Microsoft

Czysta architektura w .NET

Professional

Dino Esposito

Microsoft

Dino Esposito

Czysta architektura w .NET

Przekład: Marek Włodarz

APN Promise, Warszawa 2024

Czysta architektura w .NET

Authorized translation from the English language edition, entitled: Clean Architecture with .NET, ISBN: 978-0-13-820328-3, by Dino Esposito, published by Pearson Education, Inc, publishing as Microsoft Press, a Division of Microsoft Corporation.

Polish language edition published by APN PROMISE S.A.

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: Clean Architecture with .NET, ISBN: 978-0-13-820328-3, by Dino Esposito, opublikowanego przez Pearson Education, Inc, publikującego jako Microsoft Press, oddział Microsoft Corporation.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600
e-mail: wydawnictwo@promise.pl

ISBN: 978-83-7541-540-7 (druk), 978-83-7541-541-4 (ebook)

Przekład: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

Część III Typowe dylematy

Podziękowania

W miarę jak włosy rzedną i siwieją, powracają wspomnienia czasów, gdy to ja byłem tym najmłodszym na każdym spotkaniu lub konferencji. W ciągu 30 lat kariery byłem świadkiem wybuchu Windows jako systemu operacyjnego, powstania sieci web wraz z nieodłącznymi witrynami i aplikacjami, a później pojawieniu się technologii mobilnych i chmurowych.

Kilka razy udało mi się zauważyć, że moje wizje dotyczące rozwoju technologii programistycznych nie odbiegały zbytnio od tego, co rzeczywiście nastąpiło kilka lat później. Innym razem zaskakiwałem siebie sam, formułując osobiste projekty gdzieś w połowie drogi pomiędzy marzeniami a ambitnymi celami.

Najbardziej niewymowne ze wszystkich jest pragnienie podróżowania po świecie, przemawiania na międzynarodowych konferencjach bez presji na to, aby mówić o tym, co jest cool i trendy, ale tylko o tym, co rzeczywiście widziałem i sprawiłem – bez przebierania w słowach i bez filtrowania lub zastrzeżeń. Aby to osiągnąć, musiałem – wreszcie – codziennie pracować nad tworzeniem rzeczywistych aplikacji, które przyczyniły się do rozwoju biznesu jakiegoś rodzaju i uprościło życie jakimś odbiorcom.

Dzięki Crionet i KBMS Data Force stało się to rzeczywistością.

Po wielu latach mam pełnoetatowe zajęcie (CTO w Crionet), zespół ludzi, którzy w ciągu kilku lat rozwinęli się z juniorów w solidnych i niezawodnych profesjonalistów, oraz pragnienie dzielenia się z każdym przepisem na tworzenie oprogramowania – który nie jest ani tajny, ani magiczny.

Nie mam nic na sprzedaż, tylko opowieść. A ta książka jest dla tych, którzy chcą słuchać.

To książka dla Silvia i Francesco.

To książka dla Micheli.

To książka dla Giorgio i Gaetano.

Książka ta stała się możliwa dzięki Loretcie i Shourav, a jej ostateczny kształt zawdzięczamy Milan, Tracey, Danowi i Kate.

To moja najlepsza książka – do czasu następnej!

Wprowadzenie

Studia informatyczne ukończyłem latem 1990 roku. Wówczas w Europie nie było zbyt wielu miejsc do studiowania komputerów. Kurs magisterski nie odbywał się nawet na jego własnym wydziale informatyki, ale stanowił rozszerzenie bardziej klasycznego wydziału matematyki, fizyki i nauk przyrodniczych. Ludzie z dużym doświadczeniem komputerowym w latach 90 byli naprawdę fajni – bardzo poszukiwani, ale z niejasnymi ścieżkami kariery. Rozpocząłem jako programista dla Windows. Czasopisma komputerowe były bardzo popularne i co miesiąc wyczekiwane z niecierpliwością. Marzyłem o pisaniu do jednego z nich. Raz wygrałem tę szansę i spodobało mi się tak bardzo, że robię to nadal, 30 lat później.

Moja pasja dzielenia się wiedzą była tak intensywna, że pięć lat po rozpoczęciu pierwszej poważnej pracy jako programista właśnie to stało się moim podstawowym zajęciem. Przez ponad dwadzieścia lat zajmowałem się tylko pisaniem książek i artykułów, występowaniem na konferencjach, prowadzeniem kursów i okazjonalnymi konsultacjami. Do roku 2020 miałem bardzo ograniczony kontakt z produkcyjnym kodem i rutyną codziennego programowania. Jednak udało mi się pisać udane książki dla tych, którzy byli zaangażowani w rzeczywiste projekty.

Jednak w zakamarkach mojego umysłu stale pojawiały się drażliwe wątpliwości: czy jestem jedynie wykładowcą, czy także człowiekiem czynu? Czy byłbym w stanie zbudować prawdziwy system? Pandemia i inne zmiany życiowe doprowadziły mnie w końcu do znalezienia odpowiedzi na te pytania.

Stanąłem przed trudnym zadaniem zbudowania ogromnego i skomplikowanego systemu w ułamku pierwotnie zaplanowanego czasu, co gwałtownie przerwała pandemia. Nie było jak projektować, być zwinnym, testować ani planować – jedyną pewną rzeczą był termin. Uciekłem się do robienia tego – i pozwolenia na to kilku innym ludziom – czego właśnie uczyłem i co odkryłem podczas nauczania. Zadziałało. Ale nie tylko to. Po drodze zrozumiałem, że podejście, jakie przyjęliśmy przy budowie tego oprogramowania i powiązane z nim wzorce, ma swoją nazwę: czysta architektura. Ta książka to najlepsze, co wiem i czego się nauczyłem w trzy lata codziennego tworzenia oprogramowania po ponad dwóch dekadach uczenia się, nauczania innych i konsultacji.

W naszej firmie mamy wielu programistów, którzy dołączyli do nas jako nowicjusze i dorastali, używając i eksperymentując z zagadnieniami, które są treścią tej książki. U nas to zadziałało. Mam nadzieję, że sprawdzi się również u was!

Dla kogo jest ta książka

Odbiorcami tej książki są profesjonaliści oprogramowania, w tym architekci, czołowi deweloperzy oraz – a może szczególnie – programiści dowolnego rodzaju aplikacji .NET. Każdy, kto chce zostać architektem oprogramowania, powinien uznać tę książkę za pomocną i wartą jej ceny. A prawdziwi architekci to w większości urodzeni programiści. Jestem głęboko przekonany, że kluczem do doskonałego oprogramowania są świetni programiści, zaś świetni programiści wyrastają na dobrych nauczycielach, dobrych przykładach i – mam nadzieję – dobrych książkach i wykładach.

Czy ta książka jest tylko dla profesjonalistów .NET? Choć każdy rozdział ma posmak .NET, większość treści jest zrozumiała i powinna być przydatna dla każdego profesjonalisty oprogramowania.

Założenia

Przyjąłem założenie, że czytelnik ma przynajmniej podstawową wiedzę na temat tworzenia oprogramowania w .NET i koncepcji programowania obiektowego. Pomocne będą dobre podstawy w używaniu platformy .NET oraz znajomość pewnych technik dostępu do danych. Włożyliśmy wiele wysiłku, aby tę książkę się dobrze czytało. To nie jest książka o abstrakcyjnych koncepcjach projektowych ani też klasyczna książka o architekturze, pełna odsyłaczy lub dziwnych ciągów znaków odwołujących się do jakiegoś starego artykułu wymienionego w bibliografii na jej końcu. To książka o budowaniu systemów w latach dwudziestych XXI wieku i stawianiu czoła dylematom tego czasu, od front-endu do back-endu, po drodze przechodząc przez platformy chmurowe i problemy skalowalności.

Ta książka może nie być odpowiednia, jeśli...

Jeśli ktoś szuka podręcznika albo chce się dowiedzieć, jak używać konkretnego wzorca lub technologii, ta książka może nie być właściwym wyborem. Jej celem jest dzielenie się i przekazywanie wiedzy, aby czytelnik wiedział, co robić w dowolnym momencie. Albo przynajmniej wiedział, co kilku innych – Dino i jego zespół – zrobiło w analogicznej sytuacji.

Organizacja książki

Mówiąc w skrócie, nowoczesna architektura oprogramowania ma tylko jeden warunek wstępny: modularność. Bez względu na to, czy wybierzemy rozproszoną strukturę nakierowaną na usługi, wzorzec mikrousług czy kompaktową aplikację monolityczną, modularność jest kluczowa dla zbudowania i zarządzania bazą kodu i dla dalszego

ulepszania aplikacji zgodnie ze zmieniającymi się potrzebami biznesu. Bez modularności będziemy w stanie po prostu raz dostarczyć działający system, ale bardzo trudno będzie go rozbudowywać i aktualizować.

Część pierwsza tej książki, zatytułowana „Święty Graal modularności", kładzie podwaliny pod modułowość oprogramowania, śledząc historię architektury oprogramowaniai podsumowując istotę projektowania dziedzinowego (domain-driven design – DDD) – jednej z najbardziej pomocnych metodologii podziału dziedzin biznesowych, choć nie jest to bezwzględna konieczność każdego projektu.

Część druga, „Czyszczenie architektury", poświęcona jest pięciu warstwom, które w wizji tej książki tworzą „czystą" architekturę. Nie skupiam się nadmiernie na koncentrycznym konstruowaniu architektury spopularyzowanym przez mnóstwo książek i artykułów, ale na rzeczywistej wartości dostarczanej przez składowe warstwy: prezentacji, aplikacji, domeny, usług domeny i infrastruktury.

Na koniec część trzecia, „Typowe dylematy", skupia się na trzech często spotykanych dylematach: monolity czy mikrousługi, front-end po stronie klienta czy serwera oraz rola i waga długu technicznego.

Do pobrania: aplikacja referencyjna

Część II książki opisuje aplikację referencyjną Project Renoir, której kompletna baza kodu jest dostępna na GitHubie pod adresem:

https://github.com/Youbiquitous/project-renoir

Spakowana wersja kodu źródłowego jest również dostępna do pobrania z witryny *MicrosoftPressStore.com/NET/download.*

Uwaga Aplikacja referencyjna wymaga .NET 8 i jest aplikacją ASP.NET z front-endem opartym na Blazor. Wykorzystuje Entity Framework jako mechanizmu dostępu do danych i zakłada używanie bazy danych SQL Server (w dowolnej wersji).

Errata, aktualizacje i pomoc techniczna

Dołożyliśmy wszelkich starań, aby zapewnić dokładność informacji w tej książce i jej materiałach towarzyszących. Aktualizacje tej książki mogą być dostępne w formie erraty i listy poprawek pod adresem:

MicrosoftPressStore.com/NET/errata

W przypadku odkrycia błędu, który nie jest wymieniony na tej liście, prosimy o przesłanie nam informacji, korzystając z formularza na tej samej stronie.

