POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania, Procesów Biznesowych i Baz Danych

Marek Dudziński
Nr albumu s26754

Zastosowanie architektury warstwowej oraz heksagonalnej w wytwarzaniu aplikacji internetowych

Praca magisterska
pod kierunkiem:

Dr inż. Mariusz Trzaska

Warszawa, Czerwiec, 2024

Streszczenie

Praca jest porównaniem dwóch wzorców architektonicznych, Heksagonalnej (*ang. Hexagonal Architecture*) oraz Warstwowej *(ang. Layered Architecture*), w kontekście ich zastosowania w projektach informatycznych. Pierwszy rozdział pełni rolę wprowadzenia teoretycznego, gdzie omawiane są fundamentalne zasady i założenia architektur. Kolejny rozdział poświęcony jest omówieniu technologii wykorzystanych w procesie implementacji projektu. W tym obszarze szczegółowo opisane są narzędzia, języki programowania, biblioteki i frameworki użyte do realizacji aplikacji webowej, co umożliwia lepsze zrozumienie kontekstu implementacyjnego. Następnie praca przedstawia dziedzinę projektu, w której aplikacja ma być zaimplementowana. Omawiane są główne wymagania biznesowe oraz funkcjonalności, które aplikacja ma obsługiwać, co stanowi istotny kontekst dla porównania architektur. W ramach projektu porównawczego stworzono aplikację webową, służącą do obsługi sklepu internetowego, przy czym zrealizowano dwie wersje aplikacji, odpowiadające analizowanym architekturom. Finalna część pracy skupia się na właściwym porównaniu obu implementacji pod kątem kilku kluczowych parametrów, takich jak jakość kodu, czytelność, złożoność rozwiązania oraz poziom ryzyka związanego z wdrożeniem każdej z opcji. Analiza ta umożliwia obiektywne wyodrębnienie zalet i wad każdej z architektur, co może być cenną wskazówką dla projektantów i programistów przy podejmowaniu decyzji architektonicznych w przyszłych projektach.

Słowa kluczowe: architektura heksagonalna, architektura warstwowa, inżynieria oprogramowania

Spis treści

1. Wstęp

Architektura oprogramowania odgrywa kluczową rolę w procesie projektowania i tworzenia systemów informatycznych. Wybór odpowiedniego wzorca architektonicznego ma istotny wpływ na jakość, skalowalność, utrzymanie oraz rozszerzalność aplikacji. Architektura warstwowa jest jednym z najpopularniejszych wzorców architektonicznych, który zakłada podział aplikacji na logiczne warstwy, z każdą warstwą odpowiedzialną za określone zadania. Najczęściej spotykane warstwy to warstwa prezentacji (interfejs użytkownika), warstwa logiki biznesowej oraz warstwa dostępu do danych. Taki podział jest intuicyjny i pozwala na lepsze zarządzanie zależnościami oraz separację odpowiedzialności. Niesie to jednak ze sobą ryzyko wystąpienia problemów ze skalowalnością systemu w przypadku jego rozwoju.. Jakość oprogramowania znacząco spada wraz z poszerzaniem domeny aplikacji o nową logikę biznesową, co może prowadzić do problemów implementacyjnych i utrzymaniowych systemu. Architektura heksagonalna, znana również jako architektura portów i adapterów, jest wzorcem, który promuje silne oddzielenie logiki biznesowej od szczegółów technicznych. Domena aplikacji jest nadrzędnym bytem, która nie jest zarządzana przez warstwę aplikacji, a jest jedynie przez nią używana. Niniejsza praca stanowi analizę i porównanie obu architektur pod kątem ich struktury, elastyczności, testowalności, łatwości utrzymania oraz skalowalności. Poprzez zidentyfikowanie różnic i podobieństw między nimi, będzie możliwa ocena, który wzorzec lepiej spełnia wymagania danego projektu.

1.1. Cel Pracy

Celem niniejszej pracy jest przeprowadzenie porównania oprogramowania opartego na dwóch różnych wzorcach architektonicznych: architekturze heksagonalnej oraz architekturze warstwowej. Obie aplikacje zostały skonstruowane z wykorzystaniem takiego samego, anemicznego modelu danych, co pozwoli na dokładniejsze porównanie samego procesu wytwarzania oprogramowania.

Głównym rezultatem porównania jest dostarczenie odpowiedzi na istotne pytania dotyczące zastosowania poszczególnych wzorców. Praca skupia się na identyfikacji sytuacji, w których architektura heksagonalna lub tradycyjna struktura trójwarstwowa są bardziej odpowiednie. Przeprowadzona analiza pozwala również na szczegółowe zrozumienie zalet i wad obu rozwiązań, biorąc pod uwagę takie czynniki jak elastyczność, skalowalność, łatwość utrzymania, testowalność i wydajność.

1.2. Metodologia przyjęta w pracy

Badanie porównawcze dwóch podejść do architektury oprogramowania zostało przeprowadzone za pomocą implementacji przykładowego projektu systemu zarządzania sklepem internetowym. W obu systemach zastosowano technologię Spring Framework w wersji 3, która oparta jest na języku Java. W ramach projektu wykorzystana została specyfikacja języka Java w wersji 17.

W celu zapewnienia spójności i jednolitości, obie aplikacje zostały zaimplementowane z wykorzystaniem narzędzia Swagger do tworzenia API. Dzięki temu, modele API w obu systemach są identyczne, co umożliwia bezpośrednie porównanie ich funkcjonalności i cech.

Do testów aplikacji, został wykorzystany framework Spock, który oparty jest na składni języka Groovy. Pozwala on na wygodne pisanie testów i asercji, co umożliwia kompleksowe sprawdzenie poprawności działania obu wersji systemu.

Dzięki wykorzystaniu tych narzędzi i technologii, porównanie dwóch podejść zostanie przeprowadzone w sposób jednolity i kompleksowy. Oba systemy zostaną poddane analizie pod kątem różnych czynników, takich jak wydajność, skalowalność, łatwość rozszerzania oraz czytelność kodu.

1.3. Oczekiwany rezultat pracy

W teorii, zastosowanie architektury heksagonalnej w projekcie może stanowić wyzwanie ze względu na potencjalną złożoność kodu oraz początkowe wrażenie nadmiernego skomplikowania. Jednakże korzyści płynące z jej implementacji stają się wyraźne, zwłaszcza w kontekście zaawansowanych projektów charakteryzujących się złożoną logiką biznesową oraz potrzebą integracji z zewnętrznymi usługami.

W przypadku prostych projektów, takich jak sklep internetowy w postaci monolitu, oczekiwane korzyści z zastosowania architektury heksagonalnej są niewielkie w porównaniu do nakładu pracy potrzebnego do jej implementacji. Jednak wersja projektu z rozbudowanym modelem oraz architekturą mikroserwisów może przynieść lepszy rezultat, poprzez wprowadzenie zależności zewnętrznych do analizowanego projektu oraz bardziej skomplikowaną logikę biznesową.

Oczekiwany wniosek z porównania wskazuje na przewagę architektury heksagonalnej w bardziej złożonych projektach w porównaniu do tradycyjnej architektury trójwarstwowej. Jest to spowodowane prostotą oraz przydatnością architektury warstwowej w mniejszych projektach. Ważnym aspektem jest także skalowalność projektów, gdzie architektura heksagonalna wykazuje większą elastyczność po zastosowaniu odpowiedniego szablonu projektowego.

1.4. Stan sztuki

Architektura warstwowa z anemicznym modelem danych jest jednym z najpopularniejszych form wytwarzania oprogramowania [1]. Znaczna część materiałów dydaktycznych dla początkujących programistów skupia się właśnie na niej, ponieważ jest intuicyjna i uczy dobrych nawyków, takich jak odseparowywanie warstw odpowiedzialnych za niezależne od siebie segmenty systemu. W odpowiedzi na ten trend powstaje coraz więcej artykułów, materiałów dydaktycznych i innych publikacji, które mają na celu zwrócenie uwagi programistów na inne sposoby wytwarzania oprogramowania. Założenia DDD, architektury heksagonalnej czy CQRS sięgają początku przełomu XX i XXI wieku [2,3] i były przez ten czas wykorzystywane, ale należały do mniejszości. W Internecie można znaleźć wiele przykładów implementacji poszczególnych architektur wraz z wymieniem ich wad i zalet, ale są to na ogół krótkie publikacje jedynie wprowadzające w tematykę i pozostawiające czytelnika do dalszego rozwinięcia tematu we własnym zakresie. W ramach poszukiwań materiałów dydaktycznych do zrealizowania tej pracy napotkano na pracę poruszające zagadnienia architektur DDD z wykorzystaniem odpowiedniego modelowania domenowego, który zderzano z anemicznym modelem danych w architekturze warstwowej, natomiast nie znaleziono porównania architektury heksagonalnej z archiktekturą warstwową z identycznym modelem bazodanowym. Takie porównanie pozwala na skupienie procesu porównawczego na procesie wytwarzania oprogramowania, a nie modelowania systemu.

2. Omawiane wzorce architektoniczne

Wzorce architektoniczne w procesie wytwarzania oprogramowania to sprawdzone, powtarzalne rozwiązania na poziomie architektury systemu, które pomagają w tworzeniu oprogramowania o wysokiej jakości, elastycznym i łatwym w zarządzaniu. Są to zbiory reguł i schematy umożliwiające stworzenie oprogramowania w uporządkowany sposób. Zadaniem architektów oprogramowania jest utrzymywanie jakości i konwencji rozwijanego rozwiązania w celu umożliwienia utrzymywania i dalszej pracy nad projektem. Cytując Roberta Martina potocznie zwanego Wujkiem Bobem „*The strategy behind that facilitation is to leave as many options open as possible, for as long as possible*.” [4]. Problemem nie jest, aby oprogramowanie działało zgodnie z oczekiwaniami, jest nim poprawna implementacja umożliwiające adaptację do zmian i rozszerzeń. Podczas tworzenia architektury rozwiązania informatycznego należy uwzględnić wiele czynników takich jak:

- komponenty aplikacji
- model domeny
- sposób komunikacji komponentów
- zewnętrzne zależności
- zewnętrzne integracje
- bezpieczeństwo systemu
- zachowanie w przypadku awarii
- zachowanie w przypadku zwiększonego obłożenia

Wszystkie te czynniki mają bezpośredni wpływ na ostateczny kształt aplikacji. Uwzględnienie ich wzajemnego wpływu na siebie oraz potencjalne zmiany i rozwój aplikacji w przyszłości stanowi wyznacznik jakości dostarczonego rozwiązania.

2.1. Architektura warstwowa

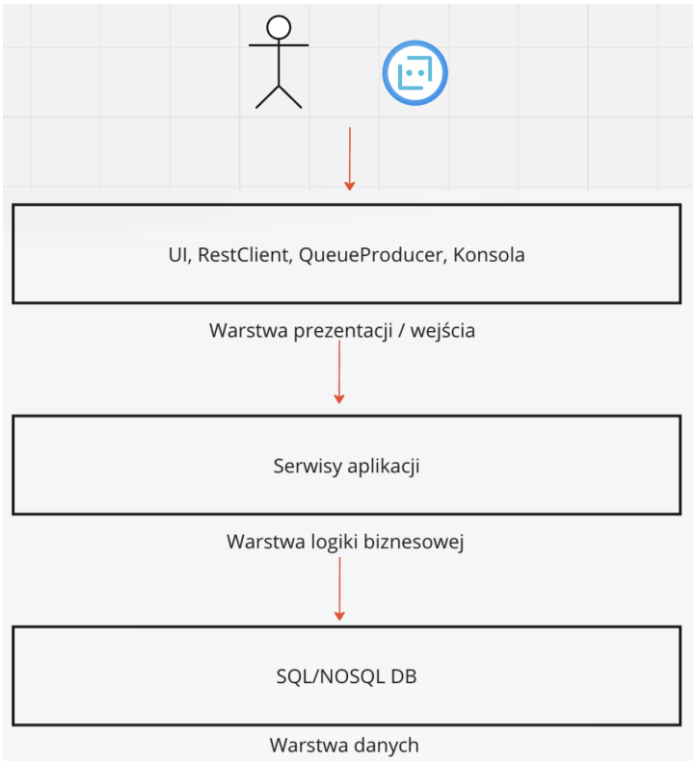
Architektury oprogramowania stanowią szablon systemu informatycznego. W dziedzinie informatyki często stosuje się analogie do codziennych obiektów lub zjawisk w celu ułatwienia zrozumienia złożonych koncepcji. Porównania takie mogą sprawić, że skomplikowane idee stają się bardziej intuicyjne i przystępne. Przykładem takiego podejścia jest architektura warstwowa, która polega na wyodrębnieniu różnych warstw odpowiedzialności w oprogramowaniu, co pozwala na jednoznaczny podział elementów systemu oraz ich funkcji.

W architekturze warstwowej system dzieli się na warstwy odpowiedzialności. Każdy komponent należy tylko do jednej warstwy, dzięki temu unika się przenikania odpowiedzialności elementów systemu. W klasycznym podejściu, najczęściej wyodrębnionymi warstwami są:

- prezentacji
- logiki biznesowej
- dostępu do danych
- bezpieczeństwa
- infrastruktury

Liczba warstw może być większa, a ich wybór oraz struktura zależą od decyzji architekta. Najczęściej stosowanym modelem architektury warstwowej jest podział trójwarstwowy, który obejmuje warstwę prezentacji, logiki biznesowej oraz dostępu do danych. Ten podział zapewnia fundamentalne oddzielenie odpowiedzialności za poszczególne funkcje systemu, co umożliwia lepsze zarządzanie kodem. Trójwarstwowa architektura jest często integrowana z architekturą mikroserwisową, która dzieli system na autonomiczne komponenty odpowiedzialne za specyficzne zadania. Dzięki temu system składa się z małych, niezależnych bloków, które komunikują się ze

sobą w sposób zorganizowany, co zwiększa elastyczność, ułatwia utrzymanie i rozwój systemu oraz poprawia jego odporność na awarie. Podział pojedynczego komponentu przedstawiono na Rys.1. Kolejne warstwy komunikują się ze sobą z zachowaniem hierarchii, czyli warstwa prezentacji nie może skomunikować się bezpośrednio z warstwą danych. Takie zachowanie świadczyłoby o błędzie projektowym.



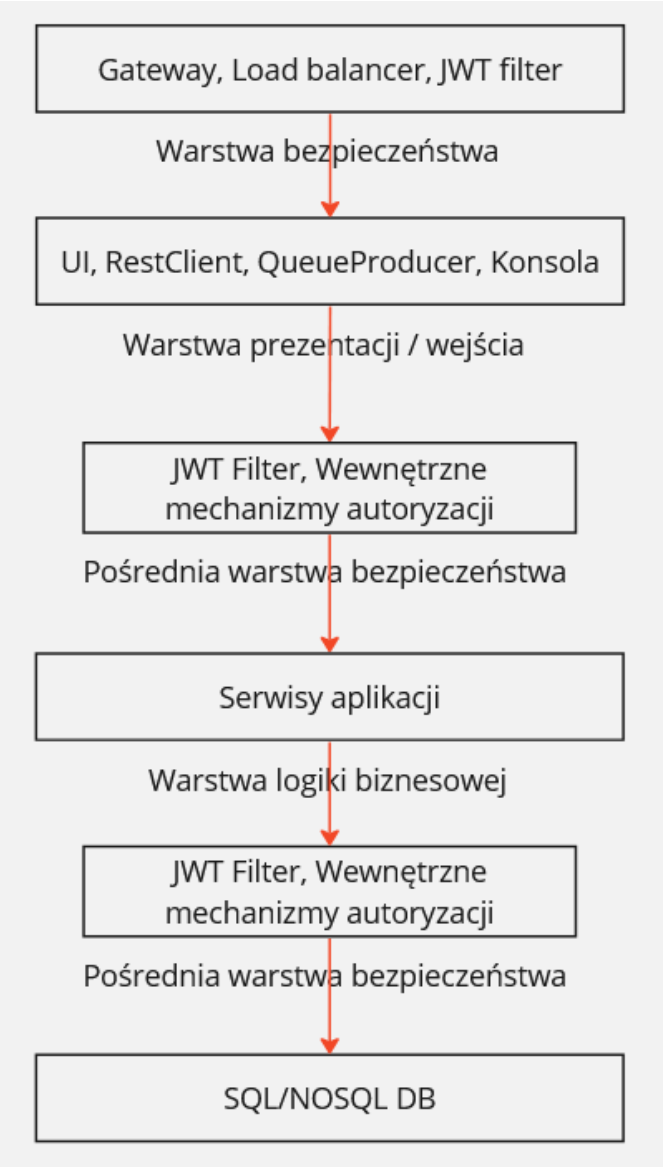
Rys. 1 Podział trójwarstwowy w architekturze warstwowej. Źródło: opracowanie własne.

Kolejnym fundamentalnym założeniem architektury warstwowej jest jednostronna komunikacja pomiędzy warstwami. Oznacza to, że warstwy niższe oczekują żądań od warstw wyższych i nie inicjują komunikacji w przeciwnym kierunku. Taki kaskadowy model interakcji gwarantuje, że procesy w systemie są uporządkowane i spójne. W praktyce oznacza to, że warstwa prezentacji, która jest odpowiedzialna za interakcję z użytkownikiem, wysyła żądania do warstwy logiki biznesowej. Warstwa logiki biznesowej, przetwarzając te żądania, komunikuje się z warstwą dostępu do danych w celu uzyskania lub modyfikacji informacji przechowywanych w systemie. Taka struktura pozwala na klarowny podział ról i obowiązków, co ułatwia zarówno rozwój, jak i utrzymanie systemu. Jednostronna komunikacja pomiędzy warstwami minimalizuje ryzyko sprzężenia zwrotnego, które mogłoby prowadzić do nieprzewidywalnych zachowań systemu. Każda warstwa ma jasno określoną odpowiedzialność i zna jedynie interfejs warstw, z którymi bezpośrednio współpracuje. Dzięki temu, modyfikacje w jednej warstwie mają ograniczony wpływ na pozostałe warstwy, co zwiększa elastyczność i modularność systemu.

Ponadto, takie podejście ułatwia testowanie i debugowanie aplikacji. Każda warstwa działa jako niezależny moduł, możliwe jest testowanie jej funkcjonalności w izolacji od innych warstw. To sprzyja szybszemu wykrywaniu i naprawianiu błędów, co przekłada się na wyższą jakość końcowego produktu.

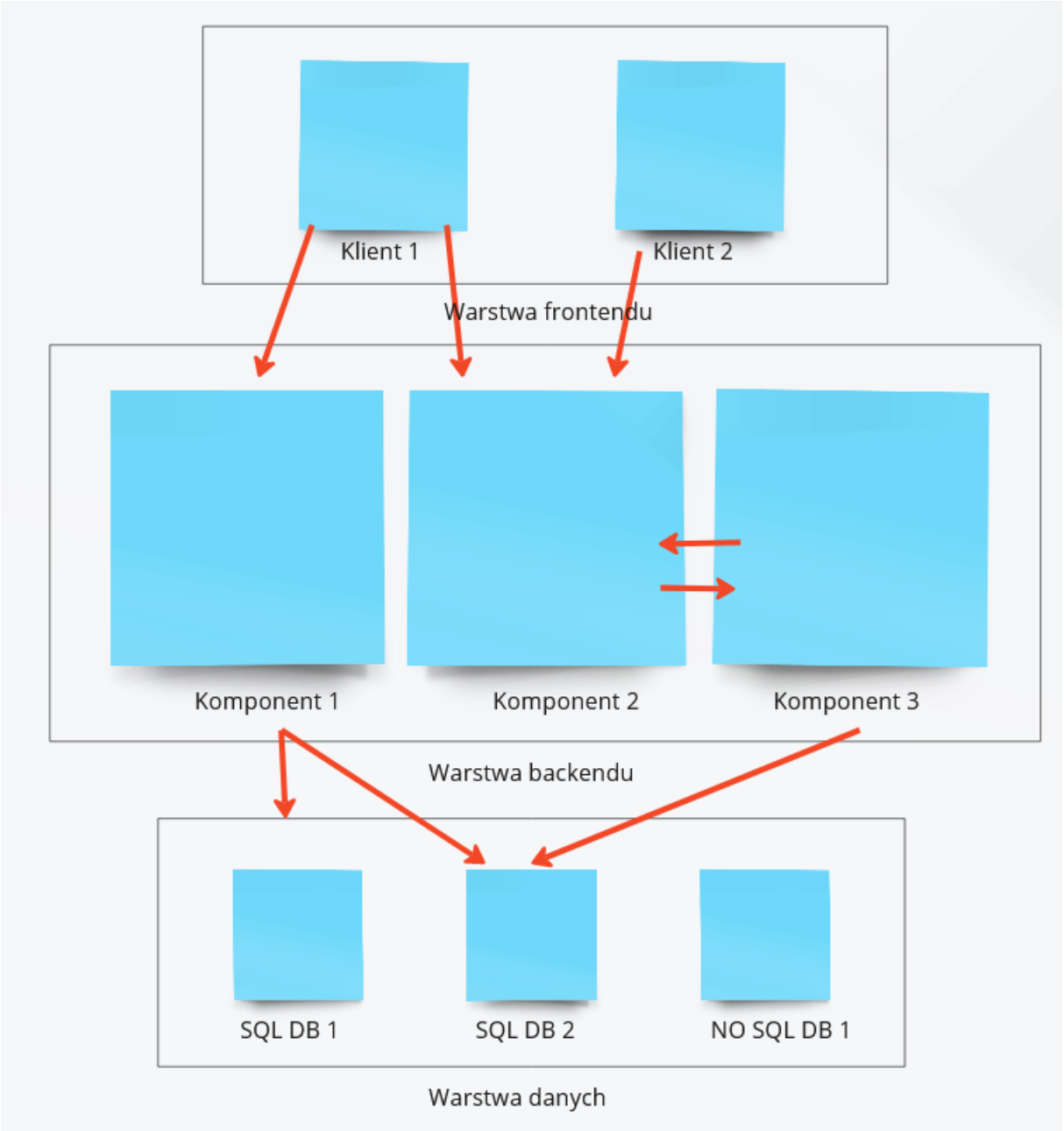
Podział trójwarstwowy nie zawsze jest wystarczający do zaspokojenia wszystkich wymagań systemu informatycznego. Na rys.2 przedstawiono rozszerzenie poprzedniego przykładu o warstwę bezpieczeństwa. Ta warstwa jest równie kluczowe w większości systemów co pozostałe, ale ze

względu na jej powszechną obecność, często jest pomijana w podstawowych diagramach architektury. Warstwa bezpieczeństwa skupia się na zapewnieniu ochrony danych oraz integralności i poufności informacji przetwarzanych przez system. Obejmuje mechanizmy autoryzacji, uwierzytelniania, szyfrowania danych oraz wykrywania i reagowania na zagrożenia. Włączenie tych dodatkowych warstw pozwala na bardziej kompleksowe zarządzanie systemem oraz lepsze zabezpieczenie przed różnorodnymi zagrożeniami. Ich obecność jest często przyjmowana jako standard, co może prowadzić do ich pominięcia w uproszczonych diagramach architektury, jednakże są one nieodzowne dla zapewnienia pełnej funkcjonalności i bezpieczeństwa systemu.



