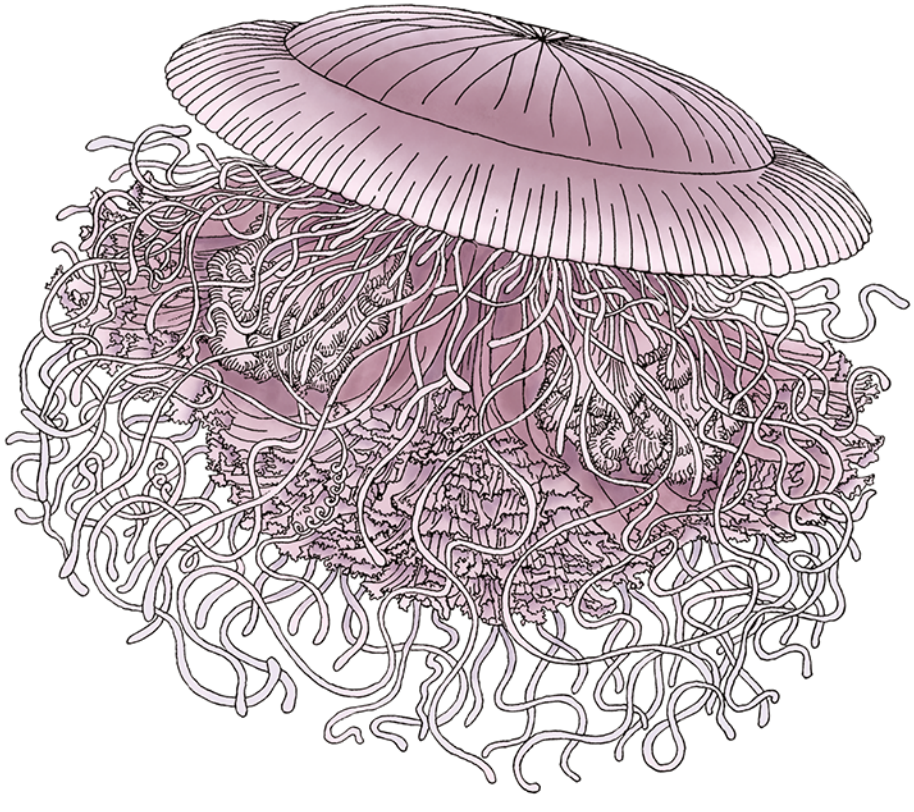
O'REILLY®

Od monolitu do mikrousług

Ewolucyjne wzorce przekształcania systemów monolitycznych



Helion

Sam Newman

Tytuł oryginału: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-6723-4

Authorized Polish translation of the English edition of Monolith to Microservices
ISBN 9781492047841 © 2020 Sam Newman

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/odmdom
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Tylko tyle mikrousług, ile potrzeba

> No, to się nazywa szybka eskalacja. To znaczy to naprawdę wymknęło się spod kontroli.
>
> — Ron Burgundy, *Legenda telewizji*

Zanim przejdę do tego, jak korzystać z mikrousług, ważne jest, aby ustalić wspólne rozumienie tego, czym są architektury oparte na mikrousługach. Chcę wspomnieć o kilku nieporozumieniach, z którymi regularnie się stykam, a także o często niedostrzeganych niuansach. Będziesz potrzebować solidnych podstaw wiedzy, aby w pełni wykorzystać dalszą zawartość tej książki. Dlatego w tym rozdziale wyjaśniam architektury oparte na mikrousługach, pokrótce omawiam powstanie mikrousług (co naturalnie oznacza przyjrzenie się systemom monolitycznym) i analizuję wybrane zalety oraz problemy związane z korzystaniem z mikrousług.

Czym są mikrousługi?

Mikrousługi są niezależnie instalowanymi usługami modelowanymi na podstawie dziedziny biznesowej. Komunikują się między sobą za pomocą sieci, a jako architektura zapewniają wiele możliwości rozwiązywania problemów, na jakie możesz natrafić. Wynika z tego, że architektura oparta na mikrousługach wykorzystuje wiele współdziałających ze sobą mikrousług.

Taka architektura jest *rodzajem* architektury opartej na usługach (ang. *service-oriented architecture* — SOA) — takim, w którym określony jest sposób wyznaczania granic usług i gdzie kluczowa jest możliwość niezależnego instalowania usług. Mikrousługi mają tę zaletę, że nie wymagają stosowania określonych technologii.

Jeśli chodzi o technologie, mikrousługi udostępniają oferowane funkcje biznesowe za pomocą jednego lub kilku sieciowych punktów końcowych. Mikrousługi komunikują się między sobą za pomocą sieci, dlatego tworzą system rozproszony. Hermetyzują też składowanie, pobieranie i udostępnianie danych za pomocą ściśle zdefiniowanych interfejsów. Dlatego bazy danych są ukryte w granicach usługi.

Wszystko to wymaga wyjaśnień, dlatego przyjrzyjmy się teraz bliżej niektórym z tych zagadnień.

Możliwość niezależnego instalowania

W *możliwości niezależnego instalowania* chodzi o to, że można wprowadzić zmianę w mikrousłudze i zainstalować tę mikrousługę w środowisku produkcyjnym bez konieczności korzystania z innych usług. Ważniejsze jest to, że nie jest to tylko *możliwość*. W ten sposób *rzeczywiście* zarządza się instalacjami w systemie. Jest to podejście stosowane do większości wersji usług. Jednak realizacja tej prostej idei jest skomplikowana.

Jeśli jest jedna rzecz, jaką powinieneś przyswoić sobie z tej książki, jest nią to: koniecznie korzystaj z możliwości niezależnego instalowania mikrousług. Wyrób sobie nawyk wprowadzania zmian w jednej mikrousłudze w środowisku produkcyjnym bez konieczności instalowania czegokolwiek innego. Wyniknie z tego wiele dobrych rzeczy.

Aby zagwarantować możliwość niezależnego instalowania, trzeba mieć pewność, że usługi są *luźno powiązane*. Oznacza to, że niezbędna jest możliwość modyfikowania jednej usługi bez konieczności wprowadzania zmian w innych komponentach. Dlatego potrzebne są bezpośrednie, dobrze zdefiniowane i stabilne kontrakty między usługami. Niektóre rozwiązania w zakresie implementacji mogą to utrudniać. Problemy sprawia zwłaszcza współdzielenie baz danych. Dążenie do uzyskania luźno powiązanych usług ze stabilnymi interfejsami wpływa na myślenie o tym, jak wyznaczać granice usług.

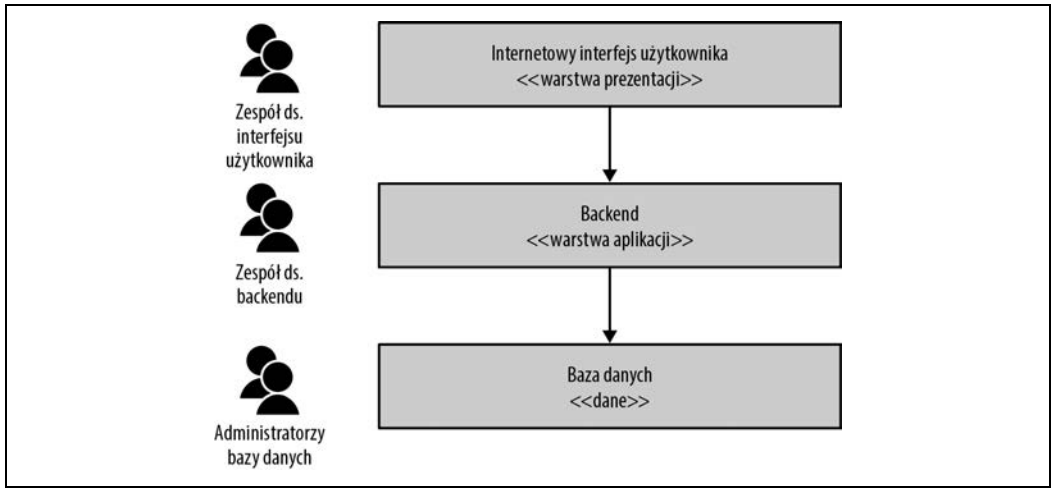
Modelowane na podstawie dziedziny biznesowej

Wprowadzanie zmian wykraczających poza granice procesu jest kosztowne. Jeśli musisz zmodyfikować dwie usługi, aby udostępnić nową funkcję, a także skoordynować instalację tych dwóch zmian, będzie to wymagać więcej pracy niż wprowadzenie tej samej zmiany w jednej usłudze (lub nawet w systemie monolitycznym). Z tego wynika, że należy szukać technik maksymalnie ograniczających zmiany obejmujące więcej niż jedną usługę.

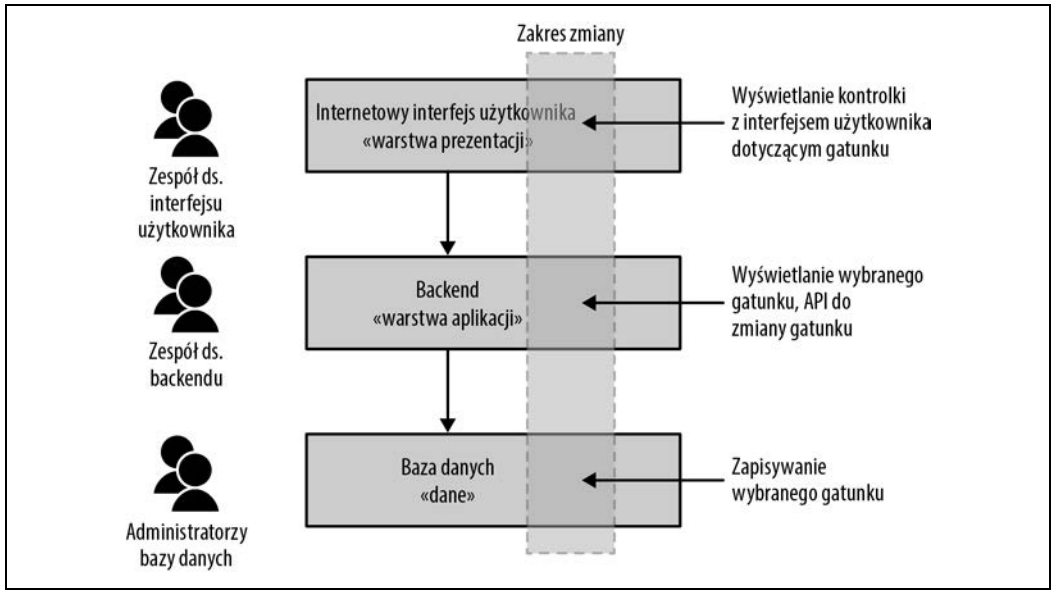
Zgodnie z podejściem, z jakiego korzystałem w książce *Budowanie mikrousług*, opisuję tu fikcyjną domenę i firmę, aby zilustrować wybrane zagadnienia, gdy nie jest możliwe posłużenie się prawdziwymi historiami. Jest to firma Music Corp, duża międzynarodowa organizacja, której w jakiś sposób udaje się utrzymać na rynku, mimo że koncentruje się prawie wyłącznie na sprzedaży płyt CD.

Zdecydowaliśmy się wprowadzić oporną firmę Music Corp w XXI wiek i w ramach tego procesu oceniamy istniejącą architekturę systemu. Na rysunku 1.1 pokazana jest prosta architektura trójwarstwowa. Dostępny jest internetowy interfejs użytkownika, warstwa logiki biznesowej w postaci monolitycznego backendu i system składowania danych w formie tradycyjnej bazy danych. Te warstwy, jak to często bywa, są rozwijane przez różne zespoły.

Chcemy wprowadzić prostą zmianę w funkcjach tego systemu i umożliwić klientom podanie ulubionego gatunku muzyki. Wymaga to zmodyfikowania interfejsu użytkownika (aby wyświetlał gatunki do wyboru), usługi backendu (aby przekazywała gatunki do interfejsu użytkownika i zmieniała wybraną wartość) i bazy danych (aby przyjmowała wprowadzane zmiany). Te modyfikacje muszą być wprowadzane przez każdy zespół, co ilustruje rysunek 1.2. Ponadto zmiany trzeba zainstalować we właściwej kolejności.



Rysunek 1.1. Systemy firmy Music Corp mają postać tradycyjnej architektury trójwarstwowej



Rysunek 1.2. Wprowadzanie zmiany we wszystkich trzech warstwach jest bardziej skomplikowane

Ta architektura nie jest zła. Każda architektura jest ostatecznie optymalizowana zgodnie z wyznaczonymi celami. Ta trójwarstwowa architektura jest tak często stosowana po części z powodu jej uniwersalności — każdy ją zna. Dlatego wybór znanej architektury, z którą już się zetknąłeś, jest jednym z powodów ciągłego natykania się na ten wzorzec. Uważam jednak, że najważniejszą przyczyną tego, że nieustannie natrafiamy na tę architekturę, jest sposób, w jaki organizuje ona pracę zespołów.

Słynne obecnie prawo Conwaya brzmi tak:

> Każda organizacja projektująca system […] bez wątpienia opracuje projekt, którego struktura jest kopią struktury komunikacji w tej firmie.
>
> — Melvin Conway, *How Do Committees Invent?*

Ta trójwarstwowa architektura jest dobrym przykładem działania tego prawa. W przeszłości firmy informatyczne grupowały pracowników przede wszystkim według ich podstawowych kompetencji. Administratorzy baz danych byli w zespole z innymi administratorami, programiści Javy znajdowali się w zespole z innymi programistami Javy, a programiści frontendu (obecnie znający egzotyczne technologie takie jak JavaScript i programowanie natywnych aplikacji na urządzenia przenośne) pracowali w jeszcze innym zespole. Grupujemy pracowników według ich podstawowych kompetencji i podobnie tworzymy zasoby informatyczne, które można przydzielać takim zespołom.