Dodatkowe wsparcie i informacje są dostępne na stronie *MicrosoftPressStore.com/Support*.

Należy zwrócić uwagę, że wsparcie techniczne dla produktów programowych i sprzętowych firmy Microsoft nie jest oferowane pod powyższymi adresami. Pomoc dotyczącą oprogramowania lub sprzętu firmy Microsoft można uzyskać pod adresem *http://support.microsoft.com*.

CZĘŚĆ I

Święty Graal modularności

ROZDZIAŁ 1

W poszukiwaniu modułowej architektury oprogramowania

Celem inżynierii oprogramowania jest kontrolowanie złożoności, a nie jej tworzenie.

– Dr Pamela Zave, Princeton University

Oprogramowanie, jakie znamy dziś, w połowie trzeciej dekady XXI wieku, jest ubocznym produktem głębszych procesów uczenia się i transformacji, których początki są głęboko zakorzenione w historii logiki i matematyki. Od XVII wieku niektóre spośród największych umysłów świata skupiały się na zbudowaniu spójnego, logicznego systemu, który mógłby pozwolić na osiągnięcie czegoś, co moglibyśmy nazwać myśleniem mechanicznym. Dowód, że nie jest to tylko marzenie, pojawił się dopiero w latach trzydziestych XX wieku wraz z twierdzeniem o niezupełności, które sformułował Kurt Gödel. Na tych podstawach Alan Turing i John von Neumann zaczęli projektować fizyczne maszyny.

Żaden z nich jednak nie śnił nawet o czymkolwiek zbliżonym do dzisiejszego oprogramowania. Ich celem było zmechanizowanie ludzkiego sposobu rozumowania, co nadal wydaje się czymś prostym, ale i niebywale ambitnym. Wczesne „myślące" maszyny z lat pięćdziesiątych były żelaznymi monolitami zbudowanymi z zaworów, tłoków i kabli – okablowany sprzęt, nie więcej. John Von Neumann miał intuicję, że instrukcje lepiej byłoby oddzielić od sprzętu, aby ta sama maszyna mogła robić różne rzeczy, takie jak obliczenia i przetwarzanie tekstu. Popularna „architektura von Neumanna" ostatecznie odnosi się do posiadania zapamiętanego programu, którego instrukcje są pobierane jedna po drugiej i przetwarzane sekwencyjnie.

Oprogramowanie uzyskało własną tożsamość i godność dopiero pod koniec lat sześćdziesiątych XX wieku, mniej więcej wtedy, gdy rasa ludzka wylądowała na Księżycu. Pierwsze znane użycie terminu „inżynieria oprogramowania" nastąpiło w połowie

tej dekady. Nikt nigdy nie starał się „stworzyć" oprogramowania. Powstało raczej jako efekt uboczny – produkt odpadowy – bardziej ambitnych badań Oddzielenie sprzętu od oprogramowania było pierwszym krokiem modularyzacji, jaki kiedykolwiek pojawił się w informatyce.

W istocie wydaje się, że ludzie zawsze podchodzili do rozwiązywania problemów za pomocą kompleksowej sekwencji kroków, wykorzystujących odniesienia i połączenia pomiędzy ustalanymi stanami i zmienianymi w razie potrzeby, aby dojść do rozwiązania. Jak pokazuje przypadek *spaghetti code*, oprogramowanie nie stanowi wyjątku.

Uwaga Poszukiwania modularności rozpoczęły się wraz z powstaniem samego oprogramowania i szybko przeniosło się z poziomu kodu aplikacji na poziom architektury aplikacji.

Na początku były trzy poziomy

Najdawniejszy przykład architektury oprogramowania, która wykracza poza dziedzinę pojedynczego komputera, został zaproponowany w latach sześćdziesiątych wraz systemem IBM 360. Idea polegała na tym, że zdalna stacja robocza mogła wysłać do centralnego komputera żądanie wykonania pewnej nieinterakcyjnej operacji, nazywanej *wsadem* (*job*). Dopracowywany w kolejnych latach, model ten stał się powszechnie znany jako *architektura klient/serwer*, który to termin pojawił się w artykule „Separating Data from Function in a Distributed File System" napisanym przez informatyków z grupy Xerox PARC w 1978 roku. Koncepcja klient/serwer była kanonicznym sposobem budowania aplikacji biznesowej, gdy dostałem pierwszą pracę jako programista zaraz po ukończeniu studiów.

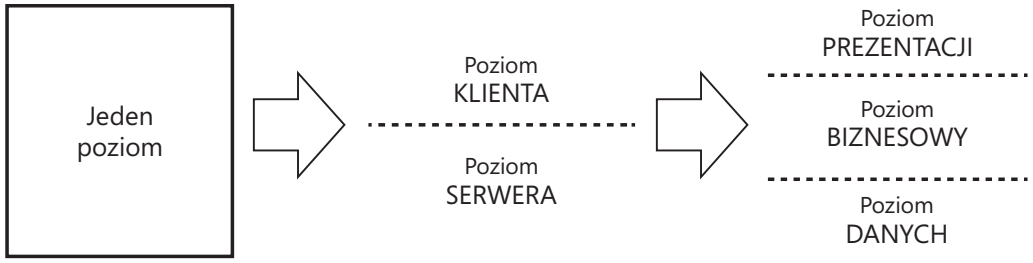
Uwaga To Von Neumann podzielił monolityczną koncepcję komputera na komponenty sprzętowe i programowe, zaś badaczom IBM i Xeroxa zawdzięczamy podział monolitu oprogramowania na komponenty klienta i serwera.

W latach dziewięćdziesiątych minionego wieku trzypoziomowa architektura była powszechnie akceptowana. W tym to czasie ujawniła się potrzeba zdefiniowania dodatkowego poziomu oprogramowania, który przejąłby część obowiązków klienta i pewne zadania serwera, aby móc sobie lepiej poradzić ze złożonością (nowych) aplikacji, które były budowane w naprawdę szybkim tempie.

Uwaga W poprzednim akapicie celowo umieściłem przymiotnik „nowych" w nawiasach, gdyż chodzi tu o aplikacje planowane i zbudowane przy użyciu architektury trójpoziomowej przed komercyjną eksplozją Internetu. W tamtym czasie ogromne aplikacje biznesowe (jako przykłady można wymienić oprogramowanie dla instytucji finansowych, firm telekomunikacyjnych, administracji państwowej, systemów ochrony zdrowia i tak dalej) nadal były bezpiecznie zakotwiczone w schemacie klient/serwer bazującym na komputerach mainframe. Nawet dziś mainframe'y wykonują ciężką pracę przetwarzania ogromnej liczby transakcji w czasie rzeczywistym, na przykład przy obsłudze kart kredytowych i bankomatów. Kluczowe uzasadnienie takich wyborów to wydajność i opłacalność, pomimo pojawienia się koncepcji chmury, przetwarzania brzegowego, technologii Blockchain i masowo rozproszonych systemów.

Podstawowe cechy systemu trzypoziomowego

Choć szkielet systemu trzypoziomowego powinien być doskonale znany każdej osobie działającej w dziedzinie oprogramowania, zdecydowałem się dołączyć ilustrujący go rysunek 1-1 jako przypomnienie stopniowego rozłupywania monolitycznych struktur w branży programowej, a przynajmniej do momentu komercyjnej eksplozji Internetu.



Rysunek 1-1 Pierwsze fazy ewolucji od pojedynczej warstwy do architektury wielopoziomowej

Pojawienie się systemów wielopoziomowych zostało powitane jako wielkie usprawnienie w porównaniu do monolitycznych aplikacji prehistorycznych. Obecnie jednak – w połowie lat dwudziestych XXI wieku – architektura wielopoziomowa jest często odrzucana bezmyślnie jako „przestarzała" i błędnie oznaczana jako „monolityczna".

Oprogramowanie monolityczne

Aktualna definicja *oprogramowania monolitycznego* różni się od tej z lat dziewięćdziesiątych ubiegłego wieku, kiedy oznaczało to „sekwencję instrukcji od początku do końca z pewną pętlą wejściową, która pozwoli utrzymać aplikację działającą i oczekującą na dalsze instrukcje". Obecnie za oprogramowanie monolityczne typowo uznawana jest aplikacja zbudowana z wielu komponentów połączonych razem w pojedynczy, autonomiczny produkt. Cała baza kodu znajduje się wewnątrz pojedynczego rozwiązania i jest wdrażana w jednym kroku na serwerze produkcyjnym, który może być

ulokowany w siedzibie firmy (użytkownika) albo w chmurze. Po wdrożeniu aplikacji wszystkie tworzące ją komponenty są niewidoczne z zewnątrz. Dowolny katastrofalny bug może *potencjalnie* wyłączyć całą aplikację, a dowolne potrzebne ulepszenia lub skalowanie muszą być stosowane do całego bloku oprogramowania. Może to prowadzić do znacznych modyfikacji kodu (lub wręcz pisania go od nowa) albo ograniczenie do czysto pionowej skalowalności opartej na udoskonaleniach sprzętu.

Warstwa czy poziom?

Poziomy zostały początkowo wprowadzone w celu odseparowania od siebie poszczególnych komponentów oprogramowania. W modelu klient/serwer zdalna stacja robocza była połączona kablem z centralnym komputerem (serwerem). Później najwyższym poziomem stał się element odpowiedzialny za logikę prezentacji, niezależnie od tego, czy wykorzystywane są dalekopisy, konsola znakowa, czy graficzny interfejs użytkownika (GUI). Poziom aplikacji był innym programem odpowiedzialnym za dostęp do serwera bazy danych i przetwarzanie logiki biznesowej.

We wspólnym branżowym żargonie pojęcia *poziom* (*tier*) oraz *warstwa* (*layer*) są często używane wymiennie. W rzeczywistości obydwa odnoszą się do różnych części aplikacji, ale różnią się znacząco z perspektywy wdrożeniowej. *Poziom* wskazuje oddzielny serwer fizyczny albo przynajmniej oddzielną przestrzeń wykonawczą procesu. *Warstwa* zaś, przeciwnie, jest kontenerem dla różnych części kodu i potrzebuje fizycznego poziomu dla wdrożenia.