Rys. 2 Rozszerzony podział komponentu w architekturze warstwowej. Źródło: opracowanie własne.

Powyższe przykłady zostały przedstawione na poziomie pojedynczego komponentu. Wzorzec architektury warstwowej może być zastosowany zarówno na poziomie pojedynczego serwisu, jak i całego systemu. Na rysunku 3 zaprezentowano diagram komponentów systemu zaprojektowanego w architekturze warstwowej. Architektura warstwowa na poziomie pojedynczego serwisu umożliwia podział jego funkcjonalności na oddzielne warstwy, co ułatwia zarządzanie, rozwój i utrzymanie serwisu. Każda warstwa odpowiada za określone zadania, co zapewnia lepszą organizację kodu oraz separację odpowiedzialności.

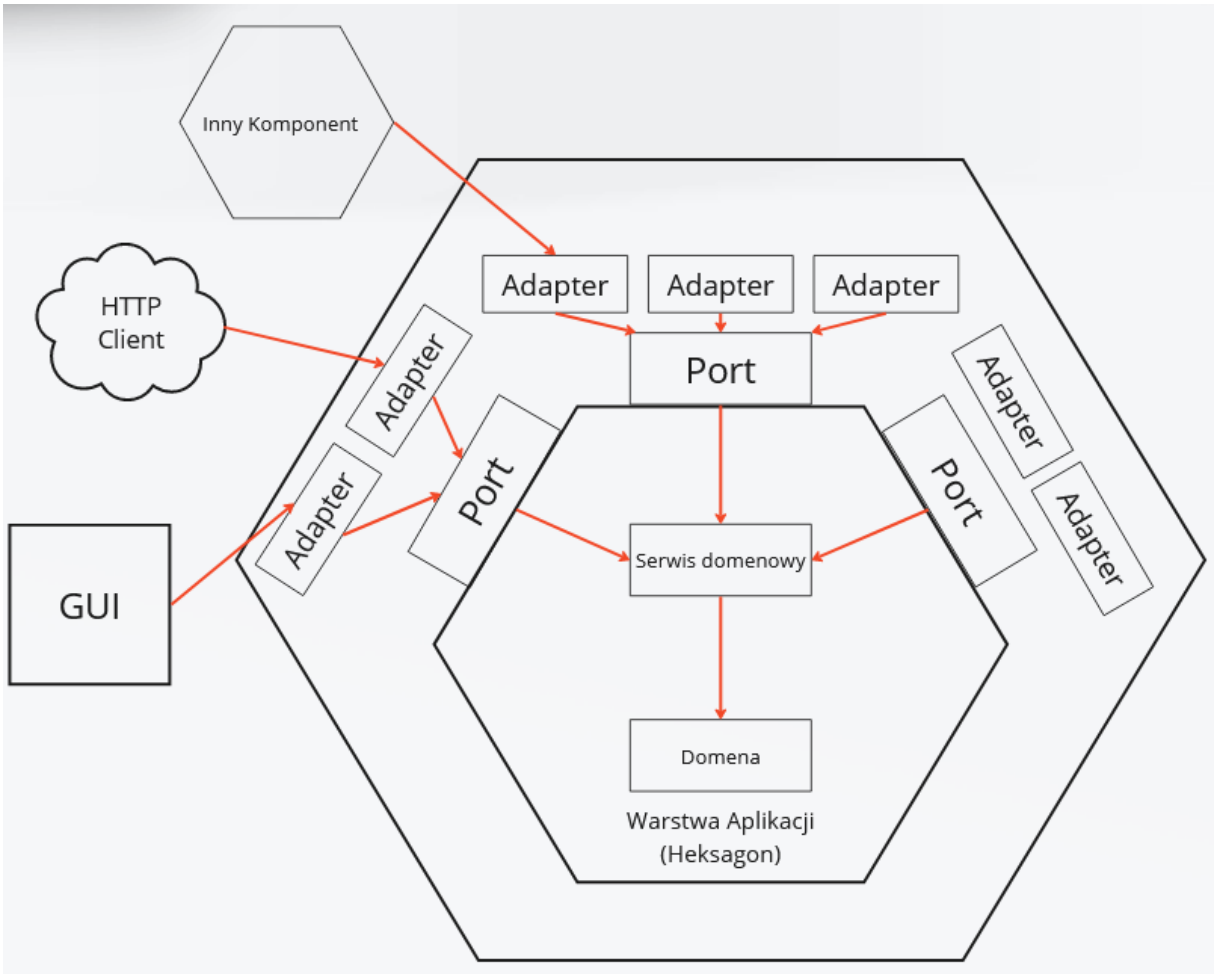


Rys. 3 Architektura warstwowa na poziomie diagramu komponentów systemu. Źródło: opracowanie własne.

Na poziomie całego systemu, zastosowanie architektury warstwowej pozwala na stworzenie spójnego i modularnego środowiska, w którym różne serwisy komunikują się ze sobą zgodnie z zasadami ustalonymi przez ich warstwową strukturę. Takie podejście zwiększa skalowalność horyzontalną systemu i ułatwia integrację nowych funkcjonalności, jednocześnie minimalizując ryzyko zakłóceń w działaniu istniejących komponentów. Diagram na rys.3 ilustruje, jak poszczególne komponenty systemu mogą być zorganizowane zgodnie z zasadami architektury warstwowej, prezentując zarówno relacje między komponentami, jak i przepływ danych oraz odpowiedzialności w systemie. Dzięki temu można łatwo zidentyfikować, które części systemu są odpowiedzialne za konkretne zadania, co sprzyja lepszej analizie, projektowaniu oraz wdrażaniu systemu informatycznego.

2.2. Architektura Heksagonalna

Architektura heksagonalna, znana również jako Architektura Portów i Adapterów, jest nowoczesnym podejściem do projektowania systemów informatycznych, które ma na celu poprawę ich modularności, testowalności oraz elastyczności. Główną motywacją do powstania tego wzorca było oddzielenie logiki biznesowej od interfejsów zewnętrznych, takich jak interfejsy użytkownika, bazy danych czy usługi sieciowe. Podstawowym założeniem architektury heksagonalnej jest to, że system powinien być niezależny od zewnętrznych elementów infrastruktury. W praktyce oznacza to, że logika biznesowa systemu jest centralnym elementem, który komunikuje się ze światem zewnętrznym poprzez jasno zdefiniowane porty i adaptery. Na rysunku 4 przedstawiono diagram koncepcyjny implementacji heksagonu dla pojedynczej domeny. Należy wyobrazić sobie, że każda domena biznesowa "otoczona" jest dodatkową warstwą, która w określony sposób definiuje zmiany, jakie mogą zajść w domenie przez konkretne adaptery.



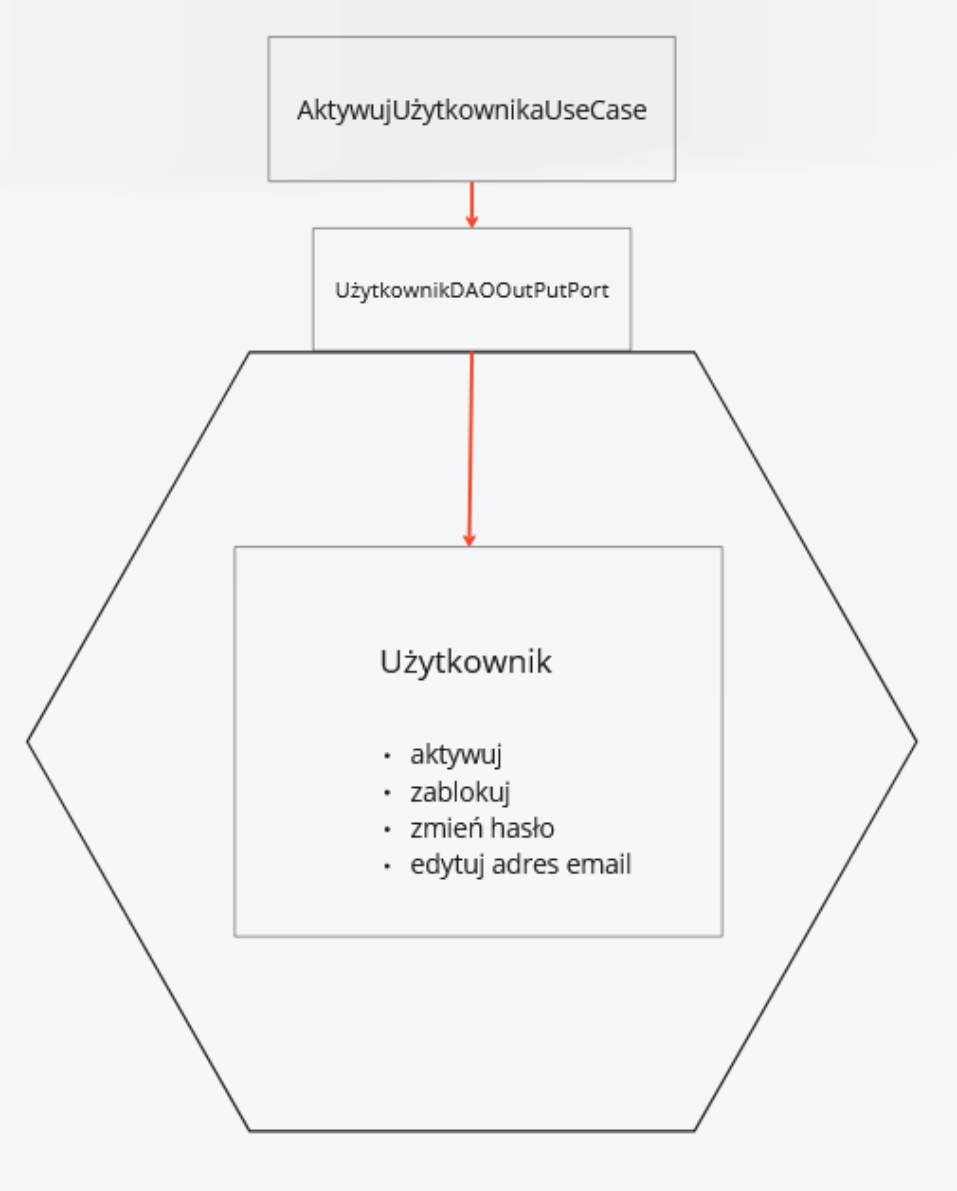
Rys. 4 Diagram architektury heksagonalnej dla pojedynczej domeny. Źródło: opracowanie własne.

Porty reprezentują interfejsy, przez które system odbiera i wysyła dane, podczas gdy adaptery są odpowiedzialne za przekształcanie tych danych do formatu zrozumiałego dla warstwy biznesowej. Dzięki takiemu podejściu możliwe jest łatwe testowanie logiki biznesowej w izolacji od pozostałych elementów systemu, co znacznie zwiększa efektywność procesu testowania i pozwala na wczesne wykrywanie błędów. Ponadto, architektura heksagonalna ułatwia wymianę i aktualizację zewnętrznych komponentów bez konieczności ingerencji w centralną logikę systemu, co przekłada się na większą elastyczność i długowieczność aplikacji.

2.2.1. Domena aplikacji

Domena w architekturze heksagonalnej stanowi centralny element systemu, koncentrując się na logice biznesowej i regułach operacyjnych. W jej obrębie znajdują się struktury danych, które przechowują kluczowe informacje biznesowe (encje) i stanowią obiektową reprezentację modelu bazodanowego. Te obiekty są odpowiedzialne za przechowywanie stanu systemu. Kolejnym aspektem domeny w architekturze heksagonalnej jest jej niezależność od zewnętrznych systemów i technologii. Założeniem domeny jest niezależność od zmian w technologii lub zewnętrznych interfejsach. Umożliwia to zachowanie spójności i integralności logiki biznesowej bez względu na zmiany w otoczeniu systemu. W idealnym przypadku obiekty domenowe są zupełnie niezależne, ale w praktycznych rozwiązaniach może to być utrudnione. Nakład technologiczny i dodatkowy kod jaki należałoby powielać w celu zachowania niezależności wypiera to założenie.

Obiekty domenowe wywoływane są przez abstrakcyjne porty, a te przez adaptery. W praktyce za dostęp do danych odpowiada Przypadek Użycia (*ang. usecase*), który jednoznacznie modyfikuje lub pobiera domenę z jądra aplikacji co zaprezentowano na rysunku 5.

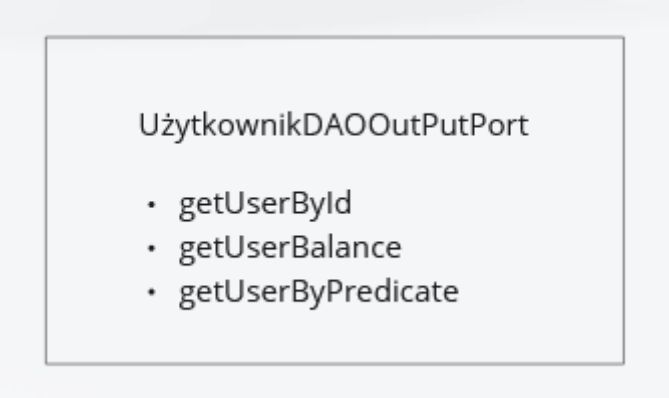


Rys. 5 Przykładowe wywołanie domeny użytkownik przez przypadek użycia. Źródło: opracowanie własne.

2.2.2. Porty

Porty pełnią rolę abstrakcyjnych interfejsów, które umożliwiają komunikację między logiką biznesową a zewnętrzną warstwą komponentu lub systemu. Architektura heksagonalna dąży do wyraźnego oddzielenia logiki biznesowej od infrastruktury technicznej, a porty są jednym z głównych elementów realizujących to oddzielenie.

Porty można podzielić na dwie główne kategorie: porty wejściowe (*ang. input ports*) i porty wyjściowe (*ang. output ports*). Porty wejściowe odpowiadają za odbieranie żądań z zewnętrznych źródeł, takich jak interfejsy użytkownika, API czy inne systemy. Te żądania są następnie przetwarzane przez odpowiednie usługi aplikacyjne, które realizują logikę biznesową. Porty wyjściowe, z kolei, są używane do komunikacji z zewnętrznymi systemami, takimi jak bazy danych, systemy płatności czy usługi zewnętrzne. Za ich pomocą aplikacja może wysyłać dane lub żądania do tych systemów w celu wykonania określonych operacji. Na rysunku 6 przedstawiono abstrakcyjną deklarację portu dla tabeli użytkownik.



Rys. 6 Abstrakcyjna deklaracje portu użytkownika. Źródło: opracowanie własne.

Porty definiują kontrakty, które muszą być zaimplementowane przez adaptery. Adaptery wykorzystują implementacje portów w celu otrzymania dostępu do domeny. Dzięki temu możliwe jest testowanie logiki biznesowej w izolacji, z wykorzystaniem zastępczych implementacji portów (*ang. mocks*), co ułatwia testowalność za pomocą testów jednostkowych. Dzięki abstrakcji, jaką wprowadzają, możliwe jest łatwe modyfikowanie lub wymienianie technologii bez wpływu na wewnętrzną logikę biznesową. Na przykład, zmiana technologii bazodanowej nie wymaga modyfikacji logiki biznesowej, a jedynie dostosowania odpowiednich adapterów, które implementują porty wyjściowe związane z dostępem do danych.

2.2.3. Adaptery

Adaptery pełnią rolę łączników między logiką biznesową a zewnętrznymi systemami i technologiami. Adaptery wykorzystują implementację portów, które są opakowane w przypadki użycia. Przyjmują na wejściu zewnętrzny model takie jakie DTO *(ang. Data Transfer Object*), Eventy (zdarzenia) lub innego rodzaju parametry wejścia. Następnie w przypadkach użycia dochodzi do konwersji danych między formatami używanymi przez domenę a formatami wymaganymi przez zewnętrzne systemy. Poprzez adaptery można zarządzać połączeniami z bazami danych, integrować z zewnętrznymi API, zarządzać autoryzacją i uwierzytelnianiem, a także zapewniać logowanie i monitorowanie operacji. Idea jaka za tym stoi polega na jej abstrakcji. Na przykład wystawienie portu dla Rest API polega na utworzeniu kontrolera z punktami końcowymi (*ang. endpoint*), który wywołuje przypadek użycia. Dzięki temu logika biznesowa pozostaje niezależna od specyficznych technologii i protokołów używanych przez zewnętrzne systemy, co znacznie ułatwia jej testowanie i rozwój.

2.2.4. Przypadek użycia

Przypadki użycia (*ang. use cases*) koncentrują w sobie realizacje konkretnych scenariuszy biznesowych. Przypadki użycia operują na obiektach domenowych i korzystają z usług aplikacyjnych oraz portów, aby zapewnić realizację konkretnych funkcji biznesowych. Są one odpowiedzialne za koordynowanie działań różnych komponentów systemu, zarządzanie przepływem danych i egzekwowanie reguł biznesowych.

Dzięki wyraźnemu oddzieleniu przypadków użycia od logiki domenowej i infrastruktury technicznej, możliwe jest łatwe testowanie, modyfikowanie i rozszerzanie funkcjonalności systemu.

Każdy przypadek użycia zaczyna się od zewnętrznego żądania, które może pochodzić z interfejsu użytkownika, zewnętrznego systemu lub innego komponentu wewnętrznego. To żądanie jest odbierane przez odpowiedni adapter, który następnie przekazuje je do odpowiedniego przypadku użycia. Przypadek użycia przetwarza dane, weryfikuje reguły biznesowe i korzysta z usług domenowych, aby wykonać wymaganą operację.

Każdy z tych przypadków użycia jest niezależnym, zamkniętym scenariuszem, który realizuje konkretną funkcję biznesową. Przypadek użycia należy traktować w intuicyjny sposób jako komplenty scenariusz jaki powstał w wyniku analizy biznesowej.

Realizacja przypadków użycia może wymagać komunikacji z zewnętrznymi systemami. W takim scenariuszu przypadek użycia korzysta z zewnętrznych portów, aby zainicjować odpowiednie operacje w zewnętrznych systemach. Adaptery implementujące te porty zajmują się szczegółami technicznymi, takimi jak formatowanie danych, zarządzanie połączeniami i obsługa błędów, co pozwala przypadkowi użycia skupić się wyłącznie na logice biznesowej.

