POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Marcin Lewandowski
Nr albumu s20082

Porównanie architektury oprogramowania opartej o wzorzec Domain Driven Design z architekturą opartą o anemiczny model danych

Praca magisterska
pod kierunkiem:

Dr. Mariusza Trzaski

Warszawa, lipiec, 2020

Streszczenie

Praca stanowi porównanie dwóch rodzajów architektury oprogramowania – *Domain Driven Design* i architektury opartej o anemiczny model danych. Pierwszy rozdział to wstęp teoretyczny do tematyki *Domain Driven Design* opisujący fundamenty i podstawowe założenia tego podejścia. Dalsza część pracy opisuje kilka wzorców architektonicznych wykorzystywanych często w parze z *Domain Driven Design*, w tym CQRS, który został wykorzystany w przykładowym projekcie porównawczym. Projekt porównawczy to prosta aplikacja typu WebAPI do obsługi pizzerii, w dwóch wersjach, odpowiadających każdej z porównywanych architektur. Obie wersje zostały opisane z perspektywy wysokopoziomowej, jak i szczegółowo, według odpowiednich warstw z podkreśleniem technicznych aspektów zastosowanych wzorców. Końcowa część pracy to właściwe porównanie obu implementacji pod kątem kilku parametrów związanych z jakością, czytelnością, złożonością i poziomem ryzyka związanym z wdrożeniem każdej z opcji.

Słowa kluczowe: Domain Driven Design, CQRS, Clean Architecture, model anemiczny, architektura oprogramowania

Spis treści

1. Wstęp

Świadomość istnienia wzorców architektonicznych jest niezbędna, aby można było wytwarzać wysokiej jakości oprogramowanie, dostosowane do dynamicznych zmian wymagań biznesowych oraz zmiennej skali liczby użytkowników korzystających z systemów informatycznych. Moje doświadczenie zawodowe oraz prywatna działalność związana z programowaniem obfitują w systemy oparte o tradycyjny, trójwarstwowy model oprogramowania, który jako centrum determinujące kształt innych warstw stawia warstwę dostępu do danych. Niestety wykorzystywanie tegoż podejścia w projektach odzwierciedlających skomplikowaną logikę biznesową prowadzi do przerostu stopnia skomplikowania implementacji nad dziedziną, którą modeluje. Innym wartym przytoczenia atrybutem takich projektów jest wyparcie podstawowych założeń programowania zorientowanego obiektowo na rzecz jak najdokładniejszego odzwierciedlenia struktury relacyjnej bazy danych. W rezultacie prowadzi to do oddzielenia danych od logiki, a zatem do drastycznego obniżenia jakości oprogramowania. Niniejsza praca przedstawia sposoby na poprawę powyższych problemów za pomocą podejścia *Domain Driven Design* i wzorca CQRS oraz zwraca uwagę na sytuacje, w których ich wykorzystanie nie jest zasadne.

1.1. Cel pracy

Celem pracy jest porównanie oprogramowania wytworzonego w oparciu o wzorce *Domain Driven Design* i CQRS z odpowiednikiem stworzonym w oparciu o tradycyjną strukturę trójwarstwową z anemicznym modelem danych. Efektem porównania powinna być odpowiedź na pytanie, kiedy należy stosować każde z tych podejść. Dodatkowym efektem przeprowadzonej analizy są zalety i wady każdego z rozwiązań.

1.2. Rozwiązanie przyjęte w pracy

Porównanie dwóch podejść do architektury oprogramowania zostało zrealizowane poprzez wytworzenie przykładowego projektu systemu do zarządzania pizzerią, w dwóch wersjach odpowiadających każdej stronie porównania. Oba systemy wykorzystują technologię .NET Core 3.1, co implikuje język programowania C#. Każda z aplikacji występuje pod postacią API z graficznym interfejsem dostarczonym przez usługę Swagger.

1.3. Rezultaty pracy

Zgodnie z przewidywaniami, efekt pracy wskazuje na zasadność wykorzystania wzorców *Domain Driven Design* oraz CQRS w sytuacjach, w których poziom skomplikowania logiki biznesowej nie pozwala na efektywne zamodelowanie systemu przy pomocy modelu anemicznego. Implementacja oparta na CQRS wykazuje się obecnością dużej liczby plików, klas pomocniczych oraz nietrywialnej konfiguracji. W przypadku prostych projektów przekłada się to na długi, nieproporcjonalny do ich wielkości czas wytwarzania oprogramowania. System do obsługi pizzerii, na obecnym, pierwotnym etapie jego rozwoju nie wykazuje korzyści z tytułu wykorzystania wyrafinowanych wzorców architektonicznych. Czas potrzebny na implementację wersji opartej o strukturę trójwarstwową, o anemicznym modelu danych był znacznie krótszy i mniej skomplikowany niż odpowiednik oparty o *Domain Driven Design* i CQRS. Niemniej jednak pewne zalety wynikające z wykorzystania powyższych wzorców z całą pewnością odwróciłyby tę proporcję na dalszym etapie rozwoju projektu,

kiedy model anemiczny stałby się zbyt prymitywny, aby skutecznie opisać rozbudowane reguły biznesowe.

Podsumowując – nie ma sensu angażować wzorców *Domain Driven Design* i CQRS do projektów prostych, ale duże, skomplikowane systemy zyskują za ich pomocą na jakości i łatwości w utrzymaniu.

1.4. Organizacja pracy

Rozdział wprowadzający opisuje podstawowe pojęcia i założenia związane z *Domain Driven Design*. Znajduje się w nim teoretyczne wyjaśnienie fundamentów oraz elementów budujących projekty oparte o to podejście.

Następny rozdział traktuje o kilku wzorcach architektonicznych często wykorzystywanych w parze z *Domain Driven Design*. Najwięcej uwagi poświęca wzorcowi CQRS, który został wykorzystany w przykładowym projekcie porównawczym – systemie do obsługi pizzerii.

Rozdział czwarty opowiada o wymaganiach biznesowych wyżej przytoczonego projektu, które muszą spełniać obie implementacje, aby porównanie było uczciwe.

Kolejny rozdział skupia się na technicznym aspekcie implementacji projektu opartego na *Domain Driven Design* i CQRS. Znajduje się w nim opis każdej z warstw, ich integracji oraz metod modelowania domeny biznesowej.

Rozdział szósty zawiera analogiczny opis projektu trójwarstwowego o anemicznym modelu danych.

Ostatni, najważniejszy rozdział to rozprawa na temat zalet i wad każdego rodzaju implementacji, która prowadzi do ostatecznych wniosków na temat scenariuszy zastosowania obu podejść.

2. Wprowadzenie do tematyki Domain Driven Design

Domain Driven Design to podejście do modelowania rzeczywistości w systemach informatycznych znane od 2003 roku, w którym to Eric Evans opublikował książkę „Domain Driven Design – Tackling Complexity in the Heart of Software" [1]. Jest to świetna podstawa teoretyczna do budowy wysokiej jakości oprogramowania, jednakże jej popularność jest dość ograniczona. Dopiero ostatnimi czasy doszło do pewnego renesansu tego tematu za sprawą większej ilości publikacji oraz kursów, jak również zwiększonej świadomości wśród programistów, z którymi mam styczność. Potrzeba matką wynalazków – w moim przypadku rozpoczęcie badań nad *Domain Driven Design* wynika z poczucia niskiej jakości produktów, które do tej pory miałem okazję wytwarzać. Niniejszy rozdział opisuje elementy składające się na projekty oparte o *Domain Driven Design*, wraz z ich właściwościami pozwalającymi na osiągnięcie wyższej jakości i dojrzałości oprogramowania.

2.1. Podstawowe założenia i motywacja

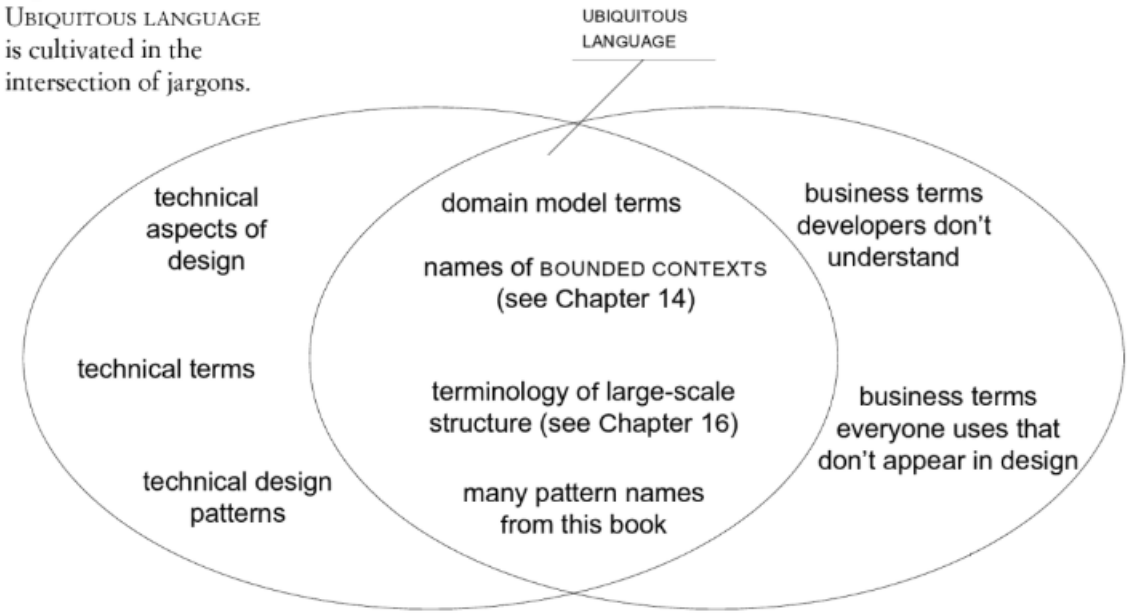
Podstawowym założeniem *Domain Driven Design* jest ustalenie hierarchii komponentów składających się na oprogramowanie w taki sposób, aby reguły biznesowe i opisująca je logika znajdowały się zawsze w centrum, czyniąc pozostałe warstwy, takie jak dostęp do danych, czy prezentacja, sobie poddanymi. Niedopuszczalna jest sytuacja, w której zastosowanie konkretnej bazy danych, czy ORM wpływa znacząco na implementacje logiki biznesowej. Często ma to miejsce w projektach trójwarstwowych, gdzie wyższe warstwy dostosowują się do modelu danych narzuconego przez konkretną technologię mapowania. Osiągnięcie całkowitej *persistence ignorance* [11] jest jednak dążeniem do utopii, gdyż wiele technicznych rozwiązań nie pozwala na zachowanie pełnej szczelności warstwy dostępu do danych, aczkolwiek śledząc rozwój projektów takich jak Entity Framework Core, można być pewnym, że moment ten nadejdzie.

Domain Driven Design jest jednak czymś znacznie więcej niż tylko wyznacznikiem rozłożenia warstw w projekcie. Zmianie ulega również sposób kontaktu z biznesem w celu ustalenia wspólnego języka opisującego domenę biznesową. Zmienia się również struktura samego modelu danych. Zamiast monolitu obejmującego cały system na kształt relacyjnej bazy danych, model zostaje podzielony na tzw. konteksty i agregaty, bogate w logikę biznesową. Z punktu widzenia *Domain Driven Design*, obiekty zawierające tylko dane, bez implementacji żadnych zachowań są antywzorcem. Takie obiekty nazywamy anemicznymi. Niniejszy rozdział opowiada w większych szczegółach o poszczególnych fundamentach *Domain Driven Design*.

2.2. Język wszechobecny

Z angielskiego – *ubiquitous language*. Eric Evans poświęca temu zagadnieniu cały rozdział swojej publikacji na temat *Domain Driven Design* [1]. Jest to absolutna podstawa budowania oprogramowania zaprojektowanego po to, aby rozwiązywać problemy biznesowe. Aby skutecznie modelować biznes, deweloperzy muszą go rozumieć. Z drugiej strony, aby skutecznie pozyskiwać informacje od biznesu, nie mogą posługiwać się żargonem technicznym, którego eksperci domenowi nie rozumieją. Istnieje zatem potrzeba wypracowania wspólnego języka, którym posługiwać się będą wszyscy członkowie zespołu deweloperskiego oraz eksperci domenowi. Jest to niezwykle ważne, aby uniknąć nieporozumień i błędnej interpretacji informacji dostarczonych przez biznes. Język wszechobecny, jak nazwa wskazuje, musi swoim zasięgiem obejmować wszystkie aspekty projektu – od dokumentacji po kod źródłowy. Co więcej, w interesie każdego członka zespołu jest pielęgnacja tego języka i wzajemne pilnowanie, aby

nie dochodziło do odchyleń od ustalonego słownika, ku czemu może zaistnieć pokusa, szczególnie wśród osób technicznych.



Rysunek 1 – Ubiquitous language. Źródło: [1] (str. 34)

Jak widać na rysunku 1, na język wszechobecny składają się pojęcia na przecięciu języków technicznych i biznesowych. Żaden inny język nie powinien być używany w kontaktach między zespołem deweloperskim, a biznesowym. Nie powinno się również korzystać z innych terminologii we wszelkich artefaktach związanych z projektem, takich jak dokumentacja, czy kod źródłowy opisujący domenę biznesową. Dla jasności, przytoczę definicję widocznego na rysunku pojęcia *large–scale structure*. Jest to język, który pozwala na dyskutowanie i zrozumienie systemu przez szerokie, wysokopoziomowe koncepcje i zasady [1].

2.3. Bounded context

Próba znalezienia poważnie brzmiącego polskiego odpowiednika pojęcia *bounded context* zakończyła się niepowodzeniem, a zatem w dalszej części tekstu będę korzystał ze słowa „kontekst", co nie powinno budzić wątpliwości w „kontekście" omawianego tematu.

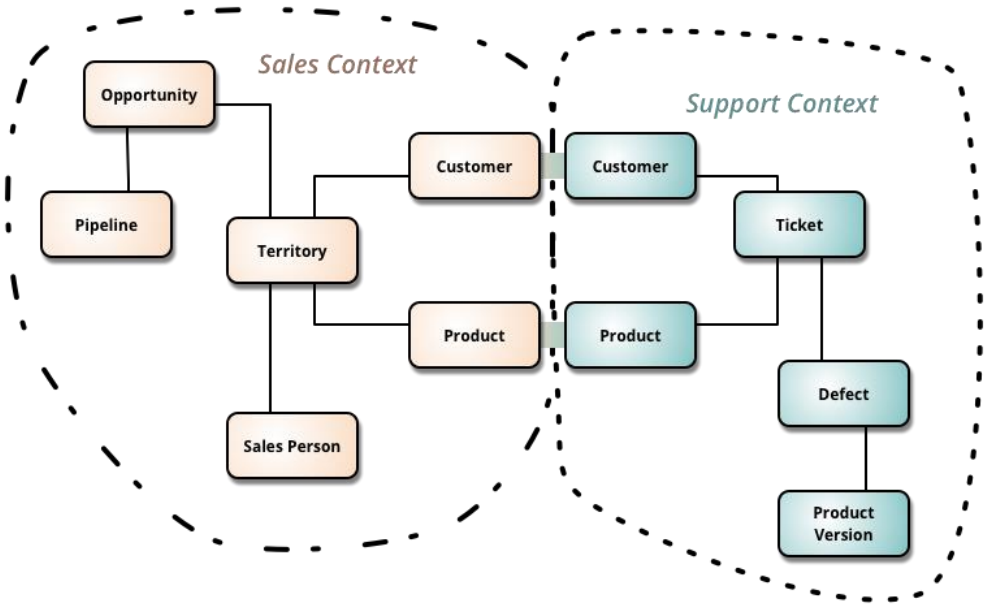
Projekty oparte o *Domain Driven Design* w odróżnieniu od tradycyjnych projektów trójwarstwowych nie skupiają całego modelu danych w jednym miejscu. Zamiast tego ustala się granice i kategorie tematyczne, do których należy każda encja i *value object* – pojęcia, których wyjaśnienie znajduje się w kolejnych podrozdziałach.

Ludzki umysł skonstruowany jest tak, że zniechęca nas potrzeba radzenia sobie z wielkimi, skomplikowanymi strukturami lub całością danego zagadnienia w jednym momencie. Dużo łatwiej zrozumieć dany problem, kiedy jest on podzielony na części w taki sposób, że możemy ograniczyć myślenie do konkretnego fragmentu, bez zamartwiania się o pozostałe. Można to zobrazować na przykładzie pisania pracy dyplomowej. Zdarzają się osoby, które decydują się na napisanie całej pracy w jeden wieczór, ale większość jednak nie byłaby w stanie nawet przyjąć do wiadomości takiego scenariusza i zamiast tego dzieli pisanie na poszczególnie rozdziały. Tworzenie systemu informatycznego obsługującego rozległe struktury biznesowe to duże wyzwanie dla umysłu i psychiki.

Podzielenie takiego projektu na tzw. konteksty pozwala na efektywniejsze zarządzanie swoim czasem i nauką dziedziny biznesowej, bo w *Domain Driven Design* nawet programista musi rozumieć biznes.

Podzielenie tematu na konteksty oznacza również łatwiejsze posługiwanie się wszechobecnym językiem – albowiem podczas modelowania systemu nie musimy brać pod uwagę relacji danego obiektu z innymi, które znajdują się poza kontekstem, do którego należy. W praktyce oznacza to, że może dochodzić do duplikacji nazw obiektów z perspektywy całego systemu. Nie powinno jednak być to dużym problemem, ponieważ każdy z nich rozpatrywany jest zawsze z perspektywy kontekstu, do którego należy.

Z punktu widzenia technicznego, mówiąc z perspektywy technologii .NET, konteksty realizuje się przez podzielenie rozwiązania na osobne projekty, w których umieszcza się fragmenty kodu należące do danego kontekstu. Oczywiście każdy kontekst może, a nawet powinien składać się z wielu projektów, np. odzwierciedlających poszczególne warstwy.



Rysunek 2 – Bounded contexts. Źródło: [12]

Rysunek 2 pokazuje przykład podziału klas na konteksty, wraz z duplikacją klasy `Customer`, która w zależności od kontekstu ma inne znaczenie. Podział na konteksty można również potraktować jako wstęp do podziału na mikroserwisy. System monolityczny podzielony na niezależne konteksty można w przyszłości łatwo przekształcić w formę rozproszoną, a jak wiadomo – w mikroserwisach duplikacja danych i denormalizacja jest na porządku dziennym.

Zagadnienie kontekstów wymaga podkreślenia jednego aspektu. Nie ma żadnych bezpośrednich połączeń pomiędzy klasami wchodzącymi w skład różnych kontekstów. Cała komunikacja odbywa się za pośrednictwem zdarzeń integracyjnych, o których piszę w podrozdziale 2.9 (str. 12).