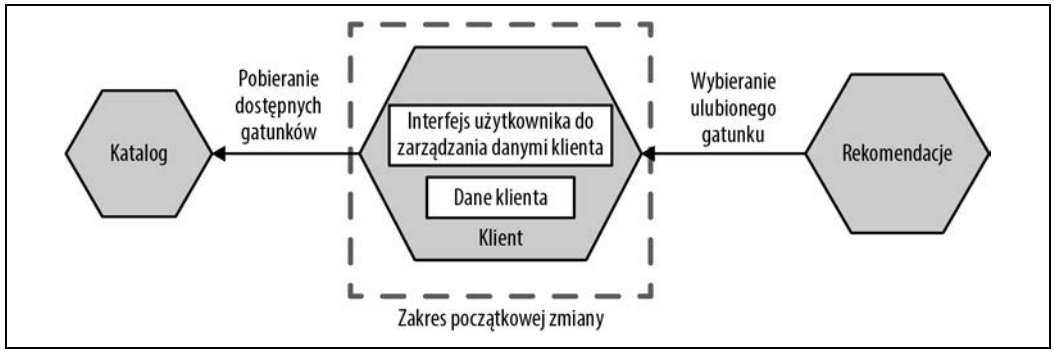
To wyjaśnia, dlaczego ta architektura występuje tak często. Nie jest ona zła. Jest tylko zoptymalizowana według jednego zestawu czynników — według tego, jak w tradycyjny sposób grupujemy pracowników na podstawie podobnych umiejętności. Jednak czynniki się zmieniły. Także nasze aspiracje związane z oprogramowaniem są dziś inne. Obecnie pracownicy są grupowani w wielozadaniowe zespoły, aby ograniczyć przekazywanie zadań i liczbę silosów. Chcemy udostępniać oprogramowanie znacznie szybciej niż w przeszłości. To wpływa na dokonywanie nowych wyborów związanych z organizowaniem zespołów, a także na sposób podziału systemów.

Zmiany w działaniu dotyczą przede wszystkim modyfikowania funkcji biznesowych. Jednak na rysunku 1.1 funkcje biznesowe są rozdzielone między trzy warstwy, co zwiększa prawdopodobieństwo, że zmiany będą obejmować kilka warstw. Jest to architektura, w której występuje wysoka spójność technologii, ale niska spójność funkcji biznesowych. Aby ułatwić wprowadzanie zmian, trzeba inaczej podejść do grupowania kodu i wybrać spójność funkcji biznesowych zamiast spójności technologii. Każda usługa może (ale nie musi) obejmować trzy wymienione warstwy, ale jest to kwestia dotycząca implementacji lokalnej usługi.

Porównaj wcześniejsze rozwiązanie z możliwą inną architekturą przedstawioną na rysunku 1.3. Widoczna jest tu specjalna usługa Klient, która udostępnia interfejs użytkownika umożliwiający klientom aktualizowanie informacji. W tej usłudze zapisywany jest też stan powiązany z klientem. Wybór ulubionego gatunku jest powiązany z danym klientem, dlatego zmiana jest bardziej lokalna. Na rysunku 1.3 pokazana jest też lista dostępnych gatunków pobierana z usługi Katalog, która zapewne jest już dostępna. Widoczna jest też nowa usługa Rekomendacja, która pobiera informacje o ulubionym gatunku. Taką usługę można łatwo utworzyć w nowych wersjach systemu.

W takim scenariuszu usługa Klient hermetyzuje wąski wycinek każdej z trzech warstw — obejmuje trochę interfejsu użytkownika, trochę logiki aplikacji i trochę kodu do składowania danych. Jednak wszystkie te warstwy są hermetyzowane w jednej usłudze.

Dziedzina biznesowa staje się tu podstawowym czynnikiem wpływającym na architekturę systemu. Można mieć nadzieję, że ułatwi to wprowadzanie zmian i organizowanie zespołów na podstawie dziedziny biznesowej. Jest to tak ważne, że przed zakończeniem tego rozdziału wrócę do kwestii



Rysunek 1.3. Specjalna usługa Klient może znacznie ułatwić zapisywanie ulubionego gatunku muzycznego klienta

modelowania oprogramowania na podstawie dziedziny biznesowej. Pozwoli mi to przedstawić kilka pomysłów związanych z podejściem DDD, które kształtują sposób myślenia o architekturze opartej na mikrousługach.

Mikrousługi są właścicielem swoich danych

Jedną z rzeczy, jakie programistom najtrudniej jest zrozumieć, jest to, że mikrousługi nie powinny współdzielić baz danych. Jeśli jedna usługa chce używać danych przechowywanych przez inną usługę, powinna zażądać od niej potrzebnych danych. Dzięki temu usługa może zdecydować, które dane są współdzielone, a które ukryte. Ponadto usługa może zarządzać odwzorowaniem z wewnętrznych szczegółów implementacji, które z różnych arbitralnych powodów mogą się zmieniać na bardziej trwały publiczny kontrakt, gwarantujący stabilne interfejsy usług. Stabilny interfejs między usługami jest niezbędny, jeśli celem jest możliwość niezależnego ich instalowania. Gdy interfejs udostępniany przez usługę stale się zmienia, wywołuje to efekt domina powodujący, że zmiany trzeba wprowadzić także w innych usługach.

Nie stosuj współdzielenia baz danych, chyba że jest to naprawdę konieczne, a nawet wtedy zrób wszystko, co możliwe, aby tego uniknąć. Moim zdaniem współdzielenie baz jest jedną z najgorszych rzeczy, jakie można zrobić, dążąc do możliwości niezależnego instalowania usług.

W poprzednim punkcie wyjaśniłem, że o usługach należy myśleć jak o kompletnych wycinkach z funkcjami biznesowymi hermetyzujących odpowiednie warstwy (interfejsu użytkownika, logiki aplikacji i składowania danych). Celem jest ograniczenie wysiłku potrzebnego przy modyfikowaniu funkcji biznesowych. Hermetyzowanie danych i operacji w ten sposób zapewnia wysoką spójność funkcji biznesowych. Ponadto ukrycie bazy danych, z której korzysta usługa, skutkuje luźniejszym powiązaniem między usługami. Do poziomu powiązania i spójności wrócę dalej.

Zrozumienie takiego rozwiązania może być trudne, zwłaszcza jeśli masz istniejący system monolityczny z bardzo rozbudowaną bazą danych, z której musisz korzystać. Na szczęście rozdział 4. jest w całości poświęcony odchodzeniu od monolitycznych baz danych.

Jakie korzyści mogą dawać mikrousługi?

Mikrousługi oferują wiele różnych zalet. Niezależność instalacji pozwala tworzyć nowe modele zwiększania skali i niezawodności systemów, a także umożliwia łączenie różnych technologii. Ponieważ nad usługami można pracować równolegle, problemem bez przeszkadzania sobie może zajmować się większa grupa programistów. Ponadto programistom może być łatwiej zrozumieć swoją część systemu, ponieważ wystarczy skupić się na tylko jednym jego wycinku. Z kolei izolacja procesów pozwala zmieniać technologie, a czasem łączyć różne języki programowania, style programowania, platformy lub bazy danych, aby znaleźć ich odpowiednią kombinację.

Zapewne najważniejsze jest to, że mikrousługi zapewniają elastyczność. Otwierają znacznie więcej możliwości rozwiązywania przyszłych problemów.

Należy jednak zwrócić uwagę na to, że wszystkie te korzyści są związane z kosztami. Jest wiele sposobów podziału systemów, a to, co starasz się osiągnąć, może wpływać na to, które podejście wybierzesz. Dlatego istotne jest, abyś zrozumiał, co chcesz osiągnąć dzięki architekturze opartej na mikrousługach.

Jakie problemy są powodowane przez mikrousługi?

Architektura oparta na usługach stała się popularna po części dlatego, że komputery potaniały, dlatego ich liczba wzrosła. Zamiast instalowania systemów w jednym gigantycznym komputerze typu mainframe lepszym rozwiązaniem stało się używanie wielu tańszych maszyn. Architektura oparta na usługach była próbą ustalenia optymalnego sposobu budowania aplikacji działających na wielu maszynach. Jedno z głównych wyzwań w tym obszarze dotyczyło narzędzia do komunikowania się między komputerami — sieci.

Komunikacja między komputerami przez sieć nie odbywa się natychmiast (musi mieć to jakiś związek z fizyką). To oznacza, że trzeba uwzględniać opóźnienie — przede wszystkim opóźnienia, które znacznie przekraczają te występujące w lokalnych operacjach wewnątrzprocesowych. Sytuacja staje się jeszcze gorsza, jeśli uwzględnić to, że te opóźnienia się zmieniają, przez co działanie systemu staje się nieprzewidywalne. Trzeba też pamiętać o tym, że sieci czasem zawodzą — pakiety giną, a kable sieciowe zostają odłączone.

Te wyzwania sprawiają, że operacje, które są stosunkowo proste w jednoprocesowych systemach monolitycznych, na przykład transakcje, stają się dużo trudniejsze. Tak trudne, że gdy system staje się bardziej złożony, zwykle trzeba zrezygnować z transakcji i zapewnianego przez nie bezpieczeństwa na rzecz innych technik (które, niestety, mają zupełnie inne wady i zalety).

Radzenie sobie z tym, że wywołania sieciowe mogą zawodzić (i tak robią), staje się problemem — podobnie jak fakt, że usługi, z którymi się komunikujesz, mogą z jakiegoś powodu stać się niedostępne lub zacząć działać w dziwny sposób. Dodatkowo trzeba zastanowić się nad tym, jak zachować spójny obraz danych na wielu maszynach.

Ponadto, oczywiście, istnieje całe mnóstwo nowych technologii zgodnych z mikrousługami, które należy uwzględnić. Są to nowe technologie, które — niewłaściwie stosowane — mogą ułatwiać popełnianie błędów w znacznie szybszy i ciekawszy (bardziej kosztowny) sposób. Szczerze mówiąc, mikrousługi wydają się beznadziejnym pomysłem, jeśli pominąć wszystkie ich zalety.

Warto zauważyć, że niemal wszystkie systemy uznawane za „monolityczne" są jednocześnie systemami rozproszonymi. Jednoprocesowa aplikacja zwykle wczytuje dane z bazy działającej na innym komputerze i wyświetla dane w przeglądarce internetowej. Oznacza to przynajmniej trzy komputery komunikujące się ze sobą przez sieci. Różnica polega na poziomie rozproszenia systemów monolitycznych w porównaniu z architekturami opartymi na mikrousługach. Im większa jest liczba komputerów komunikujących się przez większą liczbę sieci, tym większe prawdopodobieństwo wystąpienia nieprzyjemnych problemów powiązanych z systemami rozproszonymi. Opisane tu pokrótce problemy początkowo mogą nie wystąpić, jednak z czasem, wraz z rozrastaniem się systemu, prawdopodobnie natrafisz na większość z nich (lub nawet na wszystkie).

Mój dawny współpracownik, przyjaciel i ekspert z dziedziny mikrousług, James Lewis, ujął to tak: „stosując mikrousługi, kupujesz sobie możliwości". James celowo użył tych słów — *kupujesz możliwości*. Mikrousługi związane są z kosztami i musisz zdecydować, czy te koszty są warte możliwości, które chcesz wykorzystać. Zagadnienie to opisuję szczegółowo w rozdziale 2.

Interfejsy użytkownika

Zbyt często widzę, jak programiści wprowadzający mikrousługi koncentrują się wyłącznie na kodzie używanym po stronie serwera i pozostawiają interfejs użytkownika w postaci jednej, monolitycznej warstwy. Jeśli chcesz uzyskać architekturę, która ułatwia błyskawiczne instalowanie nowych funkcji, pozostawienie interfejsu użytkownika w postaci monolitycznego bloku może okazać się poważnym błędem. Można i należy zastanowić się także nad podziałem interfejsów użytkownika. To zagadnienie omawiam w rozdziale 3.

Technologie

Bardzo kuszące może być stosowanie całego zestawu nowych technologii razem z nową lśniącą architekturą opartą na mikrousługach. Gorąco zachęcam jednak do tego, abyś nie uległ tej pokusie. Wprowadzanie każdej nowej technologii jest związane z kosztami i powoduje zamieszanie. Mam nadzieję, że nowe rozwiązania okażą się tego warte (oczywiście jeśli wybrałeś właściwą technologię), jednak gdy po raz pierwszy wdrażasz architekturę opartą na mikrousługach, i tak masz wystarczająco dużo do zrobienia.

Ustalenie, jak w prawidłowy sposób rozwijać architekturę opartą na mikrousługach i zarządzać nią, wymaga poradzenia sobie z wieloma wyzwaniami dotyczącymi systemów rozproszonych. Są to wyzwania, z którymi wcześniej mogłeś się nie spotkać. Uważam, że znacznie lepiej jest zmagać się z takimi trudnościami, gdy korzysta się ze znanego zestawu technologii, a dopiero później rozważyć, czy zmiana używanych rozwiązań pomoże poradzić sobie z napotkanymi problemami.