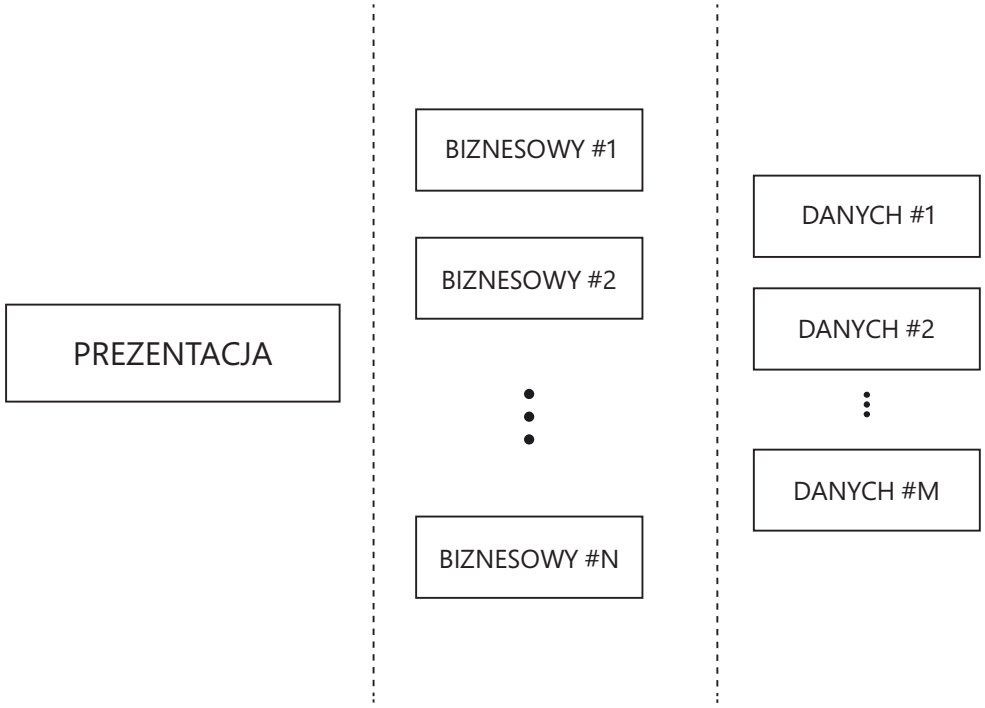
Uwaga Wszystkie warstwy są wdrażane na fizycznych poziomach, zaś różne warstwy mogą, ale nie muszą trafiać do różnych poziomów.

Kiedy zaczynamy zajmować się produkcyjnymi wdrożeniami, dyskusja skupia się na tym, czy wielowarstwowa aplikacja powinna zostać odwzorowana na wielopoziomową architekturę z dopasowaniem jeden do jednego pomiędzy warstwami i poziomami. Wiele poziomów wydaje się zapewniać większą elastyczność i ułatwia konserwację, ale odbywa się to kosztem wprowadzania opóźnień pomiędzy poziomy, a w konsekwencji każda pojedyncza operacja staje się potencjalnie wolniejsza. Dodatkowo wdrożenie w scenariuszu wielopoziomowym jest bardziej kosztowne, gdyż potrzebujemy więcej zasobów (w siedzibie lub w chmurze).

Podsumowując, poziomy mogą zapewnić strukturę dla skalowania aplikacji, ale sama ich obecność nie gwarantuje lepszej wydajności. Wydajne skalowanie obejmuje nie tylko uporządkowanie poziomów, ale również takie czynniki, jak równoważenie obciążeń, optymalizacja kodu i użycie odpowiednich technologii oddzielania pamięci masowych (na przykład magistrali). Poziomy pomagają w tym, zapewniając logiczną separację zagadnień, ale korzyści wydajnościowe osiąga się dopiero poprzez przemyślany projekt, odpowiednią alokację zasobów i staranne dostrajanie.

Wartość *N*

Termin *wielopoziomowy* (albo *wielowarstwowy*) odnosi się do pewnej liczby (*N*) poziomów (warstw), konwencjonalnie równej trzy. Czy zatem trzy jest idealną liczbą poziomów (warstw)? Aby znaleźć odpowiedź na to pytanie, przyjrzyjmy się rysunkowi 1-2.



Rysunek 1-2 Zmienna liczba poziomów w aplikacji o masowo rozproszonej architekturze

Poziomy i warstwy używają różnych skal. Popularne podejście bazujące na mikrousługach skłania do zwiększania liczby fizycznych poziomów, nawet do setek. I przeciwnie, liczba warstw w obrębie jednego poziomu rzadko przekracza cztery, co jest uważane za idealną wartość w kanonicznej architekturze opartej na metodologii projektowania dziedzinowego (*domain-driven design* – DDD). Jak niedługo się przekonamy, są to:

- **Warstwa prezentacji** Zbiera żądania i dane wejściowe od użytkowników w celu przekazania do warstw przetwarzających poniżej w stosie.
- **Warstwa aplikacji** Odbiera surowe wejście z warstwy prezentacji i orkiestruje dowolne niezbędne działania.
- **Warstwa domenowa** Zawiera nadającą się do ponownego użycia logikę biznesową.
- **Warstwa infrastruktury** Zajmuje się komunikacją z zewnętrznymi usługami (na przykład rozmaitymi API i usługami web) oraz magazynowaniem danych.

W porównaniu z architekturą trzypoziomową, architektura wielowarstwowa jest bardziej szczegółowa, gdyż dzieli na dwie części poziom biznesowy, który inaczej byłby grubszy (i zapewne bardziej splątany).

Dzisiejsze systemy wielopoziomowe

Architektura wielopoziomowa jest dobrze ugruntowanym wzorcem, który nadal obsługuje większość scenariuszy biznesowych. Przyjrzyjmy się odmianom wielopoziomowości, które możemy obserwować w dzisiejszej praktyce informatycznej.

Pierwszą odmianą jest aplikacja web. Na potrzeby tej książki termin *aplikacja web* oznacza oprogramowanie biznesowe (*line-of-business* – LoB) obsługiwane za pomocą przeglądarki po stronie klienta.

W działaniu typowa aplikacja web opiera się na dwóch poziomach: przeglądarce klienckiej oraz środowisku serwerowym (chmurowym), niekiedy określanego mianem zaplecza (*back-end*). Jak wiele poziomów i warstw istnieje wewnątrz zaplecza? Klasyczne aplikacje ASP.NET lub ASP.NET Core, a także aplikacje oparte na serwerze Blazor zwykle wyliczają dwa poziomy i wiele ($N > 3$) warstw. Jednym z tych poziomów jest zasadnicza aplikacja, a drugi reprezentuje podstawowy serwer bazodanowy – na przykład jakiś system relacyjnego zarządzania bazami danych (RDBMS).

Wyzwaniem jest zatem poznanie dobrych i złych stron każdego możliwego wzorca architektury aplikacji i dokonywanie przemyślanego wyboru na podstawie konkretnego kontekstu biznesowego, a nade wszystko uniknięcie sporów ideologicznych.

Uwaga Podczas badania pochodzenia architektury trzypoziomowej natrafiłem na zadziwiający fakt, o którym nigdy wcześniej nie słyszałem. W latach trzydziestych XX wieku, zaraz po uchyleniu prohibicji, rząd Stanów Zjednoczonych wprowadził nowy rozproszony system zapewniający dostęp ludności do alkoholu. I co się okazuje? Nazwano go *systemem trzypoziomowym*, a te poziomy (od dołu do góry) to producenci, dystrybutorzy i sprzedawcy detaliczni.

Warstwy, poziomy i modularność

Modularyzacja to główny powód wprowadzania wielu poziomów i warstw. Pracując w tej branży od przeszło 30 lat widziałem wiele prób opracowania uniwersalnego podejścia do komponentyzacji (a może należałoby powiedzieć „*Legolizacji*?) w tworzeniu oprogramowania, od komponentów pisanych w Visual Basic i Delphi, poprzez ActiveX i COM, od JavaBeans do serwerowych kontrolek Web Forms czy od usług web w starym stylu do mikrousług. Szczerze mówiąc, z wiekiem straciłem złudzenia i mogę stwierdzić, że żadne z tych podejść nie sprawdziło się dla więcej niż kilku osób/projektów albo jedynie przez ograniczony czas. Nie ma już we mnie nadziei, że gdzieś tu jest miejsce na uniwersalność.

Oprócz dobrze znanych korzyści, takich jak możliwość ponownego użycia komponentów, zrównoleglanie procesu wytwarzania i uproszczenie późniejszego utrzymywania kodu, ostatecznym celem i głównym zyskiem wynikającym z modularyzacji jest separacja zagadnień (*separation of concerns* – SoC) – uniwersalna zasada konstruowania

oprogramowania, sformalizowana w 1974 roku przez Edsgera W. Dijkstrę w artykule „On the Role of Scientific Thought".

Uwaga Choć wcześniej stwierdziłem istnienie zasadniczej różnicy pomiędzy poziomem a warstwą, od tego miejsca dla uproszczenia będę używać terminu *warstwa*, aby wskazać poziom lub warstwę, chyba że konieczne będzie rozróżnienie jednego pojęcia od drugiego.

Warstwa prezentacji

Każda z warstw omówionych skrótowo w tej części wykorzystuje wspólną koncepcję, ale różne implementacje. Na przykład w aplikacji pulpitu, takiej jak przestarzałe programy Windows Forms lub Windows Presentation Foundation (WPF), warstwą prezentacji jest interfejs użytkownika. Zawiera minimalną logikę prezentacji w celu walidacji danych wejściowych i dostosowywania interfejsu, aby odzwierciedlał bieżący stan aplikacji.

Dla kontrastu, w scenariuszu webowym interfejs użytkownika składa się z mieszanki kodu HTML, CSS i JavaScript, przetwarzanego w przeglądarce. Alternatywnie może być to wykonywalny fragment kodu WebAssembly, taki jak kod generowany przez Blazor. Jako że działa na fizycznie odrębnej maszynie, jest to prawdziwy poziom. Aplikacja może jednak zawierać także specjalną warstwę prezentacji, której głównym celem jest przekierowywanie żądań do pewnego modułu, który je obsługuje. W przypadku aplikacji ASP.NET Core warstwa prezentacji zawiera klasy kontrolera oraz w ogólności kod, który jest bezpośrednio połączony z osiągalnymi punktami końcowymi.

Warstwa biznesowa

W ujęciu abstrakcyjnym warstwa biznesowa przetwarza informacje zebrane w warstwie prezentacji w połączeniu z innymi informacjami zarządzanymi przez warstwę danych. Jest to również miejsce, w którym reguły biznesowe są znane i stosowane. W scenariuszu ASP.NET Core warstwę biznesową tworzą bloki obsługi odpowiadające na żądania kontrolerów i zwracające odpowiedzi do kontrolera w celu spakowania ich z powrotem do przeglądarki.

W ostatnich latach na szkoleniach i warsztatach często powraca pytanie o optymalne umieszczenie określonych segmentów kodu. Dla przykładu często wyłania się pytanie, jaka jest właściwa lokalizacja kodu walidującego dane wejściowe. Czy powinien on znajdować się w warstwie prezentacji, czy biznesowej? Alternatywnie, czy można rozważyć opóźnienie walidacji aż do momentu, gdy dane trafią do bazy danych, gdzie mogłaby ona być realizowana przez procedurę składowaną lub jakiś towarzyszący kod?

W podstawowym wariancie warstwa biznesowa nie rozstrzyga takich wątpliwości. To dlatego wyłoniła się architektura czterowarstwowa.