Przypadki użycia w architekturze heksagonalnej są łatwe do testowania dzięki wyraźnemu oddzieleniu od infrastruktury technicznej. Można je testować w izolacji, wykorzystując mocki lub stuby do symulacji zewnętrznych systemów i danych. To pozwala na dokładne przetestowanie logiki biznesowej i upewnienie się, że system zachowuje się zgodnie z oczekiwaniami w różnych scenariuszach z pominięciem konfiguracji zewnętrznych zależności.

Warto zauważyć, że przypadki użycia w architekturze heksagonalnej mogą być łączone ze wzorcem projektowym Fasada (*ang. Facade*). Wzorzec fasady polega na ukryciu szczegółów implementacyjnych za dodatkową warstwą kodu, co korzystnie wpływa na czytelność kodu, szczególnie gdy domena posiada wiele przypadków użycia. Dzięki zastosowaniu fasady, adapter, który musiałby wstrzykiwać wiele zależności, może ograniczyć się do jednej zależności. To uproszczenie nie tylko poprawia przejrzystość kodu, ale także ułatwia jego utrzymanie i rozszerzanie horyzontalne.

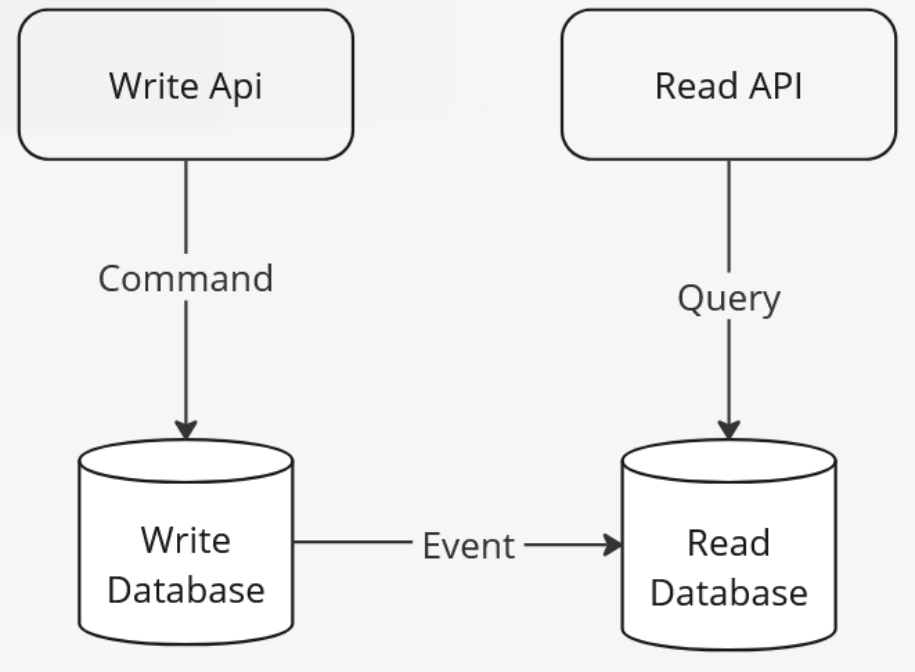
3. Inne wzorce architektoniczne

W pracy dokonane zostało porównanie pomiędzy architektura heksagonalną i warstwową, ale warto również wspomnieć o innych wzorcach, które, stanowią istotne punkty odniesienia w kontekście projektowania systemów informatycznych. Zrozumienie tych architektur pozwala na lepsze docenienie ich zalet, ograniczeń oraz kontekstu, w którym mogą być zastosowane. Wprowadzenie do różnych podejść architektonicznych dostarcza szerokiej perspektywy, która jest niezbędna dla inżynierów oprogramowania przy podejmowaniu świadomych decyzji dotyczących wyboru najbardziej odpowiednich rozwiązań technologicznych dla konkretnych problemów biznesowych i technicznych. Poniższe przykłady wybrano ze względu na częste przenikanie się tych podejść.

3.1. Command Query Responsibility Segregation (CQRS)

CQRS (*ang. Command Query Responsibility Segregation*) jest wzorcem architektonicznym stosowanym w projektowaniu systemów informatycznych, który opiera się na oddzieleniu operacji odczytu danych od operacji ich modyfikacji. CQRS wywodzi się z koncepcji *Command Query Separation,* postulującej, że każda metoda powinna realizować wyłącznie jedną z dwóch funkcji: modyfikację stanu systemu (komenda) lub odczyt danych (zapytanie), ale nigdy obu jednocześnie.

CQRS wprowadza także istotne rozdzielenie na operacje komend i zapytań. Komendy są odpowiedzialne za modyfikację stanu systemu i obejmują operacje takie jak tworzenie, aktualizacja czy usuwanie danych.



Rys. 7 Zasada działania systemu opartego o CQRS. Źródło: opracowanie własne.

Głównym założeniem CQRS jest separacja operacji odczytu i zapisu, co realizuje się poprzez wykorzystanie dwóch różnych modeli danych. Model zapisu (*ang. write model*), jest zoptymalizowany pod kątem wydajnego przetwarzania komend oraz walidacji danych. Model odczytu (*ang. read model)* jest natomiast zaprojektowany tak, aby umożliwić szybki i efektywny dostęp do danych, co często wiąże się z jego denormalizacją oraz zastosowaniem dodatkowych

indeksów i replikacji danych. Rozwiązanie to nie pasuje do wszystkich systemów informatycznych ze względu na redundancję struktur bazodanowych oraz potencjalne problemy w utrzymaniu. W teoretycznym podejściu należy implementować dwie linie procesów biznesowych w celu rozwijania systemu, jeden do odczytu drugi do zapisu. W praktyce dochodzi do zderzenia modelu odczytu i zapisu, ponieważ system zewnętrzny pytając o zasób z bazy danych najpierw musi go pobrać, aby został zmodyfikowany w warstwie kodu.

Istotnym aspektem CQRS jest również możliwość integracji z event sourcingiem, gdzie każda zmiana stanu systemu jest rejestrowana jako zdarzenie. Diagram przedstawiający zasadę działania przedstawiono na rysunku 7. Zdarzenia te mogą być następnie wykorzystywane do rekonstrukcji stanu systemu lub jako źródło danych do operacji odczytu. Taka architektura jest zalecana przy systemach opartych o asynchroniczną komunikację. Oddzielenie operacji odczytu i zapisu umożliwia niezależne skalowanie obu tych aspektów, co jest szczególnie przydatne w systemach o dużym natężeniu operacji odczytu. Ponadto, oddzielne modele danych dla operacji zapisu i odczytu pozwalają na ich optymalizację pod kątem specyficznych potrzeb, co przekłada się na wyższą wydajność w konkretnych przypadkach. Jednakże, CQRS nie jest wolne od wad. Wprowadza dodatkową złożoność do systemu, co może zwiększyć trudności związane z jego utrzymaniem i rozwojem. Zarządzanie dwoma modelami danych oraz synchronizacja pomiędzy nimi wymagają bardziej zaawansowanej infrastruktury i mogą prowadzić do problemów związanych ze spójnością danych.

3.2. Domain Driven Design (DDD)

Domain-Driven Design (DDD) to zaawansowane podejście do projektowania oprogramowania, które koncentruje się na głębokim zrozumieniu i modelowaniu domeny biznesowej, aby stworzyć systemy, które wiernie odzwierciedlają rzeczywiste zasady i złożoność biznesu. Wprowadzony przez Erica Evansa, DDD promuje ścisłą współpracę między ekspertami domenowymi a zespołem deweloperskim, co umożliwia tworzenie precyzyjnych modeli domeny, które są zarówno dokładne, jak i efektywne w implementacji. Jest to sposób wytwarzania oprogramowania znacząco różniący się od klasycznego podejścia.

Centralnym elementem DDD jest model domeny, który stanowi abstrakcyjną reprezentację kluczowych elementów i procesów biznesowych. Model ten jest wyrażony w sposób zrozumiały zarówno dla ekspertów domenowych, jak i deweloperów, co zapewnia, że system jest zgodny z rzeczywistymi wymaganiami biznesowymi. Ważnym aspektem DDD jest tworzenie wspólnego języka, zwanego językiem wszechobecnym (*ang. ubiquitous language*), który eliminuje bariery komunikacyjne między uczestnikami projektu i umożliwia precyzyjne porozumiewanie się na temat wymagań i funkcji systemu. Ta sama interpretacja domeny może mieć zupełnie inne znaczenie dla różnych kontekstów co może powodować niezrozumienia w zespole.

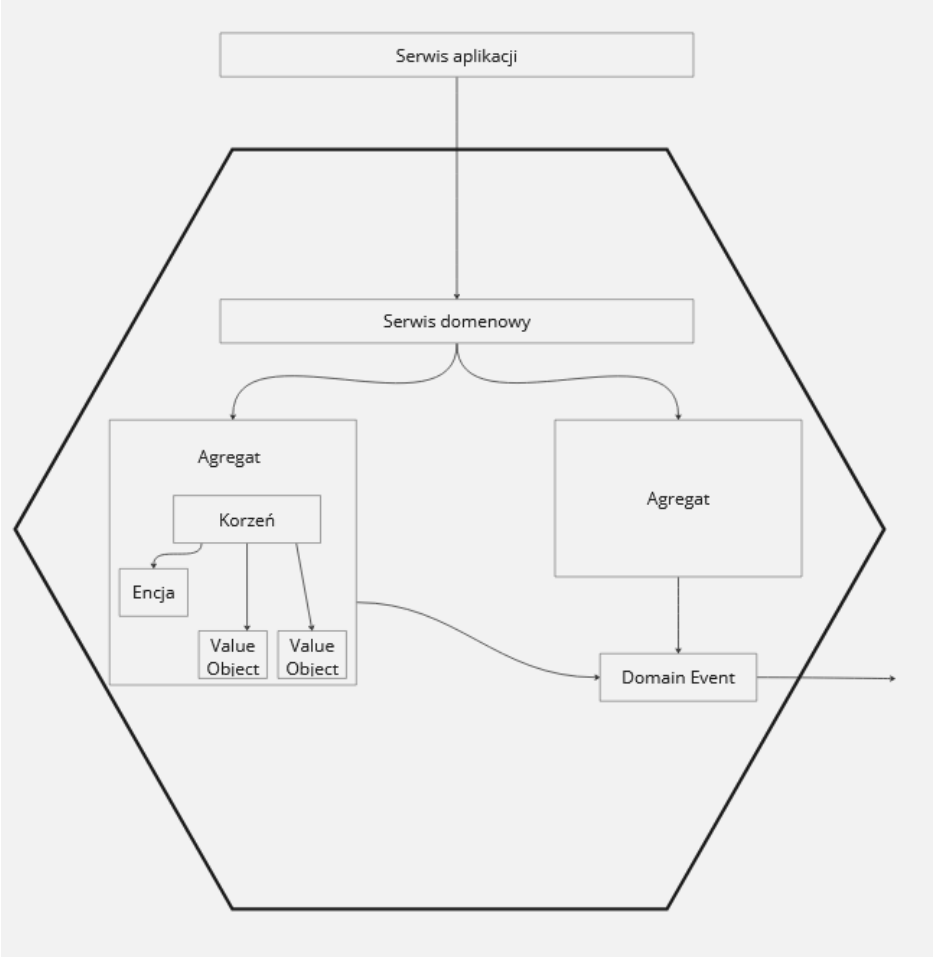
W podejściu DDD nie znajdzie zastosowania klasyczny anemiczny model danych. Podczas analizy domeny należy wyodrębnić odpowiednie grupy powiązanych obiektów, które nazywają się agregatami. Są traktowane jako niepodzielne jednostki z punktu widzenia konsystencji danych.

Agregaty definiują granice transakcji i są zarządzane przez obiekty korzeniowe (*ang. aggregate roots*), które kontrolują dostęp i modyfikację wewnętrznych stanów agregatu. W ramach agregatów wyróżnia się encje (*ang. entities*) i wartości obiektów (*ang. value objects*). Encje posiadają unikalną tożsamość i mogą zmieniać swój stan w czasie, podczas gdy wartości obiektów są niemutowalnymi obiektami, definiowanymi przez swoje atrybuty i nieposiadającymi unikalnej tożsamości.

Może się zdarzyć, że pewne operacje nie będą jasno przynależeć do konkretnej domeny, ale będą istotne dla logiki domenowej. Taki element nazywa się usługą domenową (*ang. domain*

services). Usługi te realizują kluczowe procesy biznesowe, które obejmują wiele obiektów domenowych po to, aby zapewnić trwałość i odzyskiwanie obiektów domenowych.

Warto wspomnieć, że DDD często implementuje się za pomocą architektury heksagonalnej w celu enkapsulacji domeny. Dzięki temu system staje się bardziej modułowy i łatwiejszy do zarządzania. Każdy moduł może reprezentować różne konteksty ograniczone (*ang. bounded contexts*), co jest kluczowym konceptem w DDD. Przykładowy przepływ danych zaprezentowano na rysunku 8.



Rys. 8 Enkapsulacja domeny za pomocą architektury heksagonalnej w DDD. Źródło: opracowanie własne.

Zastosowanie DDD pozwala tworzyć systemy, które dokładnie odwzorowują zasady domeny biznesowej, co sprawia, że są one bardziej zrozumiałe i łatwiejsze w utrzymaniu. Należy jednak wspomnieć, że implementacja tego podejścia może być skomplikowana i wymaga głębokiego zrozumienia zarówno technicznych, jak i biznesowych aspektów domeny. Proces ten jest czasochłonny i wymaga ścisłej współpracy między ekspertami domenowymi a deweloperami. Dodatkowo, wdrożenie DDD często wiąże się z większymi nakładami pracy na początkowych etapach projektu, co może stanowić barierę dla mniejszych zespołów lub projektów z ograniczonym budżetem. Nie jest to podejście zalecane do małych i średnich projektów, które nie będą dynamicznie rozwijane w przyszłości. Bardzo ważnym elementem jest również analiza biznesowo-techniczna, która w przypadku bycia niekompletną może znacząco wpłynąć na jakość dostarczonego rozwiązania za pomocą podejścia DDD.

4. Zastosowane technologie w projekcie

W tym rozdziale zostały opisane technologie zastosowane w projektach. W pracy zadbano o to, aby obie wersje aplikacji zawierały te same technologie w tych samych wersjach, aby porównanie skupiało się wyłącznie na wadach i zaletach przyjętych architektur.

4.1. Java

Implementacja serwisu webowego została oparta na języku Java w wersji 17 [12]. Wybór tej technologii wynikał przede wszystkim z faktu, że obecnie wiele nowoczesnych rozwiązań webowych opiera się na językach programowania obiektowego, a Java jest jednym z wiodących wyborów w tej dziedzinie. Dodatkowym czynnikiem, który wpłynął na decyzję, było doświadczenie autora projektu w obszarze Javy, co pozwoliło na efektywne wykorzystanie możliwości tego języka.

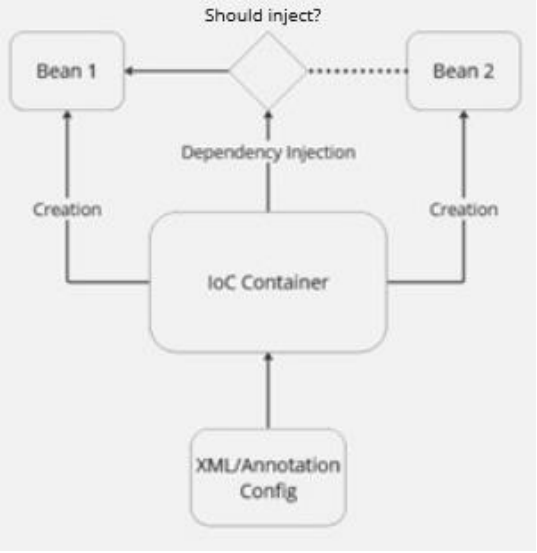
Ponadto, Java charakteryzuje się bogatym ekosystemem narzędzi i frameworków, co znacznie ułatwia zarówno rozwój, jak i utrzymanie projektu. Dzięki temu możliwe jest szybkie tworzenie wysokiej jakości aplikacji webowych, przy zachowaniu solidności, elastyczności i skalowalności projektu. Wybór Javy jako głównego języka programowania dla serwisu webowego był więc świadomą decyzją, mającą na celu zapewnienie efektywnej realizacji założonych celów.

4.2. Spring Boot

Spring Boot to framework do tworzenia aplikacji webowych w języku Java. Jest on często wybierany przez programistów ze względu na swoją gotową konfigurację i możliwość integracji z zewnętrznymi bibliotekami. Eliminuje konieczność ręcznego konfigurowania wielu aspektów aplikacji takich jak kontener aplikacyjny (domyślnie wbudowany Tomcat) lub menadżer zależności (Maven).

Ponadto, Spring Boot posiada wbudowane wsparcie dla wielu technologii, takich jak RESTful API, WebSocket, JPA (Java Persistence API), Spring Data, Spring Security i wiele innych [14].

Kolejną zaletą Spring'a jest jego silne wsparcie dla testowania jednostkowego i integracyjnego. Framework ten dostarcza narzędzia do łatwego tworzenia testów, co pozwala programistom szybko i skutecznie weryfikować poprawność działania swoich aplikacji. Za pomocą kilku adnotacji konfiguracyjnych można stworzyć odrębny kontekst aplikacyjny w celu weryfikacji testu integracyjnego.



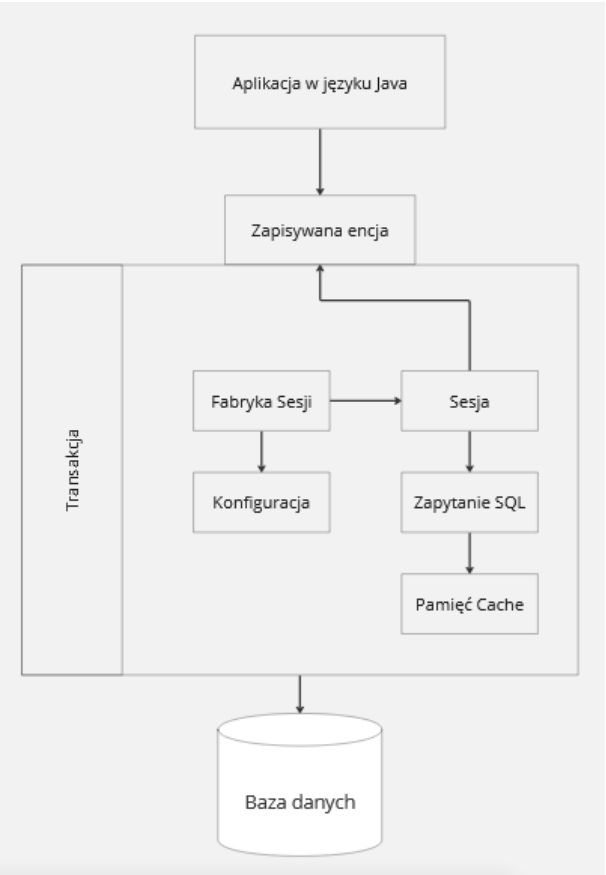
Rys. 9 Wstrzykiwanie zależności w Spring. Źródło: opracowanie własne na podstawie [14]

Spring jest frameworkiem, który działa na zasadzie wstrzykiwania zależności (*ang. Dependency Injection*). Wstrzykiwanie zależności polega na deklaracji komponentów aplikacji, zwanych ziarnami (*ang. Bean*), oraz automatycznym zarządzaniu ich zależnościami. Zamiast ręcznie tworzyć i konfigurować obiekty, Spring umożliwia definiowanie zależności pomiędzy nimi w sposób deklaratywny, co prowadzi do bardziej elastycznego i skalowalnego kodu. Zasadę wstrzykiwania zależności ilustruje rysunek 9. Poprzez wstrzykiwanie zależności, Spring implementuje zasady programowania obiektowego, takie jak Zasada Pojedynczej Odpowiedzialności (*ang. Single Responsibility Principle, SRP*) oraz Zasada Odwrócenia Zależności (*ang. Dependency Inversion Principle, DIP).*

4.3. Spring Data JPA

Serwisy webowe w celu przechowywania stanu systemu korzystają z bazy danych. Poprzez usługi napisane w Springu dochodzi do operacji na obiektach domenowych, które później zostają zapisane w bazie. Każda następna operacja na tym obiekcie również odbywa się za pośrednictwem warstwy kodu. W celu integracji modelu obiektowego z modelem bazodanowym powstała specyfikacja JPA (*ang. Java Persistence API)* [14, 15].

JPA obejmuje mapowanie obiektowo-relacyjne, dzięki któremu obiekty Java mogą być łatwo odwzorowane na struktury danych w bazie, oraz zapytania JPQL (*ang. Java Persistence Query Language*), które umożliwiają tworzenie zapytań do bazy danych przy użyciu obiektów i właściwości. JPA zarządza również cyklem życia obiektów, co oznacza, że programiści mogą operować na obiektach Java zgodnie z ich stanem w bazie danych, bez konieczności pisania złożonej logiki obsługi bazy danych. Dodatkowo, JPA obsługuje transakcje, zapewniając integralność danych poprzez grupowanie operacji w transakcje, które są atomowe, spójne, trwałe i izolowane.