Wspomniałem już, że mikrousługi zasadniczo są niezależne od technologii. Dopóki mikrousługi mogą się komunikować między sobą za pomocą sieci, możesz korzystać z dowolny rozwiązań. Bywa to ważną zaletą, ponieważ pozwala swobodnie łączyć ze sobą różne zestawy technologii.

Nie musisz korzystać z Kubernetesa, Dockera, kontenerów ani publicznej chmury. Nie musisz pisać kodu w języku Go, Rust lub jakimkolwiek innym. W architekturach opartych na mikrousługach wybór języka programowania jest w dużym stopniu nieistotny, choć niektóre języki mają bardziej rozbudowany ekosystem powiązanych bibliotek i platform. Jeśli najlepiej znasz PHP, zacznij pisać

usługi w tym języku[1]! Z niektórymi zestawami technologii związany jest przesadny techniczny snobizm, który może skutkować niemal pogardą wobec osób korzystających z określonych narzędzi[2]. Nie bądź częścią tego problemu. Wybierz podejście odpowiednie dla siebie i zmieniaj narzędzia, aby rozwiązywać napotkane problemy.

Wielkość

„Jak duża powinna być mikrousługa?" to prawdopodobnie najczęściej zadawane mi pytanie. Ponieważ w nazwie występuje słowo „mikro", nie jest to zaskoczeniem. Jednak gdy zrozumiesz, na czym polega architektura oparta na mikrousługach, wielkość okaże się jedną z najmniej interesujących kwestii.

Jak mierzyć mikrousługi? Wierszami kodu? Moim zdaniem nie ma to większego sensu. Coś, co może wymagać 25 wierszy kodu w Javie, w Clojure zapewne można napisać za pomocą 10 wierszy. Nie chcę przez to powiedzieć, że Clojure jest lepszy lub gorszy od Javy, tylko że niektóre języki są bardziej zwięzłe od innych.

Jeśli określenie „wielkość" w dziedzinie mikrousług ma mieć jakiś sens, najbliżej jego oddania moim zdaniem jest ekspert od mikrousług Chris Richardson, który kiedyś stwierdził, że celem autorów mikrousług powinno być tworzenie „jak najmniejszego interfejsu". Jest to zgodne z zasadą ukrywania informacji (którą omawiam dalej), ale reprezentuje próbę znalezienia znaczenia po fakcie. Gdy po raz pierwszy rozmawialiśmy o mikrousługach, naszym głównym celem — przynajmniej na początku — było zapewnienie łatwego zastępowania kodu.

Ostatecznie kwestia wielkości jest wysoce zależna od kontekstu. Porozmawiaj z kimś, kto pracował nad danym systemem przez 15 lat, a usłyszysz, że zdaniem tej osoby system obejmujący 100 tysięcy wierszy kodu jest naprawdę łatwy do zrozumienia. Jeśli poprosisz o opinię osobę, która dopiero zaczyna pracę nad tym projektem, usłyszysz, że ten sam system jest zdecydowanie za duży. Podobnie jeżeli zapytasz o zdanie pracowników firmy, która dopiero zaczęła przerzucać się na mikrousługi i korzysta na przykład z 10 małych mikrousług, usłyszysz inne odpowiedzi niż od organizacji o podobnej wielkości, gdzie jednak mikrousługi są normą od lat i gdzie działają ich setki.

Namawiam do tego, by nie przejmować się wielkością. Gdy zaczynasz korzystać z mikrousług, znacznie istotniejsze jest skupienie się na dwóch najważniejszych rzeczach. Po pierwsze, z iloma mikrousługami sobie poradzisz? Gdy liczba usług rośnie, to samo dotyczy także złożoności systemu. Musisz wtedy zdobyć nowe umiejętności (i prawdopodobnie wdrożyć nowe technologie), aby sobie z tym poradzić. To dlatego zdecydowanie zachęcam do stopniowego przechodzenia na architekturę opartą na mikrousługach. Po drugie, jak zdefiniujesz granice mikrousług, aby jak najlepiej wykorzystać możliwości usług i jednocześnie uniknąć poważnych kłopotów z powodu ścisłych powiązań między nimi? Te zagadnienia omawiam w pozostałej części rozdziału.

---

[1] Jeśli chcesz dowiedzieć się więcej na ten temat, polecam książkę *API nowoczesnej strony WWW*. Lorny Jane Mitchell (Helion, 2015).

[2] Po zapoznaniu się z artykułem *Contempt Culture* z bloga Aurynna Shawa (*https://blog.aurynn.com/2015/12/16-contempt-culture*) zdałem sobie sprawę, że w przeszłości sam byłem winny nieco pogardliwego stosunku do różnych technologii, a tym samym i związanych z nimi społeczności.

Historia nazwy „mikrousługi"

W roku 2011, kiedy wciąż jeszcze pracowałem dla firmy konsultingowej ThoughtWorks, mój znajomy i ówczesny współpracownik James Lewis mocno interesował się czymś, co nazywał mikroaplikacjami. James zauważył, że ten wzorzec jest stosowany przez sporo firm korzystających z architektury opartej na usługach. Te firmy optymalizowały architekturę, aby ułatwić sobie zastępowanie usług. Firmom tym zależało na szybkim wdrażaniu określonych funkcji, ale z zachowaniem możliwości przepisania ich z użyciem innego zestawu technologii, jeśli i kiedy potrzebne byłoby skalowanie systemu.

Wówczas w oczy rzucało się to, jak niewielkie były te usługi. Niektóre z nich można było napisać (lub przepisać) w kilka dni. James posunął się do stwierdzenia, że „usługi nie powinny być większe od mojej głowy". Chodziło mu o to, że zakres funkcji powinien być łatwy do zrozumienia, a tym samym i do zmodyfikowania.

Później, w 2012 roku, James prezentował te pomysły na poświęconej architekturze konferencji, w której uczestniczyłem między innymi ja, a także kilku znajomych programistów. Dyskutowaliśmy wtedy o tym, że omawiane komponenty tak naprawdę nie są niezależnymi aplikacjami, dlatego nazwa „mikroaplikacje" nie jest w pełni trafna. Zamiast tego lepsze wydało się określenie „mikrousługi"[3].

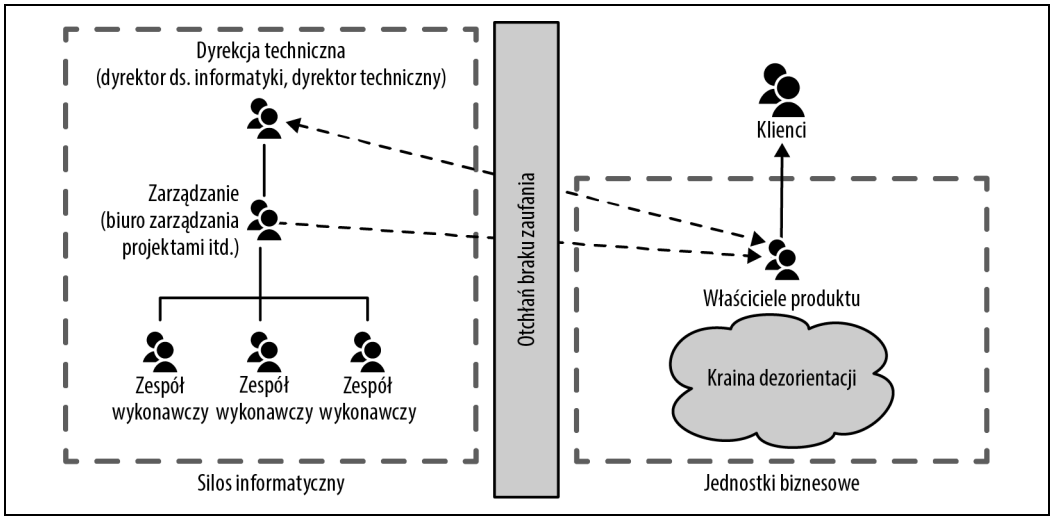
Właściciele

Gdy mikrousługi są modelowane na podstawie dziedziny biznesowej, można dostrzec dopasowanie artefaktów informatycznych (niezależnie instalowanych mikrousług) do danej dziedziny. To podejście jest zgodne z widoczną w firmach technologicznych zmianą nastawienia na eliminowanie granic między biznesem a informatykami. W tradycyjnych organizacjach informatycznych rozwojem oprogramowania często zajmuje się część firmy całkowicie odrębna od tej, która odpowiada za definiowanie wymagań i kontakty z klientami. Ilustruje to rysunek 1.4. W tego rodzaju organizacjach występuje wiele rozmaitych problemów, co zapewne nie wymaga dalszego omawiania w tym miejscu.

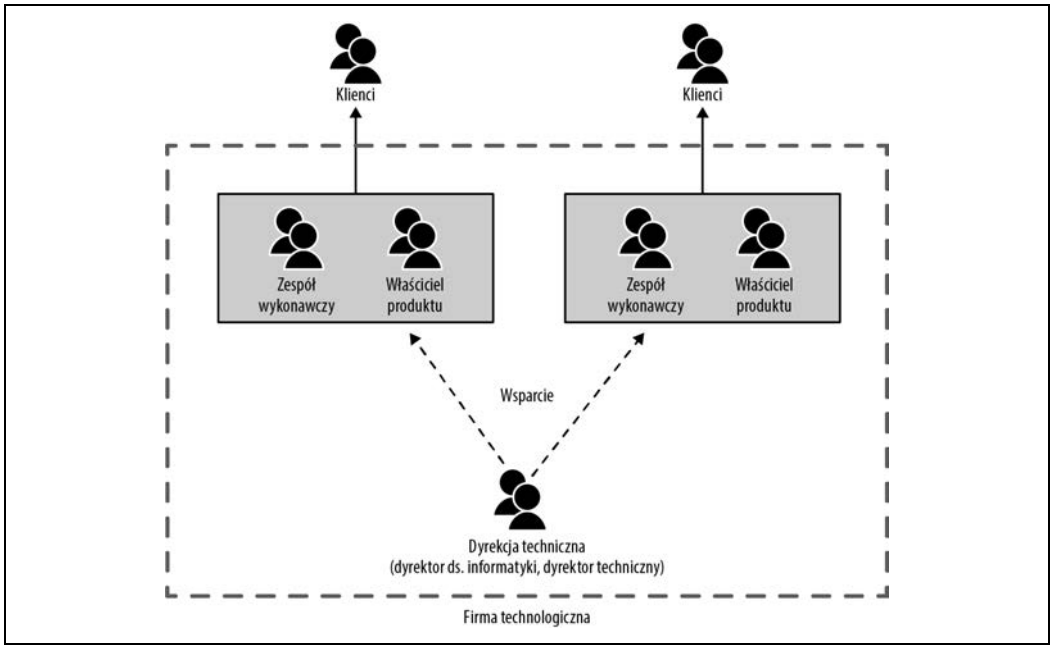
Zamiast tego widzimy, jak organizacje technologiczne łączą wcześniej odrębne silosy organizacyjne, co ilustruje rysunek 1.5. Właściciele produktu obecnie pracują bezpośrednio w zespołach wykonawczych, a te zespoły są organizowane na podstawie linii produktów przeznaczonych dla klientów, a nie na podstawie arbitralnie wybieranych umiejętności technicznych. Normą nie jest teraz centralizowanie funkcji informatycznych. Jeśli już występują scentralizowane funkcje informatyczne, celem jest wspieranie zespołów wykonawczych skoncentrowanych na kliencie.

Choć nie we wszystkich organizacjach nastąpiła taka zmiana, architektury oparte na mikrousługach znacznie ją ułatwiają. Jeśli chcesz, aby zespoły wykonawcze były organizowane według linii produktowych, a usługi były rozwijane na podstawie dziedzin biznesowych, łatwiej będzie bezpośrednio

[3] Nie pamiętam, kiedy po raz pierwszy zapisaliśmy tę nazwę. Jednak wyraźnie przypominam sobie moje nalegania (wbrew logice gramatyki), aby *nie* stosować w niej łącznika. Z perspektywy czasu było to trudne do uzasadnienia stanowisko, jednak taki zapis się przyjął, a ja podtrzymuję swoją nieracjonalną, ale ostatecznie zwycięską propozycję.



Rysunek 1.4. Struktura organizacyjna ilustrująca tradycyjny podział na jednostki informatyczne i biznesowe



Rysunek 1.5. Przykład pokazujący, jak firmy technologiczne integrują rozwój oprogramowania

przydzielać projekty zespołom wykonawczym zorientowanym na produkt. Ograniczenie liczby usług współdzielonych między wiele zespołów jest kluczem do zminimalizowania konfliktów w procesie dostarczania oprogramowania. Architektury oparte na mikrousługach organizowanych na podstawie dziedziny biznesowej znacznie ułatwiają tego rodzaju zmiany w strukturach organizacyjnych.