Warstwa danych

To tu informacje przetwarzane przez aplikację są utrwalane i odczytywane. Wielkość („grubość") tej warstwy jest bardzo zmienna. Może łączyć się ona z serwerem bazy danych – relacyjnym albo NoSQL – albo może być zbudowana przez kod generujący surowe wywołania do serwera magazynującego poprzez dedykowane odwzorowania obiektowo-relacyjne (*object-relational mappers* – ORM), takie jak Entity Framework (EF) albo Dapper.

W ostatnim czasie warstwa danych doczekała się nowej abstrakcji w formie warstwy infrastrukturalnej, dla której utrwalanie danych jest główną, ale nie jedyną odpowiedzialnością. Postrzegana jako infrastruktura, ta warstwa jest również odpowiedzialna za emaile i połączenia z zewnętrznymi API.

Ważne Podstawowym celem architektury wielowarstwowej jest osiągnięcie separacji zagadnień (SoC) i zagwarantowanie, że różne rodzaje zadań są wykonywane w odpowiednio odizolowanym środowisku. Główną konsekwencją SoC, szczególnie przy stosowaniu względem wielu warstw, jest to, że zależności pomiędzy warstwami muszą być ściśle regulowane. Omówienie tego, jak zaplanować separację zagadnień pomiędzy warstwy w kontekście aplikacji web napisanej dla stosu .NET, stanowi główny cel tej książki.

Kanoniczna architektura DDD

Na początku XXI wieku branża programistyczna stanęła w obliczu gigantycznego wyzwania: zmodernizowania, albo przynajmniej zmigrowania istniejących aplikacji biznesowych, aby wykorzystać nowe możliwości wynikające z gwałtownego rozwoju Internetu. Próby zaadaptowania aplikacji na mainframe'y w celu obsłużenia rosnącego popytu na e-commerce wprowadziły złożoności o monumentalnych rozmiarach.

Trzywarstwowa (czy też trzypoziomowa) architektura zaczęła pękać pod ciężarem tej złożoności – nie aż tak bardzo z powodu wrodzonej nieefektywności, ale raczej ze względu na potrzebę zwiększenia modularności w celu zarządzania wymaganiami biznesowymi i implementacyjnymi, a także (miejmy nadzieję) skalowalności. (To w tym czasie termin *skalowalność* zdobył popularność i uzyskał to znaczenie, które znamy dziś – zdolność systemu do zapewnienia dobrego poziomu obsługi nawet wtedy, gdy liczba żądań wzrośnie nieoczekiwanie).

Metodologia projektowania dziedzinowego (*domain-driven design* – DDD) usystematyzowała wiele praktyk i rozwiązań, które potwierdziły swoją skuteczność. Wraz z metodologią projektowania powstała również kanoniczna, wspierająca ją architektura.

Uwaga Nieprzypadkowo użyłem określenia „monumentalna złożoność". Jest to cytat z opowieści zasłyszanych od ludzi, którzy zdefiniowali DDD i hołd dla nich wszystkich.

Proponowana architektura wspierająca

Zwolennicy DDD zasugerowali użycie warstwowej architektury do implementowania modułów. Ta warstwowa architektura uogólniała trzypoziomową architekturę uzyskaną w trzech krokach:

- Poziomy zostały uogólnione do warstw.
- Warstwa biznesowa została podzielona na dwie: warstwę aplikacji do celów orkiestracji przypadków użycia oraz warstwę domenową dla czystej logiki biznesowej*. Z kolei warstwa domenowa składa się z dwóch elementów: zbioru niezależnych od utrwalania danych modeli domeny oraz zbioru usług uwzględniających utrwalanie danych. Jest to kluczowy aspekt DDD.
- Warstwa danych została przemianowana na warstwę infrastruktury, a najbardziej typową i najważniejszą usługą, jaką udostępnia, jest utrwalanie danych.

Wynikowa architektura – zawierająca również warstwę prezentacji – w swojej najprostszej formie jest monolityczna, ale ma jednoznaczne granice na każdym poziomie i dobrze zdefiniowane przepływy wymiany danych. Pod pewnymi względami jest nawet czystsza, niż wychwalana rzeczywista *czysta architektura*!

Warstwa prezentacji

Warstwa prezentacji obsługuje interakcje z zewnętrznymi systemami, które przesyłają do aplikacji dane wejściowe. Głównie mamy tu ludzkich użytkowników, ale także wywołania API z innych działających aplikacji, powiadomienia, komunikaty magistrali, wyzwalane zdarzenia i tak dalej. Ujmując to inaczej, warstwa prezentacji odbiera żądania wykonania zadań, które zostaną przetworzone w dalszej części stosu, produkując jakiś efekt w domenie. Warstwa prezentacji jest również odpowiedzialna za opakowanie wyników generowanych przez wewnętrzne warstwy w odpowiedzi na zaakceptowane żądanie i wysyłanie ich z powrotem do żądającego.

Warstwa prezentacji może przyjmować różne formy, jeśli chodzi o użyte technologie. Może być to aplikacja pulpitu (na przykład .NET MAUI, Electron, aplikacja WPF w starym stylu albo Windows Forms), aplikacja mobilna, minimalne API albo w pełni wyposażona aplikacja web (wykorzystująca ASP.NET Core, Blazor, Angular, React, Svelte i inne podobne platformy). Dodatkowo wykorzystywane protokoły mogą się nieco różnić i mogą obejmować HTTPS, gRPC, a w scenariuszach obejmujących Internet rzeczy (*Internet of Things* – IoT) rozwiązania Message Queue Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMPQ) i wiele więcej.

---

* Dążąc do możliwie jednoznacznego przekazu, terminu „dziedzina" używam tam, gdzie odnosi się do świata rzeczywistego (konkretnej branży, biznesu lub sposobu działania jakiejś organizacji), natomiast termin „domena" stosuję, gdy mowa o bycie programistycznym, odwzorowującym dziedzinę świata rzeczywistego (wszystkie przypisy pochodzą od tłumacza).

Mając to na uwadze, kluczowe jest zapamiętanie, że pomimo faktu, że nazwa warstwy (prezentacja) sugeruje obecność jakiegoś graficznego front-endu, wizualny interfejs wcale nie jest konieczny. Nawet jeśli powiązany kontekst ma być prostym API webowym, warstwa prezentacji nadal ma sens, gdyż, jak wspomniano, reprezentuje rodzaj recepcji i bramy do wewnętrznych funkcji i warstw.

Warstwa aplikacji

Podczas gdy warstwa prezentacji zbiera żądania, warstwa aplikacji koordynuje dowolne późniejsze przetwarzanie. Warstwa aplikacji jest w istocie tym miejscem, w którym biznesowe przepływy danych są uruchamiane i monitorowane. Dowolne pojedyncze żądanie obsłużone na wyższym poziomie znajduje konkretnego wykonawcę w warstwie aplikacji. Pojęciem, które dobrze opisuje zachowanie warstwy aplikacji, jest *orkiestrator*. Bardziej szczegółowy opis wykonywanego tu zadania jest taki, że warstwa aplikacji jest odpowiedzialna za implementacje różnych przypadków użycia.

Ważne W klasycznym trzypoziomowym scenariuszu opisane tu, dobrze zdefiniowane odpowiedzialności są współdzielone przez wszystkie poziomy – prezentacji, biznesowym i danych – w częściach, które zmieniają się w różnych implementacjach zależnie od poglądów i wrażliwości zaangażowanych zespołów.

Warstwa aplikacji działa ręka w rękę z warstwą prezentacji i dostarcza sposób działania dla każdego możliwego wyzwalacza wykrytego przez elementy prezentacji. Kiedy potrzebnych jest wiele wersji aplikacji (powiedzmy jedna dla przeglądarek web i druga dla urządzeń mobilnych), każda powinna mieć swoją własną warstwę aplikacji, chyba że wyzwalacze i oczekiwane reakcje są niemal takie same.

Warstwa domenowa

Możemy mieć wiele warstw aplikacji – po jednej dla każdej warstwy prezentacji. Jednak warstwa domenowa musi być jedyna i wspólna. Jest to kluczowa zasada DDD: warstwa domenowa to miejsce, w którym zakodowane są wszystkie reguły i logika biznesowa.

Warstwa domenowa składa się z dwóch powiązanych części:

- **Model domeny** Jest to prosta biblioteka klas zawierających definicje wszystkich jednostek biznesowych oraz dowolne obiekty wartości, agregacje, fabryki, wyliczenia i cokolwiek innego, co jest pomocne dla zapewnienia realistycznej reprezentacji modelu biznesowego. W wersji idealnej jest zakodowany jako oddzielna biblioteka klas. Klasy domenowe są pustymi i bezstanowymi maszynami obliczeniowymi, zawierającymi logikę przetwarzania odpowiadającą regułom biznesowym. Oto kilka ważnych aspektów modeli domeny:

- ❒ Za model domeny odpowiada jeden zespół.
- ❒ W środowisku .NET preferowane jest udostępnianie modelu domeny jako pakietu NuGet poprzez prywatny, specyficzny dla firmy lub zespołu kanał.
- ❒ Model domeny powinien mieć możliwie mało zależności względem innych pakietów lub projektów. Dowolne pojawiające się zależności (takie jak od pomocniczych pakietów innych firm) powinny być dogłębnie zbadane i włączane tylko wtedy, gdy jest to naprawdę konieczne.
- ❒ Model domeny nie ma żadnych odwołań do warstwy utrwalania (która jest częścią warstwy infrastruktury) i jest całkowicie niezależny od bazy danych.

- **Usługi domeny** Jak możemy wyszukiwać dane i ładować je do wspomnianych wyżej pustych i bezstanowych maszyn obliczeniowych? To właśnie robią usługi domeny. Są to klasy obsługujące bazę danych, które ładują informacje do klas jednostek domeny i wykonują aktualizacje danych wynikające z działania tych klas. Usługi domeny mają zależności od modelu domeny i warstwy infrastruktury (w szczególności od warstwy utrwalania).

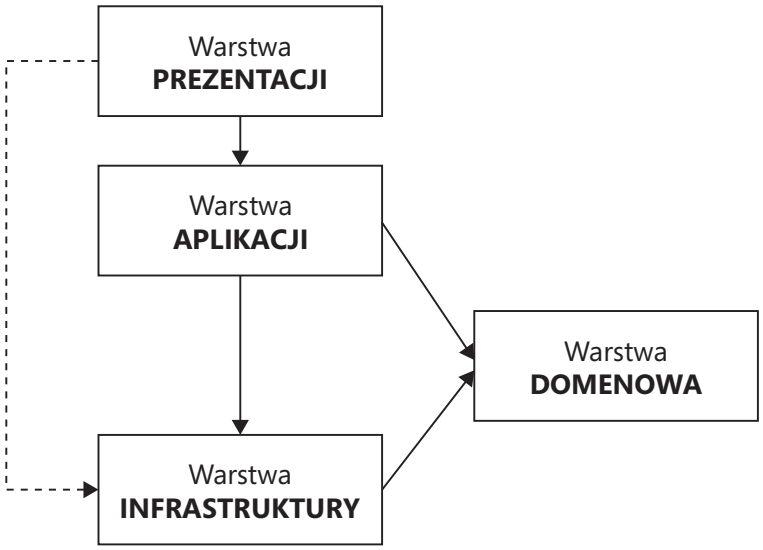
Warstwa infrastruktury

Warstwa infrastruktury jest kontenerem, w którym mieści się warstwa utrwalania, usługi zewnętrzne, takie jak email i systemy komunikacyjne, oraz łączniki do zewnętrznych API.