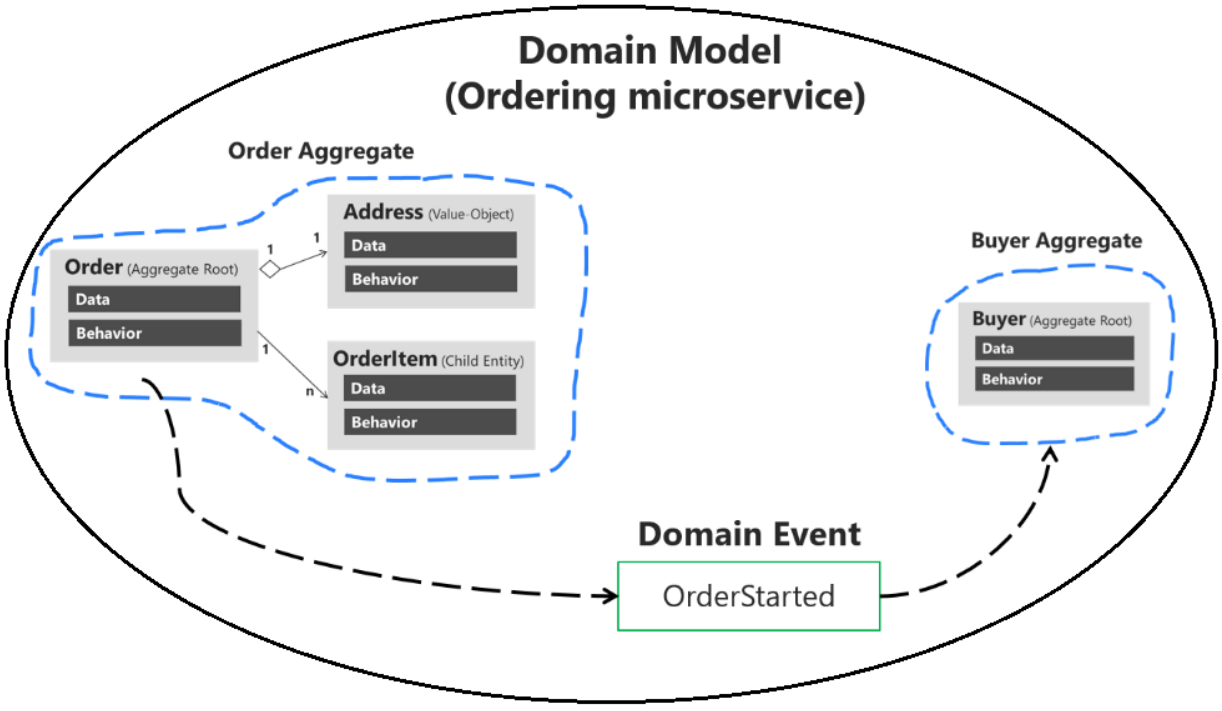
2.4. Agregaty

Każdy element modelu danych w systemie opartym o *Domain Driven Design* należy do pewnego zbioru obiektów, zwanego agregatem. Agregat składa się z korzenia agregacji oraz encji i *value objects*. Celem agregatów jest logiczny podział obiektów wewnątrz danego kontekstu. Relację między korzeniem agregacji a obiektami wchodzącymi w jego skład można przyrównać do kompozycji –

usunięcie korzenia powoduje usunięcie wszystkich obiektów od niego zależnych. W modelowaniu agregatów należy przestrzegać kilku zasad:

- Jeżeli to możliwe, pojedyncza transakcja powinna modyfikować tylko jeden agregat;
- Komunikacja między agregatami może się odbywać tylko za pośrednictwem korzeni agregatów;
- Należy unikać agregatów, których wczytanie wymaga bardzo dużej ilości pamięci;
- Korzeń agregacji jest odpowiedzialny za zachowanie spójności i niezmienników w obrębie całego agregatu;
- Repozytoria realizujące dostęp do danych mogą pobierać i zapisywać tylko korzenie agregacji;
- Nie można modyfikować encji w inny sposób niż za pośrednictwem korzenia agregatu, w którym się znajduje.

Podział danych na agregaty zapewnia porządek w strukturze modelu domenowego. Można w ten sposób uniknąć tworzenia skomplikowanych struktur o wielu trudnych do prześledzenia połączeniach między wszystkimi możliwymi obiektami. Jedną z zasad *Domain Driven Design* jest zachowanie poprawnego stanu każdego obiektu w każdym momencie poprzez enkapsulację. Koncepcja agregatów i korzeni agregacji pozwala ten cel stosunkowo łatwo osiągnąć, ponieważ zasady, które należy zaimplementować ograniczają się do pojedynczych agregatów.



Rysunek 3 – Podział na agregaty. Źródło: [2]

Rysunek 3 przedstawia przykład podziału na agregaty w słynnym projekcie firmy Microsoft – „eShopOnContainers", który jest pokazowym projektem wykorzystującym wzorce *Domain Driven Design*, CQRS oraz mikroserwisy. Widzimy tutaj agregaty „Order" i „Buyer" z korzeniami o tych samych nazwach. Komunikacja pomiędzy agregatami odbywa się za pośrednictwem zdarzenia `OrderStarted` emitowanego przez agregat „Order". W przeciwieństwie do tradycyjnych systemów trójwarstwowych, model ten nie odzwierciedla struktury bazy relacyjnej w postaci bezpośrednich referencji, aczkolwiek nic nie stoi na przeszkodzie, aby taka relację zachować na poziomie bazy danych.

2.5. Encje

Eric Evans opisuje encje, jako obiekty zdefiniowane przez ich tożsamość, które zachowują ciągłość przez cykl życia i odróżnialność niezależną od swoich atrybutów [1]. Encją może być osoba, miasto, samochód lub na przykład transakcja bankowa. Do porównania dwóch encji (stwierdzenia czy są tożsame) wystarczy porównanie ich identyfikatorów – cała reszta atrybutów nie ma znaczenia. Z punktu widzenia technicznego, możemy przeciążyć operator porównania w taki sposób, aby porównywał tylko właściwości należące do identyfikatorów porównywanych encji. Poprzedni podrozdział traktował o agregatach – tylko encje mogą być korzeniami agregacji, ponieważ cały agregat dziedziczy tożsamość po swoim korzeniu. Encje w odróżnieniu od *value objects* są mutowalne. Jest to oczywiste biorąc pod uwagę fakt, że modyfikacje encji nie mogą wywierać wpływu na ich identyfikator. Jeżeli encje byłyby niemutowalne, modyfikacja wymagałaby utworzenia nowych obiektów, o nowych identyfikatorach, co oznaczałoby zerwanie ciągłości życia encji.

Ważnym aspektem modelowania encji jest decyzja, która właściwość może posłużyć za klucz będący identyfikatorem. W przypadku niektórych encji, takich jak osoba, może się wydawać, że kompozyt imienia i nazwiska jest dobrym kandydatem na identyfikator, ale wiadomo przecież, że osób o tym samym imieniu i nazwisku jest wiele. Innym pomysłem może być zastosowanie adresu e–mail, ale wówczas problem może się pojawić w momencie, gdy wymagania biznesowe zmienią się tak, że ów adres email nie będzie dłużej potrzebny. Zatem, nawet jeśli istnieje właściwość spełniająca wymagania do potraktowania jej jako klucz, zdecydowanie lepiej poprzestać na kluczu syntetycznym, niemającym znaczenia z punktu widzenia biznesowego. Taki klucz jest odporny na zmiany wymagań oraz braki danych. Technicznie klucze syntetyczne można rozwiązać w postaci kolumny *identity* bazy danych, sekwencji w bazie danych, GUID, lub innych możliwych do wygenerowania wartości. Zaletą rozwiązań niewykorzystujących bazy danych jest możliwość ich wygenerowania na samym początku istnienia encji, co ułatwia nawiązywanie relacji i umożliwia szybki zwrot informacji do klienta, bez czekania na zakończenie transakcji, lub materializację danych.

Na koniec warto podkreślić jedną z najważniejszych kwestii w modelowaniu systemu opartego o *Domain Driven Design* – stan encji musi być zawsze poprawny. Nie wolno dopuścić do wytworzenia encji, która nie spełnia warunków walidacji, jak również nie wolno pod żadnym pozorem dopuścić do zmiany stanu encji na stan niepoprawny. W terminologii języka C# oznacza to, że nie wolno stosować publicznych akcesorów typu *set*. Całość obiektu musi być enkapsulowana. Każda zmiana wartości musi się odbywać poprzez metody, które pilnują zasad walidacji i niezmienników agregatów. W zachowaniu poprawnego stanu pomagają *value objects*.

2.6. Value objects

Celowo nie stosuje polskiego tłumaczenia, ponieważ trudno znaleźć odpowiednik, który nie budziłby wątpliwości osób, które go słyszą. *Value objects* w odróżnieniu od encji nie posiadają tożsamości – innymi słowy – reprezentują je atrybuty zamiast ustalonego klucza, identyfikatora. Eric Evans przytacza przykład kolorów, jako typowych przedstawicieli *value objects* [1]. Wykorzystując w modelu danych kolory, nie interesuje nas konkretna instancja danego koloru. Dbamy tylko to, czy odpowiada nam jego wartość. Innym typowym przykładem *value object* może być adres, czyli ulica, numer, miejscowość, kod pocztowy. Wszelkie elementy, których nie musimy współdzielić, możemy potraktować jako *value objects*. Jeżeli ustalony zestaw wartości dwóch obiektów jest taki sam, możemy stwierdzić, że są one tożsame.

Value objects co do zasady powinny być niemutowalne, to znaczy, że wszelkie modyfikacje muszą odbywać się poprzez tworzenie nowych instancji. Technicznie, w języku C# można to uzyskać np. przez modyfikator *readonly* zastosowany do pól obiektu. Niemutowalność to wbrew pozorom ułatwienie. Przyjmując założenie, że obiekt danej klasy nigdy nie ulega zmianie, możemy w bezpieczny sposób przekazywać referencje. Oprócz tego, jeżeli tożsamość obiektu determinują jego właściwości, wszelkie zmiany oznaczałyby złamanie ciągłości. Tak jak nie można zmienić identyfikatora encji, nie można zmienić atrybutów *value object*. Mutowalność jest dozwolona tylko w przypadkach, kiedy zachowanie niemutowalności jest technicznie niemożliwe, lub oznaczałoby wprowadzenie znacznych trudności.

W modelowaniu domeny biznesowej warto stosować zasadę priorytetu dla *value object*. Operacje na encjach są kłopotliwe ze względu na konieczność utrzymywania ciągłości ich tożsamości oraz referencji za pomocą identyfikatorów. Dodatkowo niemutowalność *value objects* zapewnia większe bezpieczeństwo zachowania poprawności danych. Im więcej *value objects* w stosunku do encji, tym lepiej. Dodatkowo im więcej logiki biznesowej umieścimy wewnątrz *value objects* zamiast w encjach, tym prostszy otrzymamy model.

Do konstrukcji *value objects* możemy wykorzystać konstruktory obiektów, lub fabryki, ale należy pamiętać, że tak jak w przypadku encji, niedopuszczalne jest utworzenie obiektów niespełniających reguł walidacji, zatem podczas konstrukcji reguły te muszą być sprawdzane. *Value objects* ułatwiają utrzymanie encji w poprawnym stanie. Weźmy za przykład adres e–mail. Wiadomo, że nie każdy ciąg znaków spełnia wymagania takiego adresu. Między innymi musi się w nim znajdować znak „@" oraz oddzielona kropką domena. Jeżeli przedstawimy adres e–mail wewnątrz encji jako ciąg znaków, to będziemy musieli zadbać o jego poprawność za każdym razem, gdy dochodzi do jego modyfikacji. Jeżeli jednak wykorzystamy *value object* do przechowywania tej wartości, odpowiedzialność za utrzymanie prawidłowego stanu spadnie na ten obiekt. W praktyce zamiast ciągu znaków, wykorzystalibyśmy pole typu `EmailAddress` – klasy, która przechowuje ciąg znaków i dba o jego walidację. Innym aspektem pozostawienia właściwości adresu e–mail w postaci ciągu znaków zamiast odpowiedniego *value object* jest antywzorzec *primitive obsession* [17] polegający na nadużywaniu typów prostych wbudowanych w język programowania. Prowadzi on między innymi do możliwości przypisania wartości do błędnej właściwości, np. numeru telefonu do adresu e–mail. Jeżeli obie właściwości są ciągami znaków, to znaczy, że taka operacja jest poprawna z punktu widzenia statycznego typowania. Zastosowanie *value object* `EmailAddress` i `PhoneNumber` doprowadziłoby w tej sytuacji do błędu kompilacji, ponieważ nie można przypisać wartości do właściwości innego typu. Zdecydowanie lepiej jest wykryć błąd na etapie kompilacji, niż na produkcji przez klienta.

2.7. Serwisy

W modelowaniu domeny biznesowej może dochodzić do sytuacji, w których dana czynność nie pasuje do pojedynczej encji, czy *value object*. Wartym podkreślenia jest to, że dojście do takiego wniosku powinno być ostatecznością. W pierwszej kolejności należy postarać się umieścić logikę wewnątrz *value object*, potem wewnątrz encji, a jeżeli i to się nie powiedzie, można rozważyć wykorzystanie serwisu. Serwis jest klasą, która reprezentuje tylko czynności. Nie zawiera ona żadnej enkapsulacji, ani stanu – jest bezstanowa. W odróżnieniu od systemów trójwarstwowych z anemicznym modelem danych, serwisy w *Domain Driven Design* są zazwyczaj bardzo małe i proste, ponieważ większość logiki znajduje się w odpowiednich encjach i *value objects*. *Domain Driven Design* czerpie garściami z możliwości, które zapewnia programowanie zorientowane obiektowo. Zamiast klas reprezentujących tylko i wyłącznie dane w relacyjnej bazie danych i potężnych warstw serwisów,

Domain Driven Design stawia na klasy bogate w logikę ściśle powiązaną z danymi, które są w nich zdefiniowane. Zatem serwisy w *Domain Driven Design* są strukturą pomocniczą, którą stosujemy wtedy, gdy dane zachowanie nie pasuje bezpośrednio do jednej encji lub *value object*. Przykładem takiego zachowania, przytoczonym przez Erica Evansa, jest transfer pieniędzy w banku [1]. Operacja ta nie należy do encji konta, ponieważ angażuje dwa konta i pewien zestaw reguł. Należy przy tym pamiętać, iż metody serwisu w warstwie domeny muszą spełniać założenia języka wszechobecnego (2.2).

2.8. Repozytoria

Nawet w systemach opartych na modelu anemicznym, zazwyczaj stosuje się warstwę abstrakcji nad dostęp do danych, aby ukryć wewnętrzną implementację pobierania danych z odpowiedniej bazy. Nie inaczej jest w przypadku *Domain Driven Design*. Dostęp do danych jest możliwy za pośrednictwem interfejsów repozytoriów, których implementacja obsługuje połączenie z daną bazą danych. W przypadku tradycyjnych projektów trójwarstwowych w języku C#, często spotykałem się z sytuacją, w której repozytorium wystawiało kolekcję w postaci interfejsu `IQueryable`. Taka implementacja nie jest zgodna z założeniami *Domain Driven Design*, a w szczególności z zasadą *persistence ignorance* [11]. Wystawiając interfejs `IQueryable` zmuszamy obiekty, które korzystają z repozytorium do świadomego korzystania z mechanizmu zapytań przez drzewa wyrażeń. Co gorsza, w przypadku technologii Entity Framework, lub podobnych ORM może dochodzić do konieczności doładowywania tabel przez tzw. właściwości nawigacyjne [18], co jest operacją charakterystyczną tylko dla konkretnych technologii połączeń z bazą danych. W takiej sytuacji zmiana systemu ORM lub dostawcy bazy danych jest czasochłonna lub niemożliwa, gdyż wymaga refaktoryzacji znacznej ilości kodu.

W *Domain Driven Design* tylko warstwa infrastruktury ma świadomość działania dostępu do danych, lub też tego, jaka technologia została w tym celu wykorzystana. Dobrym sposobem na udostepnienie danych będących wynikiem zapytania w języku C#, jest interfejs `IReadOnlyList`, który oprócz tego, że jest niezależny od technologii bazy danych, zabezpiecza kolekcję przed modyfikacjami, które nie są dozwolone podczas operacji odczytu.

Niestety zastosowanie konkretnych ORM często warunkuje pewne zmiany w modelu domenowym. Mówimy wtedy o tzw. zanieczyszczeniu modelu [19]. Na przykład, jeżeli chcemy zastosować Entity Framework z opcją opóźnionego ładowania, właściwości nawigacyjne muszą być wirtualne [18]. Przeczy to zasadzie *persistence ignorance* [11], na szczęście jednak nie w stopniu znaczącym. W tym konkretnym przypadku zmiana ORM nie byłaby bowiem żadnym problemem. Duży postęp w tym zakresie odnotowuje ORM Entity Framework Core, który w stosunku do swojego poprzednika wprowadza takie funkcjonalności jak *backing fields* [20], które pozwalają na utrzymanie pełnej enkapsulacji.

Warto w tym miejscu przypomnieć, iż repozytoria w *Domain Driven Design* operują tylko na korzeniach agregacji. Nie ma możliwości bezpośredniego pobierania z bazy danych encji, które nie są korzeniami. Jeżeli operacja biznesowa wymaga odczytania takiej encji, należy uprzednio wczytać cały agregat, do którego należy, a następnie po wykonaniu operacji, zapisać cały agregat z powrotem. Tylko w ten sposób możemy mieć pewność, że nie zostaną złamane niezmienniki agregatów.

2.9. Zdarzenia

W rozdziale na temat kontekstów (str. 7) znajduje się informacja o braku bezpośrednich połączeń między kontekstami. Nasuwa się więc pytanie, w jaki sposób przekazywać pomiędzy nimi informacje,

skoro nie można zrobić tego bezpośrednio. Odpowiedzią są zdarzenia. W *Domain Driven Design* możemy wyróżnić dwa rodzaje zdarzeń:

- Zdarzenia integracyjne,
- Zdarzenia domenowe.

Zdarzenia integracyjne służą do komunikacji między kontekstami. Załóżmy, że mamy do czynienia z aplikacją do obsługi pizzerii oraz zawartymi w niej kontekstami zamówień i dostaw. Kiedy zamówiona przez klienta pizza jest gotowa, aby zostać dostarczona, kontekst zamówień wysyła zdarzenie `OrderShippedIntegrationEvent`. Kontekst dostaw podejmuje zdarzenie i uzupełnia swoje źródło danych o nową dostawę, zgodnie z informacjami zawartymi w zdarzeniu.

Opisany powyżej sposób zapewnia spójność między kontekstami jednocześnie zachowując między nimi luźne powiązanie [21]. Dzięki temu istnieje możliwość niezależnej pracy nad kontekstami bez obaw, że zmiany będą dotyczyły więcej niż jednego kontekstu, pod warunkiem, że zmiany te nie dotyczą struktury samych zdarzeń, które są elementem współdzielonym. Tak długo, jak konteksty utrzymują wspólną strukturę zdarzeń i są w stanie na nie reagować, nie ma potrzeby wykonywania dodatkowej pracy w celu zapewnienia integracji. Wykluczenie bezpośrednich odwołań między kontekstami pozwala na utrzymanie porządku i prostoty modelu danych w obrębie kontekstów.