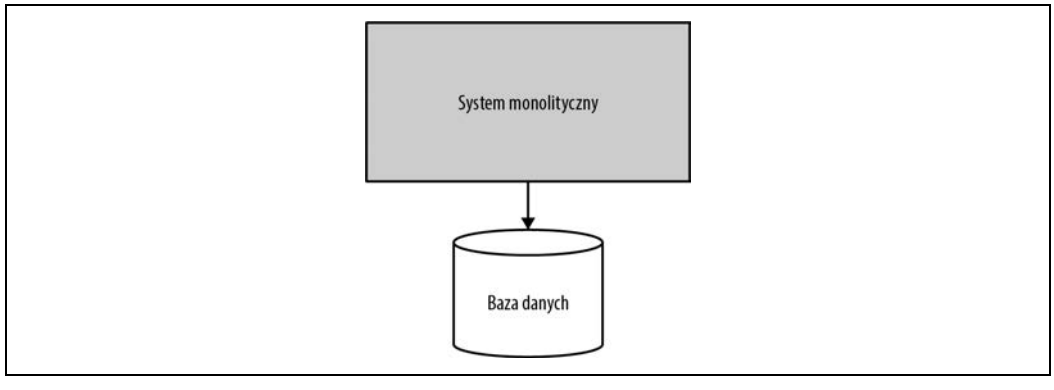
System monolityczny

Opisałem już mikrousługi, jednak ta książka dotyczy przechodzenia *od* systemów monolitycznych *do* mikrousług, dlatego trzeba też ustalić, czym jest *system monolityczny*.

Gdy w tej książce piszę o systemach monolitycznych, przede wszystkim chodzi mi o jednostki instalacji. Gdy wszystkie funkcje systemu trzeba instalować wspólnie, uznaję go za system monolityczny. Istnieją przynajmniej trzy rodzaje systemów monolitycznych zgodne z tym opisem: systemy jednoprocesowe, rozproszone systemy monolityczne i systemy typu czarna skrzynka od niezależnych dostawców.

Jednoprocesowe systemy monolityczne

Najbardziej typowym przykładem przychodzącym na myśl w trakcie omawiania systemów monolitycznych jest system, w którym cały kod jest instalowany jako *jeden proces*. Ilustruje to rysunek 1.6. Można uruchamiać wiele instancji tego procesu (na potrzeby skalowania lub zwiększenia niezawodności), jednak zasadniczo cały kod działa w ramach jednego procesu. W praktyce takie systemy jednoprocesowe same mogą być prostymi systemami rozproszonymi, ponieważ prawie zawsze wczytują dane lub zapisują je w bazie.

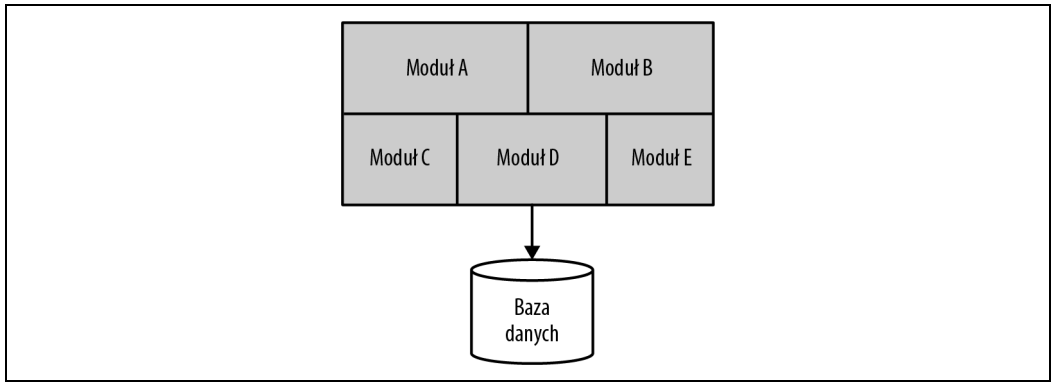


Rysunek 1.6. Jednoprocesowy system monolityczny — cały kod jest umieszczony w jednym procesie

Tego rodzaju jednoprocesowe systemy monolityczne stanowią większość systemów monolitycznych, z jakimi programiści się zmagają. Dlatego w tej książce koncentruję się na tego typu systemach. Gdy od tego miejsca będę używał pojęcia „system monolityczny”, będę miał na myśli właśnie tego rodzaju systemy (chyba że zaznaczę, że jest inaczej).

Modułowy system monolityczny

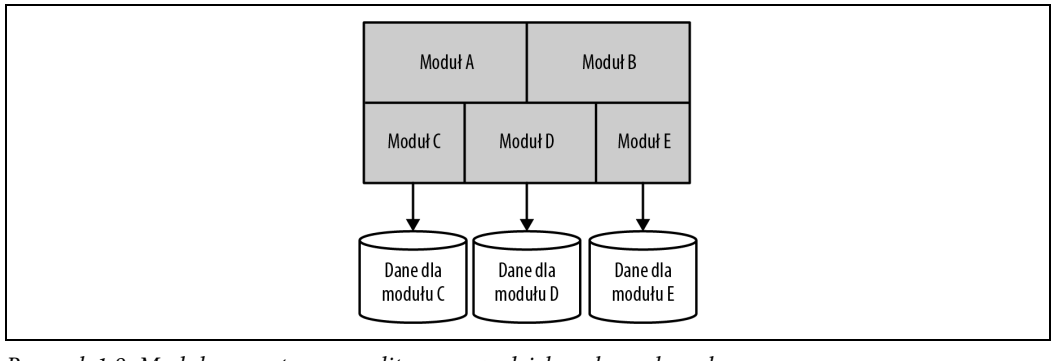
Modułowe systemy monolityczne są jedną z odmian systemów jednoprocesowych. System jednoprocesowy może obejmować odrębne moduły, które mogą być rozwijane niezależnie, ale wymagają połączenia na potrzeby instalacji. Ilustruje to rysunek 1.7. Pomysł podziału oprogramowania na moduły nie jest niczym nowym. Do związanej z tym historii wracam w dalszej części rozdziału.



Rysunek 1.7. Modułowy system monolityczny — kod wewnątrz procesu jest podzielony na moduły

W wielu organizacjach modułowy system monolityczny może okazać się doskonałym wyborem. Jeśli granice modułów są dobrze zdefiniowane, to możliwa jest równoległa praca nad nimi, a jednocześnie można uniknąć wyzwań związanych z bardziej rozproszoną architekturą opartą na mikrousługach. Dodatkowo proces instalacji modułowych systemów monolitycznych jest prostszy. Shopify jest doskonałym przykładem organizacji, w której zastosowano tę technikę zamiast podziału systemu na mikrousługi. Wygląda na to, że w tej firmie to podejście sprawdza się naprawdę dobrze[4].

Jedną z trudności związanych z modułowymi systemami monolitycznymi jest to, że w bazach danych zwykle brakuje dekompozycji stosowanej na poziomie kodu. Prowadzi to do znaczących wyzwań, z którymi możesz się zetknąć, gdy w przyszłości zechcesz podzielić system monolityczny. Widziałem, jak niektóre zespoły próbowały posunąć się o krok dalej z ideą modułowego monolitu i podzielić bazę danych w podobny sposób jak moduły. Przedstawia to rysunek 1.8. Wprowadzanie tego rodzaju zmiany w istniejącym systemie monolitycznym może być sporym wyzwaniem, nawet jeśli nie modyfikujesz kodu. Wiele wzorców omawianych w rozdziale 4. może okazać się pomocnych, jeśli zechcesz samemu wprowadzić podobne rozwiązanie.



Rysunek 1.8. Modułowy system monolityczny z podzieloną bazą danych

[4] W wypowiedzi Kirsten Westeinde w serwisie YouTube (*https://www.youtube.com/watch?v=ISYKx8sa53g*) znajdziesz przydatne informacje na temat wnioskowania, jakie doprowadziło firmę Shopify do wyboru modułowego systemu monolitycznego zamiast mikrousług.

Rozproszony system monolityczny

> System rozproszony to taki, gdzie awaria komputera, o którego istnieniu nawet nie wiedziałeś, może sprawić, że Twój własny komputer stanie się bezużyteczny[5].
>
> — Leslie Lamport

Rozproszony system monolityczny obejmuje wiele usług, przy czym z jakiegoś powodu musi być instalowany jako całość. Rozproszony system monolityczny może być zgodny z definicją architektury opartej na usługach, jednak zbyt często nie spełnia powiązanych z nią obietnic. Z mojego doświadczenia wynika, że rozproszone systemy monolityczne mają wady systemów rozproszonych, *a także* wady jednoprocesowych systemów rozproszonych, nie posiadając przy tym wystarczająco wielu zalet żadnych z nich. Napotkanie rozproszonych systemów monolitycznych w mojej pracy w dużym stopniu wpłynęło na to, że zainteresowałem się architekturą opartą na mikrousługach.

Rozproszone systemy monolityczne zwykle pojawiają się w środowiskach, w których nie przyłożono wystarczająco dużo wagi do zagadnień takich jak ukrywanie informacji i spójność funkcji biznesowych. Prowadzi to do architektur ze ścisłym powiązaniem komponentów, gdzie zmiany rozchodzą się między granicami usług i gdzie wyglądająca na prostą modyfikacja o na pozór lokalnym zasięgu może uszkodzić inne części systemu.

Systemy typu czarna skrzynka od niezależnych dostawców

Także niektóre *oprogramowanie od niezależnych dostawców* można traktować jak systemy monolityczne, które chcemy poddać dekompozycji w ramach migracji. Może to dotyczyć na przykład systemów do obsługi listy płac, systemów CRM i systemów kadrowych. Wspólną cechą takiego oprogramowania jest to, że zostało opracowane przez inne osoby i nie masz możliwości modyfikowania kodu. Możliwe, że jest to gotowe oprogramowanie, które zainstalowałeś we własnej infrastrukturze, lub produkt typu SaaS (ang. *Software-as-a-Service*). Wiele technik dekompozycji, jakie omawiam w tej książce, można zastosować nawet do systemów, których kodu nie możesz modyfikować.

Problemy związane z systemami monolitycznymi

System monolityczny (czy to jednoprocesowy, czy to rozproszony) jest często najbardziej narażony na problemy wynikające ze ścisłego powiązania, a konkretnie — ścisłego powiązania na poziomie implementacji i instalacji. Zagadnienia te omawiam szczegółowo dalej.

Gdy w jednym miejscu zaczyna pracować coraz więcej osób, zaczynają one sobie przeszkadzać. Różni programiści chcą modyfikować ten sam fragment kodu, a poszczególne zespoły chcą udostępniać funkcje w różnych momentach (lub opóźnić instalację). Nie wiadomo, kto jest właścicielem poszczególnych komponentów i kto ma podejmować decyzje. Wiele badań pokazuje problemy

[5] E-mail wysłany na listę dyskusyjną DEC SRC 28 maja 1987 roku o 12:23:29 czasu PDT (więcej informacji znajdziesz na stronie *https://www.microsoft.com/en-us/research/publication/distribution/*).

wynikające z nieporozumień co do własności komponentów[6]. Ja nazywam ten problem *konfliktami w procesie dostarczania*.

Używanie systemu monolitycznego nie oznacza, że z pewnością natrafisz na konflikty w procesie dostarczania. Podobnie używanie architektury opartej na mikrousługach nie oznacza, że nigdy nie napotkasz takich problemów. Jednak architektura oparta na mikrousługach zapewnia w systemie konkretne granice pozwalające określić własność komponentów. Daje to znacznie większą swobodę w zakresie eliminowania opisanego problemu.

Zalety systemów monolitycznych

Jednoprocesowy system monolityczny ma też jednak wiele zalet. Jego znacznie prostszy model instalacji pozwala uniknąć licznych pułapek występujących w systemach rozproszonych. Może to prowadzić do znacznie prostszych procesów pracy programistów. Ponadto to podejście pozwala znacznie uprościć monitorowanie, rozwiązywanie problemów i zadania takie jak testy typu end-to-end.

Systemy monolityczne mogą też ułatwiać ponowne wykorzystanie kodu w ramach takiego systemu. Jeśli chcesz ponownie wykorzystać kod w systemie rozproszonym, musisz zdecydować, czy chcesz skopiować kod, podzielić biblioteki, czy przenieść wspólne funkcje do usługi. W systemie monolitycznym rozwiązania są dużo prostsze i wiele osób ceni tę prostotę. Cały kod jest dostępny na miejscu, dlatego wystarczy go użyć!

Niestety, programiści zaczęli postrzegać systemy monolityczne jako coś, czego należy unikać — jako coś, co z natury rodzi problemy. Spotkałem wiele osób, dla których określenie *monolityczne* jest synonimem dla słowa *przestarzałe*. To źle. Architektura monolityczna jest jednym z modeli do wyboru i to akceptowalnym. Nie we wszystkich okolicznościach może okazać się właściwym rozwiązaniem (podobnie jak mikrousługi), ale można z niej korzystać. Jeśli wpadniemy w pułapkę ciągłego krytykowania systemów monolitycznych jako sposobu dostarczania oprogramowania, możemy zaszkodzić zarówno sobie, jak i użytkownikom naszego oprogramowania. Wady i zalety systemów monolitycznych oraz mikrousług omawiam także w rozdziale 3., gdzie analizuję także narzędzia pomagające lepiej ocenić, które z tych podejść będzie lepsze w danej sytuacji.

O powiązaniu i spójności

W kontekście definiowania granic mikrousług ważne jest zrozumienie równowagi między powiązaniem a spójnością. *Powiązanie* określa, w jakim stopniu zmiana jednego elementu wymaga modyfikacji drugiego. *Spójność* dotyczy tego, jak grupowany jest powiązany kod. Te kwestie są bezpośrednio powiązane ze sobą. Prawo Constantine'a dobrze ujmuje zależność między tymi zagadnieniami:

> Struktura jest stabilna, jeśli spójność jest wysoka, a powiązanie niskie.
>
> — Larry Constantine

[6] Jednostka Microsoft Research przeprowadziła ostatnio badania w tym obszarze. Zachęcam do zapoznania się z nimi. Na początek polecam tekst *Don't Touch My Code! Examining the Effects of Ownership on Software Quality* Christiana Birda i współpracowników (*https://www.microsoft.com/en-us/research/publication/dont-touch-my-code-examining-the-effects-of-ownership-on-software-quality/*).

