman, „Grace Hopper and Colleagues Introduce COBOL”, dostępne: <https://www.historyofinformation.com/detail.php?id=778> [Dostęp 23 stycznia 2026].
- [35] Tristan Davey's Punch Card Archive, dostępne: <https://punchcards.tristandavey.com/cards/00019/> [Dostęp 19 maja 2026].
- [36, 37, 38] IBM, „Reference format”, dostępne: <https://www.ibm.com/docs/en/i/7.4.0?topic=structure-reference-format> [Dostęp 23 stycznia 2026].
- [39] IBM, „Reserved words”, dostępne: <https://www.ibm.com/docs/en/cobol-linux-x86/1.1.0?topic=appendixes-reserved-words> [Dostęp 23 stycznia 2026].
- [40] Tokyo System House co., repozytorium Open-COBOL-ESQL, dostępne: <https://github.com/opensourcecobol/Open-COBOL-ESQL> [Dostęp 20 kwietnia 2026].

- [41] Java Pro, Simon Ritter, „25 VERSIONS IN 30 YEARS: A BRIEF HISTORY OF THE JAVA LANGUAGE”, dostępne: <https://javapro.io/2026/01/20/25-versions-in-30-years-a-brief-history-of-the-java-language/> [Dostęp 4 czerwca 2026].
- [42] TIOBE, „TIOBE Index for May 2026”, dostępne: <https://www.tiobe.com/tiobe-index/> [Dostęp 4 czerwca 2026].
- [43] IEEE Spectrum, Stephen Cass, „The Top Programming Languages 2025”, dostępne: <https://spectrum.ieee.org/top-programming-languages-2025> [Dostęp 4 czerwca 2026].
- [44] RedMonk, Stephen O’Grady, “The RedMonk Programming Language Rankings: January 2026”, dostępne: <https://redmonk.com/sograde/2026/04/14/language-rankings-1-26/> [Dostęp 4 czerwca 2026].
- [45] Spring, “Spring Boot”, dostępne: <https://spring.io/projects/spring-boot> [Dostęp 4 czerwca 2026].
- [46] PostgreSQL – dokumentacja, dostępne: <https://www.postgresql.org/docs/> [Dostęp 4 czerwca 2026].
- [47] DB-Engines, „DB-Engines Ranking”, dostępne: <https://db-engines.com/en/ranking> [Dostęp 4 czerwca 2026].
- [48] Postman, dostępne: <https://www.postman.com> [Dostęp 4 czerwca 2026].
- [49] IBM, „Introduction to CICS”, dostępne: <https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-introduction-cics> [Dostęp 20 kwietnia 2026].
- [50] Postman, “What Is a REST API? Examples, Uses and Challenges”, dostępne: <https://blog.postman.com/rest-api-examples/> [Dostęp 4 czerwca 2026].
- [51] Mariusz Trzaska: Modelowanie i implementacja systemów informatycznych 2.0. Rok 2025. Stron 487. ISBN 978-83-976442-0-5.
- [52] IBM, „System address spaces and the master scheduler”, dostępne: <https://www.ibm.com/docs/en/zos-basic-skills?topic=storage-system-address-spaces-master-scheduler> [Dostęp 2 czerwca 2026].
- [53] Free Software Foundation, GnuCOBOL, dostępne: <https://gnucobol.sourceforge.io> [Dostęp 20 kwietnia 2026].
- [54] Tokyo System House co., repozytorium Open-COBOL-ESQL, dostępne: <https://github.com/opensourcecobol/Open-COBOL-ESQL> [Dostęp 20 kwietnia 2026].
- [55] Free Software Foundation, GCC, dostępne: <https://gcc.gnu.org> [Dostęp 20 kwietnia 2026].
- [56] OpenJDK 21, dostępne: <https://openjdk.org/projects/jdk/21/> [Dostęp 7 czerwca 2026].
- [57] Apache Maven Project, Maven 3.9.12, dostępne: <https://maven.apache.org/docs/3.9.12/release-notes.html> [Dostęp 7 czerwca 2026].
- [58] IBM, John Granger, „Cloud or Mainframe? The answer is both”, dostępne: <https://newsroom.ibm.com/Cloud-or-Mainframe-The-Answer-is-Both> [Dostęp 7 czerwca 2026].
- [59] The New Stack, Joab Jackson, “20 Years in the Making, GnuCOBOL Is Ready For Industry”, dostępne: <https://thenewstack.io/20-years-in-the-making-gnucobol-is-ready-for-industry/> [Dostęp 13 czerwca 2026].