Inną ważną zaletą utrzymania separacji między kontekstami i obsługi komunikacji za pośrednictwem zdarzeń jest możliwość łatwego przystosowania takiego systemu do architektury mikroserwisowej w przyszłości. Przejście na taki rodzaj architektury wymaga jedynie rozdzielenia warstwy API i przeniesienia danych do osobnych baz. Implementacja logiki biznesowej i przypadków użycia nie wymaga modyfikacji.

Podsumowując, zdarzenia integracyjne służą do informowania o zmianach w systemie innych kontekstów, mikroserwisów lub nawet zewnętrznych aplikacji [2].

Zdarzenia domenowe w odróżnieniu od integracyjnych działają na poziomie agregatów w obrębie pojedynczego kontekstu. Można je wykorzystać do zachowania spójności między agregatami, lub do wykonywania czynności, które mają mieć miejsce po tym, gdy stało się coś w systemie. Na przykład, po wysłaniu zamówienia do klienta, można stworzyć zdarzenie, które spowoduje, że inny serwis w obrębie tej samej domeny wyśle do klienta wiadomość e–mail lub sms.

Zdarzenia domenowe można obsługiwać w obrębie pojedynczej transakcji, tuż przed zatwierdzeniem zmian, lub alternatywnie – asynchronicznie, po zatwierdzeniu transakcji. Podejście asynchroniczne zwiększa elastyczność i skalowalność, ale wymusza zaimplementowanie mechanizmów obsługi błędów.

Stosowanie zdarzeń domenowych pozwala na odciążenie warstwy domeny biznesowej z obsługi czynności będących rezultatem przeprowadzanych operacji. Oprócz tego pozwala na zamodelowanie reakcji na owe operacje w sposób jawny. Jeżeli dana operacja jest skutkiem ubocznym czynności, która go wzbudza, zdarzenia pozwalają na przedstawienie tej zależności w postaci terminologii akcja – reakcja. W ten sposób kod źródłowy staje się znacznie bardziej czytelny, a encje nie rozrastają się wraz z rozwojem systemu.

Obsługa zdarzeń zarówno domenowych, jak i integracyjnych odbywa się w warstwie aplikacji. Dzięki temu kod obsługujący zdarzenie ma dostęp do funkcjonalności warstwy infrastruktury, co często jest wymagane do przeprowadzenia pożądanej operacji, np. wysłania wiadomości email. Oprócz tego warstwa domeny biznesowej nie jest obarczona obowiązkiem obsługi operacji, które nie należą do języka opisującego logikę czysto biznesową. Innymi słowy – czynności takie jak wysyłanie wiadomości e–mail, czy modyfikowanie innych agregatów nie należą do obszaru zainteresowania samego agregatu, który skupia się na obsłudze tylko swojej logiki i niezmienników.

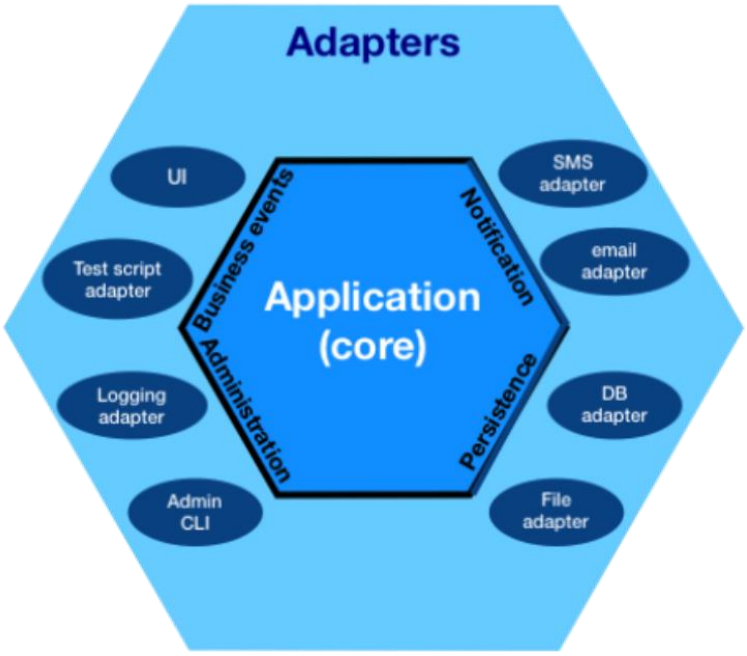
3. Wzorce architektoniczne łączone z Domain Driven Design

Domain Driven Design nie jest wzorcem architektonicznym, a raczej doktryną mówiącą o tym, jak należy modelować domenę biznesową w projektach informatycznych. Rzutuje to oczywiście na pewne rozwiązania architektoniczne, ale nie stanowi podstawy do budowy architektury oprogramowania jako całości, ograniczając zalecenia wyłącznie do centralnej warstwy logiki biznesowej. Jak wiadomo, nie samym biznesem projekty żyją. Aby poprowadzić dane od użytkownika, przez interfejs graficzny, aplikację serwerową, aż do bazy danych, potrzebny jest konkretny podział na warstwy realizujące wszystkie odpowiedzialności aplikacji. Oprócz tego, warto podkreślić, iż model zdefiniowany na gruncie *Domain Driven Design* nie określa sposobu implementacji przypadków użycia programu, a jedynie niezmienne reguły biznesowe. Aby dopełnić architekturę o wyżej wymienione, brakujące elementy, można skorzystać z jednego ze znanych wzorców architektonicznych, lub odpowiedniej ich kombinacji, dostosowanej do konkretnych potrzeb danego projektu. Niniejszy rozdział stanowi przegląd kilku takich wzorców.

3.1. Hexagonal architecture

Architektura heksagonalna, znana również pod nazwą architektury portów i adapterów została zaproponowana w 2005 roku przez znanego amerykańskiego informatyka Alistaira Cockburna [14]. Jej głównym założeniem jest podział systemu na luźno powiązane komponenty, które można łatwo wymieniać na inne. Aplikacja podążająca za architekturą portów i adapterów składa się z centralnej warstwy logiki biznesowej i aplikacji, oraz zewnętrznej warstwy adapterów, czyli konkretnych implementacji interfejsów (portów) zdefiniowanych w warstwie centralnej. Jedną z często wymienianych zalet tej architektury jest możliwość łatwego testowania jednostkowego poprzez zastąpienie prawdziwych implementacji komponentów atrapami, stosowanymi jako adaptery do tych samych portów. Jest to możliwe dlatego, że jądro aplikacji jest całkowicie niezależne od zewnętrznych komponentów, które muszą dostosować się do zdefiniowanych przez nie interfejsów [3, 4].

Architektura portów i adapterów nie jest tak szczegółowa, jak architektury, które się z niej wywodzą, czyli na przykład *onion architecture* czy *clean architecture*, zatem trudno jest zaprojektować system, który nie czerpie inspiracji z innych wzorców. Można mieć na uwadze konieczność stosowania abstrakcji zamiast konkretnych implementacji komponentów, jednakże wzrost popularności wzorca *inversion of control* doprowadził do tak powszechnego stosowania tej reguły, że w zasadzie trudno sobie obecnie wyobrazić system, który tę zasadę łamie.



Rysunek 4 – Przykład hexagonal architecture. Źródło: [4]

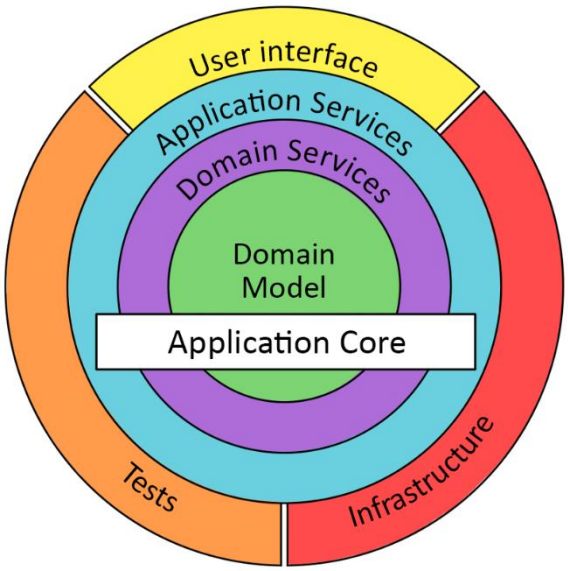
Chociaż nazwa *hexagonal* była tylko określeniem roboczym, przedstawianie tego wzorca za pomocą metafory sześciokąta spowodowało zakorzenienie tego określenia w literaturze technicznej [3].

Konfrontując schemat na rysunku 4 z *Domain Driven Design* dojdziemy do wniosku, iż miejscem na jego zastosowanie jest warstwa portów, czyli jądro aplikacji, w której należy umieścić implementację logiki biznesowej. To właśnie tam znalazłyby się konteksty komunikujące się ze sobą przez zdarzenia integracyjne. Warstwa adapterów byłaby już poza zasięgiem obejmowanym przez *Domain Driven Design*, ponieważ znajdują się w niej implementacje zewnętrznych zależności, które nie są częścią niezmiennych reguł biznesowych.

3.2. Onion architecture

Onion architecture, wprowadzona w 2008 roku przez Jeffrey'a Palermo [5], jest bardzo podobna do wcześniej opisanej architektury portów i adapterów, z tą różnicą, że stosuje nomenklaturę koncentrycznych warstw wokół jądra aplikacji. Każda warstwa jest luźno związana z warstwą leżącą poniżej, a komunikacja odbywa się za pomocą mechanizmu *inversion of control*. Podobnie jak w przypadku architektury heksagonalnej, w centrum znajduje się nie baza danych, a logika biznesowa opisana za pomocą modelu domenowego. Główne dogmaty tej architektury są następujące [5, 6]:

- Aplikacja jest zbudowana wokół niezależnego modelu domenowego;
- Warstwy wewnętrzne definiują interfejsy, a zewnętrzne je implementują;
- Kierunek powiązania warstw skierowany jest do wewnątrz;
- Kod jądra aplikacji może być skompilowany i uruchomiony niezależnie od infrastruktury.



Rysunek 5 – Onion architecture. Źródło: [13]

Jak widać na schemacie na rysunku 5, idea *onion architecture* jest w zasadzie taka sama jak główne założenia architektury portów i adapterów, jedynie jej opis i wyróżnienie warstw jest nieco bardziej szczegółowe. W dalszym ciągu nacisk kładziony jest na niezależność warstw, czyli luźne wiązanie [21] oraz uniezależnienie od infrastruktury, takiej jak baza danych, która znajduje się nie wewnątrz, a na samym wierzchu „cebuli".

Jeffrey Palermo zauważa na swoim blogu [5], że systemy, w których komponenty są ze sobą silnie powiązane stają się bardzo trudne do modyfikacji i dostosowywania do nowych technologii, ponieważ zmiana jednego komponentu powoduje propagację tych zmian do pozostałych. Jest to jedną z przyczyn, dlaczego tak dużo systemów buduje się od nowa, zamiast modernizować istniejące.

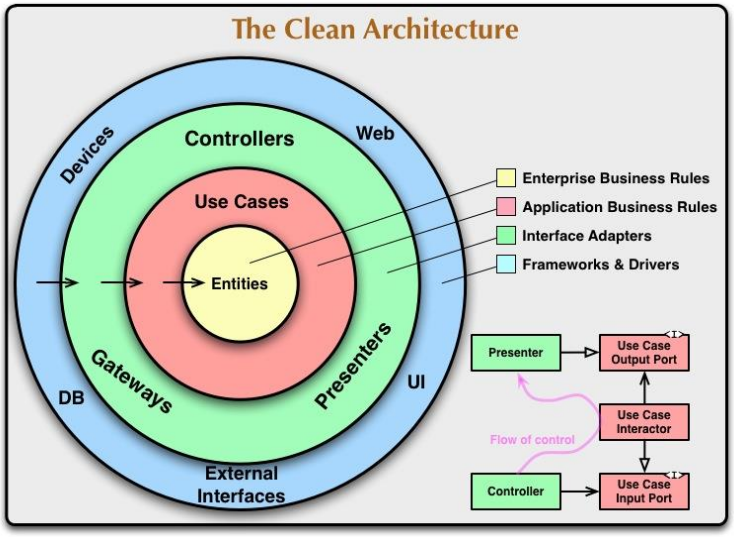
Rola *Domain Driven Design* w systemie opartym o *onion architecture* byłaby analogiczna do roli w systemie zaprojektowanym według architektury portów i adapterów. Konteksty i zdarzenia domenowe znalazłyby się w centralnej warstwie jądra systemu.

3.3. Clean architecture

Termin *clean architecture* wprowadzony przez Roberta Martina w 2012 roku [7] nie jest nowym rodzajem architektury, a zestawem założeń zdefiniowanych przez twórców architektur heksagonalnej i *onion architecture*, do których Robert Martin odwołuje się na swoim blogu. W jednym z artykułów podsumowuje on najważniejsze aspekty tych koncepcji [7]:

- Niezależność od bibliotek,
- Możliwość testowania logiki biznesowej niezależnie od infrastruktury i innych zewnętrznych warstw,
- Niezależność interfejsu użytkownika od reguł biznesowych,
- Niezależność reguł biznesowych od bazy danych,
- Niezależność reguł biznesowych od wszelkich zewnętrznych zależności.

W gruncie rzeczy architektura portów i adapterów, *onion architecture*, oraz *clean architecture* polegają na tych samych założeniach opisanych innym językiem. Podsumowanie Roberta Martina dotyczy w równym stopniu każdej z nich.



Rysunek 6 – Clean Architecture. Źródło: [7]

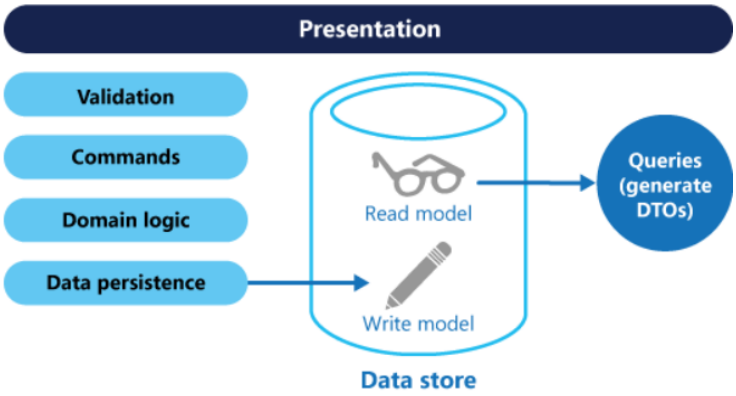
Patrząc na rysunek 6 od razu na myśl przychodzi analogiczny schemat dotyczący *onion architecture*. Można zatem powiedzieć, że punkty sformułowane przez Roberta Martina, zaczerpnięte w dużej części od Jeffrey'a Palermo i Alistaira Cockburna są uniwersalnym zbiorem reguł, które należy stosować w projektach informatycznych. Oczywiście, absolutnie zawsze należy konfrontować wszelkie wzorce i autorytety z potrzebami konkretnych produktów, choć bliski jestem stwierdzenia, że powyższe dogmaty znajdą zastosowanie w każdym projekcie, bez szczególnych modyfikacji.

3.4. CQRS

Wzorzec architektoniczny CQRS – *Command Query Responsibility Segregation* jest szczególnie ważny z punktu widzenia niniejszej pracy dlatego, że został zastosowany w przykładowym projekcie porównawczym. Wzorzec ten został sformułowany przez Grega Younga jako pojęcie, które mówi o tym, że do operacji odczytu można wykorzystać inny model niż do operacji zapisu [8]. Aby zrozumieć sens rozdzielenia modeli należy sobie uświadomić, że duża część reguł biznesowych, szczególnie walidacji, nie jest potrzebna podczas operacji odczytu. Ponadto, wszelka enkapsulacja i podział na agregaty w przypadku wczytywania danych są zbędne, a nawet utrudniają takie operacje. Poza ułatwieniem operacji odczytu, CQRS w zakresie operacji zapisu wprowadza jasny podział przypadków użycia na tzw. komendy. Warto przy tym zaznaczyć, iż wzorzec CQRS jest całkowicie zgodny z wyżej opisanymi wzorcami portów i adapterów, *onion architecture* i *clean architecture*.

Wzorcem, z którego CQRS bezpośrednio się wywodzi jest wzorzec CQS – *Command Query Separation*, który działa na poziomie interfejsów repozytoriów, wprowadzając jawny podział na metody odczytu – zapytania i zapisu – komendy. Żadna metoda nie może być jednocześnie komendą i zapytaniem, poza mało znaczącymi wyjątkami, kiedy komendy zwracają dane pomocne w ich późniejszym odczycie.

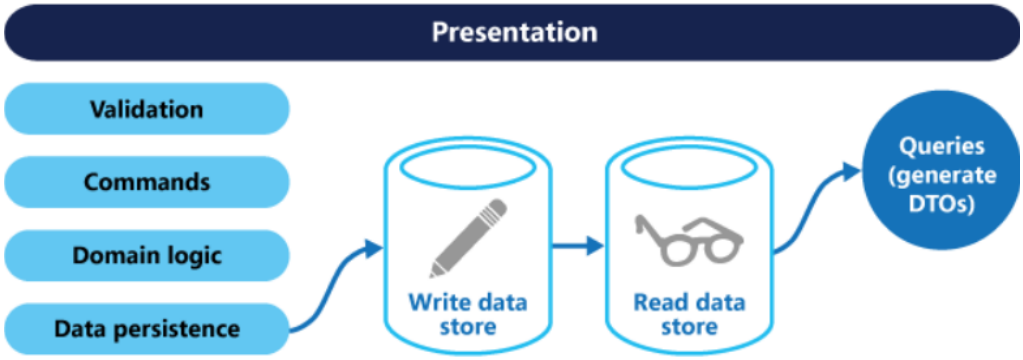
CQRS występuje w kilku odsłonach różniących się od siebie stopniem segregacji części odczytu od zapisu. Jedną z opcji jest zastosowanie podziału na poziomie modelu danych przy zachowaniu wspólnej bazy danych, jak przedstawiono na rysunku 7.



Rysunek 7 – CQRS ze wspólną bazą. Źródło: [9]

Każda operacja pochodząca od użytkownika aplikacji, która zmienia stan systemu, może zostać wprowadzona tylko za pomocą tzw. komendy, która reprezentuje dany przypadek użycia. Logika obsługująca komendę, operując na obiektach infrastruktury i domenie biznesowej dokonuje jej walidacji i decyduje, czy dokonać zmiany, czy tez odrzucić dane wejściowe. Warstwa zapytań jest tutaj bardzo cienką warstwą, położoną blisko bazy danych, która odczytuje z niej dane bezpośrednio do obiektów DTO (*Data Transfer Objects*) z pominięciem centralnej warstwy domeny biznesowej. Technologicznie, można skorzystać na przykład z lekkiej biblioteki ORM – Dapper, lub też z samego ADO.NET. Należy przy tym pamiętać, iż nie wolno pod żadnym pozorem wykorzystywać warstwy odczytu w operacjach zapisu. Jeżeli wykonanie komendy wymaga pobrania danych z bazy, np. w celu walidacji, trzeba je pobrać z repozytorium, którego interfejs należy do języka wszechobecnego domeny biznesowej. Warstwa zapytań nie operuje na obiektach domenowych, ponadto może w pewnych scenariuszach być obarczona opóźnieniem i nie nadaje się do stosowania przy obsłudze komend.