Wydaje się, że jest to sensowne i przydatne spostrzeżenie. Jeśli istnieją dwa elementy ściśle powiązanego kodu, ich spójność jest niska, ponieważ powiązane funkcje są rozdzielone między oba elementy. Występuje też ścisłe powiązanie, ponieważ gdy w takim powiązanym kodzie wprowadzane są zmiany, modyfikacji wymagają oba elementy.

Jeśli struktura takiego systemu się zmieni, wprowadzenie zmian będzie kosztowne, ponieważ koszt modyfikacji wykraczających poza granice usług w systemach rozproszonych jest wysoki. Konieczność wprowadzania zmian w jednej lub kilku niezależnie instalowanych usługach (i czasem radzenia sobie z niezgodnymi ze starszym kodem zmianami w kontraktach usług) jest zwykle poważnym obciążeniem.

Problem z systemami monolitycznymi polega na tym, że zbyt często są one zarówno niespójne, jak i ściśle powiązane. Zamiast dążyć do spójności i łączyć elementy, które zwykle są modyfikowane łącznie, programiści często umieszczają razem różne niezwiązane ze sobą fragmenty kodu. Brakuje także luźnego powiązania. Jeśli programista chce zmodyfikować jakiś wiersz kodu, może to zrobić stosunkowo łatwo, ale nie może zainstalować zmiany bez istotnego wpływu na resztę systemu monolitycznego i z pewnością musi zainstalować od nowa cały system.

Stabilność systemu jest pożądana, ponieważ celem (tam, gdzie to możliwe) jest zapewnianie niezależności instalacji. Oznacza to, że chcemy móc wprowadzić zmianę w usłudze i zainstalować tę usługę w środowisku produkcyjnym *bez konieczności wprowadzania jakichkolwiek innych modyfikacji*. Aby było to możliwe, potrzebna jest stabilność używanych usług, a także stabilny kontrakt dla usług korzystających z danego kodu.

Ponieważ na temat obu tych pojęć dostępnych jest mnóstwo informacji, niemądre byłoby zbyt dokładne omawianie ich w tym miejscu. Uważam jednak, że zasługują one na podsumowanie — przede wszystkim w celu umiejscowienia tych pojęć w kontekście architektur opartych na mikrousługach. Spójność i powiązanie w bardzo dużym stopniu wpływają na sposób myślenia o architekturze opartej na mikrousługach. Nie jest to zaskoczeniem. Oba te pojęcia dotyczą oprogramowania modułowego, a czym jest architektura oparta na mikrousługach jak nie modułami komunikującymi się przez sieć i umożliwiającymi niezależne instalowanie?

Krótka historia powiązania i spójności

Pojęcia „spójność” i „powiązanie” występują w informatyce od dawna. Początkowo zostały one opisane przez Larry’ego Constantine’a w 1968 roku. Te pokrewne zagadnienia stały się ważnym aspektem myślenia o pisaniu programów komputerowych. Książki takie jak *Structured Design* Larry’ego Constantine’a i Edwarda Yourdona (wydawnictwo Prentice Hall, 1979) wpłynęły na przyszłe generacje programistów (pozycja ta była lekturą obowiązkową na moich studiach prawie 20 lat po jej pierwszym wydaniu).

Larry po raz pierwszy opisał spójność i powiązanie w 1968 roku (był to wyjątkowo owocny rok dla informatyki) na Krajowym Sympozjum Programowania Modułowego. Była to ta sama konferencja, na której swą nazwę otrzymało prawo Conwaya. W tym samym roku miały też miejsce dwie obecnie niesławne konferencje sponsorowane przez NATO, na których ważną dziedziną stała się inżynieria oprogramowania (pojęcie to zostało wcześniej zaproponowane przez Margaret H. Hamilton).

Spójność

Oto jedna z najbardziej zwięzłych znanych mi definicji spójności: „kod, który jest razem zmieniany, należy trzymać razem". Na potrzeby tej książki jest to całkiem dobra definicja. Wcześniej wspomniałem, że celem jest optymalizacja opartej na mikrousługach architektury pod kątem łatwości wprowadzania zmian w funkcjach biznesowych. Należy więc dążyć do tego, aby funkcje były pogrupowane w taki sposób, by można było wprowadzać zmiany w jak najmniejszej ilości miejsc.

Jeśli planujesz zmienić sposób zarządzania zatwierdzaniem faktur, nie chcesz chyba szukać funkcji wymagających modyfikacji w wielu różnych usługach, a następnie koordynować udostępnianie nowo zmodyfikowanych usług w celu wprowadzenia nowej funkcji. Zamiast tego należy się upewnić, że zmiana wymaga modyfikacji jak najmniejszej liczby usług, dzięki czemu koszty wprowadzenia tej zmiany będą niskie.

Powiązanie

> Ukrywanie informacji to — podobnie jak bycie na diecie — coś, co łatwiej opisać, niż zrobić.
>
> — David Parnas, *The Secret History Of Information Hiding*

Lubimy spójność, ale obawiamy się ścisłego powiązania. Im bardziej komponenty są powiązane, w tym większym stopniu konieczne jest ich wspólne modyfikowanie. Istnieją jednak różne rodzaje powiązania, a każdy z nich może wymagać innych rozwiązań.

Jeśli chodzi o klasyfikację rodzajów powiązania, prowadzonych było w przeszłości wiele prac, przede wszystkim przez Meyera, Yourdana i Constantine'a. Tu przedstawiam własne propozycje. Nie chcę przez to stwierdzić, że wcześniejsze prace były błędne. Uważam jedynie, że moja klasyfikacja bardziej ułatwia zrozumienie aspektów związanych z powiązaniem w systemach rozproszonych. Dlatego nie jest ona kompletną klasyfikacją wszystkich rodzajów powiązania.

Ukrywanie informacji

Zagadnieniem, które wciąż się pojawia w kontekście powiązania, jest technika *ukrywania informacji*. Ta technika, po raz pierwszy opisana przez Davida Parnasa w 1971 roku, to efekt jego analiz nad definiowaniem granic modułów[7].

Podstawowa idea związana z ukrywaniem informacji dotyczy oddzielenia części kodu, które często się zmieniają, od statycznych fragmentów. Chcemy, aby granice modułów były stabilne. Należy też ukrywać te części implementacji modułu, które zapewne będą częściej modyfikowane. Chodzi o to, aby można było bezpiecznie wprowadzać wewnętrzne zmiany, o ile zachowana zostanie kompatybilność modułu.

[7] Choć jako źródło często podawana jest dobrze znana praca Parnasa z 1972 roku, *On the Criteria to be Used in Decomposing Systems into Modules*, Parnas po raz pierwszy opisał ukrywanie informacji w tekście *Information Distributions Aspects of Design Methodology*, Proceedings of IFIP Congress '71, 1971.

Osobiście stosuję podejście tworzenia jak najmniejszej granicy modułu (lub mikrousługi). Gdy coś stanie się częścią interfejsu modułu, bardzo trudno jest to wycofać. Z kolei jeśli początkowo coś ukryjesz, później zawsze będziesz mógł to udostępnić.

Hermetyzacja w oprogramowaniu obiektowym to pokrewne zagadnienie, jednak w zależności od przyjętej definicji nie musi oznaczać tego samego co ukrywanie informacji. Przyjęło się, że hermetyzacja w programowaniu obiektowym oznacza łączenie elementów w kontenerze. Wyobraź sobie klasę zawierająca pola wraz z metodami operującymi tymi polami. Możesz wtedy posłużyć się widocznością elementów w definicji klasy, aby ukryć fragmenty jej implementacji.

Dłuższe omówienie historii ukrywania informacji znajdziesz w tekście Parnasa „The Secret History of Information Hiding”[8].

Powiązanie na poziomie implementacji

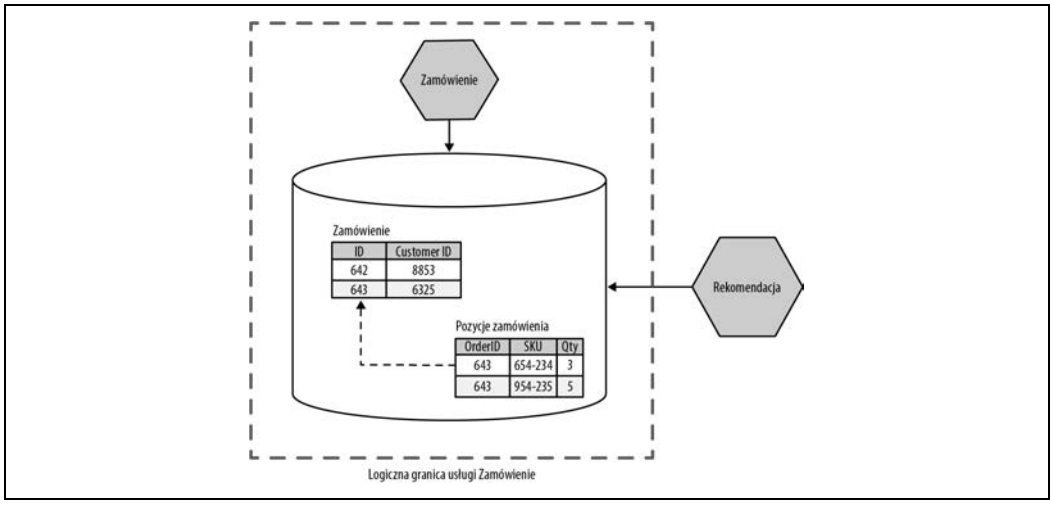
Powiązanie na poziomie implementacji to zwykle najbardziej szkodliwy rodzaj powiązania, z jakim się spotykam. Na szczęście takie powiązanie często można łatwo zmniejszyć. Powiązanie na poziomie implementacji sprawia, że A jest powiązane z B w obszarze implementacji B. Gdy zmieni się implementacja B, konieczna jest też modyfikacja w A.

Problem polega na tym, że szczegóły implementacji często są arbitralnie określane przez programistów. Istnieje wiele sposobów na rozwiązanie danego problemu. Programista może wybrać jeden z nich, a później zmienić zdanie. Taka zmiana nie powinna powodować, że jednostki korzystające z tego kodu przestaną działać (pamiętasz o niezależności instalacji?).

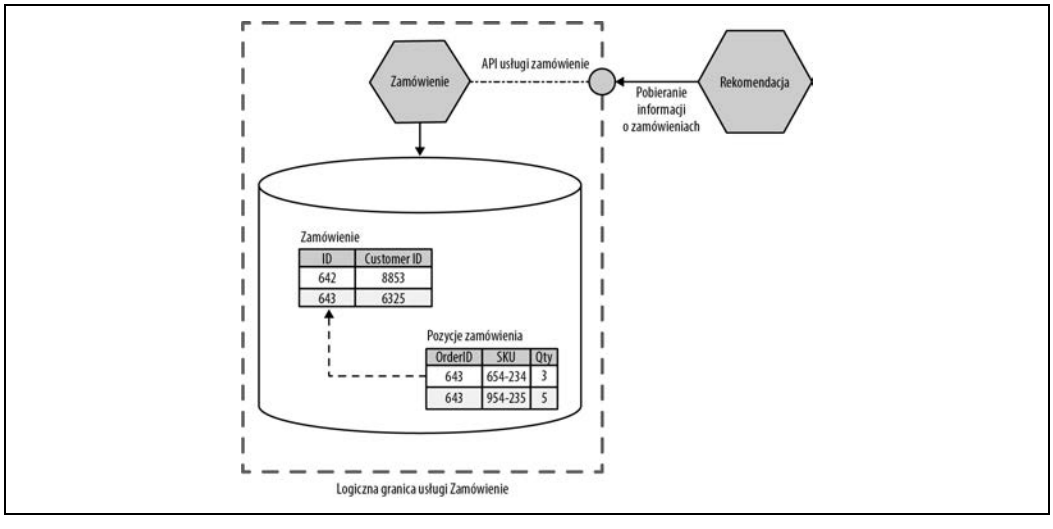
Klasyczny i często podawany przykład powiązania implementacji dotyczy współdzielonych baz danych. Na rysunku 1.9 usługa Zamówienie zawiera rekord z wszystkimi zamówieniami złożonymi w systemie. Usługa Rekomendacja proponuje na podstawie wcześniejszych zamówień płyty, które dany klient mógłby chcieć kupić. Obecnie usługa Rekomendacja bezpośrednio używa danych z bazy.

Usługa Rekomendacja wymaga informacji na temat złożonych zamówień. W pewnym sensie występuje tu nieuniknione powiązanie na poziomie *dziedziny*, które opisuję dalej. Jednak w tej konkretnej sytuacji powiązanie dotyczy też konkretnej struktury schematu, dialektu SQL-a, a może nawet zawartości wierszy. Jeśli autor usługi Zamówienie zmieni nazwy kolumn, rozdzieli tabelę Zamówienie klienta lub wprowadzi inne modyfikacje, konceptualnie usługa nadal będzie zawierać informacje o zamówieniach, jednak naruszony zostanie sposób ich pobierania przez usługę Rekomendacja. Lepszym rozwiązaniem jest ukrycie tego szczegółu implementacji, tak jak na rysunku 1.10. Teraz usługa Rekomendacja pobiera potrzebne informacje za pomocą wywołań kierowanych do API.