Podstawową usługą zapewnianą przez warstwę infrastruktury jest zazwyczaj dostęp do danych. Obejmuje on wszelkie możliwe kombinacje operacji odczytu i zapisu, jakich potrzebuje biznes. Najczęściej spotykany scenariusz to wymaganie pełnego utrwalania (odczyt/zapis). Możliwe są też inne scenariusze, takie jak tylko odczyt, gdzie dane są pobierane z istniejących usług zewnętrznych, lub tylko zapis, gdy warstwa infrastruktura jest używana jedynie do rejestrowania zdarzeń.

Komponenty odpowiedzialne za dostęp do bazy danych izolujemy poprzez użycie repozytoriów. Co interesujące, użycie repozytoriów nie łamie zasady SoC. Warstwa utrwalania, która jest częścią warstwy infrastruktury, jest miejscem przechowywania repozytoriów. Jeśli planujemy użyć jakiś interfejsów do tych repozytoriów (na przykład do celów testowych), te interfejsy można umieścić w bibliotece usług domeny. Jednak inne, prostsze podejście może polegać na umieszczeniu usług domeny i samych repozytoriów w warstwie utrwalania albo nawet na budowaniu bogatszych repozytoriów, które nie ograniczają się do metod CRUD, ale eksponują bardziej inteligentne metody. W takim przypadku jednak rozmywa się granica pomiędzy usługami domeny a warstwą utrwalania.

Rysunek 1-3 pokazuje zalecany sposób ustalania zależności pomiędzy warstwami w klasycznej architekturze DDD.



Rysunek 1-3 Powiązania pomiędzy warstwami w architekturze warstwowej DDD

Uwaga Celem tego rozdziału jest przedstawienie ogólnego obrazu architektury aplikacji i tego, jak branża informatyczna adaptowała modularność z upływem lat. To tylko zarys zagadnienia DDD i architektur warstwowych. Więcej na temat istoty DDD dowiemy się w rozdziale 2, „Prawdziwa istota DDD". W dalszej części tej książki szczegółowo omawiam warstwy architektury inspirowanej przez DDD, wraz z przykładami kodu.

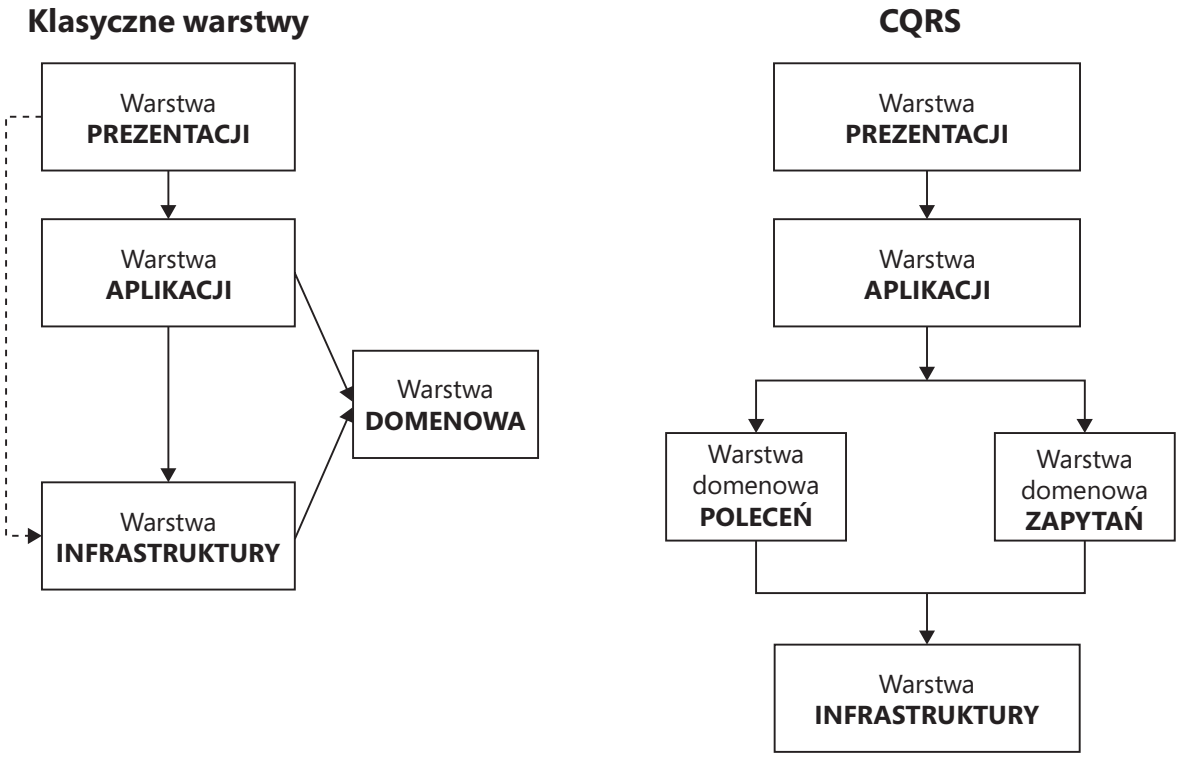
Dodatkowe składniki przepisu

Kanoniczna architektura DDD została opracowana jako pomocny punkt odniesienia. Jej stosowanie nigdy nie jest obowiązkowe. Początkowo zawierała silny akcent na orientację obiektową, ale ten aspekt również nigdy nie był wymagany. Obiektowa natura DDD ewoluowała z czasem i włączono do niej pewne możliwości funkcyjne. Ponownie wynikowy wielowarstwowy wzorzec jest tylko zaleceniem i prostsze rozwiązania (na przykład CMS, CRM, zgrupowane systemy CRUD i systemy dwuwarstwowe) są zawsze akceptowalne, o ile mogą spełnić wymagania. Ostatnio popularność zdobywają dwa dodatkowe smaki – *Command/Query Responsibility Segregation* – CQRS (rozdział odpowiedzialności za polecenia i zapytania) oraz *Event Sourcing* (pozyskiwanie zdarzeń). Obydwa można postrzegać jako dodatkowe składniki oryginalnego przepisu warstwowego.

Dodawanie CQRS dla smaku

CQRS jest po prostu wzorcem architektonicznym, który można zastosować do pewnego, konkretnego składnika potencjalnie większego systemu. Zastosowany do architektury warstwowej, CQRS dzieli warstwę domenową na dwie odrębne części. Ta separacja uzyskiwana jest poprzez zgrupowanie operacji zapytań w jednej warstwie,

a operacji poleceńw innej. Każda warstwa ma teraz swój własny model i własny zbiór usług dedykowanych odpowiednio tylko do zapytań poleceń Rysunek 1-4 pokazuje porównanie prostej warstwowej architektury (po lewej) z jej wersją opartą na CQRS (po prawej).



Rysunek 1-4 Wizualne porównanie warstw klasycznych i CQRS

W odróżnieniu od DDD, CQRS nie jest wyczerpującym podejściem do projektowania systemów klasy przedsiębiorstwa. Jak wspomniałem, jest to tylko wskazówka. Zalecanym krokiem wstępnym nadal jest analiza DDD oparta na języku używanym w biznesie w celu zidentyfikowania granic kontekstów (więcej informacji na ten temat zawiera rozdział 2). CQRS jest po prostu uprawnioną alternatywą dla implementacji konkretnego modułu całej aplikacji.

Każda operacja względem systemu programowego jest albo zapytaniem, które odczytuje pewien stan systemu, albo poleceniem zmieniającym istniejący stan. Na przykład poleceniem jest działanie wykonywane przez back-end, takie jak zarejestrowanie nowego użytkownika, przetworzenie zawartości koszyka zakupowego lub uaktualnienie profilu klienta. Z punktu widzenia CQRS zadanie jest jednokierunkowe i generuje przepływ pracy od warstwy prezentacji do warstwy domenowej, który zapewne kończy się modyfikacją jakiegoś zapisu w pamięci masowej.

Model, który zajmuje się tylko zapytaniami, będzie znacznie łatwiejszy do zbudowania, niż model, który ma wykonywać zarówno zapytania, jak i aktualizacje danych. Model czytający ma strukturę klas bardziej zbliżoną do obiektów transferu danych (*data transfer objects* – DTO), zaś właściwości zwykle są znacznie liczniejsze od metod.

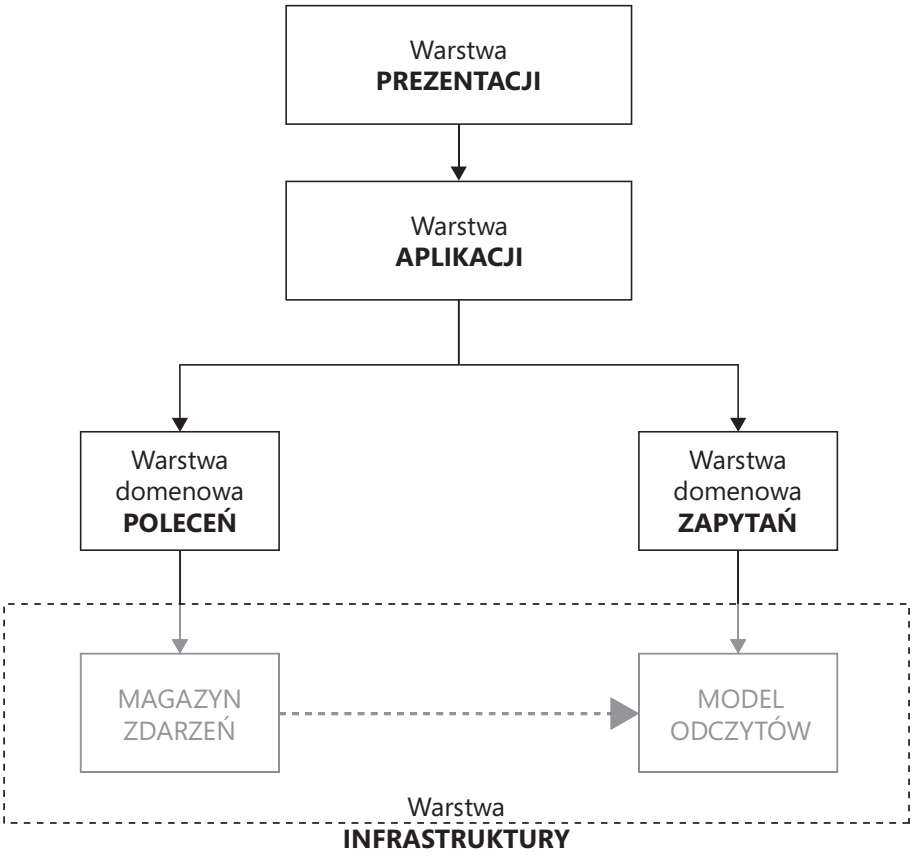
Wynikowy model jest zatem bardziej anemiczny, jako że traci wszystkie metody służące do modyfikowania stanu obiektu.