Operacje odczytu stanowią zdecydowaną większość operacji wykonywanych przez użytkowników każdego systemu i to one decydują o postrzeganiu wydajności i responsywności aplikacji. Dlatego właśnie, aby poprawić wydajność odczytów, można zastosować w tym celu osobną, dedykowaną, zdenormalizowaną bazę danych. W ten sposób otrzymamy kolejny rodzaj CQRS, który podział zapisów i odczytów wprowadza również do warstwy danych [9], co pokazano na rysunku 8.



Rysunek 8 – CQRS z osobnymi bazami danych. Źródło: [9]

Zastosowanie bazy danych zorientowanej na odczyty poprawia w znaczący sposób wydajność takich operacji, ale niestety wiąże się z bardzo poważną wadą związaną z opóźnieniem propagacji danych pomiędzy bazą do zapisu a bazą do odczytu. Skutkuje to powstaniem sytuacji nazywanej w literaturze technicznej *eventual consistency* [15], co w dosłownym tłumaczeniu oznacza ostateczną spójność. Można być pewnym, że po wykonaniu komendy nadejdzie moment, w którym dane będą spójne, ale dokładny czas zaistnienia tejże spójności można jedynie szacować. Użytkownicy żyją

w świecie, w którym wszystko jest zawsze spójne, dlatego opóźnienia w osiągnięciu całkowitej spójności są wyzwaniem nie tylko dla samej aplikacji serwerowej, ale również dla interfejsu użytkownika. Musi on przedstawiać dane w taki sposób, aby nie wprowadzić użytkowników w błąd i fałszywe poczucie nieprawidłowego działania systemu. Jedną z metod radzenia sobie z tą niedogodnością jest zmiana języka, jakim posługuje się ów interfejs. Zamiast wykonywania operacji, może przyjmować „zlecenia", które będą wykonane w bliżej nieokreślonym czasie [22]. Jako przykład obrazujący działanie systemów obarczonych *eventual consistency* można przytoczyć aplikacje bankowe. Nie tylko przelewy, ale również zmiana pinu do karty kredytowej nie odbywają się natychmiast.

Technologicznie, propagacja danych jest zazwyczaj rozwiązywana asynchronicznie, za pomocą emisji zdarzeń integracyjnych do systemów kolejkowych, takich jak np. kolejki Azure Service Bus, lub RabbitMQ [16].

Ponieważ separacja baz danych wprowadza pewną komplikację do architektury systemu, moim zdaniem powinno się ją wprowadzać stopniowo, zaczynając jednak od wspólnej bazy danych, która zapewnia przyzwoity poziom wydajności, szczególnie gdy jest to baza danych w chmurze, która pozwala na łatwe skalowanie zasobów. Jeżeli jednak wydajność takiej bazy okazuje się niewystarczająca, z powodu wysokiego obciążania, lub rozmieszczenia geograficznego użytkowników, można rozważyć wprowadzenie osobnej bazy do odczytów w technologii NoSQL. Możliwości skalowania i geo–redundancji takiej bazy są nie do zastąpienia przez bazy relacyjne [23].

Niektórzy autorzy, tacy jak na przykład Vladimir Khorikov [10] rozróżniają oprócz wyżej wymienionych rodzajów CQRS, poziom 0 oraz poziom 1. Poziom 0 oznacza paradoksalnie brak CQRS, czyli zastosowanie jednego modelu domenowego do wszystkich operacji zapisu i odczytu, jednak z zachowaniem CQS na poziomie repozytorium. Poziom 1 rozszerza poziom 0 o obiekty DTO, które zastępują obiekty domenowe w operacjach odczytu.

Projekt porównujący zastosowanie *Domain Driven Design* w stosunku do modeli anemicznych, który powstał na potrzeby tejże pracy wykorzystuje wzorzec CQRS ze wspólną bazą danych.

4. Omówienie przykładowego projektu

W celu przeprowadzenia porównania architektury opartej o Domain Driven Design z architekturą trójwarstwową o anemicznym modelu danych, został stworzony przykładowy projekt w dwóch wersjach odpowiadających każdemu z tych podejść. Niniejszy rozdział opisuje funkcjonalności tego systemu. Aby można było przeprowadzić rzetelne porównanie, oba projekty udostępniają ten sam zbiór możliwości.

4.1. Funkcjonalności systemu

Aplikacja realizuje system do obsługi pizzerii w postaci następujących modułów:

- Moduł koszyka,
- Moduł menu,
- Moduł zamówień,
- Moduł dostaw.

Każdy z powyższych modułów występuje pod postacią punktów końcowych API zgodnych ze stylem REST [24].

Moduł koszyka umożliwia użytkownikowi dodawanie i usuwanie produktów, które wejdą w skład zamówienia. Stan koszyka jest zapisywany w bazie danych, natomiast jego identyfikator przetrzymywany jest w ciasteczku przeglądarki, dzięki czemu nawet po odświeżeniu strony, dostęp do odłożonych produktów nie jest tracony. Jedną z funkcjonalności wchodzących w skład tego modułu, jest przejście do kasy, które wymaga podania danych niezbędnych do dostawy i powoduje wysłanie do modułu zamówień informacji o utworzeniu nowego zamówienia. Po tej czynności koszyk jest czyszczony.

Moduł menu umożliwia użytkownikowi przeglądanie produktów dostępnych w pizzerii. Administrator systemu może wykorzystać ten moduł do modyfikacji oferty, czyli dodania, bądź usunięcia produktów lub ich edycji. Moduł ten jest również odpowiedzialny za weryfikację zamówień pod kątem dostępności zamawianych produktów oraz ich cen.

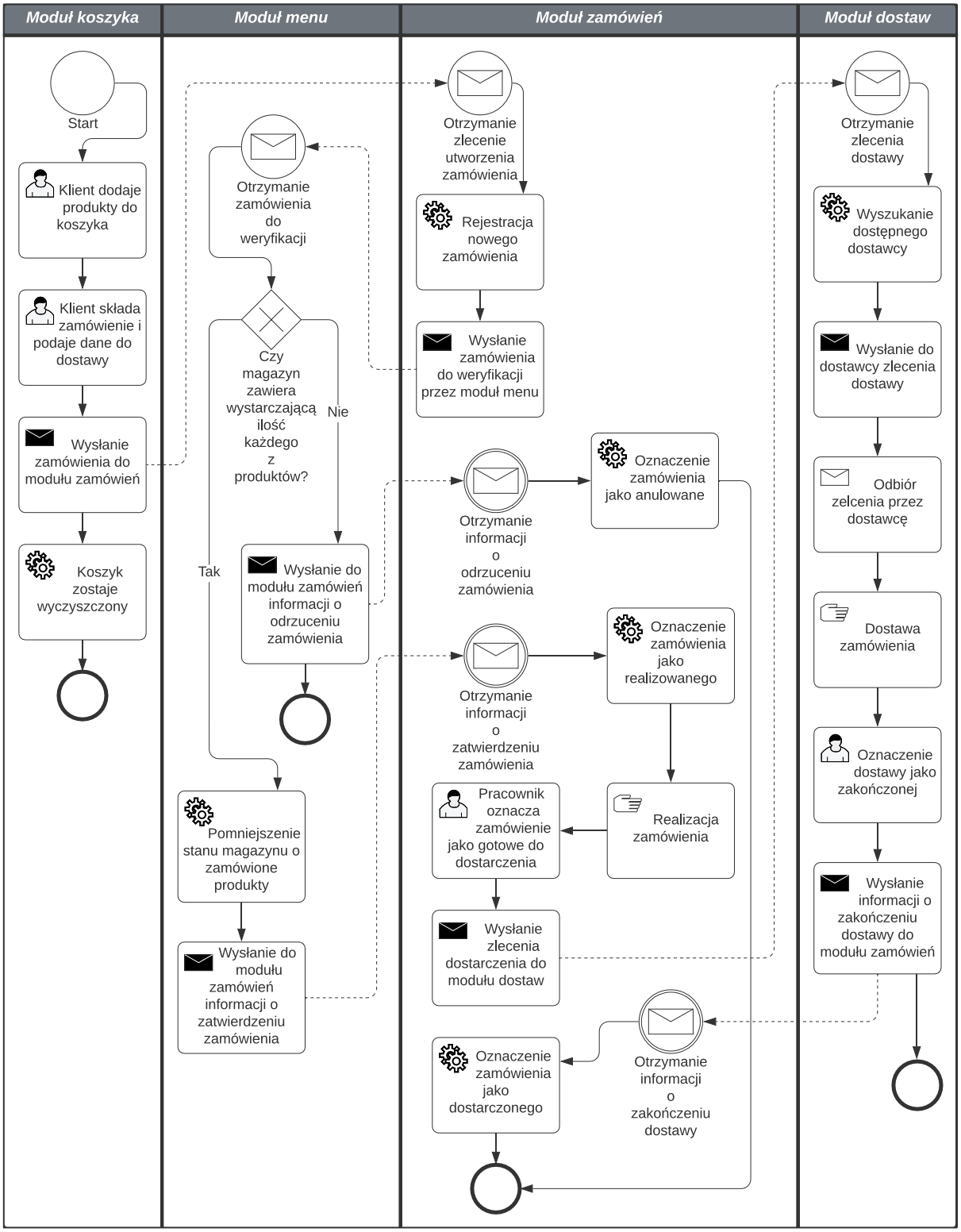
Moduł zamówień zarządza zamówieniami stworzonymi przez moduł koszyka. Jego zadaniem jest wysyłanie zamówień do modułu menu w celu ich weryfikacji oraz przekazywanie ich do modułu dostaw, kiedy pracownik pizzerii oznaczy dane zamówienie jako gotowe do dostarczenia. Funkcjonalnością tego modułu jest również prowadzenie rejestru wszystkich zamówień wraz z ich statusem.

Moduł dostaw służy do zarządzania procesem dostaw zamówień do klientów. Kiedy pracownik pizzerii wykorzysta moduł zamówień do oznaczenia zamówienia jako gotowe do dostarczenia, moduł dostaw wyszukuje dostępnego dostawcę, a następnie przydziela go do zamówienia. Dostawca powinien użyć tego modułu, kiedy zakończy dostawę, aby znów stać się dostępnym dla następnych zamówień. Historia dostaw wraz z informacją, kto dostarczył dane zamówienie jest również zapisywana.

4.2. Proces składania zamówienia

Głównym procesem obsługiwanym przez system jest składanie zamówienia przez klienta. Proces ten wymaga zaangażowania każdego z modułów i może zakończyć się zatwierdzeniem, wyprodukowaniem i dostarczeniem zamówionych pozycji, lub też odrzuceniem zlecenia z powodu

niewystarczającego stanu magazynowego co najmniej jednego z produktów. Rysunek 9 przedstawia diagram tego procesu.

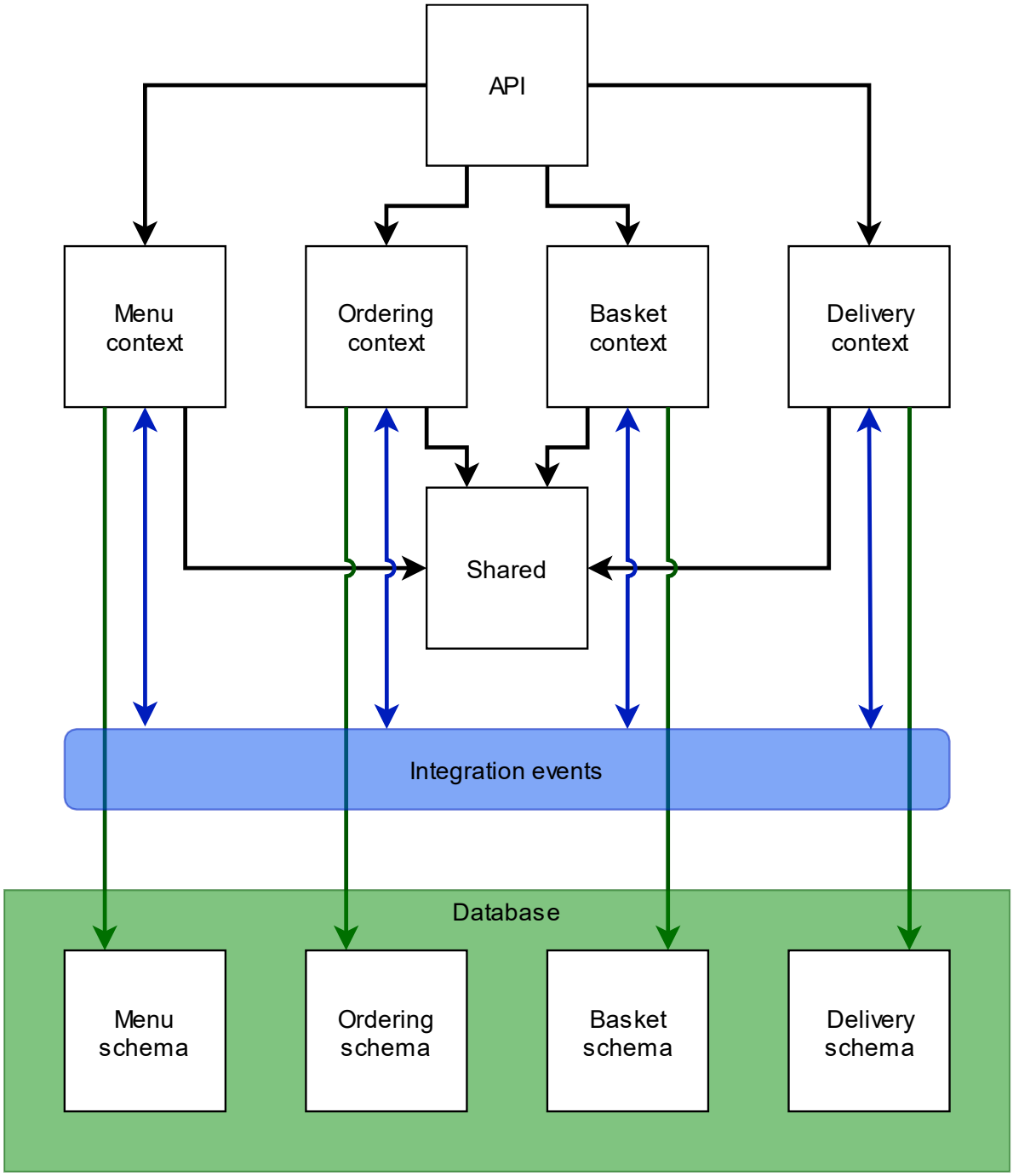


Rysunek 9 – Mapa procesu składania zamówienia. Źródło: opracowanie własne

5. Opis implementacji zgodnej z *Domain Driven Design*

Niniejszy rozdział zawiera opis technicznych aspektów przykładowego projektu w wersji opartej o *Domain Driven Design*.

5.1. Architektura wysokopoziomowa

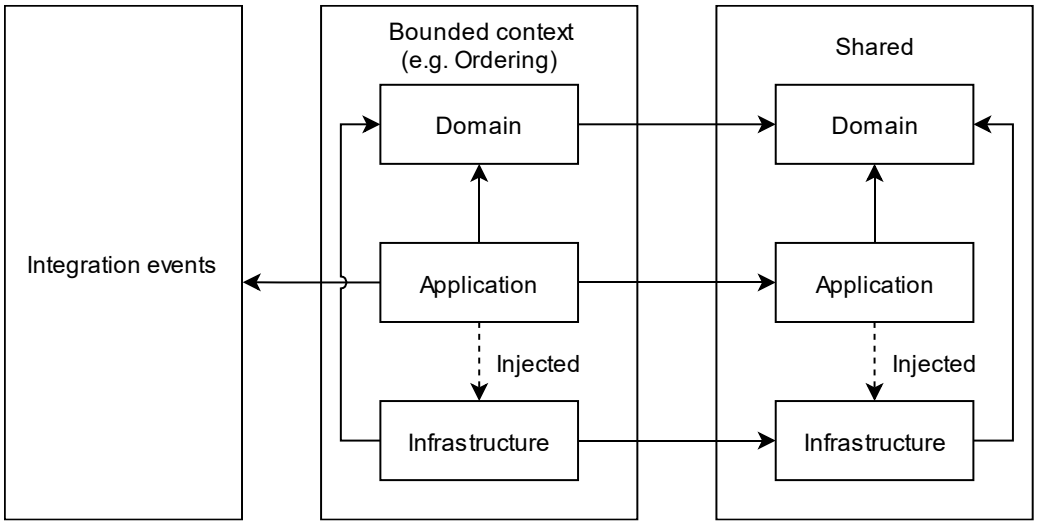


Rysunek 10 – Architektura wysokopoziomowa projektu DDD. Źródło: opracowanie własne

Jak widać na rysunku 10, mamy do czynienia z aplikacją monolityczną, ponieważ wszystkie punkty końcowe są scentralizowane w jednej aplikacji typu Web API. Aplikacja ta wykorzystuje cztery

konteksty (2.3), które dzielą zbiór klas bazowych umieszczonych w projektach *Shared*. Komunikacja między kontekstami odbywa się za pomocą zdarzeń integracyjnych, których struktura jest znana każdemu z kontekstów. Nie jest możliwa inna komunikacja kontekstów niż poprzez wyżej wymienione zdarzenia, co zapewnia luźne ich powiązanie [21]. Baza danych jest również centralna, ale jej wewnętrzna struktura podzielona jest na schematy odpowiadające każdemu z kontekstów. Analogicznie do braku bezpośrednich połączeń między kontekstami, każdy schemat bazy danych jest całkowicie niezależny od pozostałych. Kierunek zależności zobrazowany jest kierunkiem strzałek na rysunku. Konteksty nie są świadome istnienia aplikacji Web API.

5.2. Struktura kontekstu



Rysunek 11 – Struktura kontekstu. Źródło: opracowanie własne

Każdy z kontekstów składa się z trzech warstw, jak pokazano na rysunku 11. Są to:

- Warstwa domeny,
- Warstwa aplikacji,
- Warstwa infrastruktury.

Każda z tych warstw zawiera referencje do współdzielonych komponentów, zebranych w projekcie, którego struktura odpowiada strukturze kontekstu. Dodatkowo, warstwa aplikacji ma dostęp do projektu zawierającego zdarzenia integracyjne. Połączenie między warstwami aplikacji i infrastruktury nie jest bezpośrednie. Aby uniezależnić pozostałe warstwy od technologii bazy danych, zależności infrastruktury są wstrzykiwane poprzez mechanizm *inversion of control*. Każda warstwa jest zrealizowana poprzez osobny projekt typu .NET Standard.