9. Dodatki

9.1. Spis rysunków

Rysunek 1. IBM Model 701. Źródło: [22].....	7
Rysunek 2. Model System/360. Źródło: [25]	8
Rysunek 3. IBM z/16. Źródło: [27]	9
Rysunek 4. ISPF Primary Option Menu. Źródło: [32]	10
Rysunek 5. Karta perforowana COBOL. Źródło: [35]	11
Rysunek 6. Struktura danych w copybooku. Źródło: opracowanie własne.....	13
Rysunek 7. Interfejs aplikacji Postman z przykładowym żądaniem HTTP. Źródło: opracowanie własne..	19
Rysunek 8. Diagram związków encji dla zaprojektowanego systemu. Źródło: opracowanie własne.....	21
Rysunek 9. Warstwowa architektura systemu w języku COBOL. Źródło: opracowanie własne	22
Rysunek 10. Przetwarzanie żądania przez Mainframe. Źródło: opracowanie własne.....	24
Rysunek 11. Przetwarzanie żądania przez zaimplementowany system. Źródło: opracowanie własne	24
Rysunek 12. Diagram wywołań funkcji BATCH-PROCESS. Źródło: opracowanie własne	27
Rysunek 13. Diagram wywołań funkcji GET-ALL. Źródło: opracowanie własne	27
Rysunek 14. Diagram wywołań funkcji GET-BY-ID. Źródło: opracowanie własne	28
Rysunek 15. Diagram wywołań funkcji ADD-NEW-ORDER. Źródło: opracowanie własne	28
Rysunek 16. Diagram wywołań funkcji ADD-TO-ORDER. Źródło: opracowanie własne	29
Rysunek 17. Diagram wywołań funkcji REMOVE-FROM-ORDER. Źródło: opracowanie własne.....	29
Rysunek 18. Struktura sekcji w programie ORD000. Źródło: opracowanie własne.....	38
Rysunek 19. Diagram wywołań funkcji masowej aktualizacji zamówień. Źródło: opracowanie własne...	72
Rysunek 20. Diagram wywołań funkcji pobrania listy zamówień. Źródło: opracowanie własne.....	72
Rysunek 21. Diagram wywołań funkcji pobrania szczegółów zamówienia. Źródło: opracowanie własne	73
Rysunek 22. Diagram wywołań funkcji dodania nowego zamówienia. Źródło: opracowanie własne	74
Rysunek 23. Diagram wywołań funkcji dodania produktu do zamówienia. Źródło: opracowanie własne.	74
Rysunek 24. Diagram wywołań funkcji usunięcia produktu z zamówienia. Źródło: opracowanie własne	75

9.2. Spis tabel

Tabela 1. Konwencja nazewnictwa w projekcie. Źródło: opracowanie własne	23
Tabela 2. Wyniki przeprowadzonych testów wydajnościowych – pierwsze wywołanie po uruchomieniu aplikacji. Źródło: opracowanie własne	92

Tabela 3. Wyniki przeprowadzonych testów wydajnościowych – średnia z dziesięciu wywołań po uruchomieniu aplikacji. Źródło: opracowanie własne	93
Tabela 4. Maksymalny procent zużycia procesora (Apple M1) podczas wykonania żądań. Źródło: opracowanie własne	94
Tabela 5. Maksymalne zużycie pamięci przez zaimplementowane systemy podczas uruchamiania kolejno zapewnionych funkcjonalności biznesowych. Źródło: opracowanie własne.....	95