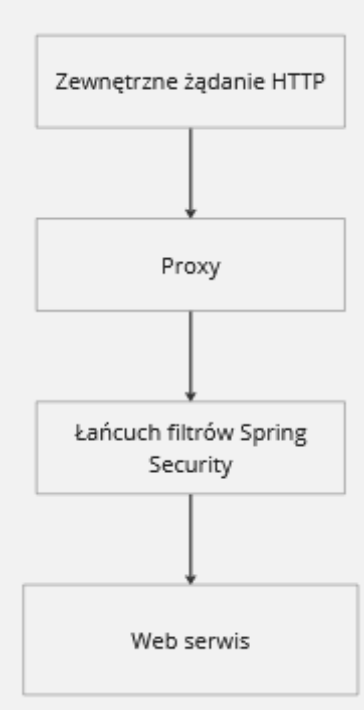


Rys. 10 Schemat działania Hibernate dla operacji zapisu encji. Źródło: opracowanie własne na podstawie [15]

JPA enkapsuluje powtarzalne konfiguracje oraz umożliwia pracę z bazą danych bez specjalistycznej wiedzy na temat składni SQL. Jest to abstrakcyjny interfejs, który posiada wiele implementacji. W ramach tego projektu został wykorzystany moduł Spring Data JPA, wykorzystujący powszechnie stosowaną implementację - Hibernate. Ta decyzja projektowa umożliwiła skuteczne zarządzanie persystencją danych w aplikacji, korzystając z zaawansowanych funkcji oferowanych przez Hibernate oraz ułatwiając integrację z innymi komponentami platformy Spring. Na rysunku 10 przedstawiono zasadę działania Hibernate.

4.4. Spring Security

Kolejnym komponentem wykorzystanym w ramach projektu jest moduł Spring Security. Ten moduł odpowiada za zapewnienie bezpieczeństwa sieciowego aplikacji. Spring Security, działając jako integralna część ekosystemu Spring, dostarcza niezbędne mechanizmy autoryzacji i autentykacji użytkowników, co stanowi kluczowy element w zapewnianiu ochrony danych i zabezpieczeniu przed nieuprawnionym dostępem [14]. Poprzez Spring Security, możliwe jest definiowanie reguł dostępu do poszczególnych zasobów aplikacji, zarządzanie sesjami użytkowników oraz obsługa różnych typów uwierzytelniania, takich jak uwierzytelnianie oparte na formularzach, uwierzytelnianie oparte na tokenach JWT (*ang. JSON Web Token*) czy uwierzytelnianie z użyciem protokołów OAuth2.



Rys. 11 Filtrowanie żądań w Spring Security. Źródło: opracowanie własne.

Dodatkowo, Spring Security oferuje szeroki zakres funkcji zabezpieczających przed atakami typu CSRF (*ang. Cross-Site Request Forgery*), XSS (*ang. Cross-Site Scripting*) czy SQL Injection, co przyczynia się do wzrostu odporności aplikacji na potencjalne zagrożenia.

4.5. Postgres

PostgreSQL, znany również jako "Postgres", jest systemem zarządzania relacyjnymi bazami danych rozwijanym w ramach projektu open–source. System został wybrany do projektu ze względu na przystępność korzystania oraz wbudowany panel dostępu [13]. Badany projekt nie wymagał szczególnego systemu zarządzania bazą danych, założeniem było wyłącznie skorzystanie z relacyjnej bazy danych.

4.6. H2

H2 również jest systemem relacyjnej bazy danych. Kluczową różnicą i motywacją do zastosowania H2 w projekcie jest fakt, że jest to baza działająca na zasadzie *in-memory,* co oznacza, że baza nie alokuje dysku twardego i nie jest potrzebne tworzenie dedykowanej instancji serwera. H2 podnosi swój kontekst automatycznie do potrzeb testowych i w momencie zakończenia testów serwer H2 kończy swoją pracę [17]. Podczas pracy baza alokuje zasoby pamięci RAM, która jest natychmiast zwalnia po zakończeniu pracy bazy. W efekcie stan bazy po ponownym uruchomieniu jest całkowicie wyczyszczony przez co H2 nie nadaje się do docelowych rozwiązań informatycznych, których stan musi być zapisywany w niezawodnym kontenerze.

H2 jest częstym wyborem do testowania aplikacji webowych metodą testów integracyjnych, gdzie asercją jest poprawne zapisanie lub zmodyfikowanie encji. W projekcie zastosowano ją dokładnie do tego celu – do przechowywania stanu aplikacji

4.7. Liquibase

Współczesne rozwiązania informatyczne zazwyczaj konfigurowane są na poziomie kodu, co oznacza, że konfiguracja środowiska, bazy danych oraz praca serwisu są przechowywane w plikach konfiguracyjnych w archiwum projektu. W przypadku serwisów webowych opartych na frameworku Spring istnieje możliwość automatycznego wygenerowania schematu bazy danych na podstawie modelu obiektowego. Alternatywnym podejściem jest dodatkowa warstwa kontroli nad schematem bazy danych oraz ręczna konfiguracja. Można to osiągnąć za pomocą skryptów generujących schemat na poziomie konsoli bazodanowej lub poprzez zastosowanie narzędzia do wersjonowania bazy danych.

Jednym z takich narzędzi jest Liquibase, które można zaimportować jako bibliotekę do projektu i przygotować skrypty generujące schemat bazy danych w plikach konfiguracyjnych serwisu [18]. Takie podejście umożliwia centralizację konfiguracji oraz zwiększa kontrolę nad modyfikacją modelu, pozwalając na zmiany zarówno w modelu obiektowym, jak i bazodanowym w ramach tego samego serwisu. Dodatkowo, synchronizacja pomiędzy modelami jest chroniona, ponieważ walidacja Liquibase odbywa się w czasie kompilacji projektu, co oznacza, że serwis nie uruchomi się w przypadku utraty spójności modeli.

4.8. Narzędzia do testowania

W niniejszej pracy zaprezentowano aplikację pełniącą rolę sklepu internetowego. Domena aplikacji obejmuje przypadki użycia, które dotyczą zarówno operacji na pojedynczych encjach, jak i operacji na wielu encjach jednocześnie. Proces weryfikacji poprawności funkcjonowania aplikacji został przeprowadzony przy użyciu testów jednostkowych oraz testów integracyjnych.

Do testów jednostkowych wykorzystano wbudowany w Spring Boot framework JUnit, który oferuje zaawansowane mechanizmy asercji, umożliwiające szczegółową kontrolę nad wynikami testów oraz ułatwiające identyfikację błędów. Testy jednostkowe pozwoliły na sprawdzenie

poprawności działania poszczególnych komponentów aplikacji w izolacji, co jest kluczowe dla zapewnienia wysokiej jakości kodu.

Do testów integracyjnych zastosowano framework Spock, którego składnia opiera się na języku Groovy, należącym do rodziny języków JVM. Framework Spock umożliwia pisanie czytelnych i łatwych do zrozumienia testów integracyjnych, które sprawdzają poprawność współdziałania różnych komponentów aplikacji w rzeczywistych warunkach [16, 19]. Dzięki temu możliwe było zweryfikowanie, czy poszczególne elementy systemu poprawnie komunikują się ze sobą oraz czy cała aplikacja działa zgodnie z założeniami.

Wdrożenie obu rodzajów testów – jednostkowych i integracyjnych – pozwoliło na kompleksowe sprawdzenie funkcjonalności aplikacji, zapewniając tym samym jej niezawodność i stabilność. Wyniki testów potwierdziły poprawność zaimplementowanych rozwiązań oraz zgodność aplikacji z wymaganiami funkcjonalnymi.

4.9. Pozostałe narzędzia wykorzystane w projekcie

Do realizacji projektu wykorzystano również narzędzia pomocnicze, które ułatwiły proces tworzenia kodu. Są to narzędzia powszechnie stosowane podczas pracy w środowisku Spring:

Lombok: służy do generowania powtarzalnego kodu, takiego jak gettery, setery i konstruktory dla obiektów DTO oraz encji. Dzięki Lombokowi możliwe jest znaczące uproszczenie kodu oraz redukcja ilości ręcznie pisanego kodu szablonowego [20].

MapStruct: umożliwia ukrycie implementacji logiki odpowiedzialnej za przekształcanie (mapowanie) obiektów z jednego typu danych na inny. Narzędzie to automatycznie generuje kod mapujący, co zwiększa czytelność kodu oraz zmniejsza ryzyko błędów związanych z ręcznym mapowaniem [21].

Swagger: pozwala na wygenerowanie interfejsu użytkownika dla API, który zawiera zestawienie zasobów wraz z przykładowymi żądaniami. Dzięki Swaggerowi możliwe jest łatwe testowanie i dokumentowanie API, co ułatwia pracę deweloperów oraz komunikację z klientami API [22].

Zastosowanie tych narzędzi wspomogło efektywność procesu tworzenia kodu oraz przyczyniło się do zwiększenia jego jakości i czytelności.

5. Opis projektu

Projekt wdrażany w ramach niniejszej pracy pełni funkcję sklepu internetowego, który umożliwia prowadzenie działalności handlowej online. System ten jest zaprojektowany z myślą o dwóch grupach użytkowników: sprzedawcach oferujących swoje produkty oraz kupujących poszukujących towarów do nabycia. Proces realizacji zamówienia przebiega poprzez kilka etapów. Klient rozpoczyna od tworzenia koszyka zakupowego, do którego dodaje wybrane produkty. Następnie zatwierdza zawartość koszyka, co inicjuje ustalanie szczegółów dotyczących dostawy, takich jak adres wysyłki i preferowany sposób dostarczenia. Ostatecznie, po potwierdzeniu wszystkich danych, zamówienie jest finalizowane i przekazywane do realizacji. System zapewnia zatem kompleksową obsługę transakcji handlowych w środowisku online, ułatwiając zarówno sprzedaż, jak i zakup towarów.

5.1. Model domenowy

W niniejszej pracy zastosowano relacyjną bazę danych jako podstawową strukturę przechowywania informacji. Wspomniany model charakteryzuje się anemicznością, co oznacza, że w kontekście bazy danych ograniczono się wyłącznie do wykorzystania atrybutów oraz relacji pomiędzy tabelami.

W procesie projektowania modelu skoncentrowano się na istotnych elementach systemu, których stan wymagał przechowywania. W efekcie tego procesu wyodrębniono kluczowe obszary systemu, które stanowią integralne encje domenowe konieczne dla prawidłowego funkcjonowania systemu. Kluczowe encje domenowe zostały wyłonione na podstawie ich istotnej roli w kontekście systemu:

- użytkownik
- produkt
- płatność
- zamówienie

W celu zachowania standardów normalizacyjnych, dodatkowo wyodrębniono tabele, które wspierają strukturę bazodanową. Działanie to ma na celu optymalizację zarządzania danymi poprzez unikanie redundancji oraz zapewnienie spójności i integralności danych w systemie. Niektóre tabele zostały wydzielone z zastosowaniem relacji 1:1, co nazywa się pionowym partycjonowaniem. Motywacją do pionowego partycjonowania jest semantyczny podział tabel oraz częstotliwość zapytań. W procesie pionowego partycjonowania kolumny są rozdzielane na różne tabele w zależności od ich charakterystyki i częstotliwości użycia. Kolumny, które są często zapytywane, wydziela się do jednej tabeli, natomiast kolumny rzadziej używane umieszcza się w oddzielnej tabeli. Taki podział pozwala na optymalizację wydajności, ponieważ dostęp do często używanych danych jest szybszy, co zmniejsza obciążenie systemu i poprawia jego efektywność. Dzięki pionowemu partycjonowaniu możliwe jest lepsze zarządzanie zasobami i zwiększenie ogólnej wydajności bazy danych [5]. W rezultacie powstały tabele:

- Adres: Encja została wydzielona z danych użytkownika w celu usunięcia redundantnych pól z encji dostawa.
- Dostawa: encja mogłaby być częścią zamówienia, ale ze względu na relację z Adresem zadecydowano o wyodrębnieniu tej encji.
- Dostępność produktu: encja wydzielona w procesie pionowego partycjonowania. Zapytanie o dostępność produktu można wydzielić, ponieważ są to często odpytywane kolumny.

- Koszyk: jest to implementacja kontenera dla produktów, który zawiera dodatkowe pola. Decyzja o wydzieleniu tej encji została podjęta podczas projektowania procesu składania zamówienia. Założono, że użytkownik w momencie „dodawania do koszyka” z poziomu GUI aplikacji już posiada instancję koszyka. Dzięki temu po powrocie na stronę internetową użytkownik nadal będzie posiadał swój koszyk widoczny na stronie.

Kolejnym ważnym elementem w systemie są tabele słownikowe, które pełnią rolę systemowego zbioru dozwolonych wartości dla konkretnych pól w systemie. Takie rozwiązanie umożliwia elastyczne zarządzenie dozwolonym stanem wartości w systemie i odseparowuje walidacje biznesowe od kodu aplikacji [6].

Do realizacji słowników stworzono dwie tabele: nazwa słownika oraz wartość słownika. Model wartości słownika przedstawiono w Kod 1. Każda wartość słownikowa musi przynależeć do konkretnej nazwy słownika, co odzwierciedla mechanizm typów wyliczeniowych w programowaniu.

Kod 1 - Deklaracja encji wartości słownikowej w podejściu warstwowym

```
@Entity
@Table(name = "t_dictionary_record")
@Getter
@Setter
@NoArgsConstructor
public class DictionaryRecordEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "value", nullable = false)
    private String value;

    @Column(name = "active", nullable = false)
    private Boolean active = Boolean.TRUE;

    @ManyToOne
    @JoinColumn(name = "dictionary_name_id", nullable = false)
    private DictionaryNameEntity dictionaryName;

    public DictionaryRecordEntity(String value,
                                  Boolean active,
                                  DictionaryNameEntity dictionaryName)
    {
        this.value = value;
        this.active = active;
        this.dictionaryName = dictionaryName;
    }
}
```

Najważniejszą różnicą w zastosowaniu tabel bazodanowych zamiast enumów jest możliwość konfiguracji wartości bez ingerencji w kod. Każda zmiana kodu w środowiskach produkcyjnych wiążę się z publikacją nowej wersji aplikacji i wymaga nakładu pracy programisty, dzięki zastosowaniu tabel słownikowych wystarczy, że administrator dokona pożądanych zmian w przygotowanym panelu poprzez REST API.

5.2. Usługi dostępne w aplikacji

Aplikacja udostępnia szereg operacji związanych z handlem internetowym. Funkcjonalności można wydzielić ze względu na aktorów w systemie, których przedstawiono na Rys. 12.

Funkcjonalności aktora Klient:

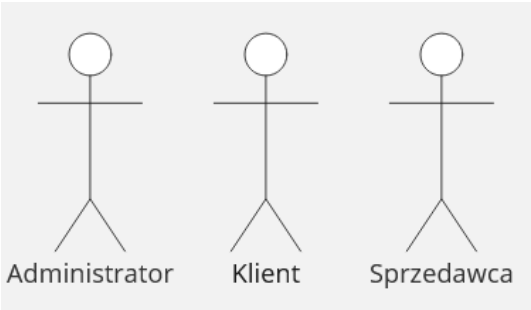
- rejestracja w systemie
- aktywacja konta
- dezaktywacja konta
- aktualizacja danych konta
- aktualizacja danych adresowych
- zasilenie salda
- usunięcie konta
- sprawdzenie czasu od rejestracji w systemie
- przeglądanie listy produktów
- przeglądanie danych o dostępności produktu
- dodanie produktów do koszyka
- wybranie formy dostawy
- złożenie zamówienia
- monitorowanie przebiegu zamówienia

Funkcjonalności aktora Sprzedawcy:

- rejestracja w systemie
- aktywacja konta
- dezaktywacja konta
- aktualizacja danych konta
- aktualizacja danych adresowych
- dodanie produktu
- zwiększenie dostępności produktu
- aktywacja produktu
- dezaktywacja produktu
- utworzenie lokalnej promocji

Funkcjonalności aktora Administrator:

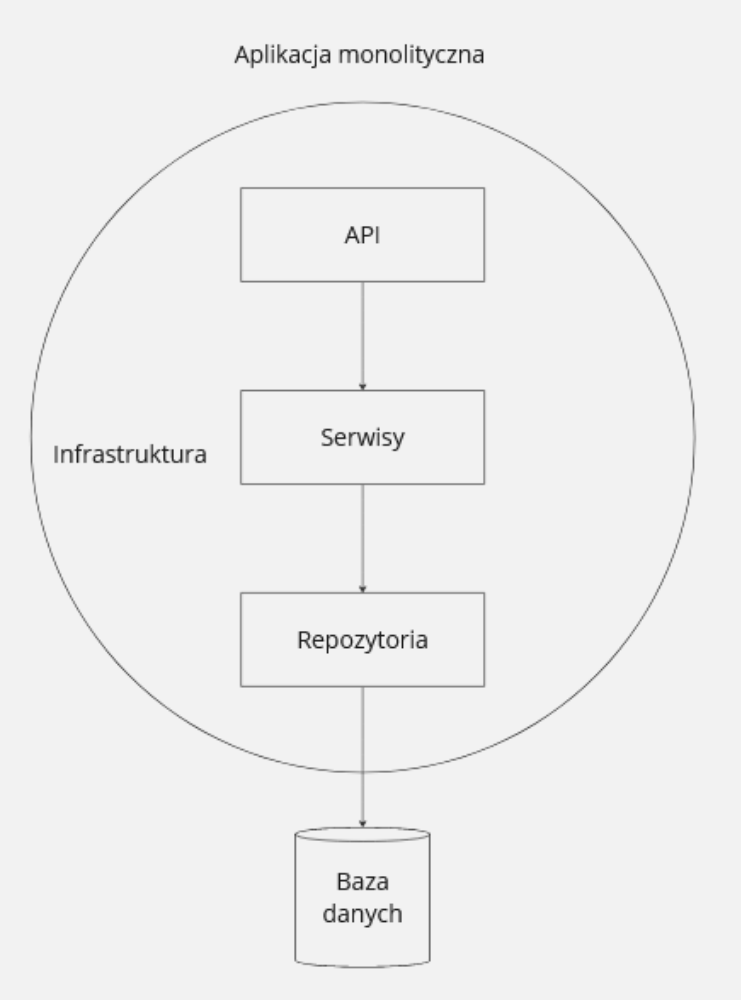
- aktywacja użytkownika
- dezaktywacja użytkownika
- usunięcie konta użytkownika
- dodanie słownika
- modyfikacja słownika
- dodanie rekordu słownikowego
- modyfikacja rekordu słownikowego
- utworzenie systemowej promocji na produkty
- utworzenie systemowej promocji na produkty



Rys. 12. Aktorzy dostępnie w systemie. Źródło: opracowanie własne.

6. Implementacja projektu w architekturze warstwowej

W procesie tworzenia aplikacji przy użyciu architektury warstwowej wyodrębniono warstwy odpowiedzialności zaprezentowane na Rys. 13.



Rys. 13. Schemat warstw w systemie. Źródło: opracowanie własne.