5.3. Warstwa domeny

Warstwa domeny to centralna i najważniejsza warstwa każdego z kontekstów. Są w niej zdefiniowane wszystkie encje, *value objects* i agregaty, do których należą. Wszystkie niezmienne reguły biznesowe oraz niezmienniki agregatów znajdują się w tej warstwie. Każda klasa i wszystkie metody w nich zawarte muszą spełniać zasady języka wszechobecnego (2.2). Wartym podkreślenia jest fakt, iż warstwa domeny nie jest odpowiedzialna za przypadki użycia.

Kod 1 – Przykładowy value object – ProductName (DDD). Źródło opracowanie własne

```
// [...]
namespace Menu.Domain.ProductAggregate
{
    public class ProductName : ValueObject
    {
        public string Value { get; private set; }

        public ProductName(string value)
        {
            if (string.IsNullOrWhiteSpace(value))
            {
                throw new DomainException(
                    new ArgumentException(
                        "Product name can't be empty.", nameof(value)
                    )
                );
            }

            Value = value;
        }

        protected override IEnumerable<object> GetAtomicValues()
        {
            return new[] { Value };
        }
    }
}
```

Kod 1 zawiera przykładową klasę typu *value object* odpowiedzialną za przechowywanie nazwy produktu. Publiczna właściwość `Value` jest enkapsulowana za pomocą prywatnego *settera*, co w parze z konstruktorem sprawdzającym poprawność wstrzykniętej wartości, gwarantuje utrzymanie prawidłowego stanu obiektu niezależnie od okoliczności. Funkcja `GetAtomicValues` to napdpisanie wirtualnej funkcji z klasy bazowej `ValueObject`. Jest ona odpowiedzialna za selekcję wartości, na podstawie których dwa obiekty tego samego typu będą porównywane. W przypadku nazwy produktu mamy do dyspozycji tylko jedną taką właściwość.

Kod 2 – Fragment klasy Order (DDD). Źródło: opracowanie własne

```
// [...]
namespace Ordering.Domain.OrderAggregate
{
    public class Order : AggregateRoot
    {
        public Client Client { get; private set; }

        public Address Address { get; private set; }

        private List<OrderItem> _items;
        public IReadOnlyList<OrderItem> Items => _items;

        public OrderStatus Status { get; private set; }

        public Order(Client client, Address address,
                  List<OrderItem> items)
        {
            Client = client;
            Address = address;
            _items = items;
            Status = OrderStatus.New;
        }

        // ReSharper disable once UnusedMember.Local
        private Order() { } // For EF

        public void Ship()
        {
            if (Status != OrderStatus.InPreparation)
            {
                throw new DomainException(
            "Cannot ship an unprepared order");
            }
            Status = OrderStatus.InDelivery;
        }
      //[...]
    }

    public enum OrderStatus
    {
        New = 1,
        InPreparation = 2,
        InDelivery = 3,
        Completed = 4,
        Cancelled = 5
    }
}
```

Kod 2 to fragment klasy `Order`, z której usunięto część metod aby zmniejszyć jej objętość w tekście. Klasa ta dziedziczy po klasie `AggregateRoot`, co oczywiście oznacza, że jest ona korzeniem agregacji. Oprócz enkapsulacji za pomocą prywatnych *setterów*, widzimy tutaj również zastosowanie kolekcji typu `IReadOnlyList`, która uniemożliwia wprowadzania zmian w publicznie dostępnym zbiorze elementów zamówienia. Metoda `Ship` to przykład implementacji logiki biznesowej. Oprócz faktycznego wykonania operacji, dokonuje walidacji stanu obiektu aby nie dopuścić do wprowadzenia

go w stan nieprawidłowy. Bezparametrowy konstruktor jest wymaganiem ORM Entity Framework Core, na szczęście jednak może on pozostać prywatny. Jest to praktyczny przykład zanieczyszczenia modelu domenowego aspektami konkretnej technologii. Model zaprezentowany w kodzie 2 to tzw. bogaty model domenowy, ponieważ oprócz danych, zawiera logikę, która na nich operuje. Na samym początku kodu klasy widać zastosowanie dwóch *value object*. Każdy z nich enkapsuluje pewne wartości za pomocą wbudowanej logiki walidacji. Dzięki temu obiekty encji, które je wykorzystują zwolnione są z obowiązku przeprowadzania tej walidacji samodzielnie. Zabezpiecza to przed duplikacją kodu kontrolującego poprawność i znacznie upraszcza implementację samych encji, które mogą skupić się na swojej części logiki biznesowej. Warto również odnotować, że konstruując obiekt klasy `Order`, nie ma możliwości omyłkowego przypisania klienta do adresu i odwrotnie, gdyż nie dopuści do tego kompilator.

Kolejnym rodzajem klasy wchodzącej w skład warstwy domeny jest serwis (2.7). W przykładowym projekcie do obsługi pizzerii znalazł się tylko jeden serwis, który obsługuje operacje, które nie pasują do żadnej encji lub *value object*. Są to operacje rozpoczęcia dostawy i zakończenia dostawy zamówienia. Każda z nich wymaga zmiany stanu dwóch encji – zamówienia i dostawcy, zatem nie mogą one należeć do jednej z nich. Kod 3 to implementacja serwisu `OrderDeliveryService` z metodami `StartDeliveryAsync` i `FinishDeliveryAsync`.

Kod 3 – Implementacja serwisu OrderDeliveryService (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
public class OrderDeliveryService
{
    public async Task StartDeliveryAsync(Order order)
    {
        var supplier = await _supplierRepository
                        .GetByIdAsync(order.SupplierId);

        if (supplier == null)
        {
            throw new RecordNotFoundException(
                        order.SupplierId, nameof(Supplier));
        }

        order.StartDelivery();
        supplier.StartDelivery();
    }

    public async Task FinishDeliveryAsync(Order order)
    {
        var supplier = await _supplierRepository
                        .GetByIdAsync(order.SupplierId);

        if (supplier == null)
        {
            throw new RecordNotFoundException(
                        order.SupplierId, nameof(Supplier));
        }

        order.FinishDelivery();
        supplier.FinishDelivery();
    }
}
```

Kod 4 – Obsługa zdarzenia dostarczenia zamówienia (DDD). Źródło: opracowanie własne

```
// [...]
using MediatR;

namespace Delivery.Domain.DomainEvents
{
    public class OrderShippedDomainEvent : INotification
    {
        public OrderShippedDomainEvent(Order order)
        {
            Order = order;
        }

        public Order Order { get; }
    }
}
```

Poza encjami, *value objects*, i serwisami, do warstwy domeny należą zdarzenia domenowe (2.9). Służą one do informowania agregatów, że coś istotnego stało się w obrębie danego kontekstu. Można je wykorzystać do komunikacji między korzeniami agregacji, lub po prostu do implementacji reakcji na te wydarzenia w warstwie aplikacji. Kod 4 to implementacja zdarzenia dostarczenia zamówienia. Jak widać, jest to tylko kontener na dane które będą dostarczone wraz ze zdarzeniem. Interfejs `INotification` pozwala na publikację zdarzenia za pomocą biblioteki MediatR. W przeciwieństwie do zdarzeń integracyjnych, zdarzenia domenowe mogą zawierać obiekty należące do języka wszechobecnego danej domeny, ponieważ ich zasięg nie obejmuje innych kontekstów.

Kod 5 – Zdarzenie domenowe OrderShipped (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
public class OrderShippedDomainEventHandler:
                        INotificationHandler<OrderShippedDomainEvent>
{
    private readonly ILogger<OrderShippedDomainEventHandler>
                                                            _logger;

    public OrderShippedDomainEventHandler(
            ILogger<OrderShippedDomainEventHandler> logger)
    {
        _logger = logger;
    }

        public Task Handle(OrderShippedDomainEvent notification,
                           CancellationToken cancellationToken)
    {
        _logger.Log(LogLevel.Information,
      $"New order has been shipped!\n{notification.Order}");

        return Task.CompletedTask;
    }
}
```

Obsługa wyżej opisanego zdarzenia nie podejmuje żadnych istotnych z punktu widzenia biznesowego akcji. Jedyne, co znajduje się w kodzie 5, to wydrukowanie informacji o wystąpieniu zdarzenia za pomocą biblioteki do logowania. Celem umieszczenia tego fragmentu w projekcie było tylko i wyłącznie zademonstrowanie działania zdarzeń domenowych.

5.4. Warstwa aplikacji

Warstwa aplikacji realizuje zapytania oraz przypadki użycia systemu. Każdy przypadek składa się z trzech części:

- Komenda,
- Walidator komendy,
- Klasa realizująca komendę (*command handler*)

Komendy swoją strukturą przypominają obiekty DTO. Służą one tylko do przekazywania danych wprowadzonych przez użytkownika do systemu. Jedną z najważniejszych cech projektów opartych o CQRS jest to, że modyfikacje danych mogą odbywać się tylko i wyłącznie za pośrednictwem komend i zdarzeń integracyjnych. Oznacza to, że analizując komendy i zdarzenia, mamy dostęp do wszystkich czynności, jakie użytkownik może wykonać w systemie.

Kod 6 – Komenda dodania produktu do koszyka (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
public class AddItemToBasketCommand : IRequest<BasketDTO>
{
    public int? BasketId { get; set; }
    public int ProductId { get; }
    public int Quantity { get; }
    public float UnitPrice { get; }

    public AddItemToBasketCommand(
        int productId,
        int quantity,
        float unitPrice)
    {
        ProductId = productId;
        Quantity = quantity;
        UnitPrice = unitPrice;
    }
}
```

Kod 6 przedstawia komendę dodania produktu do koszyka. Interfejs `IRequest` pochodzi z biblioteki MediatR i pozwala na luźne powiązanie komendy z klasą, która ją realizuje. W ten sposób, kontroler, który wysyła komendę do wykonania, nie jest świadomy obiektu, który ostatecznie wykona logikę.

Aby zabezpieczyć aplikację przed niepoprawnymi danymi, wykorzystano bibliotekę FluentValidation, która pozwala na zdefiniowanie zbioru reguł, które muszą spełniać właściwości danego obiektu, w tym przypadku – komendy.

Kod 7 Walidator komendy dodania produktu do koszyka (DDD). Źródło: opracowanie własne

```
using FluentValidation;

namespace Basket.Application.AddItemToBasketApplication
{
    public class AddItemToBasketCommandValidator
        : AbstractValidator<AddItemToBasketCommand>
    {
        public AddItemToBasketCommandValidator()
        {
            RuleFor(cmd => cmd.Quantity).GreaterThan(0);
            RuleFor(cmd => cmd.BasketId).GreaterThan(0);
            RuleFor(cmd => cmd.ProductId).GreaterThan(0);
            RuleFor(cmd => cmd.UnitPrice).GreaterThan(0);
        }
    }
}
```

W kodzie 7 można odnaleźć przykłady reguł dla właściwości typu `int` i `float`, które nie pozwalają na przypisanie im wartości mniejszych niż zero. Złamanie co najmniej jednej reguły skutkuje zwróceniem błędu 400 – *Bad Request*.

Ostatnim elementem składającym się na system komend są obiekty, które je wykonują. Kod 8 przedstawia klasę, której zadaniem jest dodanie produktu do koszyka.

Kod 8 – Klasa wykonująca komendę dodania produktu do koszyka (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
public class AddItemToBasketCommandHandler
    : IRequestHandler<AddItemToBasketCommand, BasketDTO>
{
    private readonly IRepository<CustomerBasket> _basketRepository;
    private readonly IMapper _mapper;

    public AddItemToBasketCommandHandler(
        IRepository<CustomerBasket> basketRepository,
        IMapper mapper)
    {
        _basketRepository = basketRepository;
        _mapper = mapper;
    }

    public async Task<BasketDTO> Handle(
        AddItemToBasketCommand request,
        CancellationToken cancellationToken)
    {
        CustomerBasket customerBasket;
        if (request.BasketId == null)
        {
            customerBasket = new CustomerBasket();
            await _basketRepository.AddAsync(customerBasket);
        }
        else
        {
            customerBasket =
                await _basketRepository
                    .GetByIdAsync(request.BasketId.Value);
        }

        customerBasket
            .AddItemToBasket(
                request.ProductId,
                request.Quantity,
                request.UnitPrice);

        await _basketRepository.UnitOfWork.SaveEntitiesAsync();

        return _mapper.Map<BasketDTO>(customerBasket);
    }
}
```

Interfejs `IRequestHandler` z biblioteki MediatR, działa w parze z `IRequest`, aby powiązać komendy z klasami, które je wykonują.

Inną ciekawą funkcjonalnością biblioteki MediatR są tzw. zachowania. Pozwalają one na wzbogacenie procesu wykonywania komend o dodatkową logikę, która wykonuje się w zadanej kolejności po ich wysłaniu. Mechanizm ten jest bardzo podobny do wbudowanych w ASP.NET Core *middleware*. Przykładowy projekt do obsługi pizzerii wykorzystuje trzy zachowania zainspirowane projektem eShopOnContainers [26] firmy Microsoft.

Są to:

- Zachowanie logujące,
- Zachowanie walidujące,
- Zachowanie transakcyjne.

Zachowanie logujące odpowiada za logowanie rozpoczęcia i zakończenia wszystkich prób wykonania komendy. Zachowanie walidujące uruchamia odpowiednie walidatory, pasujące do wykonywanej komendy. Najważniejsze z zachowań – transakcyjne – rozpoczyna transakcję w bazie danych i pilnuje, aby była ona cofnięta w przypadku wystąpienia błędu. Kod 9 zawiera krótki fragment implementacji tego zachowania.

Kod 9 – Zachowanie transakcyjne (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
TResponse response;
using (var transaction = await _unitOfWork.BeginTransactionAsync())
{
    _logger.LogInformation(
    "----- Begin transaction {TransactionId} " +
    "for {CommandName} ({@Command})",
    transaction.Id, typeName, request);

    response = await next();

    _logger.LogInformation(
    "----- Commit transaction {TransactionId}" +
    " for {CommandName}",
    transaction.Id, typeName);

    await _unitOfWork.SaveEntitiesAsync();
    await _unitOfWork.CommitTransactionAsync(transaction);
}
// [...]
```

Zatwierdzenie transakcji posiada wbudowany mechanizm reakcji na wyjątki, który w razie potrzeby, dokonuje jej cofnięcia. Gdyby jednak pojawił się problem na tym etapie, zwolnienie obiektu powołanego w bloku `using` gwarantuje wycofanie niezatwierdzonej transakcji.

Do warstwy aplikacji należą również zapytania wywoływane za pomocą lekkiej biblioteki ORM – Dapper. Zapytania te pomijają warstwę infrastruktury oraz domeny i materializują wyniki bezpośrednio do obiektów DTO. Podczas odczytywania danych z bazy, niepotrzebna jest walidacja, ani skomplikowany ciąg zachowań, w związku z czym warstwa zapytań jest bardzo cienka i zarazem wydajniejsza, niż rozwiązanie oparte o Entity Framework. Kod 10 zawiera przykład zapytania zwracającego zawartość koszyka.

Kod 10 – Zapytanie zwracające zawartość koszyka (DDD). Źródło: opracowanie własne

```
public async Task<BasketDTO> GetBasketAsync(int id)
{
    await using var connection = new SqlConnection(_connectionString);

    BasketDTO resultBasket = null;

    var queryResult = await connection
        .QueryAsync<BasketDTO, BasketItemDTO, BasketDTO>(
        "SELECT b.[Id] as Id, " +
        "bi.[Id] as BasketItemId, " +
        "bi.[ProductId], bi.[Quantity] " +
        "FROM [Basket].[Baskets] b " +
        "INNER JOIN [Basket].[BasketItems] bi " +
        "on bi.BasketId = b.[Id] " +
        $"WHERE b.[Id]={id}",
        (basket, basketItem) =>
        {
            if (resultBasket == null)
            {
                resultBasket = basket;
                resultBasket.Items = new List<BasketItemDTO>();
            }
            resultBasket.Items.Add(basketItem);
            return resultBasket;
        },
        splitOn: "BasketItemId");

    return queryResult.FirstOrDefault();
}
```

Zapytania wywoływane są bezpośrednio przez kontrolery aplikacji Web API, bez pośrednictwa mediatora.

Do warstwy aplikacji należą również zdarzenia integracyjne, które wymieniają informacje między kontekstami. Takie zdarzenia mogą być rozsyłane przez obiekty wykonujące komendy za pomocą przytoczonej wcześniej biblioteki MediatR. Ponieważ przykładowy system do obsługi pizzerii jest aplikacją monolityczną, nie ma potrzeby stosowania zewnętrznej szyny zdarzeń, takiej jak Azure Service Bus, czy RabbitMQ. W przeciwieństwie do komend, zdarzenia mogą być obsługiwane przez nieskończenie wiele odbiorców. W systemie monolitycznym, który nie wysyła, ani nie odbiera zdarzeń od zewnętrznych źródeł, wykonywanie zdarzeń integracyjnych może odbywać się synchronicznie, dzięki czemu obejmuje je logika zachowań, w tym transakcyjnego.

Kod 11 – Zdarzenie integracyjne zakończenia dostawy (DDD). Źródło: opracowanie własne