9.3. Spis listingów

Listing 1. Wariant A warunku IF. Źródło: opracowanie własne	12
Listing 2. Wariant B warunku IF. Źródło: opracowanie własne.....	12
Listing 3. Struktura danych w copybooku. Źródło: opracowanie własne	13
Listing 4. Dołączanie struktury copybook w programie COBOL. Źródło: opracowanie własne	13
Listing 5. Struktura copybook'a - legenda. Źródło: opracowanie własne.....	14
Listing 6. Struktura copybook'a. Źródło: opracowanie własne.....	14
Listing 7. Zastosowanie klauzuli OF. Źródło: opracowanie własne.....	14
Listing 8. Zastosowanie zmiennej poziomu 66 RENAMES. Źródło: opracowanie własne.....	15
Listing 9. Zastosowanie zmiennych poziomu 88. Źródło: opracowanie własne	15
Listing 10. Metoda main w aplikacji Java. Źródło: opracowanie własne	17
Listing 11. Plik pom.xml. Źródło: opracowanie własne.....	18
Listing 12. Copybook ORD000AR - fragment. Źródło: opracowanie własne.....	30
Listing 13. ORD000 - PROCEDURE DIVISION. Źródło: opracowanie własne	31
Listing 14. ORD000 - sekcja READ-ARGS. Źródło: opracowanie własne.....	31
Listing 15. ORD000 - sekcja MAP-INPUT. Źródło: opracowanie własne	32
Listing 16. Poprawny format pliku wejściowego do programu ORD000. Źródło: opracowanie własne....	32
Listing 17. Niepoprawny format pliku wejściowego do programu ORD000. Źródło: opracowanie własne	32
Listing 18. ORD000 - sekcja VALIDATE-INPUT. Źródło: opracowanie własne.....	33
Listing 19. ORD000 - sekcja VALIDATE-ORD002. Źródło: opracowanie własne.....	34
Listing 20. ORD000 - sekcja VALIDATE-ORD004. Źródło: opracowanie własne.....	34
Listing 21. ORD000 - sekcja PROCESS-INPUT. Źródło: opracowanie własne	35
Listing 22. ORD000 - sekcja CALL-ORD002. Źródło: opracowanie własne	35
Listing 23. ORD000 - sekcja MAP-ORD002-INPUT. Źródło: opracowanie własne	36
Listing 24. ORD000 - sekcja MAP-ORD002-OUTPUT. Źródło: opracowanie własne	36
Listing 25. Plik wyjściowy z programu ORD000 po wykonaniu funkcji GET-ALL. Źródło: opracowanie własne	37

Listing 26. ORD001 - PROCEDURE DIVISION. Źródło: opracowanie własne	39
Listing 27. Copybook ORD001AR. Źródło: opracowanie własne.....	39
Listing 28. ORD001 - sekcja BATCH-PROCESS-ORDERS. Źródło: opracowanie własne.....	40
Listing 29. ORD001 - sekcja INITIALIZE-VARIABLES. Źródło: opracowanie własne.....	40
Listing 30. ORD001 - sekcja VALIDATE-FILE. Źródło: opracowanie własne.....	41
Listing 31. ORD001 - sekcja PROCESS-ORDERS. Źródło: opracowanie własne	41
Listing 32. ORD001 - sekcja CALL-ORD100-OPEN. Źródło: opracowanie własne.....	42
Listing 33. ORD001 - sekcja CALL-ORD100-NEXT. Źródło: opracowanie własne.....	43
Listing 34. Przykładowy fragment wejściowego pliku CSV. Źródło: opracowanie własne.....	43
Listing 35. Copybook ORDER001. Źródło: opracowanie własne	44
Listing 36. ORD001 - sekcja MAP-ORD100-OUTPUT. Źródło: opracowanie własne	44
Listing 37. ORD001 - sekcja VALIDATE-ORDER-DATA. Źródło: opracowanie własne	45
Listing 38. ORD001 - sekcja VALIDATE-ORDER-TYPE. Źródło: opracowanie własne	46
Listing 39. ORD001 - sekcja MAP-ORD020-INPUT. Źródło: opracowanie własne	46
Listing 40. ORD001 - sekcja MAP-ORD020-OUTPUT. Źródło: opracowanie własne	47
Listing 41. ORD001 - sekcja INSERT-NEW-ORDER. Źródło: opracowanie własne	48
Listing 42. ORD001 - sekcja MAP-ORD110-INPUT. Źródło: opracowanie własne.....	48
Listing 43. ORD001 - sekcja VALIDATE-PRODUCT-ORDER-DATA. Źródło: opracowanie własne	49
Listing 44. ORD001 - sekcja INSERT-PRODUCT-ORDER. Źródło: opracowanie własne.....	49
Listing 45. ORD001 - sekcja END-PROGRAM. Źródło: opracowanie własne	50
Listing 46. ORD010 - PROCEDURE DIVISION. Źródło: opracowanie własne	50
Listing 47. ORD010 - sekcja INITIALIZE-VARIABLES. Źródło: opracowanie własne.....	51
Listing 48. ORD010 - sekcja CONNECT-TO-DATABASE. Źródło: opracowanie własne.....	52
Listing 49. ORD010 - sekcja GET-LIST. Źródło: opracowanie własne.....	53
Listing 50. ORD010 - sekcja DECLARE-ORDERS-CURSOR. Źródło: opracowanie własne.....	54
Listing 51. ORD010 - sekcja OPEN-ORDERS-CURSOR. Źródło: opracowanie własne.....	54
Listing 52. ORD010 - sekcja FETCH-ORDERS. Źródło: opracowanie własne.....	55
Listing 53. ORD010 - sekcja MAP-CURSOR-OUTPUT. Źródło: opracowanie własne.....	56
Listing 54. ORD010 - sekcja CLOSE-ORDERS-CURSOR. Źródło: opracowanie własne.....	57
Listing 55. ORD100 - PROCEDURE DIVISION. Źródło: opracowanie własne	58
Listing 56. ORD100 - sekcja INITIALIZE-VARIABLES. Źródło: opracowanie własne.....	59
Listing 57. ORD100 - sekcja OPEN-CSV. Źródło: opracowanie własne.....	60
Listing 58. ORD100 - sekcja DISCARD-HEADER-LINE. Źródło: opracowanie własne	60
Listing 59. ORD100 - sekcja READ-CSV-LINE. Źródło: opracowanie własne.....	61
Listing 60. ORD100 - sekcja GET-NEXT-ROW. Źródło: opracowanie własne	61