[8] Zobacz: David Parnas, *The Secret History of Information Hiding*, w: „SoftwarePioneers”, red. M. Broy i E. Denert (Berlin Heidelberg: Springer, 2002).

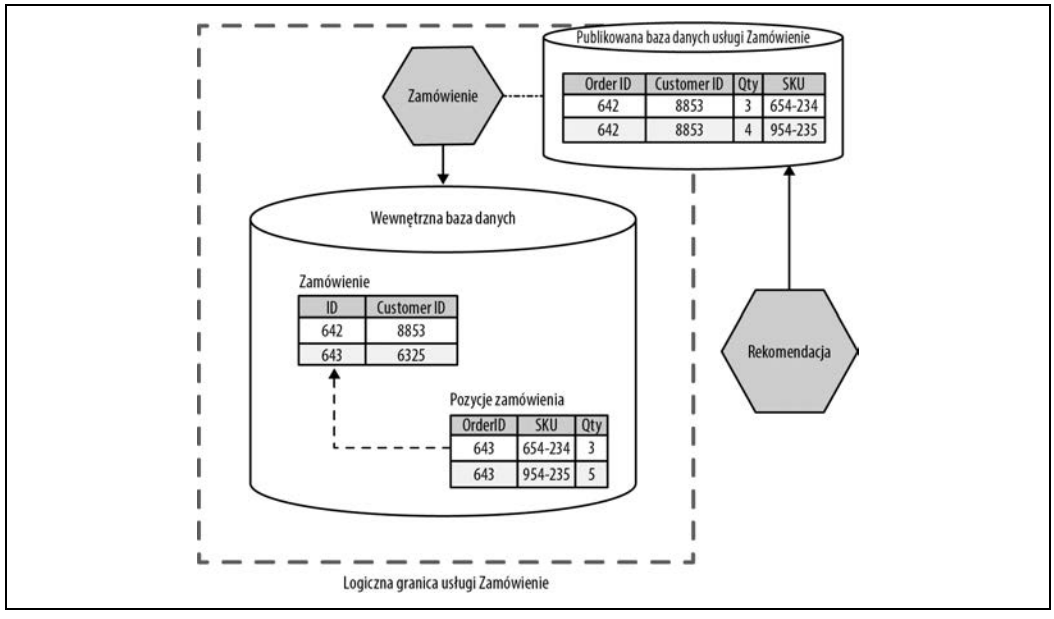


Rysunek 1.9. Usługa Rekomendacja bezpośrednio używa danych zapisanych w usłudze Zamówienie



Rysunek 1.10. Teraz usługa Rekomendacja pobiera informacje o zamówieniach za pomocą API, co pozwala ukryć wewnętrzne szczegóły implementacji

Ponadto usługa Zamówienie może też publikować zbiór danych w formie bazy przeznaczonej dla klientów do dostępu masowego. Ilustruje to rysunek 1.11. O ile usługa Zamówienie potrafi poprawnie publikować dane, zmiany wprowadzane w tej usłudze są niewidoczne dla klientów, ponieważ przestrzega ona publicznego kontraktu. Otwiera to możliwość usprawniania modelu danych udostępnianego klientom zgodnie z ich potrzebami. Wzorce tego rodzaju są opisane szczegółowo w rozdziałach 3. i 4.



Rysunek 1.11. Usługa Rekomendacja pobiera obecnie informacje o zamówieniu za pomocą udostępnianej bazy danych, której struktura różni się od wewnętrznej bazy danych

W ten sposób w obu opisanych przykładach stosowane jest ukrywanie informacji. Ukrycie bazy danych za dobrze zdefiniowanym interfejsem usługi pozwala ograniczyć zakres udostępnianych danych, a jednocześnie zmodyfikować sposób ich reprezentowania.

Inną pomocną sztuczką jest myślenie „od zewnątrz do środka" w trakcie definiowania interfejsu usługi. Interfejs usługi należy rozwijać, myśląc najpierw z perspektywy klienta, a następnie zastanawiając się, jak zaimplementować dany kontrakt. Inne podejście (które, niestety, wedle moich obserwacji występuje zbyt często) polega na zastosowaniu odwrotnej kolejności. Zespół pracujący nad usługą bierze model danych lub inny wewnętrzny szczegół implementacji, a następnie zastanawia się, jak udostępnić go zewnętrznym jednostkom.

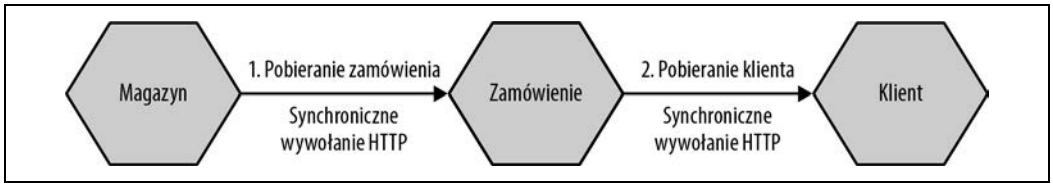
Stosując myślenie „od zewnątrz do środka", najpierw należy zadać pytanie: „czego potrzebują klienci mojej usługi?". Nie chodzi mi o to, abyś *sobie* zadał pytanie o potrzeby klientów — powinieneś zapytać o zdanie ludzi, którzy będą kierować wywołania do Twojej usługi!

Interfejsy udostępniane przez mikrousługę traktuj jak interfejs użytkownika. Zastosuj myślenie „od zewnątrz do środka", aby zbudować projekt interfejsu we współpracy z ludźmi, którzy będą kierować wywołania do Twojej usługi.

Kontrakt usługi opisujący komunikację z zewnętrznym światem traktuj jak interfejs użytkownika. Gdy projektujesz interfejs użytkownika, pytasz użytkowników o to, czego potrzebują, i iteracyjnie dopracowujesz projekt razem z klientami. Kontrakt usługi powinieneś kształtować w ten sam sposób. Pomijając nawet to, że uzyskasz w ten sposób usługę łatwiejszą w użytku dla klientów, może to pomóc Ci w rozdzieleniu zewnętrznego kontraktu od wewnętrznej implementacji.

Powiązanie na poziomie czasu

Powiązanie na poziomie czasu jest zagadnieniem dotyczącym wykonywania kodu, związanym z jednym z najważniejszych wyzwań obsługi wywołań synchronicznych w środowisku rozproszonym. Powiązanie na poziomie czasu występuje, gdy komunikat jest wysyłany i sposób jego obsługi jest zależny od czasu. Może to brzmieć dziwnie, dlatego przyjrzyj się konkretnemu przykładowi pokazanemu na rysunku 1.12.



Rysunek 1.12. Trzy usługi korzystające z wywołań synchronicznych, aby wykonać operację, są powiązane na poziomie czasu

Widać tu, że usługa Magazyn kieruje synchroniczne wywołanie HTTP do działającej na późniejszym etapie usługi Zamówienie, aby pobrać potrzebne informacje na temat zamówienia. Aby obsłużyć to żądanie, usługa Zamówienie musi pobrać informacje z usługi Klient, także za pomocą synchronicznego wywołania HTTP. Aby cała ta operacja została ukończona, wszystkie usługi (Magazyn, Zamówienie i Klient) muszą być aktywne i dostępne. Są one powiązane na poziomie czasu.

Ten problem można byłoby ograniczyć na kilka sposobów. Można rozważyć zastosowanie pamięci podręcznej. Jeśli usługa Zamówienia będzie zapisywać w pamięci podręcznej potrzebne jej informacje od usługi Klient, będzie mogła w niektórych sytuacjach uniknąć powiązania na poziomie czasu z dalszymi usługami. Można też pomyśleć o asynchronicznym przekazywaniu żądań, na przykład z użyciem brokera komunikatów. Pozwala to przesyłać komunikaty do dalszych usług i obsługiwać te komunikaty wtedy, gdy dana usługa będzie dostępna.

Kompletne omawianie typów komunikacji między usługami wykracza poza zakres tej książki. Omawiam je bardziej szczegółowo w rozdziale 4. książki *Budowanie mikrousług*.

Powiązanie na poziomie instalacji

Wyobraź sobie jeden proces obejmujący wiele statycznie dołączanych modułów. W jednym z tych modułów modyfikowany jest jeden wiersz kodu, po czym należy zainstalować kod z tą zmianą. Aby to zrobić, trzeba zainstalować cały monolityczny system — nawet moduły, które nie zostały zmodyfikowane. Wszystko trzeba zainstalować razem, występuje więc *powiązanie na poziomie instalacji*.

Powiązanie na poziomie instalacji może być wymuszone, tak jak w przykładzie z procesem ze statycznym dołączaniem, ale może być też kwestią wyboru i wynikać ze stosowania technik takich jak „pociąg wersji dystrybucyjnych” (ang. *release train*). W tym podejściu na początku opracowywany jest harmonogram udostępniania, zwykle z powtarzającymi się cyklami. Gdy następuje data udostępnienia oprogramowania, instalowane są wszystkie zmiany wprowadzone od czasu poprzedniej instalacji. Dla niektórych osób pociąg wersji może być wygodną techniką, ja jednak zdecydowanie preferuję traktowanie jej jako kroku przejściowego do odpowiednich metod udostępniania opro-

gramowania na żądanie (pociąg wersji nie powinien być rozwiązaniem ostatecznym). Pracowałem w organizacjach, które instalowały jednocześnie wszystkie usługi z systemu w ramach pociągu dystrybucyjnego, w ogóle nie zastanawiając się nad tym, czy poszczególne usługi wymagają zmian.

Instalowanie czegoś związane jest z ryzykiem. Istnieje wiele sposobów na ograniczenie takich zagrożeń i jednym z nich jest modyfikowanie tylko tego, co wymaga zmian. Jeśli można ograniczyć powiązanie na poziomie instalacji, na przykład w wyniku podziału większych procesów na niezależnie instalowane mikrousługi, można zmniejszyć ryzyko każdej instalacji, ograniczając jej zakres.

Mniejsze instalowane wersje oznaczają mniejsze ryzyko. Mniej rzeczy może się nie powieść. Jeśli już wystąpią problemy, łatwiej można stwierdzić, gdzie wystąpił błąd i jak go naprawić, bo zmiany mają mniejszy zakres. Znajdowanie sposobów zmniejszania wielkości instalowanych wersji związane jest z istotą ciągłego dostarczania, gdzie istotne są szybkie informacje zwrotne i udostępnianie oprogramowania na żądanie[9]. Im mniejszy zakres instalowanej wersji, tym łatwiej i bezpieczniej można ją udostępnić i tym szybciej uzyskiwane są informacje zwrotne. Sam zainteresowałem się mikrousługami dzięki wcześniejszemu skupieniu na ciągłym dostarczaniu. Szukałem architektur, które ułatwią wdrażanie ciągłego dostarczania.

Aby ograniczyć powiązanie na poziomie instalacji, oczywiście nie trzeba korzystać z mikrousług. Środowiska uruchomieniowe takie jak Erlang umożliwiają instalowanie nowych wersji modułów w działającym procesie bez konieczności zatrzymywania go. Możliwe, że w przyszłości więcej osób będzie miało dostęp do takich możliwości w używanych na co dzień zestawach technologii[10].

Powiązanie na poziomie dziedziny

W systemie składającym się z wielu niezależnych usług muszą występować jakieś interakcje między jednostkami. W architekturze opartej na mikrousługach powstaje więc *powiązanie na poziomie dziedziny*. Interakcje między usługami odpowiadają interakcjom w danej dziedzinie. Jeśli chcesz złożyć zamówienie, musisz wiedzieć, jakie elementy znajdowały się w koszyku klienta. Jeżeli chcesz wysłać produkt, musisz znać adres dostawy. W architekturze opartej na mikrousługach te informacje z definicji znajdują się w różnych usługach.

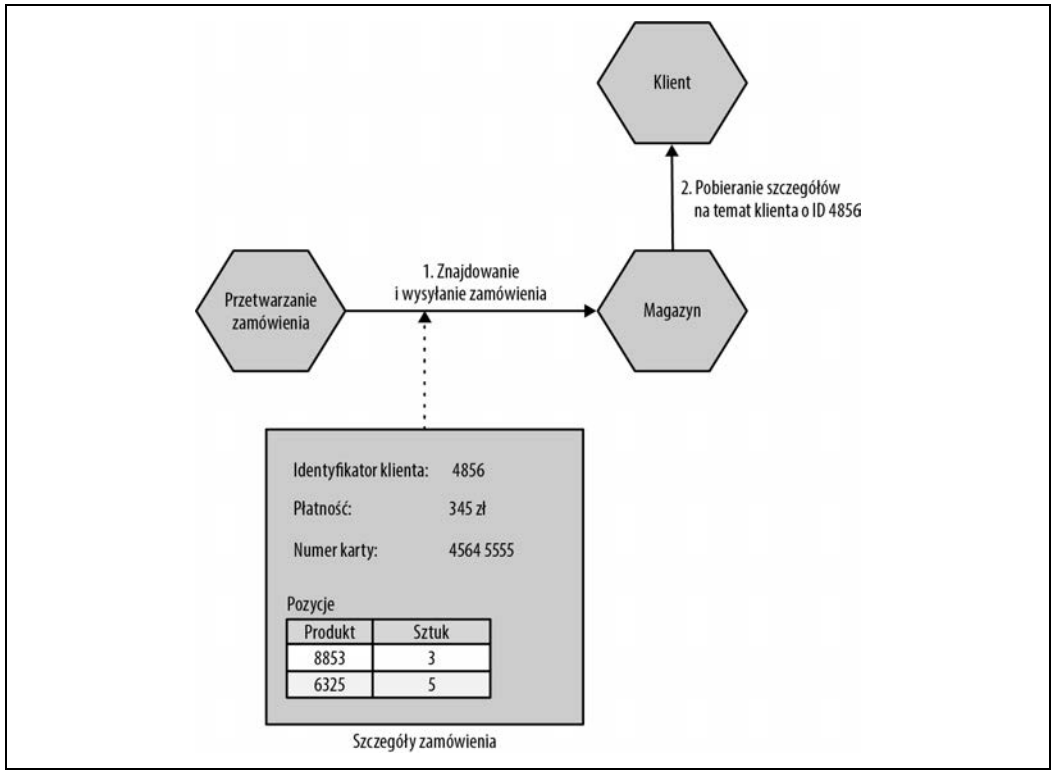
W ramach konkretnego przykładu zastanów się nad firmą Music Corp. Ma ona magazyn, gdzie składowane są produkty. Gdy klient zamawia płyty CD, pracownicy magazynu muszą ustalić, jakie produkty należy znaleźć i spakować oraz gdzie wysłać paczkę. Dlatego pracownikom magazynu trzeba przekazać informacje na temat zamówienia.