```
using MediatR;

namespace Shared.IntegrationEvents.Delivery
{
    public class OrderDeliveryFinishedIntegrationEvent : INotification
    {
        public OrderDeliveryFinishedIntegrationEvent(int orderId)
        {
            OrderId = orderId;
        }

        public int OrderId { get; }
    }
}
```

Kod 11 to przykład zdarzenia integracyjnego, który komunikuje zakończenie dostawy zamówienia. Struktura zdarzeń jest analogiczna do struktury komend – są to obiekty, których jedynym celem jest przekazanie danych do obiektów, które je obsługują.

Kod 12 – Klasa obsługująca zdarzenie zakończenia dostawy (fragment) (DDD). Źródło: opracowanie własne

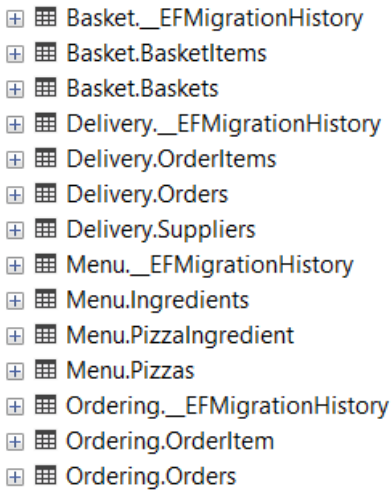
```
// [...]
public class OrderDeliveryFinishedIntegrationEventHandler
    : INotificationHandler<OrderDeliveryFinishedIntegrationEvent>
{
    private readonly IRepository<Order> _orderRepository;

    public OrderDeliveryFinishedIntegrationEventHandler(
        IRepository<Order> orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task Handle(
        OrderDeliveryFinishedIntegrationEvent notification,
        CancellationToken cancellationToken)
    {
        var order = await _orderRepository
            .GetByIdAsync(notification.OrderId);

        if (order == null)
        {
            throw new RecordNotFoundException(
                notification.OrderId,
                nameof(Order));
        }

        order.Complete();

        await _orderRepository.UnitOfWork.SaveEntitiesAsync();
    }
}
```

Kod 12 przedstawia klasę obsługującą pokazane wcześniej zdarzenie zakończenia dostawy zamówienia. Zasada powiązania interfejsów `INotification` i `INotificationHandler` z biblioteki MedtaR

jest taka sama jak w przypadku komend i odpowiadających im interfejsom, za wyjątkiem braku ograniczenia do tylko jednej klasy obsługującej dane zdarzenie.

5.5. Warstwa infrastruktury

Warstwa infrastruktury odpowiada za pośredniczenie między warstwą aplikacji i domeny a bazą danych. Jest to implementacja kontekstów bazy danych w technologii Entity Framework Core. Każdy z kontekstów biznesowych (zamówienia, dostawy, menu, koszyk) posiada własny kontekst bazy danych (`DbContext`), co w połączeniu z migracjami *Code First* przekłada się na wytworzenie osobnego schematu w bazie danych dla każdej domeny, co pokazano na rysunku 12. W związku z powyższym, dane należące do osobnych kontekstów biznesowych są całkowicie od siebie niezależne.



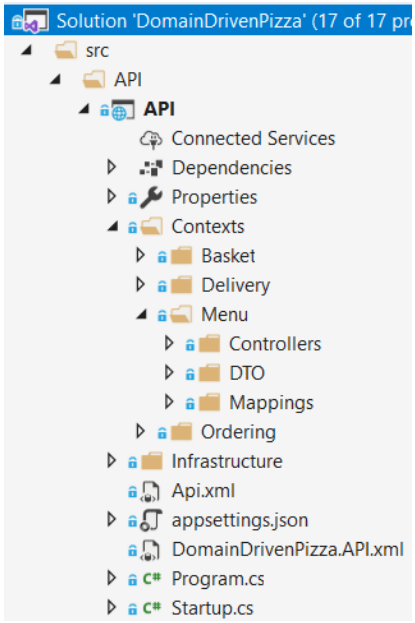
Rysunek 12 – Tabele w bazie danych. Źródło: opracowanie własne

Repozytoria zaimplementowane w warstwie infrastruktury służą do modyfikacji i odczytu danych przez obiekty wykonujące komendy. Nie są one wykorzystywane do odczytu danych przez klienta, ponieważ tę funkcjonalność realizuje cienka warstwa odczytów wraz z biblioteką Dapper.

5.6. Aplikacja Web API

Aplikacja typu Web API w technologii ASP.NET Core 3.1 to centralny punkt wejściowy do systemu. Składa się ona z kontrolerów udostepniających punkty końcowe odpowiadające przypadkom użycia zdefiniowanym w warstwie aplikacji (5.4). Oprócz tego, znajduje się w niej konfiguracja kontenera *inversion of control*, a zatem dokonuje się tutaj zespolenie interfejsów z konkretnymi implementacjami. Do tego celu wykorzystano bibliotekę Autofac, która zawiera kilka przydatnych rozszerzeń względem kontenera wbudowanego w ASP.NET Core, które okazują się niezwykle przydatne podczas rejestracji klas wykorzystywanych przez bibliotekę MediatR.

Podział projektu na foldery odpowiada zasadom tzw. *screaming architecture* [25]. Polega ona na tym, że osoba, która pierwszy raz widzi projekt, od razu wie, do czego on służy i na czyje potrzeby został stworzony. W przeciwieństwie do klasycznego podejścia, które segreguje klasy według ich znaczenia technicznego, *screaming architecture* podkreśla biznesowe znaczenie komponentów systemu poprzez „wykrzyczenie" pojęć charakterystycznych dla języka wszechobecnego domeny.



Rysunek 13 – Struktura plików projektu API. Źródło: opracowanie własne

Jak widać na rysunku 13, struktura plików projektu API podzielona jest na foldery według kontekstów biznesowych. Dopiero podfoldery każdego z nich wskazują na techniczny rodzaj elementów, które się w nich znajdują. Zobaczywszy foldery takie jak *Basket* czy *Ordering*, od razu wiadomo, że system zajmuje się pewnego rodzaju sklepem internetowym, a folder *Menu* zawęża ten wniosek do sklepu realizującego zamówienia w restauracji.

Odpowiedzialności kontrolerów są następujące:

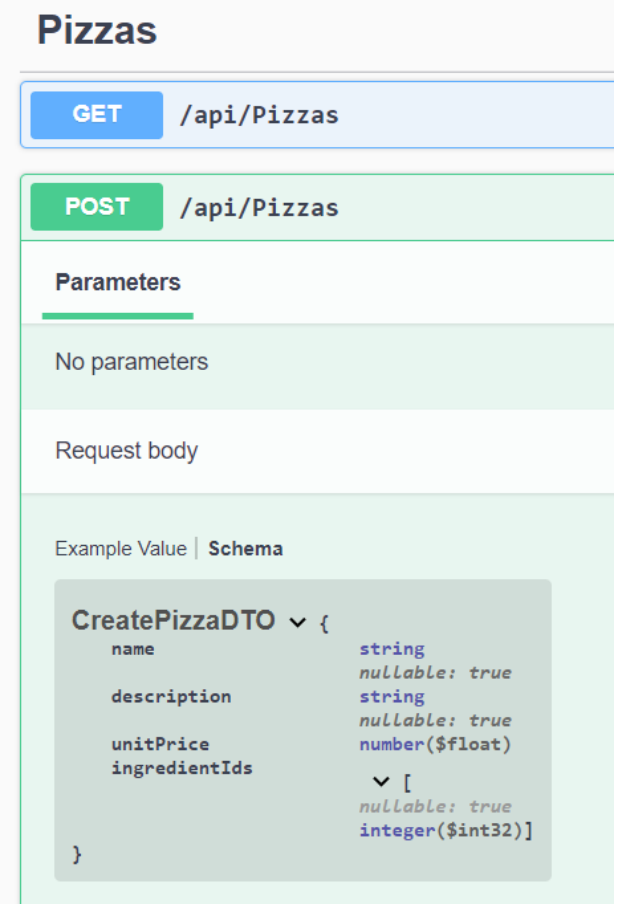
- Przyjmowanie danych od użytkownika,
- Konwersja danych wejściowych do formatu zgodnego z warstwą aplikacji,
- Przekazanie wygenerowanych komend, lub argumentów zapytań do wykonania w warstwie aplikacji,
- Zwrócenie prawidłowego statusu http.

Kontrolery nie wykonują logiki biznesowej, ani nie oddziałują bezpośrednio na obiekty należące do warstwy domeny. Komunikacja wejściowa, jak i wyjściowa odbywają się za pomocą obiektów DTO (*data transfer object*).

Kod 13 – PizzaController (fragment) (DDD). Źródło: opracowanie własne

```
// [...]
[Route("api/[controller]")]
[ApiController]
public class PizzasController : ControllerBase
{
    [HttpGet("{id}")]
    [ProducesResponseType(typeof(PizzaDTO), (int)HttpStatusCode.OK)]
    [ProducesResponseType((int)HttpStatusCode.NotFound)]
    public async Task<ActionResult> Get(int id)
    {
        var pizza = await _productQueries.GetPizzaByIdAsync(id);
        if (pizza == null)
        {
            return NotFound();
        }
        return Ok(pizza);
    }

    [HttpPost]
    [ProducesResponseType((int)HttpStatusCode.BadRequest)]
    [ProducesResponseType(typeof(PizzaDTO),
                                (int)HttpStatusCode.Created)]
    public async Task<ActionResult> Create(
                               [FromBody] CreatePizzaDTO pizzaDto)
    {
        var command = _mapper.Map<CreatePizzaCommand>(pizzaDto);

        var result = await _mediator.Send(command);

        return CreatedAtAction(
                  nameof(Get),
                        new { result.Id }, result
            );
    }
// [...]
```

Kod 13 zawiera fragment kontrolera do obsługi kontekstu produktu pizzy. Metoda `Get` odczytuje dane z części zapytań warstwy aplikacji i w zależności od otrzymanego rezultatu, decyduje, czy zwrócić błąd 404, czy status 200 wraz z wynikiem zapytania. Metoda `Create` dokonuje mapowania danych wejściowych biblioteką AutoMapper do postaci komendy CQRS, aby następnie wykorzystać bibliotekę MediatR do jej wykonania. Dodatkowe atrybuty typu `ProducesResponseType` służą do generowania interfejsu graficznego za pomocą biblioteki Swagger, którego fragment pokazano na rysunku 14.



Rysunek 14 – Swagger. Źródło: opracowanie własne

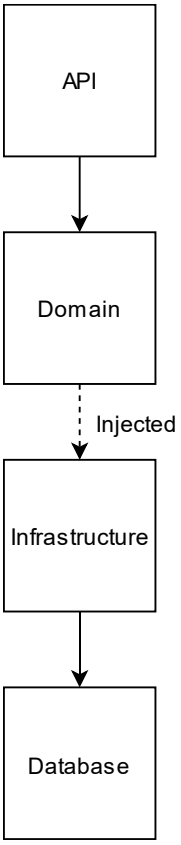
Aplikacja Web API zajmuje się również kwestiami pomocniczymi, takimi jak globalna obsługa wyjątków, czy konfiguracja profili mapowania obiektów. Dzięki globalnemu filtrowi wyjątków, który w razie wystąpienia błędu, dokonuje selekcji kodu odpowiedzi http, zgodnego z przyczyną nieprawidłowości, objętość kontrolerów ulega znacznemu skróceniu. W ten sposób uniknięto również znaczącej duplikacji kodu źródłowego.

6. Opis implementacji opartej o model anemiczny

Niniejszy rozdział zawiera opis implementacji systemu do obsługi pizzerii w oparciu o anemiczny model danych, bez wykorzystania wzorca CQRS, który zastąpiony został architekturą klasyczną, wielowarstwową.

6.1. Architektura wysokopoziomowa

W przeciwieństwie do rozwiązania opartego o *Domain Driven Design*, architektura systemu wielowarstwowego jest trywialna.



Rysunek 15 – Architektura wysokopoziomowa aplikacji wielowarstwowej. Źródło: opracowanie własne

Jak widać na rysunku 15, nie ma tutaj podziału na konteksty, zarówno na poziomie samej aplikacji, jak i bazy danych. W związku z tym, nie ma również potrzeby stosowania zdarzeń integracyjnych. Projekt oparty o taką architekturę jest tak monolityczny, jak to tylko możliwe. Centralny punkt wejściowy – aplikacja typu Web API komunikuje się z warstwą domeny biznesowej, do której poprzez mechanizm *inversion of control* wstrzykiwane są obiekty z warstwy infrastruktury, która pośredniczy w wymianie danych z bazą.

6.2. Warstwa domeny

Warstwa domeny w projekcie wielowarstwowym, anemicznym, którą nazywam po prostu *Core*, łączy odpowiedzialności warstw aplikacji i domeny projektu opartego o *Domain Driven Design*. Znajdziemy w niej model danych odzwierciedlający relacyjną bazę danych, oraz zbiór serwisów

realizujących przypadki użycia. Kody 14 i 15 przedstawiają modele koszyka i produktu w koszyku. Najbardziej widoczną cechą obu klas jest całkowity brak enkapsulacji, wyrażony przez publiczne *settery*. Nie ma tutaj podziału na encje i *value objects*. Model anemiczny składa się tylko z encji, które nie są pogrupowane w agregaty, gdyż wszystkie tworzą jeden, monolityczny model danych. Nie znajdziemy tutaj również absolutnie żadnej logiki związanej z danymi, które przechowują dane obiekty, co uzasadnia nazywanie takich obiektów anemicznymi.

Kod 14 – Model koszyka (model anemiczny). Źródło: opracowanie własne

```
using System.Collections.Generic;

namespace AnemicPizza.Core.Models.Basket
{
    public class CustomerBasket : Entity
    {
        public IList<BasketItem> Items { get; set; }

        public CustomerBasket()
        {
            Items = new List<BasketItem>();
        }
    }
}
```

Kod 15 – Model produktu w koszyku (model anemiczny). Źródło: opracowanie własne

```
using AnemicPizza.Core.Models.Products;

namespace AnemicPizza.Core.Models.Basket
{
    public class BasketItem : Entity
    {
        public int ProductId { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
        public CustomerBasket Basket { get; set; }
        public int BasketId { get; set; }
    }
}
```

Serwisy w warstwie domeny pełnią rolę komend i zapytań warstwy aplikacji projektu opartego o *Domain Driven Design*. Ponieważ zabrakło tutaj wzorca, który rozgraniczałby te dwie kwestie, repozytorium wstrzykiwane przez odpowiednie interfejsy obsługuje zapis danych oraz odczyt na potrzeby użytkownika i logiki modyfikującej zawartość bazy danych. Kod 16 zawiera funkcję realizującą zapytanie o koszyk klienta. Zamiast bezpośredniego wykonania zapytania SQL biblioteką Dapper, robi to z wykorzystaniem repozytorium.

Kod 16 – Metoda serwisu zwracająca koszyk (model anemiczny). Źródło: opracowanie własne

```
public Task<CustomerBasket> GetBasketAsync(int basketId)
{
    return _basketRepository.GetByIdAsync(basketId);
}
```

Model anemiczny wymusza na serwisach dokonywanie walidacji stanu obiektów, które modyfikują. Kod 17 to implementacja metody dodającej produkt do koszyka. Analogiczna metoda, w postaci klasy obsługującej komendę, została zaprezentowana w kodzie 8 (str.30). Oprócz faktycznego dodania

produktu do koszyka, widzimy tutaj logikę, która sprawdza, czy dany produkt już znajduje się w koszyku, aby w razie potrzeby tylko zwiększyć jego ilość. Ponieważ model koszyka jest anemiczny, logika tego rodzaju musi znajdować się na zewnątrz – w tym przypadku w serwisie.

Kod 17 – Metoda dodająca produkt do koszyka (model anemiczny). Źródło: opracowanie własne

```
public async Task<CustomerBasket> AddItemToBasketAsync(int? basketId,
            int productId,
            int quantity)
{
    CustomerBasket customerBasket;
    if (basketId == null)
    {
        customerBasket = new CustomerBasket();
        await _basketRepository.AddAsync(customerBasket);
    }
    else
    {
        customerBasket = await _basketRepository
            .GetByIdAsync(basketId.Value);
    }

    var basketItem = new BasketItem
    {
        ProductId = productId,
        Quantity = quantity,
        Basket = customerBasket
    };

    var productExistsInBasket = customerBasket.Items
        .Any(bi => bi.ProductId == productId);

    if (productExistsInBasket)
    {
        var existingBasketItem = customerBasket
            .Items
            .First(item => item.ProductId == productId);
        existingBasketItem.Quantity += quantity;
    }
    else
    {
        customerBasket.Items.Add(basketItem);
    }

    return customerBasket;
}
```

Projekt oparty o model anemiczny nie jest podzielony na konteksty, a zatem metody, które należą do innego obszaru biznesowego, mogą być wywoływane bezpośrednio poprzez interfejsy odpowiednich serwisów. W rozwiązaniu opartym o *Domain Driven Design*, utworzenie zamówienia na podstawie koszyka wymaga wysłania zdarzenia integracyjnego z kontekstu koszyka, które następnie zostanie odebrane przez kontekst zamówień. Kod 18 zawiera logikę realizującą tę samą czynność za pomocą wywołania serwisu zamówień z poziomu serwisu koszyka. Jest to możliwe, ponieważ model danych jest całkowicie zintegrowany.

Kod 18 – Metoda tworząca zamówienie w oparciu o koszyk (model anemiczny). Źródło: opracowanie własne

```
public async Task CheckoutAsync(
            int basketId,
            string firstName,
            string lastName,
            string emailAddress,
            string phoneNumber,
            string city,
            string addressLine1,
            string addressLine2,
            short zipCode)
        {
            var basket = await _basketRepository.GetByIdAsync(basketId);

            if (basket == null)
            {
                throw new RecordNotFoundException(
                                        basketId, nameof(CustomerBasket));
            }

            await _orderingService.CreateOrderAsync(
                firstName,
                lastName,
                emailAddress,
                phoneNumber,
                city,
                addressLine1,
                addressLine2,
                zipCode,
                basket.Items
                    .ToDictionary(bi => bi.ProductId,
                        bi => bi)
            );

            basket.Items.Clear();
        }
```

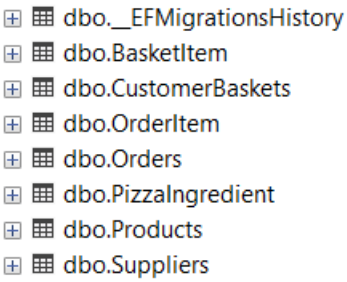
Zdarzenia domenowe, podobnie jak integracyjne, w rozwiązaniu anemicznym nie istnieją. Wszelka dodatkowa logika, która wykonywałaby się w formie reakcji na takie zdarzenie, musi być umieszczona bezpośrednio wewnątrz odpowiedniej metody w serwisie.

W odróżnieniu od projektu zbudowanego w oparci o *Domain Driven Design*, projekt z modelem anemicznym nie wykorzystuje biblioteki pośredniczącej MediatR, a zatem nie może skorzystać z jej mechanizmu zachowań. W związku z tym, rozpoczynanie, zatwierdzanie i cofanie transakcji odbywa się w warstwie API, na poziomie metod kontrolerów.

6.3. Warstwa infrastruktury

W aplikacji opartej o anemiczny model danych, model ten jest zintegrowany, co przekłada się na monolityczną strukturę bazy danych. W projekcie z wykorzystaniem *Domain Driven Design*, każdy kontekst posiadał własną warstwę infrastruktury z kontekstem bazy danych, który generował migrację odpowiadającą tylko jednemu schematowi bazy. Model anemiczny odzwierciedla bezpośrednio model relacyjny, a zatem wspólna warstwa infrastruktury generuje jedną migrację *Code First*, która obejmuje

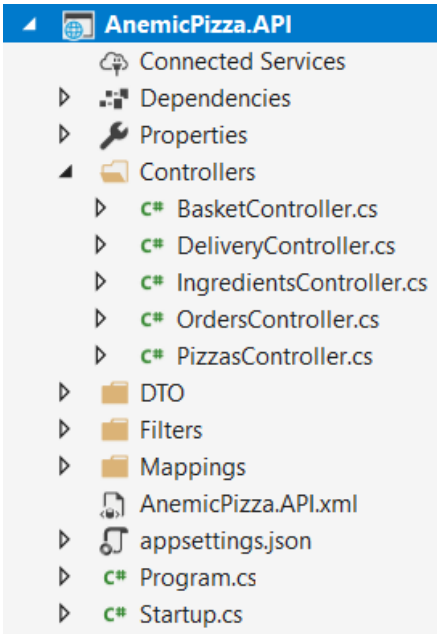
swoim zasięgiem wszystkie tabele w bazie danych. Rysunek 16 przedstawia wygenerowane przez tę migrację tabele. Zastosowana technologia ORM to Entity Framework Core.



Rysunek 16 – Tabele w bazie danych projektu z modelem anemicznym. Źródło: opracowanie własne

6.4. Aplikacja Web API

Aplikacja Web API realizuje takie same odpowiedzialności, jak jej odpowiednik w projekcie opartym o *Domain Driven Design*. Zastosowana technologia również nie uległa zmianie – jest to ASP.NET Core 3.1. Klasy w projekcie API podzielone są tym razem według ich przeznaczenia technicznego, co pokazano na rysunku 17.



Rysunek 17 – Struktura projektu API w aplikacji opartej o model anemiczny. Źródło: opracowanie własne

Główną różnicą w zawartości kontrolerów jest wykonywanie przypadków użycia nie przez bibliotekę MediatR i komendy, a przez bezpośrednie wywoływanie metod interfejsów odpowiednich serwisów. Ponieważ warstwa domeny, pełniąca również rolę zarządzania przypadkami użycia, nie posiada wbudowanego mechanizmu zachowań, który udostępnia biblioteka MediatR, transakcyjność akcji musi zapewnić kontroler. Kod 19 to przykładowa metoda, która służy do rozpoczęcia dostawy zamówienia. Zanim jednak wywoła ona metodę serwisu, tworzy obiekt jednostki pracy, który zakłada transakcję na bazie danych. Przed wyjściem z metody kontrolera, transakcja musi zostać zatwierdzona, a w przypadku wystąpienia błędu – wycofana, za co odpowiada wyżej wspomniany obiekt jednostki pracy.

Kod 19 – Metoda kontrolera aplikacji (model anemiczny). Źródło: opracowanie własne