Warstwy zostały wyodrębnione za pomocą najpowszechniejszej metody:

- warstwa wejścia do aplikacji
- warstwa logiki biznesowej
- warstwa dostępu do danych

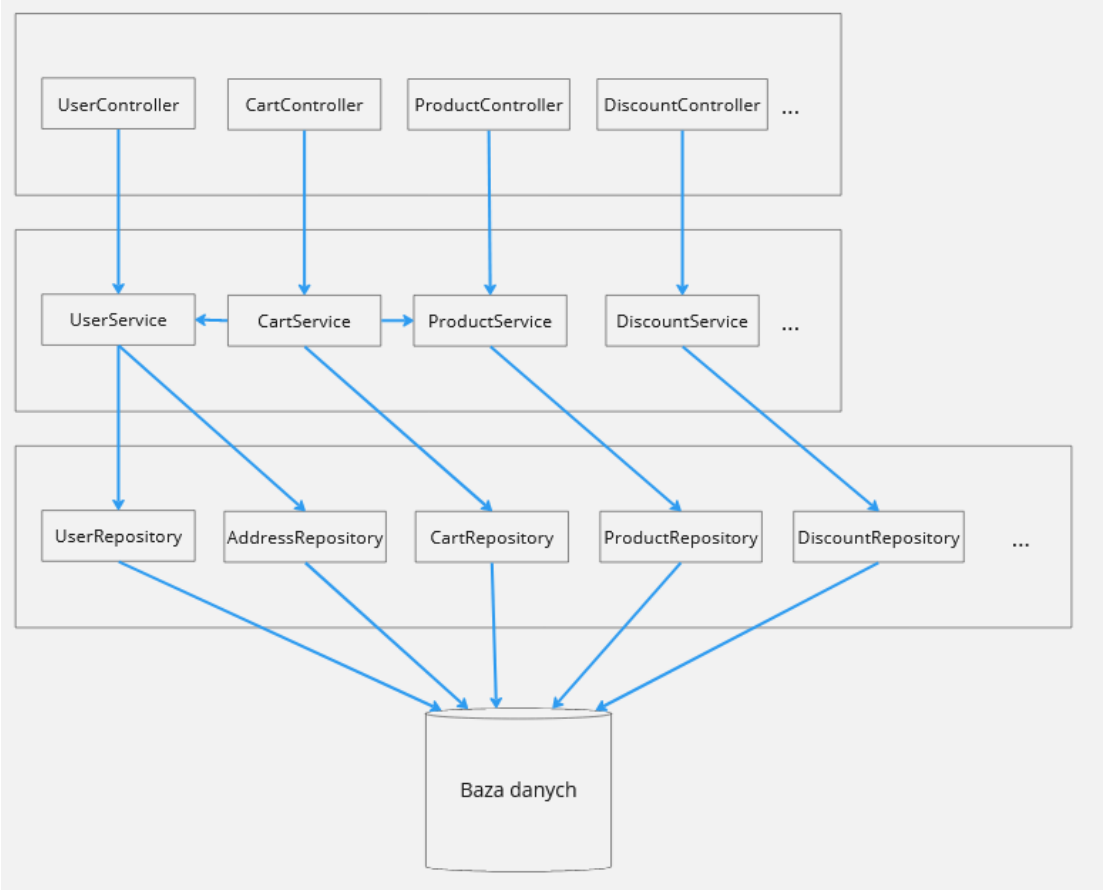
6.1. Opis warstw systemu

Każdy proces aplikacji rozpoczyna się od warstwy API, w której zdefiniowane są odpowiednie endpointy. Endpointy te służą jako punkty wejściowe dla zapytań przychodzących od klientów, takich jak przeglądarki internetowe czy aplikacje mobilne. Kiedy klient wysyła żądanie do określonego endpointu, aplikacja rozpoznaje je i kieruje do odpowiedniego punktu wejściowego w warstwie API.

Po odebraniu żądania, warstwa API przetwarza je i wywołuje odpowiedni serwis z warstwy serwisowej. Warstwa serwisowa jest odpowiedzialna za realizację głównych operacji biznesowych aplikacji. To tutaj następuje walidacja danych wejściowych, co oznacza sprawdzenie czy dane przekazane przez klienta są zgodne z oczekiwaniami systemu i spełniają określone kryteria.

Walidacja ta może obejmować sprawdzenie formatów danych, obecność wymaganych pól, a także spełnianie określonych reguł biznesowych.

Po pomyślnej walidacji danych wejściowych, warstwa serwisowa realizuje logikę biznesową, czyli wykonuje operacje związane z funkcjonalnością aplikacji. Może to obejmować różnorodne procesy, takie jak obliczenia, przetwarzanie danych, integracje z innymi systemami czy zarządzanie sesjami użytkowników. Kiedy logika biznesowa zostanie zrealizowana, aplikacja przystępuje do zapisania stanu w bazie danych. Proces ten jest zarządzany przez warstwę repozytorium, która abstrahuje bezpośrednią interakcję z bazą danych, zapewniając uporządkowany i bezpieczny sposób komunikacji z systemem przechowywania danych. Proces został zwizualizowany na Rys. 14.



Rys. 14 Wizualizacja kaskadowości operacji w pracy. Źródło: opracowanie własne.

Ustawienie silnej granicy odpowiedzialności w architekturze warstwowej umożliwia skalowanie wertykalne. W przypadku dołożenia nowej tabeli w większości przypadków wystarczy dodać implementację odpowiednio: repozytorium, serwis oraz kontroler. Tego typu rozwiązanie jest intuicyjne i stosunkowo łatwe do wdrożenia. Programiści pracujący w zespole nie muszą przestrzegać skomplikowanych założeń, lecz mogą rozszerzać istniejące rozwiązanie w ramach gotowych już warstw abstrakcji. Takie podejście jest szczególnie odpowiednie dla małych i średnich projektów z nieskomplikowaną domeną, gdzie priorytetem jest szybkie i efektywne wdrożenie. Programiści mogą łatwo dodawać nowe funkcjonalności, nie martwiąc się o skomplikowane zależności czy architektoniczne wyzwania.

Jednak w przypadku rozbudowanego modelu bazodanowego i złożonej logiki biznesowej mogą pojawić się problemy. Zależności pomiędzy różnymi częściami systemu mogą zacząć się przecinać, co utrudnia utrzymanie i rozwój kodu. Może również dojść do sytuacji, w której

odpowiedzialność poszczególnych klas zacznie rosnąć w sposób niekontrolowany, prowadząc do trudności w zarządzaniu kodem oraz zwiększając ryzyko wystąpienia błędów. W takich przypadkach konieczne może być zastosowanie bardziej zaawansowanych technik projektowych, aby zachować czytelność i skalowalność systemu. Przykładem może być logika utworzenia zamówienia przedstawiona we fragmencie Kod 2, która jest najbardziej zaawansowanym procesem w aplikacji.

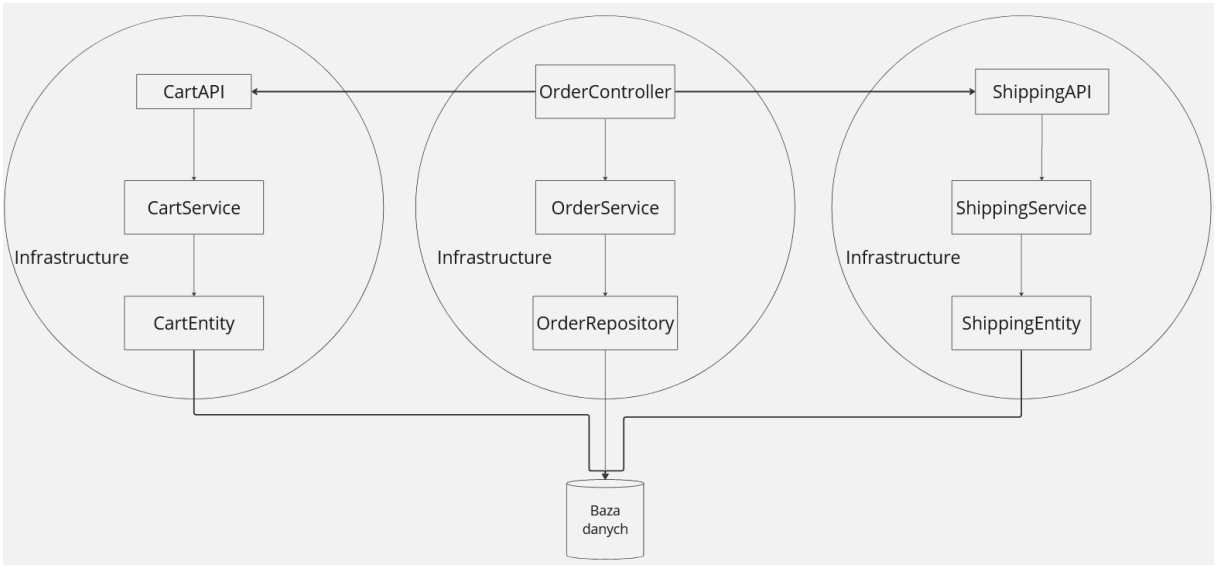
Kod 2 – Logika tworzenia zamówienia.

```
@Transactional
public UUID createOrder(CreateOrderDTO dto) {
    ShippingEntity shipping = shippingService.findByUUID(dto.getShippingUUID());
    CartEntity cart = cartService.findEntityByUUID(dto.getCartUUID());
    Set<DiscountEntity> totalDiscounts = getDiscounts(dto, cart);

    BigDecimal totalAmount = calculateTotalAmount(cart);
    if (!totalDiscounts.isEmpty()) {
        totalAmount = totalAmount
                .subtract(calculateDiscountAmount(cart, totalDiscounts));
    }

    cartService.finalizeCart(cart.getUuid());
    OrderEntity order = new OrderEntity(cart, shipping);
    order.setTotalPrice(totalAmount);
    orderRepository.save(order);

    return order.getUuid();
}
```

Pierwszymi krokami w procesie jest pobranie danych z domen: dostawy, koszyka oraz zniżek. Obecna implementacja serwisu Zamówienia korzysta z innych serwisów wyłącznie do odczytu, co jest zgodne z konwencją architektury warstwowej. Niepożądanym zachowaniem w obecnym rozwiązaniu byłaby sytuacja, w której wstrzykiwany serwis zewnętrzny wykonuje operacje logiki biznesowej na tych domenach, co mogłoby prowadzić do zmian w tabelach, za które ten serwis nie odpowiada.

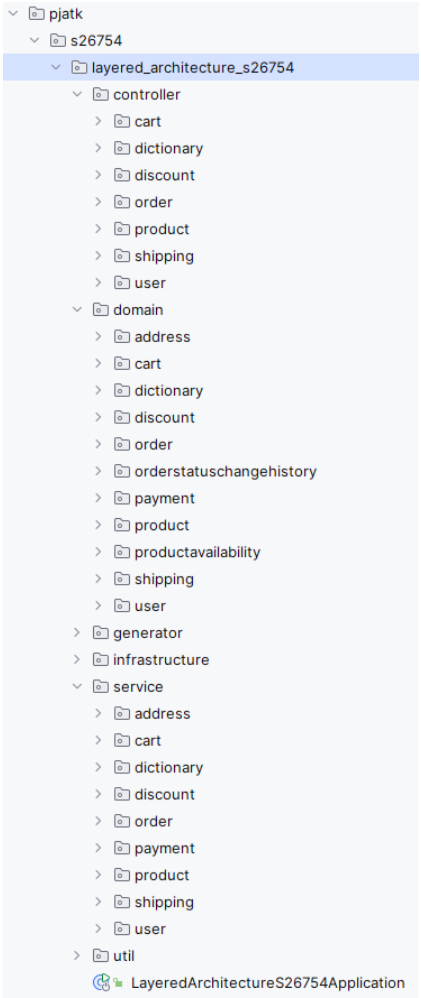
Obecna implementacja kodu dopuszcza takie zachowanie, ponieważ wszystkie serwisy są publiczne, a ich metody są udostępnione. Pomimo tego, że w teorii taka operacja jest możliwa ze względu na deklarację wszystkich metod jako publiczne, uniknięcie tego błędu wynika ze świadomości programisty. Zgodność z konwencją architektury warstwowej i odpowiedzialne korzystanie z metod serwisów jest kluczowe dla utrzymania integralności danych i odpowiedzialności poszczególnych komponentów systemu.

W architekturze warstwowej należy zarządzać wstrzykiwaniem zależności w taki sposób, aby w każdym momencie możliwe było wydzielenie całej encji do innego mikroserwisu. W praktyce oznacza to, że zależności między komponentami powinny być jasno zdefiniowane i ograniczone do minimalnego, niezbędnego zakresu. Każda warstwa systemu, czy to warstwa prezentacji, logiki biznesowej, czy dostępu do danych, powinna komunikować się z innymi warstwami za pośrednictwem dobrze zdefiniowanych interfejsów. Dzięki takiemu podejściu wystarczy zmienić implementację serwisu, a komponent wstrzykujący ten serwis jest agnostyczny względem jego implementacji.



Rys. 15 Schemat komponentów po wydzieleniu mikroserwisu dla domeny Koszyk oraz Dostawa. Źródło: opracowanie własne.

Architektura warstwowa jest również intuicyjna pod względem uporządkowania kodu. Każda z warstw abstrakcji pełni rolę pakietu. Schemat pakietyzacji projektu został przedstawiony na Rys. 16.



Rys. 16 Zestawienie pakietów aplikacji. Źródło: opracowanie własne.

W projekcie, oprócz wcześniej wymienionych trzech warstw, zaimplementowano również warstwę pakiet przechowujący statyczne metody pomocnicze. Pakiet ten zawiera funkcje, które poprawiają czytelność i utrzymanie kodu, ułatwiając wykonywanie często powtarzających się operacji. Te metody są umieszczone poza kontenerem kontekstu Spring, co oznacza, że nie naruszają obiektowego podejścia aplikacji i nie są zarządzane przez Spring. Dzięki temu kod pozostaje klarowny i modularny, a jednocześnie zachowana jest separacja odpowiedzialności.

Enkapsulacja i wydzielanie logiki kodu do odrębnych klas chroni przed częstą refaktoryzacją kodu. W rozwiązaniu warstwowym skorzystano z klas walidacyjnych, które należą do warstwy logiki biznesowej. Są to klasy przechowujące statyczne metody sprawdzające poprawność warunków ustalonych przez biznes. Wydzielenie klas walidacyjnych sprawia, że główna klasa serwisu pozbawiona jest prywatnych metod, które dotyczą pojedynczego procesu w aplikacji. Kod 3 przedstawia przykładową walidację tranzycji pomiędzy statusami zamówienia. System sprawdza czy operacja modyfikująca zamówienie próbuje zmienić status zamówienia w dozwolony sposób.

Kod 3 – Fragment metody walidacji zmiany statusu zamówienia.

```
[...]

private Map<OrderStatus, Set<OrderStatus>> initializeTransitionMap() {
    Map<OrderStatus, Set<OrderStatus>> allowedTransitionsMap = new
HashMap<>();
    allowedTransitionsMap.put(
            OrderStatus.CREATED,
            new HashSet<>(List.of(OrderStatus.PAID))
    );
    allowedTransitionsMap.put(
            OrderStatus.PAID,
            new HashSet<>(List.of(OrderStatus.IN_PROGRESS))
    );
    allowedTransitionsMap.put(
            OrderStatus.SENT,
            new HashSet<>(List.of(OrderStatus.DELIVERED))
    );
    allowedTransitionsMap.put(
            OrderStatus.DELIVERED,
            new HashSet<>(Collections.emptyList())
    );
    return allowedTransitionsMap;
}
[...]
    if (Objects.isNull(orderStatusChangeHistoryEntity)) {
        throw new OrderDomainOperationException(
                ORDER_STATUS_NULL,
                "Order status cannot be null! "
        );
    }
    Set<OrderStatus> allowedTransitions =
allowedTransitionMap.get(orderStatusChangeHistoryEntity.getCurrentStatus());
    if (!isTransitionValid(dto, allowedTransitions)) {
        throw new OrderDomainOperationException(
                ORDER_STATUS_TRANSITION_INVALID,
                "Order status change transition is not valid! "
        );
    }
}

[...]
```

6.2. Domena aplikacji

Reprezentacja modelu obiektowego została zaprojektowana z wykorzystaniem konfiguracji frameworka Spring oraz modułu Spring Data. Proces ten obejmował odpowiednie oznaczenie pól encji za pomocą adnotacji, co umożliwiło mapowanie obiektów na struktury danych w bazie danych relacyjnej.

W ramach tego podejścia, pola encji oznaczone adnotacją `@Column` zostały skonfigurowane tak, aby odzwierciedlały odpowiednie kolumny w bazie danych. Z kolei adnotacja `@JoinColumn` została użyta w celu wskazania kolumny, która zawiera klucz obcy do innej tabeli. Ponadto w zaprezentowanym Kod 4 zadeklarowane zostały relacje pomiędzy encjami za pomocą adnotacji `@ManyToOne`.

Kod 4 - Deklaracja encji w podejściu warstwowym

```
@Getter
@Setter
@Entity
@Table(name = "t_payment")
public class PaymentEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "uuid", nullable = false, unique = true)
    private UUID uuid;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private UserEntity user;

    @ManyToOne
    @JoinColumn(name = "promotion_name")
    private DictionaryRecordEntity promotionName;

    @Enumerated(EnumType.STRING)
    @Column(name = "payment_type", nullable = false)
    private PaymentType paymentType;
```

Do modyfikowania wartości pól encji zastosowano metody mutujące (*ang. Setters*), które zostały wygenerowane za pomocą narzędzie Lombok poprzez adnotację `@Setter`. Taki sam zabieg zastosowano do metod dostępowych za pomocą adnotacji `@Setter`. Umotywowano zwiększeniem czytelności i redukcji ilości kodu koniecznego do ręcznego definiowania tych metod

7. Implementacja projektu w architekturze heksagonalnej

Projekt realizowany zgodnie z konwencją architektury heksagonalnej w założeniu ma pełnić te same funkcje co wersja warstwowa. Posiada również tak samo zaprojektowane REST API. Możliwe jest to dzięki wydzieleniu warstwy kontrolerów od pozostałej części systemu.

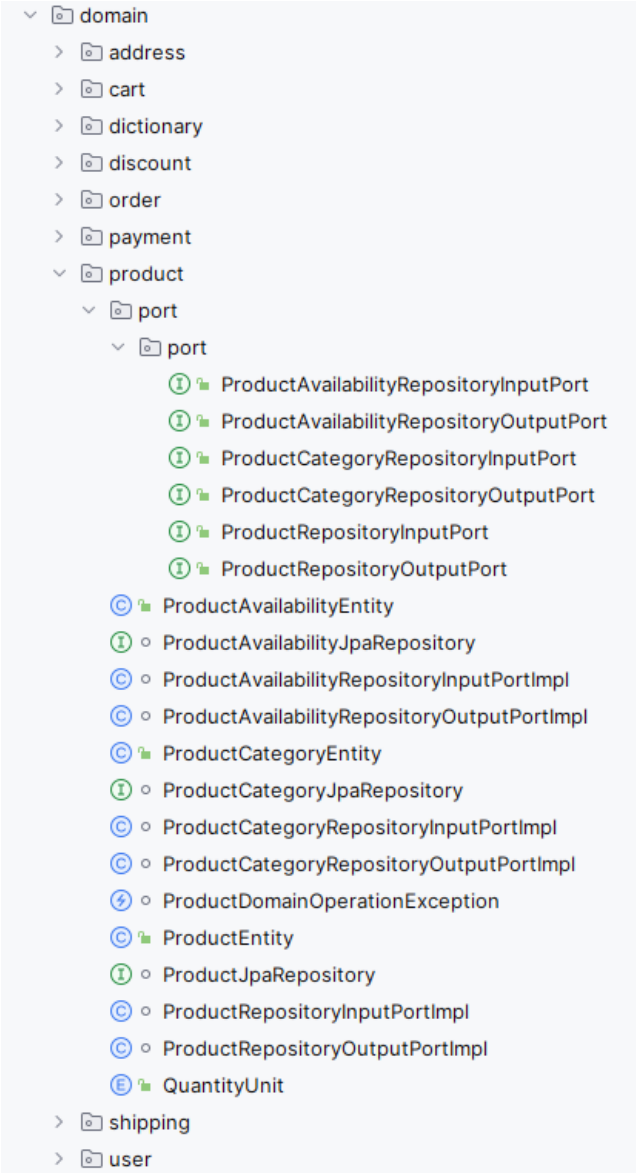
Praca skupia się na procesie wytwarzania kodu i implementacji serwisu. W związku z tym zdecydowano o zastosowaniu również analogicznego – anemicznego modelu danych. Próba analizy domeny w celu wyodrębnienia dalszych poziomów abstrakcji i zachować zbliżyłaby pracę do podejścia DDD, co mogłoby wpłynąć na formę porównania [23]. W tym podejściu zwrócono szczególną uwagę na szczegóły implementacyjne, które potrafią zaważyć nad ogólnym kształtem architektury. Przykładem może być dbałość o dobór specyfikatora dostępu w klasach napisanych w języku Java. Robert Martin w swojej książce napisał „*It's almost as if we, as developers, instinctively use the public keyword without thinking. It's in our muscle memory*." [4]. Zgadzam się z tym stwierdzeniem; jest ono szczególnie prawdziwe w przypadku deweloperów z krótkim stażem pracy. Wraz z nabytym doświadczeniem deweloperzy poznają znaczenie modyfikatora dostępu nie tylko na poziomie metody, co wspiera jakość i czytelność kodu. Co więcej, dzięki niemu można również budować solidne fundamenty pod cały system. W podejściu warstwowym również zwrócono na to uwagę, ale ze względu na jej własności większość kluczowych komponentów jest publiczna.

7.1. Struktura projektu

W projekcie występują cztery główne katalogi. Schemat katalogów ilustruje Rys. 17. Pakiety te odzwierciedlają założenie architektury heksagonalnej zaprezentowane w rozdziale 2.2.1. Pakiet domeny jest jądrem aplikacji, pakiet aplikacji odpowiada za wykonywanie logiki biznesowej i stanowy łącznik pomiędzy zewnętrznymi zależnościami a domeną natomiast warstwa adapterów stanowi zewnętrzne technologie komunikujące się z systemem. Na tym poziomie abstrakcji przypomina to implementację architektury warstwowej. Różnie zaczynają się wyodrębniać na bardziej szczegółowym poziomie implementacji.