Listing 61. ORD100 - sekcja PARSE-CSV-LINE. Źródło: opracowanie własne	62
Listing 62. ORD100 - sekcja BUILD-ORDERS-REC. Źródło: opracowanie własne	63
Listing 63. ORD100 - sekcja CONVERT-TO-INTEGER. Źródło: opracowanie własne	64
Listing 64. ORD100 - sekcja CLOSE-CSV. Źródło: opracowanie własne	65
Listing 65. ORD110 - PROCEDURE DIVISION. Źródło: opracowanie własne	66
Listing 66. ORD110 - sekcja ADD-NEW-ORDER. Źródło: opracowanie własne	66
Listing 67. ORD110 - sekcja INSERT-NEW-ORDER. Źródło: opracowanie własne	67
Listing 68. ORD110 - sekcja MAP-OUTPUT. Źródło: opracowanie własne	68
Listing 69. Polecenia przygotowujące środowisko do kompilacji. Źródło: opracowanie własne	69
Listing 70. Polecenia do przetworzenia wstępnego programów komunikujących się z bazą danych. Źródło: opracowanie własne	69
Listing 71. Polecenia do kompilacji programów komunikujących się z bazą danych. Źródło: opracowanie własne	69
Listing 72. Polecenia do kompilacji programów niekomunikujących się z bazą danych. Źródło: opracowanie własne	70
Listing 73. Polecenie służące do uruchomienia programu. Źródło: opracowanie własne.....	70
Listing 74. OrdersController - fragment. Źródło: opracowanie własne	75
Listing 75. OrderService - fragment. Źródło: opracowanie własne	77
Listing 76. OrderTypes - enumeracja odpowiadająca tabeli Order_Type. Źródło: opracowanie własne....	78
Listing 77. OrdersRepository - fragment. Źródło: opracowanie własne	78
Listing 78. Klasa Order – fragment. Źródło: opracowanie własne	79
Listing 79. Order - metoda addProductOrders. Źródło: opracowanie własne	80
Listing 80. Rekord OrderDataDto. Źródło: opracowanie własne.....	80
Listing 81. CsvParser - fragment. Źródło: opracowanie własne	81
Listing 82. OrderMapper - fragment. Źródło: opracowanie własne.....	81
Listing 83. CobolExecutor - fragment. Źródło: opracowanie własne	83
Listing 84. CobolExecutor - metody executeCobolProgram i getProcess. Źródło: opracowanie własne...	84
Listing 85. GetOrderResponseCpy - fragment. Źródło: opracowanie własne	85
Listing 86. ORD003AR - fragment. Źródło: opracowanie własne.....	86
Listing 87. ProductDataResponseCpy - fragment. Źródło: opracowanie własne.....	86
Listing 88. CopybookParser - fragment. Źródło: opracowanie własne	87
Listing 89. Zastosowanie refleksji w metodzie extractCobolFieldMetadata. Źródło: opracownie własne .	88