Warstwa aplikacji nie doświadcza zasadniczych zmian w scenariuszu CQRS. Zajmuje się wyzwalaniem zadania serwerowego powiązanego z żądaniem. To jednak nie jest prawdą w przypadku warstwy infrastruktury. W tym miejscu wkracza inny dodatek – *Event Sourcing*.

Dodawanie Event Sourcing

Event Sourcing idzie kolejny krok dalej w oddzielaniu poleceń zapytań, postulowanym przez CQRS. Dane przechowuje jako szereg niezmiennych zdarzeń, przechwytując każdą zmianę stanu systemu. Zdarzenia te zapewniają pełny historyczny zapis ewolucji systemu, pozwalając na inspekcję, odtwarzania i złożone analizy danych, w tym analizy „co jeśli". Event Sourcing jest szczególnie cenne w systemach, w których zmiany danych są częste lub gdy kluczowy jest szczegółowy kontekst historyczny, przez co można go traktować jako ewolucję zasad CQRS.

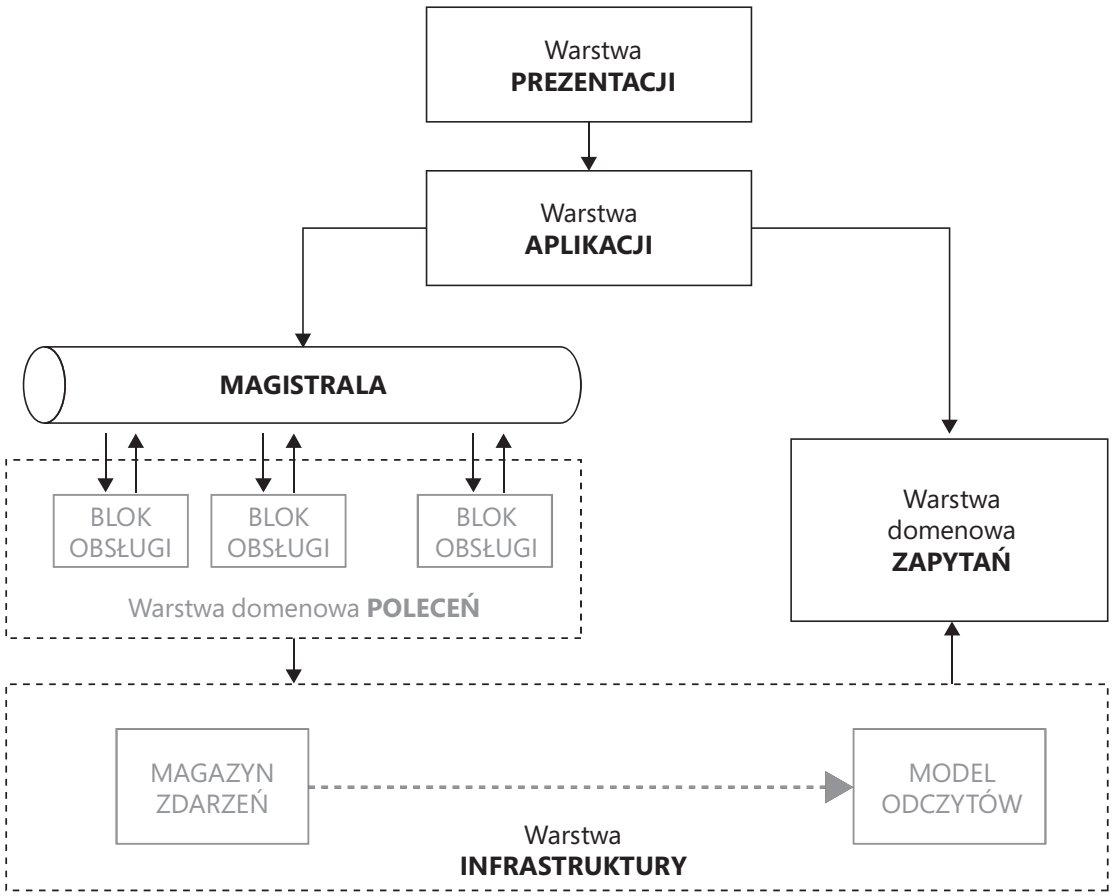
W ogólności, kiedy częstotliwość poleceń znacząco przekracza liczbę odczytów, można rozważyć dedykowany podsystem utrwalania wewnątrz warstwy infrastruktury. Rysunek 1-5 pokazuje możliwy projekt.



Rysunek 1-5 Abstrakcja architektury systemu opartego na Event Sourcing

W rzeczywistym świecie zaledwie obserwujemy zdarzenia. Jednak z jakiegoś powodu czujemy potrzebę zbudowania modelu, który będzie przechwytywać dowolne informacje niesione przez te zdarzenia i je przechowywać. Modele są niebywale pomocne, gdy zajmujemy się zapytaniami, ale już nie tak bardzo w przypadku poleceń. Dla tych ostatnich najlepsze są systemy oparte na zdarzeniach. Możemy nawet pójść dalej i stwierdzić, że systemy zdarzeniowe powinny być normą, a modele wyjątkiem. Ilekroć używamy modelu, posługujemy się jakimś „dostatecznie dobrym" przybliżeniem.

Model z rysunku 1-5 można nawet rozszerzyć, aby zmienić wewnętrzną organizację warstwy aplikacji. Zazwyczaj logika konieczna do implementacji przypadków użycia jest pisana jako oparty na kodzie przepływ pracy, orkiestrowany przez klasy i metody w warstwie aplikacji. Kiedy jednak wybierzemy postrzeganie aplikacji oparte na zdarzeniach, wówczas wszystko, co trzeba robić w warstwie aplikacji, to wypchnięcie komunikatu opisującego odebrane żądanie. Komunikat jest dostarczany do magistrali, na której nasłuchują programy obsługi (handlery) usług domeny. Każdy handler reaguje na zdarzenia, które go interesują, wykonując działania i wypychając do magistrali pozostałe komunikaty, aby inni słuchający mogli zareagować. Jak widać na rysunku 1-6, cała logika biznesowa każdego zadania zostaje ostatecznie zakodowana jako sekwencja komunikatów, a nie sekwencja aktywności przepływu pracy opartego na kodzie.



Rysunek 1-6 Logika biznesowa oparta na komunikatach

Największym problemem dotyczącym zdarzeń w architekturze oprogramowania jest koncepcja „ostatniego znanego dobrego stanu" systemu, która od dziesięcioleci należy do głównych pojęć. Teraz jest ona zastępowana podejściem „co się stało", który traktuje zdarzenia domenowe jako podstawę architektury.

Odgrywanie przez zdarzenia tak centralnej roli w architekturze oprogramowania stwarza pewne nowe wyzwania i może nawet powodować jakiś opór bezwładności. Oto kilka powodów, dla których zdarzenia mają głęboki wpływ na architekturę oprogramowania:

- **Niczego nie przeoczymy** Projektując architekturę opartą na zdarzeniach dajemy sobie moc łatwego śledzenia niemal wszystkiego, co dzieje się w systemie. Nowe zdarzenia mogą być dodawane niemal w każdej chwili, pozwalając na coraz bardziej precyzyjne kopiowanie przestrzeni biznesowej.
- **Rozszerzalność reprezentacji biznesowej** Używanie modelu do utrwalania biznesowych przypadków użycia ogranicza to, co można przechować i reprezentować w ramach tego modelu. Używanie zamiast tego zdarzeń usuwa większość tych ograniczeń; jak wspomniałem, dodawanie lub modyfikowanie scenariuszy biznesowych jest teraz możliwe i względnie niedrogie.
- **Dobra pozycja dla skalowalności** Event Sourcing używany w połączeniu z CQRS zapewnia podstawy skalowalności każdego systemu, jeśli nadejdzie czas, w którym skalowalność stanie się kluczowa.

W ujęciu abstrakcyjnym warstwowa architektura DDD pozostaje niezwykle logicznym sposobem organizowania indywidualnych aplikacji. Jednak dzisiejsze czasy stworzyły nowe ozdoby i lukry dla finalnego tortu. Podobnie jak w każdym przepisie na wypieki, właściwe proporcje CQRS i Event Sourcing są kwestią smaku.

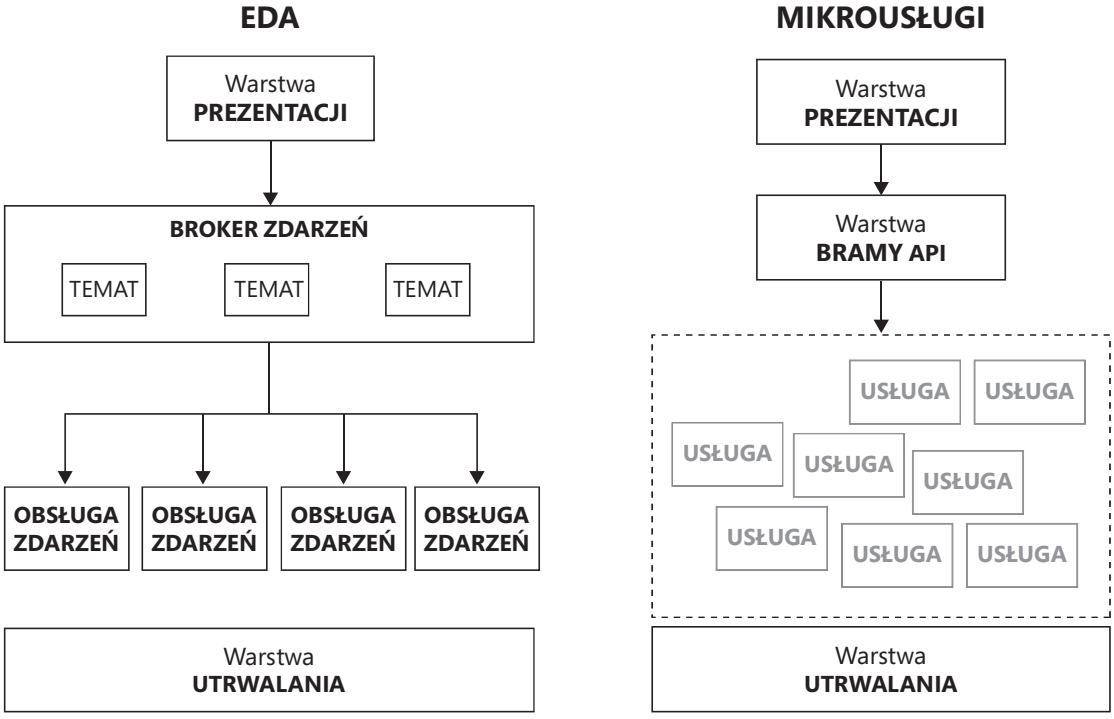
Z powodów, które mogę jedynie zgadywać, w pewnym momencie architektura warstwowa była rozważana nie tyle dzięki zapewnianej przez nią abstrakcji, ale jako konkretny, horyzontalny i obiektowy sposób budowania aplikacji. Tym samym pojawiły się inne odmiany architektur aplikacji o fantazyjnych nazwach. Jednak z mojego punktu widzenie nie są one niczym innym, jak różnymi sposobami ubierania warstw oprogramowania.