Rys. 17 Główne pakiety aplikacji napisanej w architekturze heksagonalnej. Źródło: opracowanie własne.

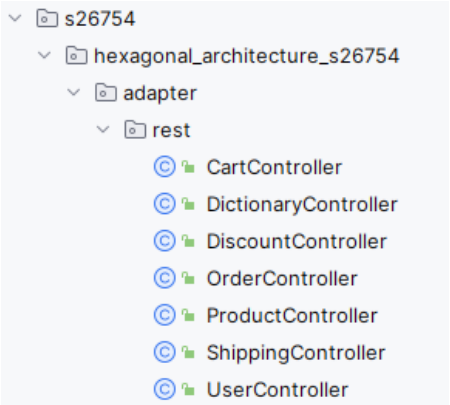
Pakiet domeny podzielony jest na pakiety dedykowane głównym encjom systemu. W tych pakietach znajdują się encje, JPA repozytoria, implementacje tych repozytoriów z podziałem na operacje odczytu i zapisu wraz z wystawionymi portami tych domen. Podział repozytoriów wynika z specyfikacji architektury heksagonalnej, w której jest podział na porty wejściowe i wyjściowe. Taki podział daje większą kontrolę nad zmianami zachodzącymi w encjach i zwiększa modularność kodu. Rys. 18 przedstawia zawartość pakietu wraz z zaprezentowanymi specyfikatorami dostępu. Brak ikony otwartej kłódki przy nazwie klasy oznacza, że jest ona ustawiona na zasięg widoczności w obrębie pakietu (ang.*package – private)* co zapewnia enkapsulację modułu i dostęp do metod encji tylko za pomocą portów.



Rys. 18 Zawartość pakietu Produkt. Źródło: opracowanie własne.

Warstwa aplikacji jest realizuje logikę biznesową zgodnie z wymaganiami systemu. W kontekście tej warstwy, kod jest zorganizowany w sposób modułowy, gdzie każda klasa enkapsuluje konkretne zadanie biznesowe określone w dokumentacji systemowej.

Enkapsulacja ta pozwala na izolację poszczególnych fragmentów funkcjonalności, co prowadzi do wyodrębnienia niezależnych bytów, które reprezentują określone przypadki użycia. W praktyce oznacza to, że każdy przypadek użycia stanowi spójną jednostkę, która niezależnie od innych fragmentów systemu może być rozwijana, testowana i utrzymywana. Takie podejście ma kluczowe znaczenie dla utrzymania przejrzystości i łatwości zarządzania kodem aplikacji. Poprzez modularyzację kodu i izolację funkcjonalności, warstwa aplikacji staje się bardziej elastyczna i łatwiejsza w modyfikacji.



Rys. 19 Warstwa adapterów w aplikacji. Źródło: opracowanie własne

Warstwa adapterów przechowuje logikę zewnętrznych zależności, które komunikują się z warstwą aplikacji. W pracy zastosowano jedną formę komunikacji jaką jest REST API.

7.2. Model domeny

Architektura heksagonalna zwraca szczególną uwagę na oddzielenie części biznesowej od części technicznej systemu. W tym celu analiza warunków biznesowych i sposób mutacji encji domenowych deklaruje się w klasie encji a nie w warstwie aplikacji. Przykład przedstawiono w Kod 5.

Kod 5 - Metody w klasie UserEntity.

```
[…]
public void creditBalance(BigDecimal creditRealAmount,
                          BigDecimal creditFreeAmount) {
    this.realAmount = realAmount.add(creditRealAmount);
    this.freeAmount = freeAmount.add(creditFreeAmount);
}

public void debitBalance(BigDecimal creditRealAmount,
                         BigDecimal creditFreeAmount) {
    validateBalance(realAmount, freeAmount);
    this.realAmount = realAmount.subtract(creditRealAmount);
    this.freeAmount = freeAmount.subtract(creditFreeAmount);
}

public void linkAddressToUser(Long id) {
    this.addressId = id;
}
[…]
```

W tym podejściu to encja deklaruje w jaki sposób może ulec zmianie. Kluczową różnicą od podejścia warstwowego jest brak metod ustawiających każde pole klasy. Nie jest to obowiązkowe założenie architektury heksagonalnej. Taki zabieg stosuje się w podejściu domenowym, które zazwyczaj jest implementowane przy użyciu tej architektury.

Kod 6 - Konfiguracja encji UserEntity.

```
@Getter
@Entity
@Table(name = "t_user")
public class UserEntity {
 […]
```

W ten sposób programista jawnie definiuje dozwolone zachowania w ramach encji. Dodatkowo zastosowano odpowiednio dostosowane i jawnie zdefiniowane konstruktory. Specyfikacja standardu JPA wymaga od programisty zastosowania konstruktora bezparametrowego dla każdej encji. Nie oznacza to jednak, że taki konstruktor ma być pozbawiony logiki, która ustawia domyślne wartości pożądanych pól. Konstruktor bezparametrowy jest wymagany przez JPA, ponieważ framework ten używa go do tworzenia instancji encji poprzez refleksję.

Kod 7 przedstawia przykładowe konstruktory encji płatności. Pierwszy konstruktor jest bezparametrowy – zgodny z wymaganiami JPA. Może on zawierać logikę inicjalizującą domyślne wartości dla niektórych pól encji, co zapewnia spójność stanu nowo utworzonych obiektów. Drugi konstruktor, będący konstruktorom docelowym, jest używany przez system do tworzenia obiektów z pełnym zestawem danych. Konstruktor ten przyjmuje parametry odpowiadające atrybutom encji, umożliwiając pełne zainicjalizowanie obiektu w momencie jego tworzenia.

Kod 7 - Konstruktory encji Payment.

```
public PaymentEntity() {
    this.uuid = UUID.randomUUID();
    this.createDate = OffsetDateTime.now();
}

public PaymentEntity(Long userId,
                     PaymentType paymentType,
                     BigDecimal freeAmount,
                     BigDecimal realAmount) {
    this.uuid = UUID.randomUUID();
    this.createDate = OffsetDateTime.now();
    this.userId = userId;
    this.paymentType = paymentType;
    this.freeAmount = freeAmount;
    this.realAmount = realAmount;
}
```

Ostatnią kluczową różnicą od podejścia warstwowego, charakterystycznego dla architektury heksagonalnej jest odejście od deklaracji relacji w encji i przekazywania ich jako pole innej encji. Wiele źródeł [1,3,7] podaje, że model aplikacji powinien być jak najmniej uzależniony od zewnętrznych technologii. W idealnym scenariuszu oznacza to napisanie klasy w czystej Javie bez wsparcia żadnej adnotacji lub konfiguracji pochodzenia zewnętrznego. W pracy nie zrezygnowano całkowicie z frameworków. We fragmencie Kod 8 zaprezentowano kod encji Zamówienie, która zachowała adnotację `@Entity` w celu objęcia ją przez kontekst Spring Data JPA, dzięki czemu w późniejszym etapie skorzystano z JPARepository i zapytań Query Method i JPQL.

Kod 8 - Pola encji Zamówienie z deklaracją pól kluczy obcych zamiast całej encji.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

@Column(name = "uuid", nullable = false, unique = true)
private UUID uuid;

@Column(name = "order_date")
private OffsetDateTime orderDate;

@Column(name = "cart_id", nullable = false)
private Long cartId;

@Column(name = "shipping_id", nullable = false)
private Long shippingId;
[…]
```

7.3. Zastosowane porty

W aplikacji wykorzystano technologię Spring Data JPA. Każda encja w systemie została oznaczona adnotacją `@Entity`, co umożliwia mechanizmowi skanowania encji identyfikację tych klas i zapewnia integrację z pakietem JPARepository.

Spring Data JPA to rozbudowane narzędzie, które dostarcza kompleksowy zestaw funkcjonalności usprawniających pracę z bazą danych w aplikacji opartej na architekturze JPA. Adnotacja `@Entity` informuje mechanizm ORM (*ang. Object-Relational Mapping*), że klasa Java reprezentuje encję w bazie danych, co pozwala na mapowanie pól klasy na kolumny w tabelach bazodanowych. W konsekwencji, Spring Data JPA umożliwia tworzenie interfejsów repozytoriów (np. repozytoria JPA), które automatycznie generują gotowe metody umożliwiające operacje na encjach, takie jak zapis, odczyt, aktualizacja i usuwanie.

Repozytoria JPA udostępnia dziesiątki metod. Jest to interfejs, który można wstrzykiwać jako zależność bezpośrednio do klas w celu wykonywania operacji na encjach. Zastosowanie tej praktyki może rodzić pewne niebezpieczeństwo ze względu na znaczną odpowiedzialność oraz rozbudowane możliwości, jakie niesie ze sobą pojedyncza klasa. Istnieje ryzyko naruszenia zasady pojedynczej odpowiedzialności. W pracy zastosowano mechanizm publicznych portów do repozytoriów, które ograniczają repozytoria JPA wyłącznie do metod zadeklarowanych przez programistę. W ten sposób programiści nie mają dostępu do nadmiarowych metod. Mogą je zaimplementować, ale dopiero w przypadku wystąpienia takiej potrzeby co jest zgodne z zasadą YAGNI (*ang. You aren't gonna need it*) [8].

Kod 9 przedstawia deklarację portów wejściowych i wyjściowych w systemie. Warstwa aplikacji korzysta z tych portów do pobierania i modyfikowania encji, ale dzieje się to w kontrolowany, odseparowany sposób.

Kod 9 - Deklaracja portów domeny Produkt.

```
@Port
public interface ProductRepositoryInputPort {

    void save(ProductEntity product);

    void delete(UUID productUUID);
}
[...]
@Port
public interface ProductRepositoryOutputPort {

    ProductEntity getByUUID(UUID productUUID);

    List<ProductEntity> findAllByUUIDin(Set<UUID> productList);

    List<ProductEntity> findAllByIdIn(Set<Long> productList);
}
```

Komponent wstrzykujący taki port nie zna szczegółów implementacji. Nie jest to w jego zasięgu odpowiedzialności. Rozdzielenie portów do odczytu i zapisu w kodzie jest przykładem praktyki, która przyczynia się do samodokumentującej natury kodu. Poprzez wyraźne określenie, który port służy do odczytu danych, a który do zapisu, programiści oraz inni czytający kod łatwiej mogą zrozumieć jego funkcjonalność i intencje. Implementacja portu odczytu domeny produkt przedstawiono w Kod 10.

Kod 10 - Implementacja jednego z portów domeny Produkt.

```
@Repository
@RequiredArgsConstructor
class ProductRepositoryOutputPortImpl implements ProductRepositoryOutputPort {

    private final ProductJpaRepository productJpaRepository;

    @Override
    public ProductEntity getByUUID(UUID productUUID) {
        return productJpaRepository.findByUuid(productUUID)
                .orElseThrow(EntityNotFoundException::new);
    }

    @Override
    public List<ProductEntity> findAllByUUIDin(Set<UUID> productList) {
        return productJpaRepository.findAllByUuidIn(productList);
    }

    @Override
    public List<ProductEntity> findAllByIdIn(Set<Long> productList) {
        return productJpaRepository.findALlByIdIn(productList);
    }
}
```

7.4. Adaptery

W pracy zastosowano jeden typ adaptera – REST API. Specyfikacja projektu nie wymagała wprowadzenia dodatkowego adaptera. Założeniem pracy było stworzenie serwisu webowego. Nie oznacza to jednak, że rozwiązanie zostało stworzone tylko pod protokół HTTP. W kontekście teoretycznego rozwoju aplikacji można rozważyć implementację komunikacji asynchronicznej lub komunikacji opartej na zdarzeniach. W obydwu podejściach zastosowano identyczną implementację kontrolerów, przedstawioną w Kod 11.

Kod 11 - Deklaracja kontrolera domeny Użytkownik.

```
@RestController
@RequestMapping(value = "/user")
@RequiredArgsConstructor
public class UserController {

    private final UserFacade userFacade;

    @GetMapping("/{userUUID}")
    public ResponseEntity<UserDTO> getUser(@PathVariable UUID userUUID) {
        return ResponseEntity.ok(userFacade.getUser(userUUID));
    }

    @PostMapping("/register")
    public ResponseEntity<UUID> registerUser(@RequestBody @Valid RegisterUserDTO
dto) {
        UUID user = userFacade.registerUser(dto);
        return ResponseEntity.ok(user);
    }
[…]
```

Jedyną zauważalną różnicą jest wstrzyknięcie zależności fasady zamiast serwisu, ale dla zewnętrznej warstwy aplikacji jaką jest adapter nie ma to znaczenia.

7.5. Przypadki użycia

Każda funkcjonalność systemowa jest zenkapsulowana do oddzielnej klasy, która stanowi przypadek użycia. Te klasy wstrzykują kolejne porty domen, które są wymagane do wykonania operacji a następnie przetwarzają żądanie. Kod 12 przedstawia przypadek użycia zmiany statusu zamówienia. Wśród zależności przypadku widoczne są wyłącznie porty do domen aplikacji. Warto zauważyć, że główną rolę w tym przypadku użycia odgrywa encja `OrderStatusChangeHistoryEntity` i tylko do niej został wstrzyknięty `InputPort` co świadczy o tym, że encja stan encji ulegnie zmianie w bazie danych. Pozostałe domeny korzystają tylko z portów zewnętrznych służących do odczytu.

Kod 12 - Wykonanie przypadku użycia zmiany statusu zamówienia.

```
@UseCase
@RequiredArgsConstructor
class ChangeOrderStatusUseCase {

    private final OrderRepositoryOutputPort orderRepositoryOutputPort;
    private final OrderStatusChangeHistoryRepositoryOutputPort
orderStatusChangeHistoryRepositoryOutputPort;
    private final OrderStatusChangeHistoryRepositoryInputPort
orderStatusChangeHistoryRepositoryInputPort;

    @Transactional
    public void execute(OrderStatusChangeDTO dto) {
        OrderEntity order =
orderRepositoryOutputPort.getByUUID(dto.getOrderUUID());

        OrderStatusChangeHistoryEntity latestStatusChange =
orderStatusChangeHistoryRepositoryOutputPort
                .getAllByOrderId(order.getId())
                .stream()
                .max(comparing(OrderStatusChangeHistoryEntity::getChangeDate))
                .orElse(null);

validateOrderStatusChange(dto, latestStatusChange);
OrderStatusChangeHistoryEntity statusChange =
        new OrderStatusChangeHistoryEntity(
                order.getId(),
                requireNonNull(latestStatusChange).getCurrentStatus(),
                dto.getOrderStatus()
        );
        orderStatusChangeHistoryRepositoryInputPort.save(statusChange);
    }
```

Takie rozwiązanie również sprzyja samodokumentacji. Programista znający to podejście patrząc użyte zależności może domyślić się zakresu działań poszczególnych encji. W całym systemie zaprojektowano przypadki użycia w analogiczny sposób.

Warto jeszcze wspomnieć o serwisach aplikacyjnych, które stanowią pomocnicze rozszerzenie przypadku użycia. Fragment kodu z zastosowaniem serwisu aplikacyjnego zaprezentowano w Kod 13.

Kod 13 - Przypadek użycia wykorzystujący serwis aplikacyjny

```
@UseCase
@RequiredArgsConstructor
class RegisterUserUseCase {

    private final UserRepositoryInputPort userRepositoryInputPort;

    private final AddressInputService addressInputService;

    @Transactional
    public UUID execute(RegisterUserDTO dto) {

        Long addressId = addressInputService.createAddress(
                dto.getAddress()
        );

        UserEntity user = createUser(dto);
        user.linkAddressToUser(addressId);
        userRepositoryInputPort.save(user);

        return user.getUuid();
    }
[…]
```

Klasa `AddressInputService` wskazuje na operacje zapisu Adresu, która jest encją ściśle związaną z Użytkownikiem. W architekturze heksagonalnej przypadki użycie nie mogą wstrzykiwać siebie nawzajem. Są agnostyczne względem siebie. W ramach encji Adresu powstała potrzeba wyodrębniania logiki zmapowania, utworzenia i zapisu adresu co w architekturze kwalifikuje się na przypadek użycia. Rozwiązaniem na takie zagadnienie są serwisy aplikacyjne, które stanowią warstwę pośrednią pomiędzy przypadkami użycia. Kod serwisu zawiera prostą logikę mapowania zewnętrznego modelu DTO *(ang. Data Transfer Object*) na encję domenową i zapis do bazy danych. W ramach analizy przypadków użycia wykryto, że metoda `createAddress` wykorzystywana będzie co najmniej w trzech różnych przypadkach użycia. Zastosowanie warstwy serwisu aplikacyjnego pozwoliło na zachowanie spójności z założeniami architektury heksagonalnej.

Kod 14 – Kod serwisu aplikacyjnego

```
@Service
@RequiredArgsConstructor
public class AddressInputService {

    private final AddressRepositoryInputPort addressRepositoryInputPort;

    public Long createAddress(AddressDTO dto) {
        AddressEntity address;
        if (Objects.nonNull(dto.getFlatNumber())) {
            address = new AddressEntity(
                    dto.getStreetName(),
                    dto.getStreetNumber(),
                    dto.getFlatNumber(),
                    dto.getPostalCode(),
                    dto.getCountry(),
                    dto.getCity()
            );
        }
[…]
        return addressRepositoryInputPort.save(address).getId();
    }
}
```

7.6. Wzorzec fasada

Logika warstwy aplikacyjnej może być skomplikowana i składać się z wielu klas. Zewnętrzny adaptery nie powinny implementować bezpośrednio przypadków użycia, w szczególności, gdy ich liczba jest znacząca. Rozwiązaniem tego problemu jest wzorzec projektowy fasada, która przejmuje odpowiedzialność zarządzania zależnościami przypadków użycia i ukrywa ich implementacje [10]. Dzięki temu zewnętrzne adaptery muszą wstrzyknąć tylko fasadę, aby móc odpytywać warstwę aplikacji. Fasada domeny Użytkownik została zaprezentowana w Kod 15. W przypadku wielu adapterów może zajść potrzeba przygotowania dedykowanej fasady dla każdej z nich w celu izolacji kontekstów. W pracy natomiast jest tylko jedna rodzina adapterów, dlatego każda domena wyposażona jest tylko w jedną fasadę.

Kod 15 - Deklaracja fasady domeny Użytkownik

```
@Facade
@RequiredArgsConstructor
public class UserFacade {

    private final GetUserUseCase getUserUseCase;
    private final RegisterUserUseCase registerUserUseCase;
    private final UpdateUserBasicDataUseCase updateUserBasicDataUseCase;
    private final CreditBalanceUseCase creditBalanceUseCase;
    private final DebitBalanceUseCase debitBalanceUseCase;
    private final ActivateUserUseCase activateUserUseCase;
    private final DeactivateUserUseCase deactivateUserUseCase;
    private final DeleteUserUseCase deleteUserUseCase;
    private final GetTimeWithUsUseCase getTimeWithUsUseCase;
    private final UpdateUserAddressUseCase updateUserAddressUseCaseUse;

    public UserDTO getUser(UUID userUUID){
        return getUserUseCase.execute(userUUID);
    }

    public UUID registerUser(RegisterUserDTO dto) {
        return registerUserUseCase.execute(dto);
    }
[…]
```

Wszystkie przypadki użycia są dostępne w zakresie pakietu dzięki specyfikatorowi dostępu package-private, jedynie fasada wstrzykująca przypadki użycia jest publiczna. Dzięki temu, szczegóły implementacyjne pozostają ukryte, a jedynie interfejs zewnętrzny jest widoczny dla innych pakietów.

7.7. Obsługa wyjątków

W aplikacji obowiązują określone reguły biznesowe. Na wejściu do aplikacji wykorzystywane są modele DTO, które stanowią ustandaryzowany sposób komunikacji z serwisem. W tych modelach przeprowadzane są wstępne walidacje pól wejściowych, takie jak sprawdzenie istnienia pola lub poprawności jego wartości. W warstwie DTO nie waliduje się reguł biznesowych, ponieważ te powinny być określone dopiero w warstwie domenowej. W niniejszej pracy walidacja reguł biznesowych została zadeklarowana w encjach, ponieważ tam odbywają się operacje biznesowe. W przypadku niespełnienia warunków, które są w systemie nieoczekiwana dochodzi do rzucenia wyjątku domenowego z odpowiednim komunikatem i kodem. Mechanizm ten został stworzony na abstrakcyjnej bazie `DomainOperationException`, zaprezentowanej w Kod 16.

Kod 16 - Deklaracja abstrakcyjnego wyjątku domenowego

```
@Getter
public abstract class DomainOperationException extends RuntimeException {

    private final String message;

    protected DomainOperationException(String message) {
        this.message = message;
    }

    public abstract String getErrorCode();
}
```

Przykładowa implementacja takiego wyjątku przez domenę Produkt została przedstawiona w Kod 17. W systemie została również stworzona klasa nasłuchująca na rzucane wyjątki w systemie w celu konwersji odpowiedzi HTTP zgodnie z wymaganiami systemu.

Kod 17 - Deklaracja implementacji wyjątku domenowego przez encję Produkt

```
class ProductDomainOperationException extends DomainOperationException {

    private final ProductDomainOperationErrorCode errorCode;

    public ProductDomainOperationException(
            ProductDomainOperationErrorCode errorCode,
            String message) {
        super(message);
        this.errorCode = errorCode;
    }

    @Override
    public String getErrorCode() {
        return errorCode.toString();
    }

    public enum ProductDomainOperationErrorCode {
        QUANTITY_UNIT_MUST_BE_PROVIDED,
        QUANTITY_MUST_BE_PROVIDED,
        QUANTITY_UNIT_INVALID,
        NEGATIVE_QUANTITY_PROVIDED
    }
}
```

Domyślnie aplikacje webowe w Springu rzucają wyjątek 500 w przypadku nieobsłużenia go przez programistę co oznacza, że błąd występuje po stronie serwera [11]. To zachowanie domyślne należy nadpisać w celu poprawnego skonfigurowania API oraz zwrócenia informacji na temat przetworzonego żądania. Kod 18 przedstawia implementację wzorca obserwator [10], który jest interceptorem rzucanych wyjątków w systemie.

Kod 18 - Interceptor wyjątków w systemie

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ResponseBody
    @ExceptionHandler(DomainOperationException.class)
    public ResponseEntity<ErrorDTO>
handleDomainOperationException(DomainOperationException exception) {

        return ResponseEntity.status(HttpStatus.PRECONDITION_FAILED)
                .body(
                        new ErrorDTO(
                                ErrorType.BUSINESS_VALIDATION,
                                exception.getMessage(),
                                exception.getErrorCode()
```

```
                        )
                );
        }
        @ResponseBody
        @ExceptionHandler(EntityNotFoundException.class)
        public ResponseEntity<ErrorDTO>
handleEntityNotFoundException(EntityNotFoundException exception) {

                return ResponseEntity.status(HttpStatus.NOT_FOUND)
                        .body(
                                new ErrorDTO(
                                        ErrorType.UNEXPECTED,
                                        exception.getMessage(),
                                        null
                                )
                        );
        }
}
```

W pracy zaprojektowano mapowanie wyjątku `EntityNotFoundException` oraz abstrakcyjnego `DomainOperationException` na dokładniejsze kody HTTP. Nieznalezienie szukanego zasobu zwraca kod 404 z typem technicznym `UNEXPECTED` natomiast wyjątki domenowego mapowane są na 412 co ma zasugerować niepoprawność żądania.