---

[9] Więcej szczegółów znajdziesz w: Jez Humble i David Farley, *Ciągłe dostarczanie oprogramowania. Automatyzacja kompilacji, testowania i wdrażania* (Helion, 2015).

[10] Dziesiąta reguła Greenspuna brzmi tak: „Każdy wystarczająco skomplikowany program w języku C lub Fortran obejmuje doraźnie opracowaną, nieformalną, pełną błędów i powolną implementację połowy języka Common Lisp". Na podstawie tej reguły powstał nowszy żart: „Każda architektura oparta na mikrousługach obejmuje częściową niepoprawną nową implementację Erlanga". Uważam, że jest w tym dużo prawdy.

Na rysunku 1.13 pokazany jest ilustrujący to przykład. Usługa Przetwarzanie zamówienia przesyła wszystkie szczegóły na temat zamówienia do usługi Magazyn, która następnie uruchamia pakowanie produktów. W ramach tej operacji usługa Magazyn przesyła identyfikator klienta, aby pobrać informacje na jego temat z odrębnej usługi Klient. Dzięki temu usługa Magazyn będzie wiedzieć, jak powiadomić klienta o wysłaniu zamówienia.

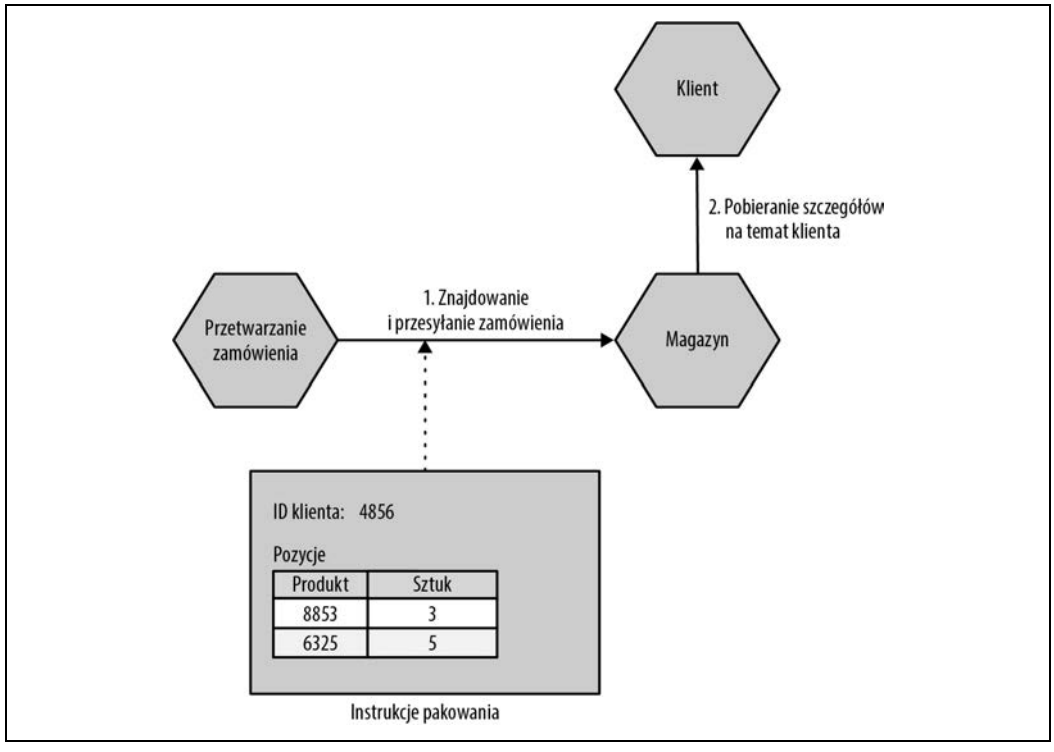


Rysunek 1.13. Zamówienie jest przesyłane do magazynu, aby umożliwić spakowanie produktów

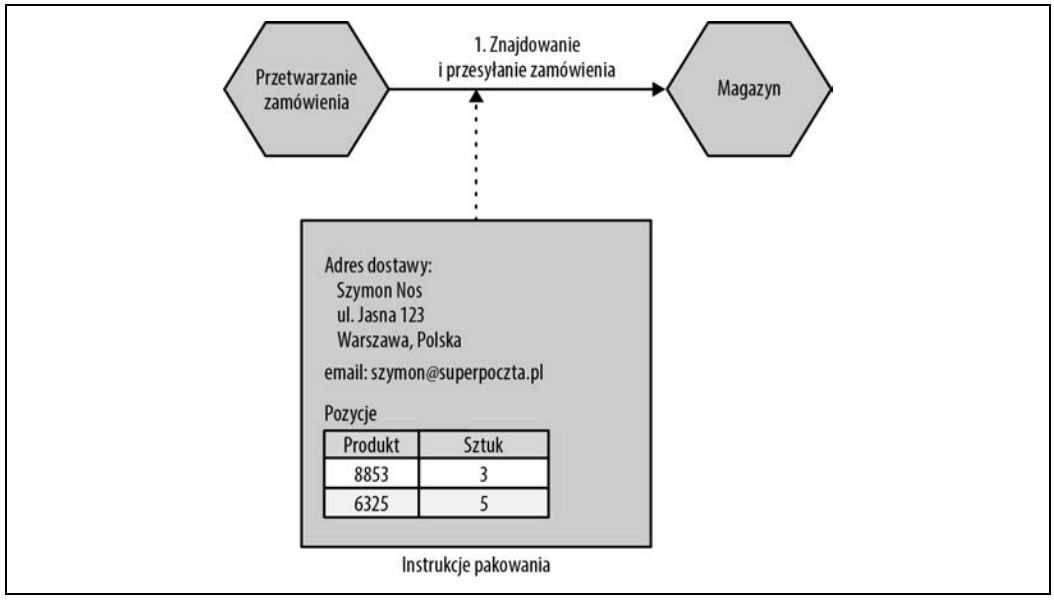
W tym scenariuszu do magazynu przekazywane jest całe zamówienie, co może nie mieć sensu. Magazyn potrzebuje tylko informacji o tym, co spakować i gdzie to wysłać. Informacje o cenie produktu nie są niezbędne (jeśli do paczki trzeba dodać fakturę, można ją przekazać w formie wygenerowanego dokumentu PDF). Występuje też problem ze zbyt szerokim dostępem do chronionych informacji. Jeśli udostępniane jest kompletne zamówienie, szczegółowe informacje o karcie kredytowej mogą być udostępniane usługom, które ich nie potrzebują.

Zamiast takiego rozwiązania można utworzyć nowy komponent z omawianej dziedziny — Instrukcje pakowania — zawierający tylko informacje potrzebne usłudze Magazyn. Ilustruje to rysunek 1.14. Jest to kolejny przykład ukrywania informacji.

Można jeszcze bardziej ograniczyć poziom powiązania i sprawić, by usługa Magazyn nie musiała nawet wiedzieć o istnieniu usługi Klient. Zamiast tego można byłoby udostępniać wszystkie potrzebne szczegóły w usłudze Instrukcje pakowania, tak jak na rysunku 1.15.



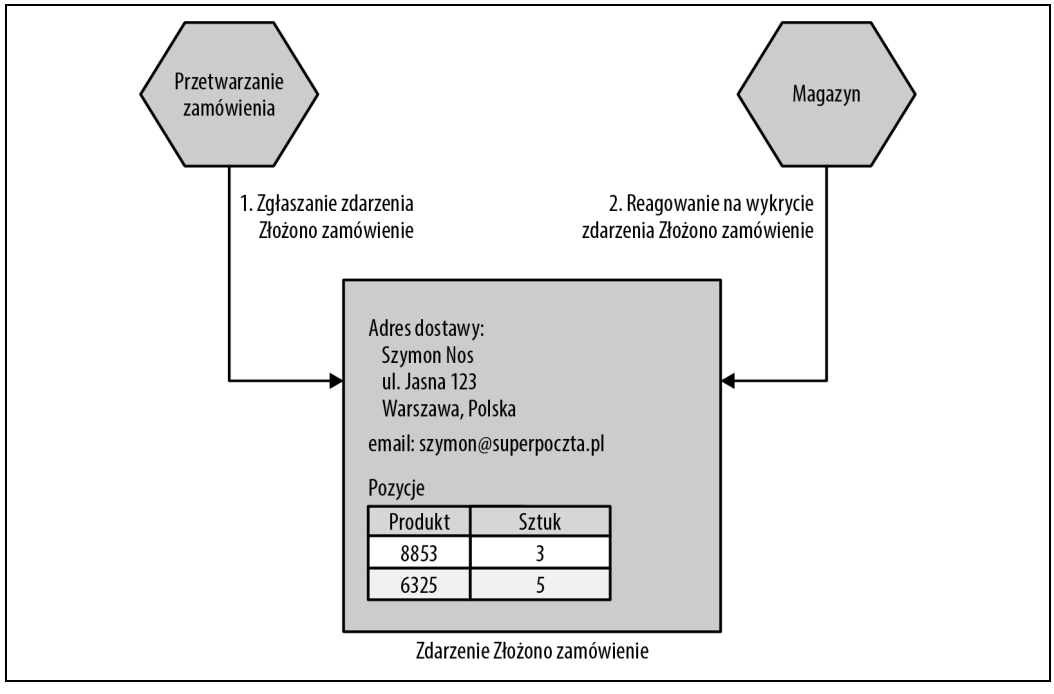
Rysunek 1.14. Używanie usługi Instrukcje pakowania, aby ograniczyć ilość informacji przesyłanych do usługi Magazyn



Rysunek 1.15. Umieszczenie dodatkowych informacji w usłudze Instrukcje pakowania pozwala uniknąć konieczności wywoływania usługi Klient

Aby to podejście zadziałało, w pewnym momencie usługa Przetwarzanie zamówienia musi uzyskać dostęp do usługi Klient, by móc wygenerować instrukcje pakowania. Prawdopodobne jest jednak, że usługa Przetwarzanie zamówienia i tak z innych powodów będzie potrzebować informacji o kliencie, dlatego nie stanowi to problemu. Opisany proces przesyłania instrukcji pakowania sugeruje, że usługa Przetwarzanie zamówienia kieruje wywołanie API do usługi Magazyn.

Inna możliwość to generowanie przez usługę Przetwarzanie zamówienia zdarzenia na potrzeby usługi Magazyn, tak jak na rysunku 1.16. Generowanie zdarzenia używanego przez usługę Magazyn skutkuje odwróceniem zależności. Teraz to nie usługa Przetwarzanie zamówienia zależy od usługi Magazyn, aby móc wysłać zamówienie, ale usługa Magazyn oczekuje na zdarzenia od usługi Przetwarzanie zamówienia. Oba te podejścia mają zalety, a wybór jednego z nich zapewne będzie wynikać z lepszego zrozumienia interakcji między logiką usługi Przetwarzanie zamówienia i funkcjami ukrytymi w usłudze Magazyn. W tym może pomóc modelowanie dziedziny, którym to tematem zajmę się w dalszej kolejności.



Rysunek 1.16. Generowanie zdarzenia, które może zostać odebrane przez usługę Magazyn i zawiera tylko tyle informacji, aby możliwe było spakowanie i wysłanie zamówienia

Aby usługa Magazyn mogła wykonywać jakiekolwiek zadania, potrzebne są jakieś informacje o zamówieniu. Nie da się uniknąć tego poziomu powiązania na poziomie dziedziny. Jednak jeśli starannie przemyślisz, co i jak jest udostępniane, nadal będziesz mógł ograniczyć poziom powiązania.

Tylko tyle DDD, ile potrzeba

Wyjaśniłem już, że modelowanie usług na podstawie dziedziny biznesowej ma istotne zalety w kontekście architektury opartej na mikrousługach. Jak opracować taki model? Odpowiedzią na to pytanie jest podejście DDD (ang. *domain-driven design*).

Dążenie do tego, by programy lepiej reprezentowały rzeczywisty świat, w którym te programy będą działać, nie jest niczym nowym. Języki obiektowe, takie jak Simula, opracowano po to, aby umożliwiały modelowanie rzeczywistych dziedzin. Jednak aby wcielić tę ideę w życie, potrzeba czegoś więcej niż możliwości języka programowania.