```
[HttpPost("ship/{id}")]
public async Task<IActionResult> ShipOrder(int id)
{
    using var uow = _unitOfWorkFactory.Create();
    await _deliveryService.StartDeliveryAsync(id);
    uow.Commit();
    return NoContent();
}
```

Warstwa Web API wyposażona jest w globalny filtr wyjątków, który działa na zasadzie identycznej w stosunku do projektu opartego o *Domain Driven Design*. Jego celem jest reakcja na określone wyjątki w postaci przechwytywania zapytań http i zwracania odpowiednich kodów błędów. Konfiguracja kontenera *inversion of control* tym razem odbywa się w oparciu o mechanizmy wbudowane w ASP.NET Core, bez biblioteki Autofac. Nie zabrakło tu również interfejsu graficznego, wygenerowanego biblioteką Swagger, który jest taki sam, jak w przypadku projektu opartego o *Domain Driven Design*, jako że oba projekty udostępniają ten sam zestaw funkcjonalności.

7. Porównanie implementacji

Niniejszy rozdział porównuje implementacje projektu systemu do obsługi pizzerii. Z jednej strony mamy do czynienia z rozwiązaniem opartym o *Domain Driven Design* i CQRS, z drugiej zaś z systemem o architekturze tradycyjnej, wielowarstwowej, z anemicznym modelem danych. Porównanie bierze pod uwagę następujące parametry:

- Jakość rozwiązania, czyli odporność na błędy programisty;
- Czytelność kodu źródłowego;
- Złożoność rozszerzania o kolejne funkcjonalności;
- Złożoność wprowadzania zmian w istniejącej architekturze;
- Możliwość rozszerzenia do architektury rozproszonej;
- Ryzyko związane z zachowaniem spójności danych.

7.1. Jakość

Model domenowy w *Domain Driven Design* jest „bogaty", co oznacza, że klasy odzwierciedlające obiekty biznesowe, służą nie tylko do przechowywania danych, ale również do przeprowadzania na nich operacji. Odpowiedzialnością warstwy domeny w takich projektach jest również utrzymywanie poprawności danych w ramach tzw. niezmienników agregatów. Najważniejszą cechą *Domain Driven Design* jest to, że systemy modelowane są tak, by spełniały wymagania biznesowe i opisywały je językiem wspólnym dla osób biznesowych i technicznych, niezależnym od technologii, czy struktury bazy danych oraz innych elementów infrastruktury. Klasy cechują się restrykcyjną enkapsulacją, która uniemożliwia wprowadzenia ich w niepoprawny stan. Walidacja właściwości jest wbudowana w każdą encję i *value object*. Minimalizuje to ryzyko wprowadzenia błędu przez programistę, ponieważ w wypadku wystąpienia nieprawidłowości, system zareaguje wyjątkiem bardzo szybko – na długo wcześniej niż dane trafią do bazy. Na część błędów nie pozwoli już kompilator, dzięki zastosowaniu statycznie typowanych pól typu *value object* zamiast typów wbudowanych, takich jak `string`.

Model anemiczny, w przeciwieństwie do „bogatego", pozbawiony jest enkapsulacji i wszelkiej logiki, operującej na danych. Ryzyko popełnienia błędów podczas korzystania z takich obiektów jest duże, w szczególności, gdy stosuje się typy proste do opisu wszystkich właściwości. Walidacja odbywa się na zewnątrz, w warstwie serwisów, co bez skrupulatnego pokrycia kodu testami jednostkowymi może prowadzić do wielu uchybień, których odkrycie może nastąpić już na środowisku produkcyjnym, co z kolei często wymusza naprawę uszkodzonych danych w bazie. Problem ten można częściowo rozwiązać za pomocą wbudowanych atrybutów walidacyjnych stosowanych do właściwości obiektów, co jest dość powszechną praktyką

Elementem *Domain Driven Design*, który nie występuje w projektach o anemicznym modelu danych są zdarzenia domenowe i integracyjne. Zdarzenia domenowe opisują w sposób jawny wystąpienia istotnych wydarzeń w obrębie jednego agregatu. Ponieważ subskrybentów zdarzenia może być nieskończenie wiele, nie dochodzi do niekontrolowanego rozrostu warstwy aplikacji, do której należałoby w przeciwnym wypadku dokładać linie kodu za każdym razem, gdy dana operacja biznesowa wzbudza reakcję innych serwisów.

W projektach opartych o anemiczny model danych, serwisy warstwy aplikacji są zazwyczaj bardzo rozbudowane, co wynika z potrzeby zrównoważenia braku logiki zawartej w samych klasach modelu

danych. Sprzyja to duplikacji kodu, ponieważ logikę subskrybentów zdarzeń należy rozłożyć wśród metod serwisów, które dokonują podobnych zmian w danych.

Kwestia zdarzeń integracyjnych działa na niekorzyść *Domain Driven Design*, ponieważ ich zastosowanie wynika z decentralizacji modelu na niezależne konteksty, a nie z przesłanek jakościowych. Projekty z modelem anemicznym, ze względu na monolityczną strukturę danych, są wolne od dodatkowego utrudnienia związanego z koniecznością zachowania spójności.

Podsumowując, jakość modelu zgodnego z *Domain Driven Design* jest wyższa niż odpowiednika anemicznego, który przypomina bardziej opis schematu bazy danych, niż klasy języka zorientowanego obiektowo. Rzutuje to na dodatkowe skomplikowanie warstwy aplikacji, która musi rekompensować brak implementacji logiki biznesowej w klasach modelu danych. Ryzyko popełnienia błędu przez programistę w projektach opartych o *Domain Driven Design*, jest nieporównywalnie mniejsze, ze względu na restrykcyjne podejście do enkapsulacji i stosowania odpowiednich *value objects* zamiast typów prostych.

7.2. Czytelność kodu

Zwiększoną czytelność kodu pisanego w zgodności z *Domain Driven Design* gwarantuje wykorzystanie języka wszechobecnego. Nacisk na wykorzystanie terminów biznesowych zamiast żargonu technicznego sprawia, że programiści, którzy dopiero dołączyli do zespołu, lub wrócili po długiej przerwie łatwiej odnajdują sens linii kodu, z którymi przychodzi im obcować. Osoby zajmujące się biznesową stroną projektu nie czytają kodu źródłowego, zatem nie należy ulegać myśleniu, że taka forma jest ukłonem w ich stronę. Istotnie, wykorzystywanie języka wszechobecnego pomaga jednak w rozmowie z biznesem i dochodzeniu do wspólnego poziomu niezbędnej wiedzy. Aby zobrazować wpływ *Domain Driven Design* na czytelność kodu, wystarczy posłużyć się prostym przykładem.

Kod 20 – Metoda wykonująca komendę dodania produktu do koszyka (DDD). Źródło: opracowanie własne

```
public async Task<BasketDTO> Handle(AddItemToBasketCommand request,
    CancellationToken cancellationToken)
{
    CustomerBasket customerBasket;
    if (request.BasketId == null)
    {
        customerBasket = new CustomerBasket();
        await _basketRepository.AddAsync(customerBasket);
    }
    else
    {
        customerBasket =
            await _basketRepository
                .GetByIdAsync(request.BasketId.Value);
    }

    customerBasket
        .AddItemToBasket(
            request.ProductId,
            request.Quantity,
            request.UnitPrice);

    await _basketRepository.UnitOfWork.SaveEntitiesAsync();

    return _mapper.Map<BasketDTO>(customerBasket);
}
```

Kod 20 przedstawia metodę, która wykonuje komendę dodania produktu do koszyka w projekcie opartym o *Domain Driven Design*. Logika związana z samym koszykiem została zaznaczona kolorem czerwonym. Dla porównania, kod 21 zawiera analogiczną metodę w systemie z anemicznym modelem danych, w którym kolorem czerwonym zaznaczono fragment realizujący tę samą logikę biznesową. Różnica polega na tym, że w kodzie 20, dodanie produktu do koszyka jest jedną z funkcji obiektu koszyka, a więc osoba analizująca kod klasy koszyka od razu widzi wszystkie możliwe do wykonania na niej operacje. W kodzie 21, dodawanie produktu do koszyka znajduje się w klasie serwisu, a więc jest odseparowane od klasy, na której operuje. Takich fragmentów może być w projekcie bardzo wiele, w różnych, niepowiązanych ze sobą miejscach.

Kod 21 – Metoda dodająca produkt do koszyka (model anemiczny). Źródło: opracowanie własne

```
public async Task<CustomerBasket> AddItemToBasketAsync(
    int? basketId,
    int productId,
    int quantity)
{
    CustomerBasket customerBasket;
    if (basketId == null)
    {
        customerBasket = new CustomerBasket();
        await _basketRepository.AddAsync(customerBasket);
    }
    else
    {
        customerBasket = await _basketRepository
            .GetByIdAsync(basketId.Value);
    }

    var basketItem = new BasketItem
    {
        ProductId = productId,
        Quantity = quantity,
        Basket = customerBasket
    };

    var productExistsInBasket = customerBasket.Items
        .Any(bi => bi.ProductId == productId);

    if (productExistsInBasket)
    {
        var existingBasketItem = customerBasket
            .Items
            .First(item => item.ProductId == productId);
        existingBasketItem.Quantity += quantity;
    }
    else
    {
        customerBasket.Items.Add(basketItem);
    }