8. Porównanie wyników

Porównanie implantacji przeprowadzono za pomocą zróżnicowanych kryteriów. Wspólnym założeniem obydwu rozwiązań było zastosowanie takiego samego modelu danych oraz identycznego REST API w celu zachowania takiego samego zakresu logiki biznesowej. W ocenie osiągniętych wyników wzięto pod uwagę:

- ilość kodu (liczba klas, linie kodu)
- ilość wstrzykiwanych komponentów (siatka zależności)
- złożoność kodu
- podatność na zmiany
- podatność na rozszerzanie
- testowalność
- skalowalność
- błędogenność programisty
- złożoność rozwiązania

8.1. Analiza metryk kodu

Do przeprowadzenia porównania metryk kodu skorzystano z wtyczki do programu IntelliJ IDEA – Statistic. Do zestawienia skorzystano z ilości klas, niepustych linii kodu i średniej liczby linii kodu na klasę. Takie zestawienie przygotowano dla całego projektu oraz z granulacją per pakiet oraz per domena w celu precyzyjnego przedstawienia różnic w objętości kodu.

Tab. 1 Zestawienie metryk całego projektu

	Architektura Warstwowa	Architektura Heksagonalna	Różnica ilościowa	Różnica procentowa [%]
Liczba klas	128	203	75	58,59
Najmniej linii	4	4	0	0
Najwięcej linii	148	165	17	11,49
Średnia liczba linii kodu na klasę	31	28	-3	-9,68
Niepuste linie kodu	3007	4428	1421	47,26

Implementacja architektury heksagonalnej wymagała stworzenia o prawie 59% więcej klas co przełożyło się na ponad 47% więcej linii kodu. Ten wynik jest efektem rozdzielenia odpowiedzialności portów oraz utworzenia dedykowanych klas dla przypadków użycia w aplikacji. Każda taka klasa poza deklaracją musi wstrzyknąć odpowiednie zależności co jest zliczane w ramach agregacji kodu przez wtyczkę. W tabeli 2 przedstawiono zestawienie na poziomie warstwy logiki biznesowej, czyli zestawiono warstwę aplikacji z warstwą serwisy domeny Zamówienie.

Tab. 2 Zestawienie metryk pakietu zamówienie

	Architektura Warstwowa	Architektura Heksagonalna	Różnica ilościowa	Różnica procentowa [%]
Pakiet Zamówienie	207	393	186	90
Utworzenie zamówienia	76	111	35	46
Zmiana statusu zamówienia	22	44	22	100
Usunięcie zamówienia	2	12	10	500
Pobranie zamówienia	48	76	28	58

Do porównania wybrano pakiet Zamówienie, ponieważ jest to najbardziej złożony pakiet systemu. W większości projektów małych i średnich, a także tych opartych na architekturze mikroserwisów, w modelu zauważalna jest kluczowa encja, która pełni centralną rolę w zarządzaniu danymi i procesami. Taka encja zazwyczaj odpowiada za gromadzenie i integrację informacji z różnych modułów systemu, zapewniając spójność i efektywność operacji. W systemie zaprojektowanym w ramach niniejszej pracy tę centralną rolę pełni encja Zamówienie, która integruje dane dotyczące klientów, produktów, płatności i dostaw, stanowiąc fundament dla działania całego systemu. Różnice metry w implementacji tego pakietu znacznie przekraczają wyniki z porównania globalnego, które stanowią wynik średni.

Różnice te wynikają z liczby zależności jakie należy wstrzyknąć do przypadków użycia i metodologii operacji na danych.

Kod 19 - Fragment kodu metody tworzenia zamówienia w podejściu heksagonalnym

```
//Podejście warstwowe
[…]
public OrderDTO getByUUID(UUID orderUUID) {
    OrderEntity order = orderRepository.getByUUID(orderUUID);
    return orderMapper.toOrderDTO(order);
}
[…]

//Podejście heksagonalne
[…]
public OrderDTO execute(UUID orderUUID) {
    OrderEntity order = orderRepositoryOutputPort.getByUUID(orderUUID);
    CartEntity cart = cartRepositoryOutputPort.getById(order.getCartId());
    ShippingEntity shipping =
shippingRepositoryOutputPort.getById(order.getShippingId());
    UserEntity user = userRepositoryOutputPort.getById(cart.getUserId());

    return new OrderDTO(
            toBaseUserDataDTO(user),
            createProductOrderDTOList(cart.getId()),
            toShippingDTO(shipping)
    );
}
[…]
```

W architekturze warstwowej zastosowano Spring JPA do mapowania encji, dzięki czemu do obiektów zagnieżdżonych relacją można dostać się jak do zwykłego pola klasy. W rozwiązaniu opartym na architekturze heksagonalnej, zamiast przekazywać obiekt, przekazywany jest identyfikator obiektu, co wymaga od programisty wstrzyknięcia portu służącego do odczytu oraz jawnego pobrania obiektu zagnieżdżonego. Takie podejście wspiera modularność, jest zgodne z ideą luźnego wiązania (*ang. loose coupling*) i zapobiega zależnościom technologicznym. Co więcej w ten sposób programista jest bardziej świadomy tego co dzieje się w procesie i jakie encje są wymagane do utworzenia zamówienia.

Metoda napisana w taki sposób przypomina niejawne wywołania zapytań SQL, które Hibernate wykonuje pod spodem w podejściu warstwowym. Takie podejście jest zdecydowanie bardziej skomplikowane, ponieważ wymaga od programisty większej pracy, dbałości o więcej szczegółów oraz kompleksową wiedzę na temat relacyjnych baz danych i transakcji [24, 25].

Powyższe rozwiązanie, połączone z zaimplementowaną segregacją portów i adapterów, ma dodatkową korzyść polegającą na uniemożliwieniu modyfikacji encji, dla których nie został wstrzyknięty odpowiedni port do zapisu. Taka funkcjonalność została osiągnięta poprzez wykorzystanie referencji przez identyfikator, co omija mechanizm Hibernate oferujący kaskadowość

operacji. Jest to istotne, gdyż wiele operacji, które programiści wykonują przy użyciu tego frameworku, może odbywać się bez ich pełnej świadomości co do procesów zachodzących niejawnie.

Warto również zauważyć, że metody oznaczone adnotacją `@Transactional` w Springu nie muszą bezpośrednio wywoływać JPARepository w celu zapisania zmian stanu encji. Dzieje się tak, ponieważ podczas wykonywania transakcji Hibernate korzysta z mechanizmu brudnego sprawdzania *(ang. Dirty Check*). Proces ten polega na porównaniu projekcji encji przed i po wykonaniu się transakcji, co umożliwia automatyczne wykrycie zmian i ich zapisanie w bazie danych [26].

Wszystkie te czynniki sprawiają, że w mniej skomplikowanych pakietach, które korzystają z mniejszej liczby zależności zewnętrznych zauważono mniejsze różnice w ilości kodu. Tabela 3 stanowi zestawienie metryk pakietu Zamówienie, który korzysta wyłącznie z zależności do tabeli Adres.

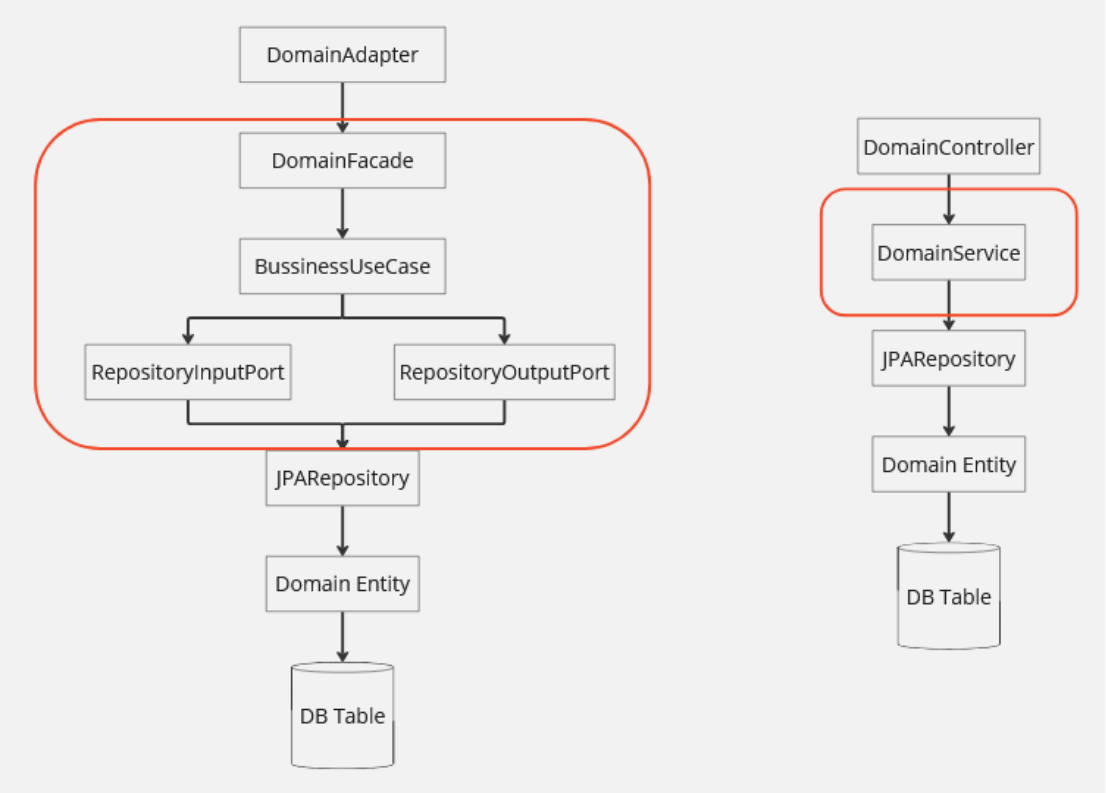
Tab. 3 Zestawienie metryk pakietu dostawa

	Architektura Warstwowa	Architektura Heksagonalna	Różnica ilościowa	Różnica procentowa [%]
Pakiet dostawa warstwy logiki biznesowej	175	232	57	33
Utworzenie dostawy	44	44	0	0
Aktualizacja dostawy	40	49	9	23
Usunięcie dostawy	2	2	0	0
Pobranie dostawy	30	31	1	3

Operacje w pakiecie dostawa są podstawowymi operacjami CRUD (*ang. Create, Read, Update, Delete*), które przyjmują żądanie, przekształcają model wejściowy na model bazodanowy i dokonują zapisu do bazy danych.

8.2. Struktura kodu

Obydwa rozwiązania charakteryzuje zróżnicowana struktura kodu. W architekturze warstwowej są jawnie wyodrębnione granice odpowiedzialności danych klas natomiast w architekturze heksagonalnej podział ten jest zrealizowany w inny sposób. Porównanie pojedynczego pakietu domenowego zaprezentowano na Rys. 20.



Rys. 20 Porównanie struktury kodu obydwu rozwiązań. Źródło: opracowanie własne

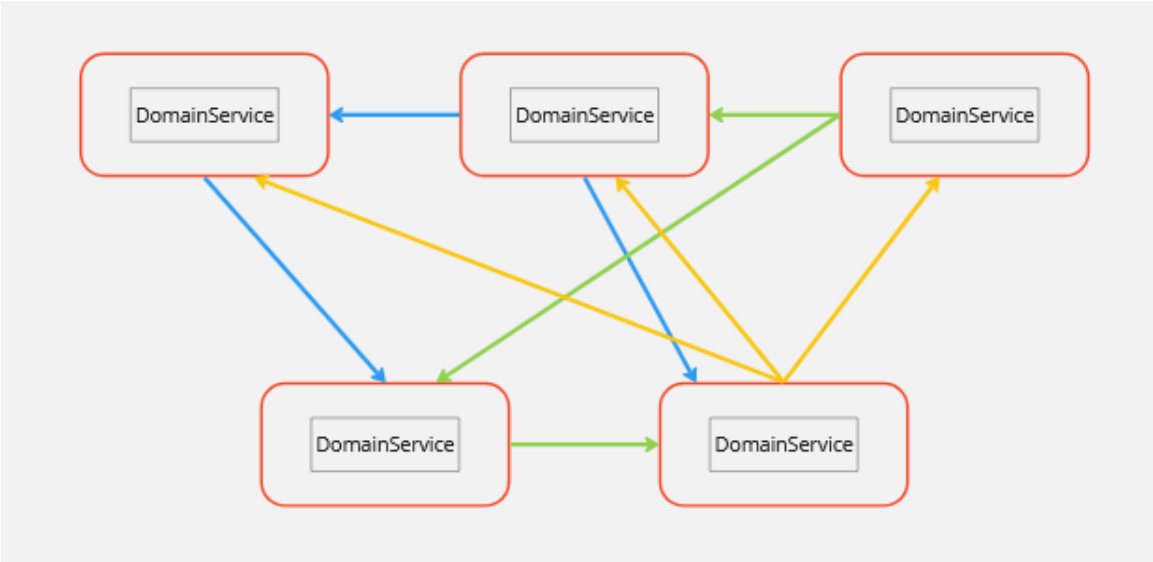
Kolorem czerwonym zaznaczono obszary kodu, w których występują największe różnice. Tymi obszarami są fragmenty realizujące logikę biznesową, które jednocześnie mogą wywoływać kontekst innych zależności. Przestrzeń wykonywania logiki biznesowej w architekturze warstwowej to po prostu warstwa serwisowa, która odpowiada za przechowywanie metod realizujących operacje zapisu, odczytu czy modyfikacji encji. Takie podejście jest bardzo wygodne w przypadku nieskomplikowanego modelu bazodanowego, niewielkiego projektu lub odpowiednio podzielonego systemu w architekturze mikroserwisów.

W architekturze heksagonalnej przestrzenią realizującą logikę biznesową jest „warstwa" aplikacji, która składa się z portów wejścia i wyjścia, fasad oraz przypadków użycia. Jest to podejście bardziej skomplikowane wymagające większej ilości powtarzalnego kodu w celu stworzenia samego szablonu.

8.3. Skalowalność

Obie architektury różnią się pod względem skalowalności, co wynika z sieci zależności pomiędzy komponentami systemu. W architekturze warstwowej domeny krzyżują się w jednym punkcie - warstwie serwisów, co naturalnie prowadzi do wzajemnego wstrzykiwania serwisów. Jest to rozwiązanie elastyczne, które umożliwia swobodne dodawanie nowych funkcjonalności poprzez rozbudowę warstwy serwisów.

Natomiast w architekturze heksagonalnej wstrzykiwanie zewnętrznych zależności jest enkapsulowane w przypadkach użycia, co sprzyja izolacji komponentów i ułatwia modyfikację poszczególnych funkcji. Warstwa fasady, analogiczna do warstwy odpowiedzialności w architekturze warstwowej, jest neutralna wobec implementacji przypadków użycia i ich zależności. Jest to warstwa abstrakcji, która pozwala na oddzielenie interfejsu użytkownika od logiki biznesowej, co z kolei ułatwia skalowanie aplikacji poprzez możliwość szybkiego dostosowania do zmieniających się wymagań i warunków środowiska. Rys. 21 zaprezentuje przykładową siatkę zależności serwisów w ramach wykonywania poszczególnych procesów oznaczonych unikalnymi kolorami.

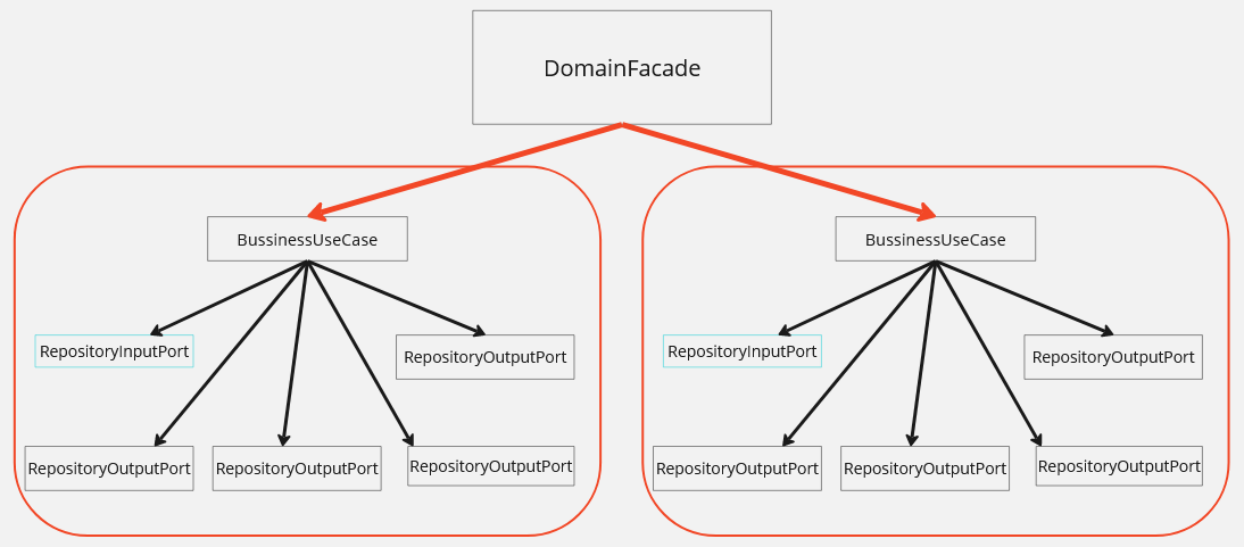


Rys. 21 Siatka zależności w warstwie serwisów rozwiązania warstwowego

Kolejnym istotnym wyzwaniem w kontekście skalowalności rozwiązania opartego na architekturze warstwowej jest narastająca liczba publicznych metod, co w konsekwencji może prowadzić do łamania zasady pojedynczej odpowiedzialności (*ang. Single Responsibility Principle)*. Oznacza to, że klasa może być odpowiedzialna za zbyt wiele zadań, co utrudnia jej utrzymanie i modyfikację w przyszłości. Dodatkowo, często stosowanym zabiegiem jest ekstrakcja fragmentów kodu do metod prywatnych w celu zwiększenia czytelności oraz próby wyodrębnienia wspólnej logiki. Niemniej jednak, nadmierne korzystanie z tego podejścia może prowadzić do nadmiernego rozrostu rozmiaru klasy, co może negatywnie wpłynąć na jej złożoność oraz skomplikować proces rozwijania serwisu. W rezultacie, konieczne staje się zachowanie umiaru i regularna refaktoryzacja kodu w celu zapewnienia jego czytelności, elastyczności i łatwości rozbudowy.