Bezwarstwowa architektura oprogramowania

Idea modularności idzie w parze z warstwami. Choć obie koncepcje wydają się nieco inaczej spolaryzowane, warstwy można rozpoznawać w każdym miejscu. Niemniej jednak wewnętrzne szczegóły projektowe warstw czasami tak się przenikają, że przelewają się przez granice warstw i same stają się architekturą bezwarstwową. Dla przykładu rozważmy architekturę sterowaną zdarzeniami (*event-driven architecture* – EDA).

Zasadniczo EDA jest tym, co pokazuje rysunek 1-6, zakładając, że wszystko – w tym żądanie odczytania modelu – przychodzi z magistrali albo bardziej ogólnie, poprzez komponent brokera (patrz lewa strona rysunku 1-7).

Innym pozbawionym warstw typem architektury są mikrousługi. Rozdział 9, „Mikrousługi kontra modularne monolity" jest poświęcony mikrousługom. Na razie kluczowe jest proste zrozumienie zamierzonego znaczenia tego terminu, a żeby to osiągnąć, trzeba mieć na uwadze wielkość komponentów. Oznacza to znalezienie odpowiedzi na pytanie „Jak duże jest mikro?" Jeśli mikro jest dostatecznie duże, wróciliśmy do architektury warstwowej. Jeśli mikro jest naprawdę małe, zbliżamy się do EDA. W kontekście architektury natywnie chmurowej mikrousługi są względnie prostymi i często bezstanowymi programami obsługi zdarzeń. Logika orkiestrowania aktywności wielu mikrousług w działaniu biznesowym może istnieć w różnych miejscach – front-end, w jakimś oprogramowaniu pośredniczącym typu GraphQL lub w usłudze bramy (patrz prawa strona rysunku 1-7).



Rysunek 1-7 Architektura sterowana zdarzeniami kontra architektura mikrousług

Główną korzyścią oferowaną przez architekturę bezwarstwową jest wrodzona decentralizacja funkcji, jeśli porównać to z monolityczną naturą rozwiązań warstwowych. Jednak tworzenie oprogramowania zawsze oznacza jakieś kompromisy. Tak więc decentralizacja nie musi być koniecznie lepsza w każdym z przypadków, a rozwiązania monolityczne nie zawsze są brudne i zagmatwane. Jak architekci mogą rozstrzygać, czego użyć? No cóż, to zależy!

Różne odmiany warstw

Architektura warstwowa często występuje w różnych wizualizacjach i pod różnymi nazwami. Zrozumienie tego wykracza poza moje możliwości, ale z tego co widzę, przynajmniej część tych nazw oznacza po prostu nieznacznie różniące się odmiany architektury warstwowej. W tym podrozdziale omówię kilka z tych odmian: architekturę heksagonalną, czystą oraz sterowaną funkcjami.

Uwaga Wszystkie omawiane tu wzorce architektoniczne (oraz inne, które można spotkać) mają wspólny cel (modularność i separacja zagadnień) oraz wspólną taktykę (warstwy oprogramowania).

Architektura heksagonalna

Architektura heksagonalna (HA) bazuje na idei, że centralną częścią aplikacji jest podstawowa biblioteka, która wchodzi w interakcje z zewnętrznym światem poprzez dobrze zdefiniowane interfejsy. Jeśli porównamy ją z architekturą warstwową, rdzeń aplikacji HA odpowiada warstwom domenowej i aplikacji (implementacja przypadków użycia). Kluczowym czynnikiem w HA jest to, że każda komunikacja pomiędzy rdzeniem aplikacji a resztą świata odbywa się poprzez kontraktowane interfejsy, nazywane *portami*. Adaptery są łącznikami pomiędzy portami a różnymi warstwami aplikacji. Jako minimum, muszą istnieć adaptery dla warstwy prezentacji i utrwalania (rysunek 1-8).

Rysunek 1-8 Schemat architektury heksagonalnej port/adapter

Uwaga Twórcą koncepcji (i nazwy) architektury heksagonalnej jest Alistair Cockburn, który opracował ją w roku 2005. Cockburn, który jest również jednym z autorów Manifestu Zwinnego (*Agile Manifesto*), był blisko ludzi, którzy opracowali DDD, a jego architektura heksagonalna może się wydawać próbą złagodzenia możliwych pułapek modelowania obiektowego, stanowiącego ośrodek DDD. Jednak choć koncepcja heksagonalna pojawiła się mniej więcej w tym samym czasie, co DDD i jego architektura warstwowa, ma z nim niewiele wspólnego. Jej głównym celem było uzyskanie jeszcze większej modularności poprzez umieszczenie interfejsu (portu) za każdym zadaniem ze świata rzeczywistego, które może wykonać rdzeń aplikacji.

Zbiór portów wejściowych tworzy warstwę API definiowaną przez aplikację, dzięki czemu świat zewnętrzny (czyli warstwa prezentacji) może z nią wchodzić w interakcje. Te porty są również nazywane *portami sterującymi* (*driver ports*). Porty umieszczone po drugiej stronie heksagonu to *porty sterowane* (*driven ports*), które tworzą definiowany przez aplikację interfejs do komunikacji z zewnętrznymi systemami, takimi jak bazy danych. Metody w tych portach są wszystkim, o czym wie aplikacja i usługi domeny. Z tego samego powodu adaptery są implementacjami interfejsów portów, które naprawdę wiedzą, jak pakować obiekty przesyłania danych wejściowych i jak odczytywać/zapisywać w bazie danych.

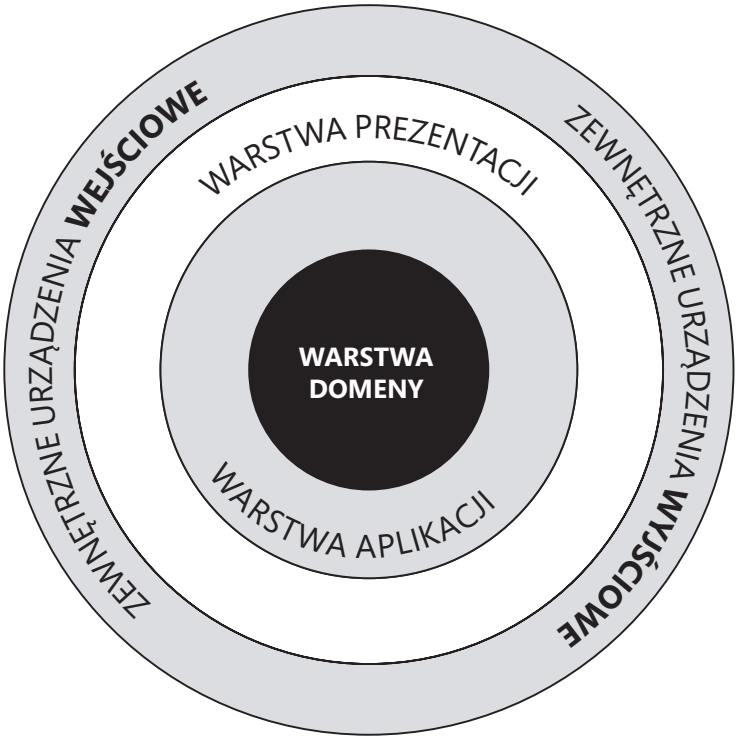
Uwaga Ze względu na to, jak ważne są porty i adaptery w architekturze heksagonalnej, inną popularną nazwą tej koncepcji jest „port/adapter".

Czysta architektura

Spopularyzowana przez Roberta Martina (znanego jako Uncle Bob) około roku 2012, czysta architektura (*clean architecture* – CA) jest po prostu inną odmianą architektury warstwowej, która czerpie zarówno z idei DDD, jak i HA. Nie wprowadza do gry niczego nowego poza dążeniem do uporządkowania i (zapewne) ujednoznacznienia koncepcji. Jednak ostatecznie dodanie kolejnej nazwy i jeszcze jednego stylu diagramu zapewne wprowadziło również nowe zamieszanie. Nie będziemy daleko od prawdy mówiąc, że CA to sposób nazwania nieznacznie różniących się pomysłów nową, jednoczącą nazwą.

CA przedstawia się przy użyciu koncentrycznych kręgów, a nie pionowo rozmieszczonych pasków lub sześciokąta. Najbardziej zewnętrzny krąg reprezentuje dowolną możliwą komunikację ze światem zewnętrznym, którym mogą być webowe front-endy, interfejsy użytkowników lub bazy danych. Najbardziej wewnętrzny krąg to warstwa domeny – repozytorium niezmiennych reguł biznesowych. Warstwa domeny jest otoczona warstwą przypadków użycia, czyli warstwą aplikacji. Następnie mamy warstwę

prezentacji, w której przetwarzane są dane wejściowe od zewnętrznych użytkowników (rysunek 1-9).



Rysunek 1-9 Schemat czystej architektury

Dobrze byłoby porównać schemat z rysunku 1-9 z tym pokazanym na rysunku 1-3. Jak widać, tamten schemat ilustruje te same koncepcje w pionie, grupując zewnętrzne urządzenia wyjściowe pod nazwą infrastruktury i zakłada (bez ich rysowania) istnienie urządzeńwejściowych nad blokiem prezentacji.

Uznawane zalety CA są takie same, jak każdej innej (właściwie wykonanej) architektury warstwowej. Można je podsumować następująco:

- Wrodzona testowalność logiki biznesowej, która pozostaje odseparowana od zewnętrznych zależności wnoszonych przez interfejs użytkownika, usługi i bazy danych.
- Niezależność interfejsu użytkownika, jako że architektura nie jest w żaden sposób związana z używaniem ASP.NET, .NET MAUI, bogatych front-endów ani czegokolwiek innego; frameworki nie mają żadnego wpływu na reguły biznesowe.
- Niezależność od utrwalania, gdyż znajomość bazy danych jest ograniczona do najbliższej warstwy i ignorowana na każdym wyższym poziomie.

Warto dodaćjeszcze końcowy komentarz na temat potrzeby stosowania interfejsów do przekraczania granic pomiędzy warstwami: w HA te interfejsy (porty) są w pewnym sensie obowiązkowe i stanowią wyróżniającą częśćsamej architektury. W CA i ogólnie w architekturach warstwowych używanie interfejsów jest decyzją implementatorów.