    return customerBasket;
}
```

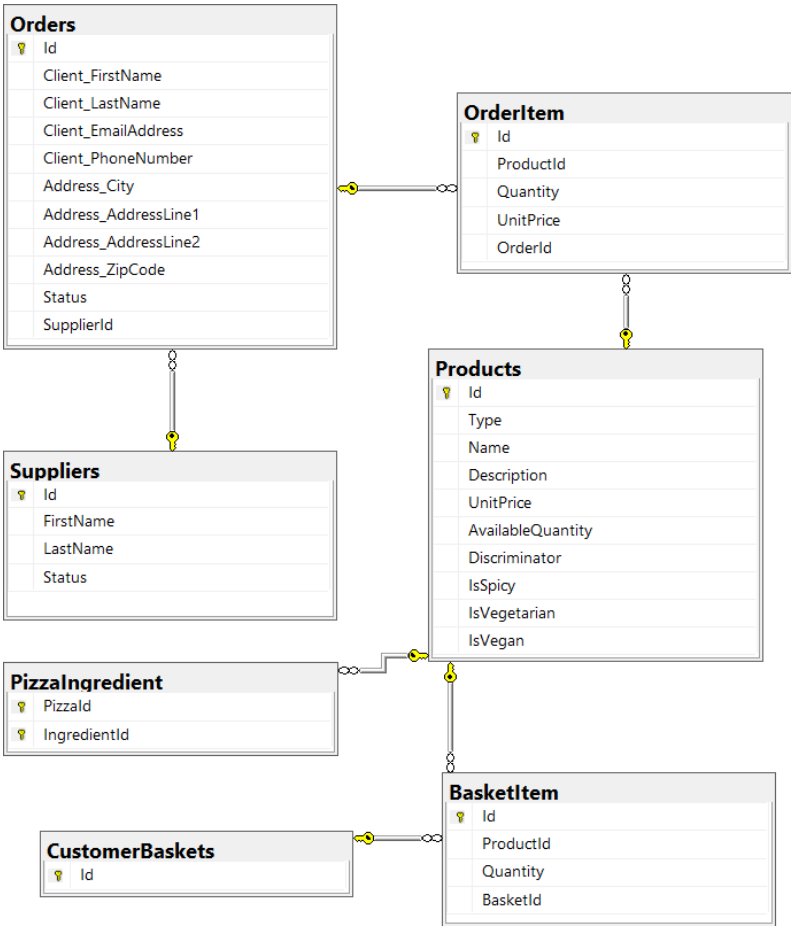
Osobną kwestią związaną z czytelnością kodu jest rozmieszczenie poszczególnych plików w solucji. Jak wiadomo, kod w projektach opartych o *Domain Driven Design* podzielony jest na

konteksty (2.3). Dzięki temu poszukiwanie pliku, który należy zmodyfikować, aby wprowadzić pożądaną przez biznes zmianę ogranicza się do wąskiego zbioru klas należących do kontekstu zajmującego się danym obszarem biznesowym. Wadą podzielenia na konteksty jest zagadnienie ich integracji, na przykład za pomocą zdarzeń integracyjnych (2.9). Operacje, które propagują zmiany na wiele kontekstów są trudniejsze do analizy, niż sekwencyjny ciąg wywołań odpowiednich serwisów w projektach z modelem anemicznym. Warto również nadmienić, iż konfiguracja obsługi zdarzeń integracyjnych wprowadza istotny poziom komplikacji, który nie jest potrzebny, gdy model jest monolitem.

Warto również podkreślić pozytywny wpływ *screaming architecture* (str. 34) na czytelność projektów w solucji, aczkolwiek jego wykorzystanie nie jest zarezerwowane dla konkretnego rodzaju architektury.

Nieodłącznym elementem systemów przetwarzających dane, jest baza danych, której schemat również można analizować pod kątem jakości, czy czytelności. Podział bazy na niezależne schematy odpowiadające kontekstom biznesowym być może ułatwia zrozumienie struktury tabel w obrębie danego obszaru, ale spojrzenie całościowe staje się nieczytelne z uwagi na brak relacji miedzy niektórymi tabelami, co widać na rysunku 18, gdzie poszczególne konteksty otoczono wielokątami. Kwestia spójności danych jest jednym z najpoważniejszych problemów *Domain Driven Design*. Z punktu widzenia bazy, nic nie stoi na przeszkodzie, aby doprowadzić do jego braku. Dodatkowo, duplikacja kolumn między schematami utrudnia stwierdzenie, w którym miejscu znajduje się źródło prawdy. Oddelegowanie części odpowiedzialności bazy danych do aplikacji zwiększa ryzyko uszkodzenia danych w systemie. Problem ten nie występuje w projektach o zintegrowanym modelu anemicznym, w których schemat bazy danych jest znormalizowany i zachowuje wszystkie relacje i ograniczenia, co pokazano na rysunku 19. Diagram przedstawiony na rysunku 19 obrazuje również poziom skomplikowania i jakość schematu samej bazy, który w projekcie o modelu anemicznym jest po prostu lepszy.

Rysunek 18 – Diagram bazy danych w projekcie DDD

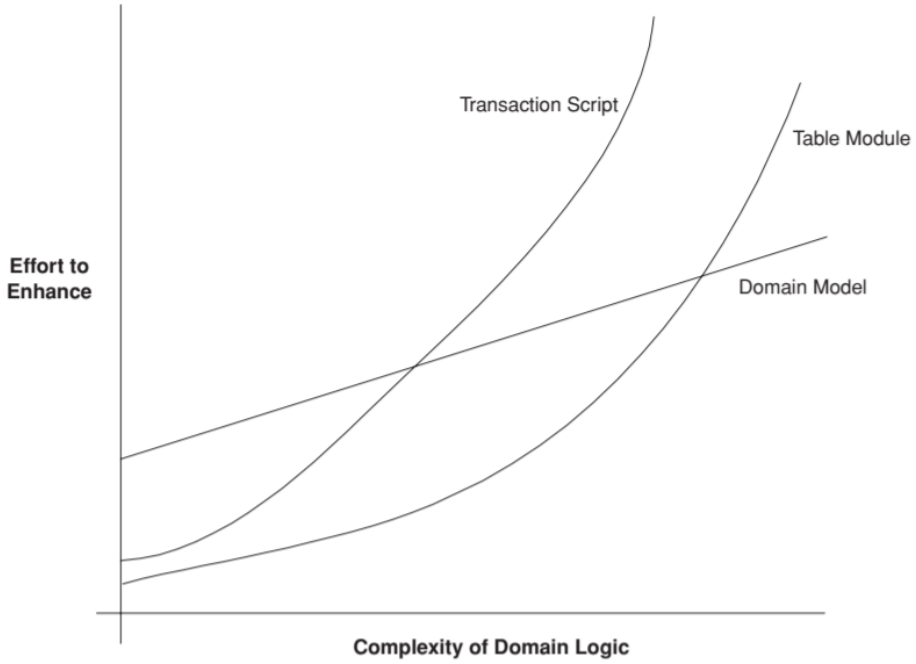


Rysunek 19 – Diagram bazy projektu o modelu anemicznym

7.3. Złożoność rozszerzania o kolejne funkcjonalności

Skoro architektura oparta o model anemiczny jest mniej skomplikowana, można by odnieść wrażenie, że jej rozszerzanie o kolejne elementy jest łatwiejsze niż w przypadku *Domain Driven Design*. Problemem jest jednak rosnąca złożoność zintegrowanego, monolitycznego modelu danych. Im więcej funkcjonalności zawiera taki system, tym więcej serwisów operuje na jednym, dużym schemacie danych, który również rośnie wprost proporcjonalnie do ilości funkcji projektu. Każdy nowo wprowadzany element musi być kompatybilny z całością dotychczas działającego systemu, a jak wiadomo, rozpatrywanie dużych projektów z perspektywy ich całości może być wyzwaniem dla umysłu programisty, czy architekta. *Domain Driven Design* rozwiązuje ten problem przez podział domeny na konteksty odpowiadające pojedynczym sektorom obsługiwanego biznesu. Jeżeli nowy obszar pasuje do jednego z kontekstów, wystarczy dopasować go do wąskiego zbioru obiektów wchodzących w skład owego modułu. Inną możliwością jest wprowadzenie zupełnie nowego kontekstu, co zupełnie eliminuje ryzyko związane z dopasowaniem do już istniejącego modelu danych. Problemem, który pojawia się zawsze podczas pracy z *Domain Driven Design* jest konieczność integracji kontekstów za pomocą mechanizmu zdarzeń, co w przypadku projektów o niewielkiej złożoności biznesowej, poddaje pod wątpliwość zasadność wykorzystania takiego wzorca. Wszystko zatem sprowadza się do dostosowania architektury systemu do potrzeb kreowanych przez dany biznes. Jeżeli dziedzina jest skomplikowana, a architektura zbyt prosta, jej implementacja może zostać doprowadzona do stanu, w którym wszelkie rozszerzenia pociągają za sobą konieczność refaktoryzacji dużej części już istniejącego systemu.

Oznacza to wysokie koszty i ryzyko uszkodzenia działających elementów, czyli dokładne przeciwieństwo tego, co architekt chciał osiągnąć stosując zbyt proste wzorce. Oczywiście identyczna zależność działa w drugą stronę, gdy wyrafinowana architektura spowalnia wzbogacanie systemu, który modeluje zbyt ubogą dziedzinę biznesu. Ważne jest jednak, aby przewidzieć kierunek rozwoju projektu, gdyż początkowo niezbyt skomplikowane wymagania mogą z czasem przybrać formę na tyle złożoną, że czas zainwestowany w implementację *Domain Driven Design* zwróci się w przyszłości.



Rysunek 20 – Porównanie złożoności rozszerzania różnych rodzajów logiki domenowej. Źródło: [27]

Rysunek 20 pochodzi z książki Martina Fowlera „Patterns of Enterprise Application Architecture" [27], w której pisze, że modelowanie zorientowane na domenę, z początku wydaje się nieatrakcyjne, ponieważ dodatkowy czas poświęcony na jego implementację może się nie zwrócić. Inne podejścia osiągają jednak pewien punkt graniczny, powyżej którego złożoność ich utrzymywania i rozszerzania zaczyna rosnąć wykładniczo. Zadaniem architekta jest stwierdzenie, w którym miejscu na poziomej osi tego wykresu znajduje się jego aplikacja i gdzie może znaleźć się w przyszłości [27].

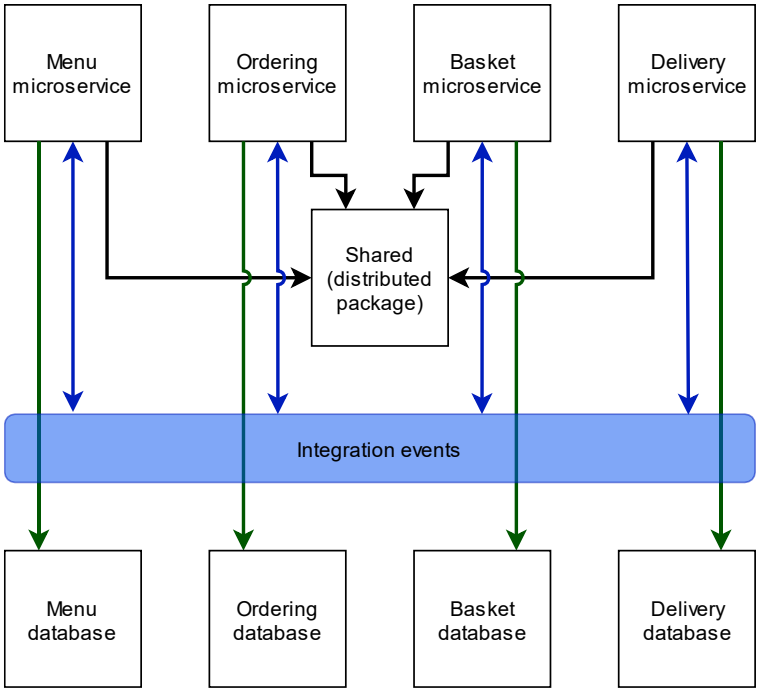
7.4. Złożoność wprowadzania zmian w istniejącej architekturze

Prawidłowości, które rządzą modyfikacjami istniejących projektów są bardzo podobne do zasad panujących podczas ich rozszerzania o nowe funkcjonalności. Należy przy tym pamiętać, że struktura projektów o modelu jednolitym jest z reguły bardziej hermetyczna niż w przypadku modelu podzielonego na konteksty. Pod warunkiem, że nowa funkcjonalność nie wymaga modyfikacji więcej niż jednego kontekstu, pozostałe moduły nie ulegają ryzyku wprowadzenia nowych błędów. Oprócz tego, zintegrowany model danych może dosyć łatwo paść ofiarą efektu domina polegającego na tym, że zmiana w jednym z serwisów wymaga dostosowania pozostałych, co w rezultacie może doprowadzić do konieczności refaktoryzacji całego systemu. Rysunek 18 przedstawiający schemat danych w projekcie opartym o *Domain Driven Design* obrazuje przewagę takich systemów w postaci wzajemnego odizolowania domen biznesowych na poziomie danych, co pozwala na bezpieczne

modyfikowanie danej domeny bez wpływu na pozostałe, w myśl luźnego wiązania komponentów [21]. Oczywiście jest to okupione opisywanymi wcześniej wadami, takimi jak potencjalne problemy z utrzymaniem spójności danych czy też kwestie związane z integracją kontekstów i ogólnym obniżeniem jakości schematu bazy danych, jeżeli rozpatrywana jest ona w całości. W porównaniu *Domain Driven Design* i projektów opartych o model anemiczny dominuje zasada dostosowania wzorców projektowych do poziomu skomplikowania modelowanego biznesu. Jeżeli dziedzina biznesowa jest bardzo prosta, to refaktoryzacja kilku interfejsów jest zadaniem dalece mniej złożonym od analizy powiazań między agregatami i kontekstami w projekcie opartym o *Domain Driven Design*. Szala zwycięstwa zostaje przechylona na stronę bardziej złożonych architektur w pewnym granicznym momencie przedstawionym symbolicznie na Rysunek 20. Znalezienie takiego momentu i podjęcie prawidłowej decyzji to esencja optymalizacji kosztów utrzymania oprogramowania.

7.5. Możliwość rozszerzenia do architektury rozproszonej

Podział systemu opartego na *Domain Driven Design* na konteksty sprzyja możliwości jego konwersji do architektury mikroserwisowej. Jedyne elementy, które spajają ze sobą moduły projektu w monolit, to aplikacja API, baza danych i klasy do serializacji zdarzeń integracyjnych. Z punktu widzenia bazy danych, wystarczy przenieść tabele należące do poszczególnych kontekstów do osobnych baz. Aplikacja typu API zawiera konfigurację, którą należałoby powtórzyć w każdym mikroserwisie, ale same kontrolery i klasy reprezentujące obiekty DTO pogrupowane są według kontekstów, co oznacza, że ich odseparowanie nie stanowi dużego problemu. Zdarzenia integracyjne wystarczy po prostu rozdysponować między serwisami, do których należą. Rysunek 21 przedstawia wysokopoziomowy schemat ekosystemu mikroserwisów, po pozbyciu się wyżej wymienionych wspólnych elementów. Projekt *Shared* został tutaj przedstawiony jako pakiet rozdystrybuowany, co w terminologii technologii .NET oznacza pakiet Nuget.



Rysunek 21 – Wysokopoziomowa architektura systemu DDD po konwersji do mikroserwisów. Źródło: opracowanie własne

Potrzeba konwersji do architektury rozporoszonej może być podyktowana zwiększonym obciążeniem systemu z powodu rosnącej liczby użytkowników, lub na przykład rozmieszczeniem geograficznym klientów. Zalety i wady takiej architektury to temat na osobną pracę dyplomową, więc pozostając przy tematyce porównania *Domain Driven Design* do modeli anemicznych, należy podkreślić, że tradycyjna architektura wielowarstwowa z takim właśnie modelem nie jest w żaden sposób przystosowana do konwersji do systemu rozproszonego. Podział na konteksty w projektach opartych o *Domain Driven Design* jest niejako fazą wstępną do podziału na mikroserwisy według tych samych granic biznesowych. Reguły integracji mikroserwisów za pomocą zdarzeń integracyjnych nie ulegają zmianie. Dokonując wyboru architektury, nawet jeżeli projekt w początkowej fazie będzie bardzo mały i niezbyt skomplikowany, trzeba brać pod uwagę jego przyszłość i potrzeby ewentualnych rozszerzeń, które mogą okazać się warte inwestycji w *Domain Driven Design* na samym początku prac.

7.6. Ryzyko związane z zachowaniem spójności danych

W przypadku projektów o zintegrowanym modelu danych, zachowanie spójności jest wbudowane w mechanizm dostępu do danych. Baza danych w takich systemach jest znormalizowana i nie ma potrzeby replikacji danych pomiędzy schematami. *Domain Driven Design* wprowadza jednak podział na konteksty, który obejmuje również magazyn danych. W związku z tym odpowiedzialność za zachowanie spójności między kontekstami spada na programistę, który musi ręcznie zaimplementować proces replikacji i ewentualnego cofania skutków ubocznych, gdy dojdzie do błędu. Jest to poważna wada *Domain Driven Design,* która istotnie zwiększa ryzyko uszkodzenia danych. Dlatego przed podjęciem decyzji o adaptacji takiej architektury, bardzo ważną kwestią jest upewnienie się, że każdy członek zespołu deweloperskiego rozumie to zagrożenie. W przykładowej aplikacji do obsługi pizzerii, aby zminimalizować ryzyko, zastosowano synchroniczną obsługę zdarzeń integracyjnych oraz transakcje obejmujące wszystkie konteksty na raz. Chroni to przed błędami podczas zapisu danych, ale oczywiście nie zatrzyma błędów logicznych w ustawianiu wartości kolumn. Rozdział 7.5 mówi o *Domain Driven Design* jako o punkcie przejściowym na drodze do mikroserwisów. Poziom ryzyka związanego z zachowaniem spójności danych wpisuje się w tę ideę, ponieważ wraz z rozproszeniem systemu znika możliwość zastosowania transakcji obejmującej cały system, a zdarzenia trzeba obsługiwać asynchronicznie.

8. Podsumowanie

Tabela 1 zawiera skonsolidowane porównanie obu rodzajów architektury z podziałem na zalety i wady każdej z nich.

Tabela 1 – Zalety i wady DDD i modeli anemicznych. Źródło: opracowanie własne

***Domain Driven Design* – zalety**	***Domain Driven Design* – wady**
• Wysokiej jakości, odporny na błędy walidacji i reguł biznesowych model danych; • Brak duplikacji walidacji dzięki zastosowaniu *value objects;* • Model danych jest wyrażony językiem wszechobecnym; • Niska złożoność i czas potrzebny na rozszerzanie projektu w rozbudowanych biznesowo systemach; • Możliwość łatwej konwersji do architektury rozporoszonej.	• Skomplikowany, zdenormalizowany schemat bazy danych; • Ryzyko utraty spójności danych; • Wysoki koszt rozpoczęcia prac nad nowym projektem; • Zbyt wysoki poziom skomplikowania przy prostych projektach; • Dość skomplikowana implementacja integracji między kontekstami.
Model anemiczny – zalety	**Model anemiczny – wady**
• Niski koszt i złożoność projektów o niezbyt rozbudowanych wymaganiach biznesowych; • Znormalizowany, spójny schemat bazy danych; • Brak ryzyka utraty spójności danych; • Brak konieczności integracji kontekstów.	• Niska jakość modelu danych, w niektórych sytuacjach podatność na błędy walidacji (przy braku alternatywnych rozwiązań, jak atrybuty walidacyjne) i reguł biznesowych; • Model danych odzwierciedla strukturę bazy danych, zamiast wymagań biznesowych; • Brak zabezpieczenia przed duplikacją walidacji (aczkolwiek można ją złagodzić za pomocą atrybutów walidacyjnych); • Zmniejszona czytelność warstwy aplikacji, która musi przejąć logikę warstwy domeny; • Wykładnicza relacja złożoności rozszerzania projektu względem wielkości projektu; • Utrudniona konwersja do architektury rozproszonej.

Porównanie nie wskazuje jednoznacznie zwycięstwa jednej ze stron. Zarówno *Domain Driven Design*, jak i model anemiczny sprawdzają się w konkretnych sytuacjach. Punktem przełomowym, w którym *Domain Driven Design* zaczyna być opcją bardziej opłacalną, jest moment wskazany przez Martina Fowlera, w którym poziom złożoności, czy też wielkości modelowanego systemu zaczyna

czerpać korzyści z wprowadzenia dodatkowych elementów porządkujących projekt [27]. Systemy oparte o model anemiczny sprawdzają się lepiej, gdy wymagania biznesowe są na tyle proste, że dużo szybciej można dostarczyć produkt bez poświęcania dodatkowego czasu na modelowanie agregatów i kontekstów. Sięgając w przeszłość, jestem wstanie odnaleźć liczne przykłady projektów, w których brałem udział, które były tworzone w oparciu o model anemiczny, z uwagi na brak znajomości innych wzorców przez wszystkich członków zespołu, w tym oczywiście mnie. Wspólną cechą tych projektów był moment, w którym stwierdzaliśmy, że większy sens ma rozpoczęcie pracy od początku, niż kontynuacja komplikacji, którą nieświadomie wprowadziliśmy. Teraz wiem, że był to objaw przekroczenia punktu granicznego, opisanego przez Martina Fowlera [27]. Wprowadzenie *Domain Driven Design* do tych projektów rzuciłoby całkowicie nowe światło na możliwość dalszego ich rozwoju, bez pogrążania się w chaosie. Podczas tworzenia przykładowej aplikacji do obsługi pizzerii, trudno było oprzeć się wrażeniu, że powstaje ogromna ilość kodu i plików, które przekładają się na bardzo mało funkcjonalności. Oznacza to, że na obecnym etapie rozwoju, zastosowanie *Domain Driven Design* nie ma sensu, ponieważ uzyskanie wysokiej jakości i elastyczności okupione jest zbyt dużym nakładem czasu, który rujnuje uzyskany efekt z punktu widzenia biznesowego. Jeżeli można w pewien sposób uzasadnić wykorzystanie tego wzorca w takim projekcie, to należy powoływać się na jego ewentualne rozszerzanie w przyszłości, co przybliżyłoby osiągnięcie punktu granicznego według Martina Fowlera [27]. Niemniej jednak naukowa wartość dodana wniesiona przez ów projekt jest wysoka, ponieważ pozwoliła mi nabrać doświadczenia w pracy z *Domain Driven Design* i CQRS, które są cenionymi umiejętnościami w portfolio programisty, czy architekta oprogramowania.

Bibliografia

[1] *Domain Driven Design*, Eric Evans, wyd. Addison – Wesley, 2004. ISBN: 978-0-321-12521-7

[2] Microsoft: Domain Events: design and implementation (dostęp: 21.03.2020) https://docs.microsoft.com/pl-pl/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation

[3] NDepend (Phil Vuollet): Hexagonal Architecture: What Is It and How Does It Work? (dostęp: 28.03.2020) https://blog.ndepend.com/hexagonal-architecture

[4] Wikipedia: Hexagonal architecture (dostęp: 28.03.2020) https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)

[5] Jeffrey Palermo: The Onion Architecture: part 1 (dostęp: 28.03.2020) https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[6] Jeffrey Palermo: The Onion Architecture: part 3 (dostęp: 28.03.2020) https://jeffreypalermo.com/2008/07/the-onion-architecture-part-3/

[7] Robert Martin: The Clean Architecture (28.03.2020) https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[8] Martin Fowler: CQRS (dostęp: 28.03.2020) https://martinfowler.com/bliki/CQRS.html

[9] Microsoft: Command and Query Responsibility Segregation (CQRS) pattern (dostęp: 28.03.2020) https://docs.microsoft.com/pl-pl/azure/architecture/patterns/cqrs

[10] Vladimir Khorikov: Types of CQRS (dostęp: 28.03.2020) https://enterprisecraftsmanship.com/posts/types-of-cqrs/

[11] Microsoft: Patterns in Practice – The Unit Of Work Pattern And Persistence Ignorance (dostęp: 4.04.2020) https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/june/the-unit-of-work-pattern-and-persistence-ignorance

[12] Martin Fowler: Bounded Context (dostęp: 4.04.2020) https://martinfowler.com/bliki/BoundedContext.html

[13] DZone (Grzegorz Ziemoński): Onion Architecture Is Interesting (dostęp: 4.04.2020) https://dzone.com/articles/onion-architecture-is-interesting

[14] Alistair Cockburn: Hexagonal architecture (dostęp: 4.04.2020) https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture

[15] Microsoft (Dino Esposito): Cutting Edge: Documents, Databases and Eventual Consistency (dostęp: 4.04.2020) https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/august/cutting-edge-documents-databases-and-eventual-consistency

[16] Microsoft: Implementing event-based communication between microservices (integration events) (dostęp: 4.04.2020) https://docs.microsoft.com/en-gb/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications

[17] Medium (Arpit Jain): Primitive Obsession – A Code Smell that Hurts People the Most (dostęp: 4.04.2020)

https://medium.com/the-sixt-india-blog/primitive-obsession-code-smell-that-hurt-people-the-most-5cbdd70496e9
[18] Microsoft: Loading Related Data (dostęp: 4.04.2020) https://docs.microsoft.com/en-gb/ef/core/querying/related-data
[19] ThoughtWorks (Andrew Hamel-Law): Domain-Driven design needn't be hard. Here's how to start (dostęp: 4.04.2020) https://www.thoughtworks.com/insights/blog/domain-driven-design-neednt-be-hard-heres-how-start
[20] Microsoft: Backing Fields (dostęp: 4.04.2020) https://docs.microsoft.com/en-gb/ef/core/modeling/backing-field
[21] SearchNetworking (Margaret Rouse): loose coupling (dostęp: 4.04.2020) https://searchnetworking.techtarget.com/definition/loose-coupling
[22] Jimmy Bogard: Eventual consistency, CQRS and interaction design (dostęp: 4.04.2020) https://lostechies.com/jimmybogard/2012/06/26/eventual-consistency-cqrs-and-interaction-design/
[23] Microsoft: Relational vs. NoSQL data (dostęp: 4.04.2020) https://docs.microsoft.com/en-gb/dotnet/architecture/cloud-native/relational-vs-nosql-data
[24] REST API Tutorial: What is REST (dostęp: 12.04.2020) https://restfulapi.net/
[25] Robert Martin: Screaming Architecture (dostęp: 20.04.2020) https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html
[26] Microsoft: eShopOnContainers (dostęp: 25.04.2020) https://github.com/dotnet-architecture/eShopOnContainers
[27] *Patterns of Enterprise Application Architecture*, Martin Fowler, wyd. Addison – Wesley, 2003, ISBN: 0-321-12742-0

Spis rysunków

Spis kodów źródłowych

Spis tabel