Architektura heksagonalna, poprzez enkapsulację zewnętrznych zależności na poziomie dedykowanych przypadków użycia, skutecznie zapobiega problemowi powstawania złożonej siatki zależności. W tej architekturze każda klasa jest traktowana jako odrębny, niezależny byt, co prowadzi do wyraźnego rozdzielenia odpowiedzialności i zapewnia klarowność struktury systemu.

Dzięki temu podejściu, klasy w architekturze heksagonalnej są odpowiedzialne wyłącznie za konkretne zadania zdefiniowane w ich przypadkach użycia. Enkapsulacja zewnętrznych zależności oznacza, że te klasy są izolowane od zmian w innych częściach systemu lub od zewnętrznych czynników, co przyczynia się do zwiększenia elastyczności i łatwości rozbudowy systemu.



Rys. 22 Siatka zależności w architekturze heksagonalnej

Architektura warstwowa narażona jest na zbudowanie trudnej do utrzymania siatki zależności natomiast architektura heksagonalna jest zaprojektowana w taki sposób, że nawet usunięcie całego przypadku użycia nie powinno wypłynąć na integralność pozostałych obszarów systemu [27].

8.4. Testowalność

Testy zostały opracowane zgodnie z metodologią testów jednostkowych i integracyjnych. Każda metoda warstwy usługowej została poddana testowi, który sprawdzał poprawny przebieg scenariusza (*ang. Happy Path*), a także niektóre scenariusze sprawdzające walidacje reguł biznesowych (*ang. Unhappy Path*). Podczas procesu tworzenia oprogramowania, nie zaobserwowano istotnych różnic między zastosowanymi architekturami. Zarówno w architekturze warstwowej, jak i heksagonalnej, wyodrębnienie klas odpowiedzialnych za logikę biznesową (takich jak serwisy i fasady) umożliwiło testowanie logiki biznesowej w konwencji given, when, then. Główne różnice wynikały z konieczności wstrzykiwania portów do testów integracyjnych w celu zapewnienia wymaganych zależności dla asercji weryfikujących poprawność testu.

Kod 20 - Porównanie zależności w klasie testów integracyjnych domeny Użytkownik

```
//Podejście warstwowe
[…]
@Autowired
UserService userService;

@Autowired
UserJpaRepository userJpaRepository;

@Autowired
PaymentRepositoryImpl paymentRepository;

[…]
```

```
//Podejście heksagonalne
[…]
@Autowired
UserFacade userFacade;

@Autowired
UserRepositoryOutputPort userRepositoryOutputPort;

@Autowired
AddressRepositoryOutputPort addressRepositoryOutputPort;

@Autowired
PaymentRepositoryOutputPort paymentRepositoryOutputPort;

@Autowired
DictionaryOutputService dictionaryOutputService;
[…]
```

W przypadku bardziej rozbudowanych testów złożonej logiki biznesowej, rozwiązanie oparte na architekturze heksagonalnej przynosi korzyść poprzez możliwość enkapsulacji testów w obrębie konkretnych przypadków użycia, a nie całego serwisu aplikacyjnego.

Dzięki architekturze heksagonalnej, gdzie logika biznesowa jest enkapsulowana w dedykowanych przypadkach użycia, testowanie staje się bardziej precyzyjne i zorganizowane. Możemy skoncentrować się na testowaniu konkretnych funkcjonalności lub scenariuszy bez konieczności uwzględniania całego kontekstu aplikacji. Każdy przypadek użycia stanowi klarowną jednostkę testową, co ułatwia zarządzanie testami i identyfikację ewentualnych błędów. W przypadku bardziej złożonej logiki biznesowej, gdzie funkcjonalności mogą być powiązane i zależne od siebie, enkapsulacja testów w obrębie przypadków użycia pozwala na izolację testów oraz uniezależnienie ich od innych części systemu. To z kolei ułatwia utrzymanie testów, ponieważ zmiany w jednym przypadku użycia nie mają wpływu na inne testy.

9. Podsumowanie

W ramach niniejszej pracy został opracowany projekt, który można zakwalifikować jako mały projekt informatyczny. Projektowanie systemu informatycznego wymaga zawsze uwzględnienia potencjalnego rozwoju aplikacji oraz możliwości modyfikacji dostępnych zasobów w przypadku wzrostu obciążenia systemu. Wybór odpowiedniej architektury zależy także od czynników takich jak początkowa, docelowa i potencjalna skala projektu.

Podczas projektowania małych projektów informatycznych, należy szczególnie zwrócić uwagę na elastyczność i skalowalność systemu. Nawet jeśli projekt zaczyna się jako mały, należy przewidzieć możliwość jego rozbudowy w przyszłości, aby sprostać zmieniającym się wymaganiom biznesowym oraz wzrostowi liczby użytkowników. W tym kontekście kluczową rolę odgrywa właściwy dobór architektury, która umożliwi łatwe dostosowanie systemu do ewentualnych zmian. Dodatkowymi czynnikami przy tworzeniu oprogramowania jest liczba osób w zespole, czas i budżet jaki jest przeznaczony na projekt. Oprogramowanie realizujące te same cele i funkcjonalności można stworzyć na wiele różnych sposobów, które różnić się będą sposobem realizacji.

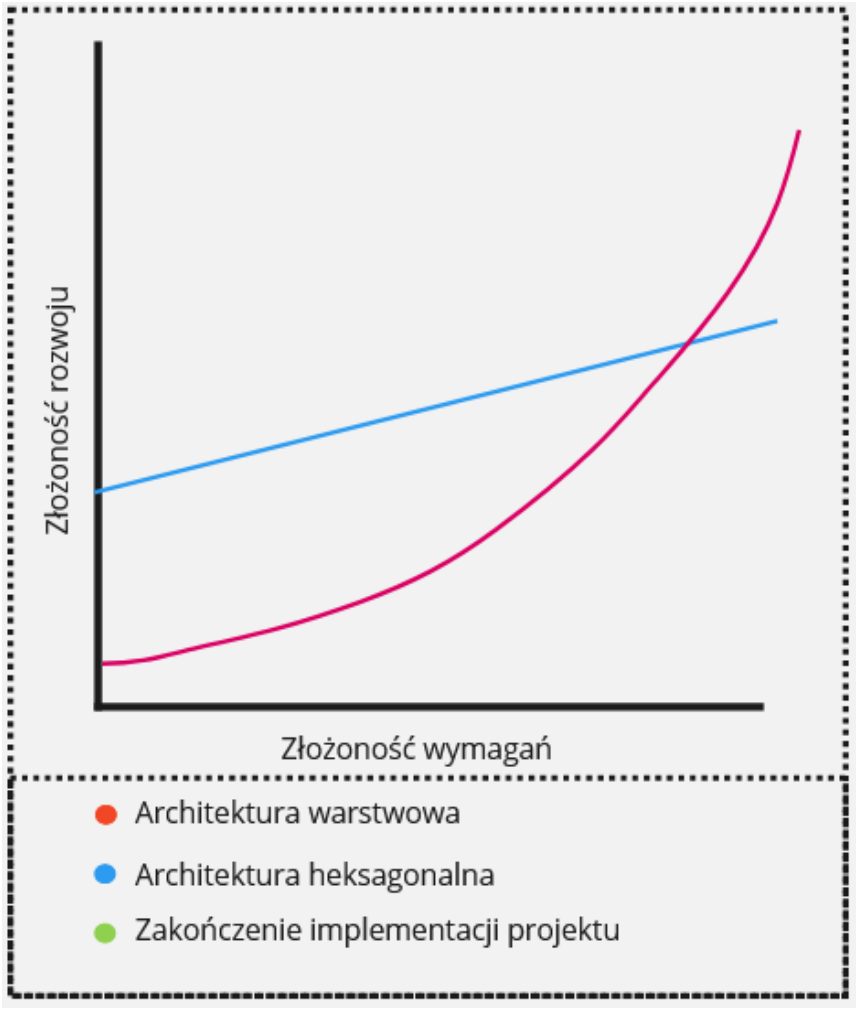
Architektura warstwowa, będąca metodologią projektowania oprogramowania, wyróżnia się swoją łatwością implementacji, intuicyjnym podejściem oraz szerokim zastosowaniem w praktyce. W dzisiejszych czasach, artykuły naukowe oraz poradniki programistyczne często opierają się na architekturze warstwowej, wykorzystując ją w swoich materiałach. To sprawia, że większość osób rozpoczynających swoją przygodę z programowaniem staje się z nią zaznajomiona. Dzięki temu, nowi programiści mogą szybciej zrozumieć złożoność projektowania oprogramowania oraz efektywniejsze podejście do rozwiązywania problemów. Jest to łatwe rozwiązanie pozwalające osiągnąć działający systemu w stosunkowo krótkim czasie. Umiejętnie napisana architektura warstwowa umożliwia wydzielenie kluczowych elementów (na przykład warstwy serwisu) do oddzielnego komponentu aplikacji. Modularność na takim poziomie jest niezwykle przydatna we współczesnych rozwiązaniach opartych o architekturę mikroserwisów, której założeniem jest wydzielanie komponentów systemu do jak najmniejszych jednostek [28].

Architektura warstwowa narażona jest na przeplatanie się logiki różnych tabel czy elementów danych w poszczególnych warstwach. Takie splątanie może prowadzić do utrudnienia czytelności procesu oraz narastania zależności między poszczególnymi komponentami systemu. W rezultacie może dojść do powstania tzw. "siatki zależności", gdzie zmiana w jednym elemencie architektury wymaga modyfikacji w wielu innych miejscach, co utrudnia zarządzanie systemem i wpływa negatywnie na jego skalowalność. Problem ten jest szczególnie istotny w kontekście rosnącej skali systemu oraz złożoności jego funkcjonalności. W miarę rozwoju projektu, liczba interakcji między poszczególnymi elementami systemu może wzrastać, co prowadzi do wzrostu ryzyka utraty klarowności w projektowaniu oraz trudności w utrzymaniu i rozbudowie systemu. W celu radzenia sobie z tym wyzwaniem, istotne jest zastosowanie odpowiednich technik i praktyk projektowych, takich jak np. zastosowanie wzorców projektowych, ścisłe oddzielenie odpowiedzialności poszczególnych warstw lub migracja monolitycznej aplikacji do mikroserwisów.

Architektura heksagonalna, w swoich teoretycznych założeniach, stanowi odpowiedź na wiele problemów, które mogą pojawić się w architekturze warstwowej. Jest to podejście bardziej zaawansowane i złożone, które oferuje większą elastyczność i możliwości, jednakże wymaga również bardziej zaawansowanego doświadczenia i głębszego zrozumienia przez zespół deweloperski. W przeciwieństwie do architektury warstwowej, która opiera się na prostym podziale systemu na warstwy, architektura heksagonalna proponuje bardziej abstrakcyjne podejście, w którym aplikacja jest rozumiana jako zbiór wewnętrznych funkcji i operacji, niezależnych od konkretnych warstw czy technologii. Jest to rozwiązanie o większej elastyczności, co pozwala na różnorodne sposoby implementacji, w zależności od potrzeb i kontekstu projektu. Jednakże, złożoność

architektury heksagonalnej oraz jej elastyczność mogą być również przyczyną problemów implementacyjnych. Wymaga to od zespołu deweloperskiego większej wiedzy oraz umiejętności, aby właściwie zastosować tę architekturę w praktyce. Brak doświadczenia lub niezrozumienie założeń projektowych może prowadzić do błędów w implementacji oraz trudności w utrzymaniu i rozwijaniu systemu. Ważnym aspektem w podejściu opartym na architekturze heksagonalnej jest konieczność utrzymania spójności w obranej konwencji projektowej przez cały zespół programistyczny. Konsekwencja w stosowaniu przyjętych założeń oraz wspólna konwencja programistyczna są kluczowe dla zachowania jakości kodu i zapobieżenia chaotycznemu rozwojowi aplikacji. Utrata kontroli nad przyjętą konwencją może prowadzić do pogorszenia czytelności kodu, zwiększenia ryzyka błędów oraz utrudnienia współpracy między członkami zespołu.

Architektura heksagonalna zajmuje więcej czasu na wdrożenie, wymaga większego nakładu pracy analitycznej i implementacyjnej. W niniejszej pracy rozwiązanie oparte o architekturę heksagonalną zajęło prawie 60% więcej kodu. Założeniem pracy było stworzenie systemu z zastosowaniem takiego samego modelu bazodanowego, pomimo tego, że architektura heksagonalna często implementowana jest w ramach rozwiązań korzystających z podejścia DDD. Motywacją do takiej decyzji była chęć przeanalizowania zalet i wad obydwu rozwiązań. Rys. 23 zawiera wykres będący opracowaniem własnym z publikacji Martina Flowera, który twierdzi, że podejście czystej architektury i skupienia się na domenie przynosi korzyści dopiero w dużych projektach.

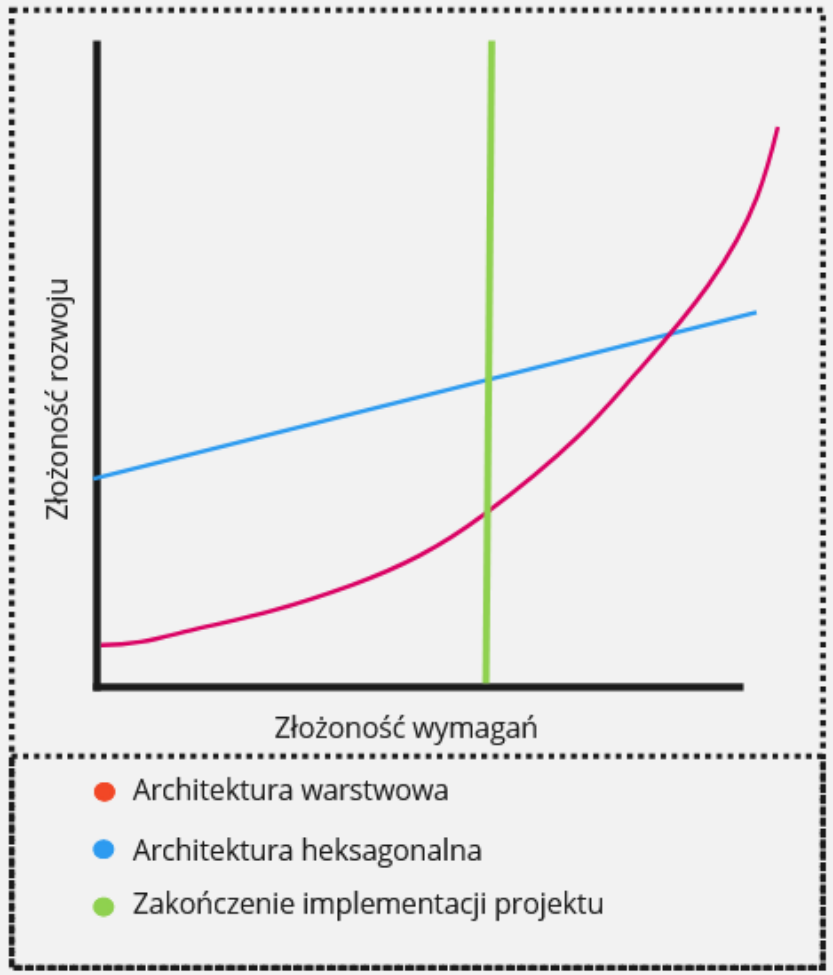


Rys. 23 Stosunek złożoności rozwoju projektu do złożoności wymagań systemu.

Źródło: Opracowanie własne na podstawie [1]

W pracy wykazano zgodność z tym stwierdzeniem. potwierdzając, że w mniejszych projektach korzyści te mogą być mniej zauważalne. Z drugiej strony większych systemach czysta architektura przyczynia się do lepszej organizacji kodu, łatwiejszego zarządzania złożonością oraz zwiększonej skalowalności, co przekłada się na efektywność i trwałość projektu w dłuższej perspektywie czasowej. W pracy nie doprowadzono do sytuacji, w której architektura warstwowa stała się ciężka w utrzymaniu. Finalna wersja aplikacji oferuje podstawowe operacje prowadzenia sklepu internetowego z wykorzystaniem jednego modułu aplikacji (podejście monolityczne). W przypadku doprowadzenia do komplikacji w kodzie zalecana byłaby refaktoryzacja lub podział systemu na mikroserwisy.

Posługując się wykresem z Rys. 23, stwierdzono, że realizowane projekty znajdują się przed punktem przecięcia linii wykresu, co sugeruje przewagę architektury warstwowej. Zostało to zilustrowane na Rys. 24.



Rys. 24 Wpasowanie zakończonego etapu projektu w wykres
Źródło: opracowanie własne na podstawie [1].

W pracy udowodniono, że nie zawsze należy wytwarzać oprogramowanie z zachowaniem wszystkich najlepszych praktyk w celu osiągnięcia pożądanego rezultatu. Ponieważ może to niepotrzebnie wydłużyć i skomplikować czas realizacji projektu. Dobór architektury oprogramowania powinien być dostosowany do potrzeb klienta, skali i specyfikacji domeny projektu.

Bibliografia

[1] Flower M. “Patterns of Enterprise Application Architecture”, wydanie z 2002r
[2] Evans E. “Domain-Driven Design: Tackling Complexity in the Heart of Software”, wydanie z 2003r
[3] Meyer B. “Object-oriented Software Construction” wydanie z 1997r
[4] Robert C. Martin Clean Architecture: A Craftsman's Guide to Software Structure and Design
[5] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou, Stanford University 1984: Vertical Partitioning Algorithms for Database Design"
[6] AT&T: Database management system with active data dictionary
[7] Artykuł o kontrolowaniu mapowania encji w Hibernate https://dev-vibe.medium.com/think-twice-before-you-map-entity-associations-with-hibernate-a09173dda9c0 [dostęp z 31.05.2024]
[8] Extreme Programming Installed, Jeffries, Ronald E
[9] Bloch, J. (2017). Java. Efektywne programowanie. Wydawnictwo Helion.
[10] Shvets, A. (2019). ”Dive Into Design Patterns”. https://refactoring.guru/pl/design-patterns/book
[11] Dokumentacja HTTP, [dostęp z 31.05.2024]https://httpwg.org/specs/
[12] Java, dokumentacja, [dostęp z 31.05.2024]
https://docs.oracle.com/en/java/javase/17/docs/api/inde .html
[13] PostgreSQL, dokumentacja [dostęp z 31.05.2024] https://www.postgresql.org/docs/
[14] Spring, dokumentacja [dostęp z 31.05.2024] https://docs.spring.io/spring framework/reference
[15] Hibernate, dokumentacja [dostęp z 31.05.2024] https://hibernate.org/orm/documentation/6.2/
[16] JUnit, dokumentacja [dostęp z 31.05.2024] https://junit.org/junit5/docs/current/user-guide/
[17] H2, dokumentacja [dostęp z 31.05.2024] https://h2database.com/html/quickstart.html
[18] Liquibase, dokumentacja [dostęp z 05.06.2024] https://docs.liquibase.com/home.html
[19] Spock, dokumentacja [dostęp z 31.05.2024]
https://spockframework.org/spock/docs/2.3/index.html
[20] Lombok, dokumentacja [dostęp z 05.06.2024] https://projectlombok.org/api/
[21] MapStruct dokumentacja [dostęp z 05.06.2024] https://mapstruct.org/documentation/
[22] Swagger, dokumentacja [dostęp z 31.05.2024] https://swagger.io/docs/
[23] Artykuł o anemicznym modelu danych, [dostęp z 31.05.2024] https://www.baeldung.com/java-anemic-vs-rich-domain-objects
[24] Artykuł o błędach przy pracy z Hibernate [dostęp z 31.05.2024] https://thorben-janssen.com/5-common-hibernate-mistakes-that-cause-dozens-of-unexpected-queries/
[25] Artykuł o dobrych praktykach pracy z Hibernate [dostęp z 31.05.2024] https://thorben-janssen.com/hibernate-best-practices/
[26] Artykuł o dirty checking [dostęp z 31.05.2024] https://medium.com/jpa-java-persistence-api-guide/dirty-checking-magic-in-hibernate-how-it-works-and-why-its-important-3cdb422dc4d4
[27] Artykuł o siatce zależności w architekturze warstwowej [dostęp z 31.05.2024]
https://medium.com/@johnnywiller10/layered-architecture-and-service-based-development-why-no-9378b146b0ef
[28] Artykuł o architekturze warstwowej z mikroserwisami [dostęp z 31.05.2024]
https://medium.com/microservices-in-practice/microservices-layered-architecture-88a7fc38d3f

Spis Ilustracji

Spis kodu źródłowego

Spis Tabel