Użycie interfejsów – i kodowanie dla interfejsów, a nie implementacji – jest uniwersalną zasadą oprogramowania z niskim sprzężeniem. Jednak rezygnacja z interfejsów przy jasnym określeniu ich roli jest doskonałą oznaką samodyscypliny i pragmatyzmu.

Co naprawdę rozumiemy poprzez „czyste?"

Kiedy ktoś mówi „czysta architektura", nie jest oczywiste, jaką „czystość" ma na myśli – koncentryczny wzorzec czystej architektury opisany przez Roberta Martina, czy po prostu dobrze wykonaną, modułową architekturę. Tak czy inaczej, gdy rozmawiam o architekturze oprogramowania, czystość może być postrzegane jako synonim warstwowości.

Co naprawdę oznacza „warstwowość?" No cóż, warstwa oprogramowania, czy to logiczna, czy fizyczna, jest oddzielnym modułem połączonym z innymi modułami poprzez kontraktowe interfejsy. W rezultacie stosunkowo łatwe jest jej testowanie, zarówno w izolacji, jak i w trybie integracji. Tak więc gdy mowa o architekturze oprogramowania, warstwowość jest synonimem dla modularności.

Ale co to jest „modularność"? Ta koncepcja sięga wstecz do uniwersalnej zasady SoC sformułowanej przez Dijkstrę. Można ją przetłumaczyć na podział całego zbioru funkcjonalności na niezależne bloki, z których każdy zawiera wszystkie części niezbędne do perfekcyjnego działania funkcji. Moduł jest autonomiczny i połączony z innymi modułami przez identyfikowalne wtyczki oraz jednoznaczne, najlepiej wstrzykiwane zależności.

Czuły punkt modularności w mniejszym stopniu dotyczy konstruowania modułów i funkcji, a w większym ich kodowania – utrzymywania pod kontrolą punktów połączeń bez tworzenia ukrytych zależności. Tak napisany kod, czy to monolityczny, czy rozproszony, można ochrzcić mianem „czystego".

Uwaga Czystość kodu jest atrybutem uniwersalnym. Nie uzyskamy czystego kodu, po prostu stosując konkretny stos technologiczny (na przykład .NET, Java, Android, Python, Go czy TypeScript) ani jego określoną wersję.

Architektura funkcjonalna

Architektura funkcjonalna (*feature-driven architecture* – FDA) to podejście do architektury oprogramowania podkreślające organizowanie struktury programu wokół komponentów funkcjonalnych, które są kluczowe dla działania aplikacji. FDA nie jest czymś zasadniczo odmiennym od wzorców architektonicznych, które poznaliśmy do tej pory; po prostu oferuje alternatywne spojrzenie na projektowanie i budowanie systemu. FDA koncentruje się na identyfikowaniu kluczowych funkcji oprogramowania i koordynuje architekturę tak, by nadawać priorytet ich kompleksowemu wsparciu. Znaczącą

korzyścią podejścia FDA jest to, że często prowadzi do architektury modułowej opartej na komponentach.

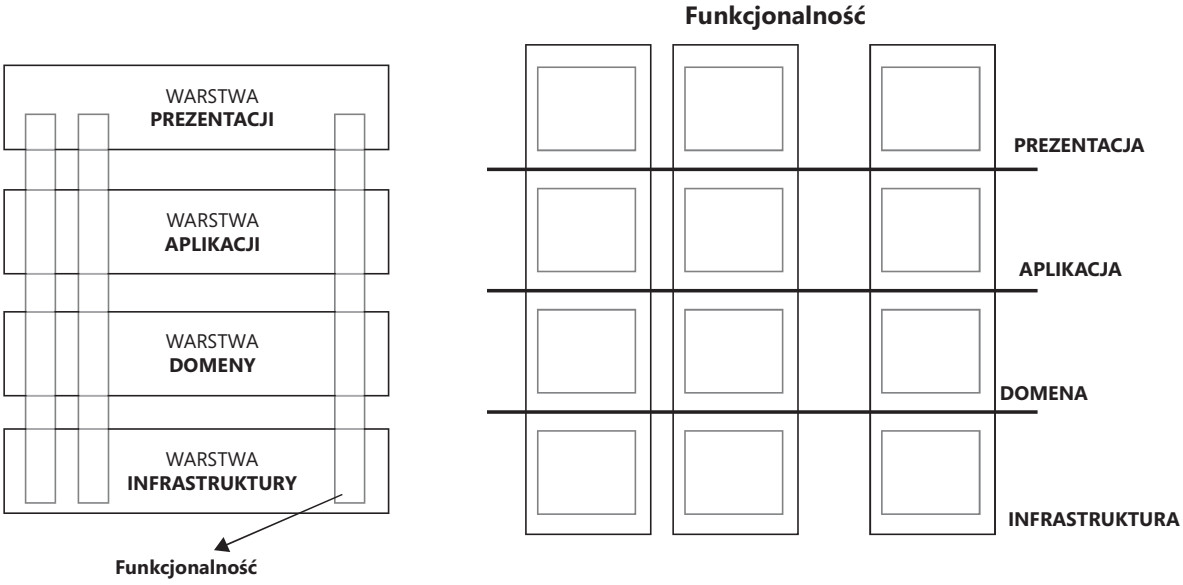
Zbiór podstawowych funkcjonalności reprezentuje najmniejszą możliwą złożoność nieodłącznie powiązaną z systemem. Rzeczywistą złożoność tworzy połączenie organicznej złożoności samych funkcjonalności plus dokładane do tego wszelkiego rodzaju przypadkowe złożoności wynikające z nieporozumień, długu technicznego, odziedziczonego kodu lub niedokładnych wyborów projektowych.

Architektura pionowych plasterków

Podejście funkcjonalne do wytwarzania systemu zazwyczaj idzie w parze z architekturą pionowych plasterków (*vertical slice architecture* – VSA). Dla przykładu zespół deweloperski może przyjąć jako podejście projektowe architekturę funkcjonalną, po czym użyć pionowych podziałów w celu przyrostowego implementowania i wydawania tych funkcji, dostarczając wartość w każdym kroku wytwarzania.

Rozumowanie na podstawie funkcji nie ma większego wpływu na architekturę, która pozostaje warstwowa – od logiki przypadków użycia, poprzez logikę domenową, aż do utrwalania. Jednak pomaga uporządkować wytwarzanie, włącznie z kodowaniem i testowaniem. W istocie VSA oznacza implementowanie kolejnych plasterków funkcjonalności jeden po drugim, przy czym każdy rozciąga się na cały stos warstw, w tym interfejs użytkownika, logikę aplikacji i magazynowanie danych.

Projektowanie systemu w pionowych segmentach oznacza przekształcenie schematu z rysunku 1-3 poprzez dodanie obok siebie pionowych podziałów, jak na rysunku 1-10.



Rysunek 1-10 Pionowe dzielenie funkcjonalności

Zwinność i kompromisy

Myślnie funkcjonalne i VSA mogą pomóc w oszacowaniu kosztów wytwarzania. VSA powstało w związku z metodologiami zwinnymi (Agile) i jest często używane do zbudowania minimalnego działającego produktu (*minimum viable product* – MVP) lub tworzenia przyrostowych wydań. Celem jest dostarczenie w pełni funkcjonalnej części oprogramowania, która może być używana, testowana i demonstrowana interesariuszom, jednocześnie zapewniając dostarczenie wartości użytkownikom docelowym wcześnie w procesie wytwarzania.

Jednak jak pokazuje prawy schemat na rysunku 1-10, projektowanie oparte na funkcjonalności może pofragmentować wszystkie poziomy warstwy, co stwarza pewne ryzyko duplikowania kodu. Co więcej, jeśli jedna lub więcej warstw jest wdrażana niezależnie w swoich własnych usługach aplikacji, koszty mogą wzrosnąć, jeśli będziemy wdrażać każdą warstwę oddzielnie dla każdej funkcji. Aby tego uniknąć, pewne specyficzne dla danej funkcji części tej samej warstwy (na przykład warstwy aplikacji) mogą zostać zespawane razem, tworząc coś, co DDD nazywa *współdzielonym jądrem* (*shared kernel*). Obszerniejszą dyskusję tego zagadnienia zawiera rozdział 2.

W rezultacie podejście sterowane funkcjonalnością brzmi trochę jak droga na skróty, jeśli jest stosowane na poziomie architektury, ale jest efektywnym sposobem organizowania codziennej pracy programistycznej, a także plików i folderów w repozytorium kodu. Najistotniejsze jest to, że analiza DDD jest najlepszą drogą, gdyż przechodzi przez żądane funkcje i izoluje je w ograniczonym kontekście i współdzielonym jądrze, zapewniając ostateczną listę komponentów oprogramowania, które trzeba zakodować.

Podsumowanie

Jeśli w ogóle istniał taki czas, w którym zagadnienie architektury oprogramowania było pomijalne, na pewno nie jest tak dziś. Zwłaszcza dla nowoczesnego oprogramowania dobra architektura jest wymogiem strukturalnym, a nie luksusem.

Jeśli postawimy pytanie, jaka architektura oprogramowania jest najbardziej odpowiednia w dzisiejszych czasach, zapewne usłyszymy jedną wspólną odpowiedź: mikrousługi. Patrząc obiektywnie, mikrousługi są nadużywanym terminem, który znaczy niewiele lub nic, o ile nie powiążemy go z definicją zamierzonego rozmiaru komponentów i otaczającego środowiska. Z mojego punktu widzenia, mikrousługi aż nazbyt często są wybierane w ramach czystej wiary w ten wzorzec. Rozdział 9 zawiera więcej informacji na temat mikrousług; mówiąc w skrócie, dobrze pasują do natywnej chmurowo architektury sterowanej zdarzeniami, o ile są hostowane na platformie bezserwerowej.

Jeśli jednak nie mikrousługi, to co? W trwającej publicznej debacie przeciwstawia się mikrousługi monolitom modularnym. Zabawne jest to, że mikrousługi wyłoniły się jako sposób zastąpienia ciasno związanych monolitycznych aplikacji. Podział był

jednak nadmiernie szczegółowy i stwarzał inne problemy – zasadniczo dotyczące zbierania razem rozproszonych i małych komponentów. Jak głoszą popularne przysłowia (to akurat ma swój odpowiednik w niemal każdym języku), prawda (albo cnota) leży pośrodku. W tym kontekście „środkiem" jest modułowa architektura warstwowa.

W tym rozdziale przedstawiłem historyczny przegląd koncepcji „poziomów" i „warstw", kończąc na najbardziej abstrakcyjnej ich wersji: architekturze warstwowej zdefiniowanej w DDD i ostatnio przemianowanej na *czystą architekturę*. Akronim DDD pojawił się tu wielokrotnie. Kolejny rozdział pokazuje, czym w istocie jest DDD.