W książce *Domain-Driven Design*[11] Erica Evansa zaprezentowano szereg ważnych pomysłów, które pomagają lepiej reprezentować dziedzinę problemu w programach. Kompletne omawianie tych pomysłów wykracza poza zakres tej książki, przedstawiam jednak krótki przegląd najważniejszych idei związanych z architekturami opartymi na mikrousługach.

Agregat

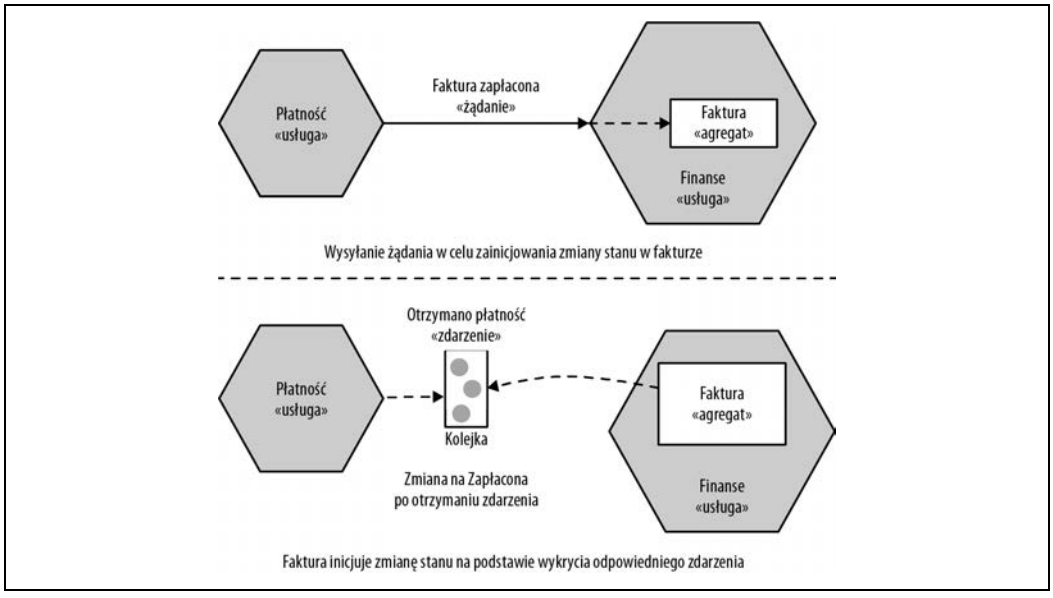
W DDD *agregat* jest dość zagadkową koncepcją o wielu różnych definicjach. Czy jest to dowolna kolekcja obiektów? A może najmniejsza jednostka, jaką należy pobierać z bazy danych? Technika, która zawsze sprawdzała się w moim przypadku, polega na rozpoczęciu analizy agregatu jako reprezentacji obiektu z danej dziedziny, na przykład zamówienia, faktury, produktu itd. Agregaty zwykle mają określony cykl życia, dlatego nadają się do implementowania jako maszyny stanowe. Agregaty należy traktować jak niezależne jednostki. Trzeba się upewnić, że kod, który obsługuje zmiany stanu agregatu, znajduje się w jednej jednostce razem z tym stanem.

Jeśli chodzi o związek między agregatami a mikrousługami, to jedna mikrousługa powinna obsługiwać cykl życia i składowanie danych jednego lub kilku rodzajów agregatów. Jeśli mechanizm z innej usługi chce zmodyfikować jeden z tych agregatów, musi albo bezpośrednio zażądać takiej zmiany, albo spowodować, aby agregat sam zareagował na inne zdarzenia w systemie i zainicjował zmianę swojego stanu. Przykłady są pokazane na rysunku 1.17.

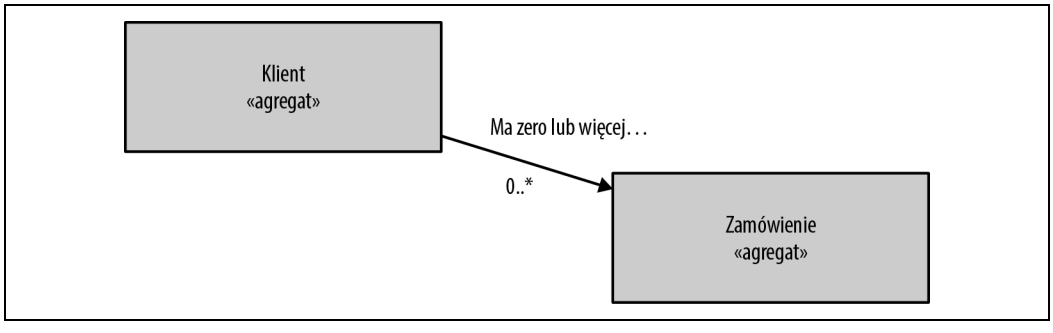
Najważniejszą rzeczą do zrozumienia jest tu to, że jeśli zewnętrzna jednostka zażąda zmiany stanu agregatu, agregat może odmówić. Najlepiej implementować agregaty w taki sposób, aby niedozwolone zmiany stanu były niemożliwe.

Agregaty mogą pozostawać w relacji z innymi agregatami. Na rysunku 1.18 pokazany jest agregat Klient powiązany z zerem lub większą liczbą Zamówień. Zdecydowałem się utworzyć w modelu Klienta i Zamówienie jako odrębne agregaty, którymi mogą zarządzać różne usługi.

[11] Eric Evans, *Domain-Driven Design. Zapanuj nad złożonym systemem informatycznym* (Helion, , 2015).



Rysunek 1.17. Różne sposoby, w jakie usługa Płatność może uruchomić zmianę stanu na Zapłacona w agregacie Faktura



Rysunek 1.18. Jeden agregat Klient może być powiązany z zerem lub jakąś liczbą agregatów Zamówienie

Istnieje wiele sposobów podziału systemu na agregaty. Niektóre rozwiązania są wysoce subiektywne. Możesz na przykład w celu zwiększenia wydajności lub ułatwienia implementacji zdecydować się w przyszłości zmienić formę agregatów. Jednak uważam, że początkowo kwestie implementacji są drugorzędne. Na początku — zanim istotne staną się inne czynniki — pozwalam, aby to modele mentalne użytkowników systemu wyznaczały wstępny projekt. W rozdziale 2. przedstawiam zespołowe ćwiczenie Event Storming, które pomaga tworzyć modele dziedziny z pomocą nieprogramistów.

Ograniczony kontekst

Ograniczony kontekst (ang. *bounded context*) zwykle reprezentuje duży ograniczony obszar organizacji. W ramach tego obszaru trzeba wykonywać jasno określone zadania. Ten opis jest dość mętny, dlatego przyjrzyj się innemu konkretnemu przykładowi.

W magazynie firmy Music Corp ciągle coś się dzieje — pracownicy zarządzają wysyłanymi zamówieniami (i rzadkimi zwrotami), przyjmują nowe produkty, urządzają wyścigi wózków widłowych itd. Pracownicy działu finansów może mniej lubią zabawę, ale też mają ważne zadania w organizacji — obsługują płatności, płacą za dostawy itd.

Ograniczony kontekst pozwala ukryć szczegóły implementacji. Są one kwestiami wewnętrznymi. Na przykład model używanych wózków widłowych jest mało istotny dla wszystkich osób, które nie pracują w magazynie. Te wewnętrzne zagadnienia należy ukryć przed światem zewnętrznym, który ani nie musi ich znać, ani nie powinien się nimi interesować.

W implementacji ograniczony kontekst obejmuje jeden lub więcej agregatów. Niektóre agregaty mogą być udostępniane poza ograniczonym kontekstem. Inne mogą być ukryte i dostępne tylko wewnętrznie. Ograniczone konteksty, podobnie jak agregaty, mogą być powiązane z innymi ograniczonymi kontekstami. Po odwzorowaniu na usługi te zależności stają się zależnościami między usługami.

Odwzorowywanie agregatów i ograniczonych kontekstów na mikrousługi

Zarówno agregaty, jak i ograniczone konteksty tworzą jednostki spójności z dobrze zdefiniowanymi interfejsami w ramach większego systemu. Agregat jest niezależną maszyną stanową, która w systemie dotyczy jednego zagadnienia z dziedziny. Ograniczony kontekst reprezentuje kolekcję powiązanych agregatów ze zdefiniowanym interfejsem dla zewnętrznego świata.

Dlatego i agregaty, i ograniczony kontekst mogą sprawdzić się jako granice usług. Na początku, jak już wspomniałem, warto moim zdaniem ograniczyć liczbę używanych usług. Dlatego uważam, że prawdopodobnie powinieneś tworzyć usługi obejmujące cały ograniczony kontekst. Gdy już przyzwyczaisz się do nowego podejścia i zdecydujesz się rozbić usługi na mniejsze części, rozważ ich podział na podstawie granic agregatów.

Sztuczka polega tu na tym, że nawet jeśli zdecydujesz się w przyszłości rozbić usługę modelującą cały ograniczony kontekst na mniejsze usługi, nadal będziesz mógł ukryć tę decyzję przed światem zewnętrznym — na przykład udostępniając klientom bardziej ogólny interfejs. Można przyjąć, że decyzja podziału usługi na mniejsze części dotyczy implementacji, dlatego — jeśli jest to wykonalne — można ją ukryć.

Dalsza lektura

Dokładne omówienie podejścia DDD byłoby przydatne, jednak zagadnienie to wykracza poza zakres tej książki. Jeśli chcesz dowiedzieć się więcej na ten temat, zachęcam do lektury pierwotnej pracy Erica Evansa, *Domain Driven Design*, lub książki *Domain-Driven Design Distilled* Vaughna Vernona[12].

[12] Zobacz: Vaughn Vernon, *DDD. Kompendium wiedzy* (Helion, 2018).

Podsumowanie

W tym rozdziale napisałem, że mikrousługi to niezależnie instalowane usługi modelowane na podstawie dziedziny biznesowej. Mikrousługi komunikują się między sobą przez sieć. Należy na podstawie zasad ukrywania informacji i podejścia DDD tworzyć mające stabilne granice usługi, nad którymi łatwiej jest pracować osobno. Należy też zrobić to, co możliwe, aby ograniczyć powiązanie na różnych poziomach.

Przedstawiłem też krótką historię pochodzenia mikrousług, a nawet znalazłem miejsce na przyjrzenie się małemu wycinkowi dużej ilości wcześniejszych prac, na bazie których opracowano mikrousługi. Ponadto pokrótce opisałem wybrane wyzwania związane z architekturami opartymi na mikrousługach. To zagadnienie omawiam szczegółowo w następnym rozdziale, gdzie wyjaśniam też, jak zaplanować przejście do architektury opartej na mikrousługach. Opisuję też wskazówki pomagające ocenić, czy mikrousługi w ogóle będą dla Ciebie właściwym rozwiązaniem.

Dążysz do sukcesu? Obierz kurs na mikrousługi!

Mikrousługi są relatywnie świeżą koncepcją w świecie systemów IT, mimo to coraz więcej organizacji decyduje się na wdrażanie opartej na nich architektury. Uznaje się, że zapewnia ona wówczas lepszą skalowalność, łatwość łączenia poszczególnych niezależnych elementów, a przede wszystkim możliwość szybszej reakcji na zmiany i skuteczniejsze wdrażanie nowych funkcjonalności. Zanim jednak organizacja da się skusić tymi obietnicami, powinna gruntownie przeanalizować swoją sytuację i decyzję o ewentualnej migracji systemu oprzeć na racjonalnych przesłankach. Konieczne jest również opracowanie planu takiego przejścia, zwłaszcza jeśli nie można sobie pozwolić na dłuższe przestoje w działalności.

To wyczerpujący poradnik dla inżynierów, którzy stoją przed wyzwaniem przekształcenia monolitycznego systemu w architekturę opartą na mikrousługach bez przerywania funkcjonowania firmy. Książka jest przeznaczona dla organizacji, które muszą płynnie zmienić istniejący system, a nie zbudować go od nowa. Zawiera wiele cennych wskazówek odnoszących się do celowości samej migracji oraz przedstawia liczne scenariusze i strategie przekształcania: od etapu planowania aż po dekompozycję aplikacji i baz danych. Znalazł się tu zestaw sprawdzonych wzorców i technik wraz z omówieniem sytuacji, w jakich można je bezpiecznie zastosować. Nie zabrakło ważnych szczegółów związanych z wzorcami refaktoryzacji architektury czy problematyki naruszeń integralności w wyniku podziału baz danych.

W książce między innymi:

- podstawowe koncepcje związane z mikrousługami
- ocena przydatności mikrousług w konkretnych sytuacjach
- planowanie wdrażania architektury opartej na mikrousługach
- wzorce migracji, dekompozycja aplikacji i inne zagadnienia techniczne
- wykrywanie i rozwiązywanie problemów związanych z mikrousługami

Sam Newman jest programistą, architektem, autorem książek technicznych i doświadczonym prelegentem. Pracował dla firm z różnych branż na całym świecie. Jest freelancerem, specjalizuje się głównie w systemach opartych na chmurze, ciągłym dostarczaniu i mikrousługach. Typowy niespokojny duch, wciąż angażuje się w różne przedsięwzięcia. Mieszka we wschodniej części hrabstwa Kent.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-6723-4
9 788328 367234
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 54,90 zł