O'REILLY®

Złożone zagadnienia architektury oprogramowania

Jak analizować kompromisy i podejmować trudne decyzje

Neal Ford
Mark Richards
Pramod Sadalage
Zhamak Dehghani

Helion

Tytuł oryginału: Software Architecture: The Hard Parts:
Modern Trade-Off Analyses for Distributed Architectures

Tłumaczenie: Piotr Pilch

ISBN: 978-83-283-9527-5

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/zlozag.zip

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Co się dzieje przy braku „najlepszych praktyk”?

Dlaczego specjalista, taki jak architekt oprogramowania, występuje na konferencji lub pisze książkę? Dzieje się tak, ponieważ takie osoby poznały to, co kolokwialnie jest określane mianem „najlepszych praktyk”. Ten termin jest tak bardzo nadużywany, że osoby posługujące się nim w coraz większym stopniu doświadczają gwałtownej reakcji odbiorców. Niezależnie od terminu specjaliści piszą książki, gdy opracują nowatorskie rozwiązanie ogólnego problemu i chcą się nim podzielić z szerszą grupą odbiorców.

Co jednak w przypadku sporego zbioru problemów, które nie mają dobrych rozwiązań? Z architekturą oprogramowania związane są całe grupy problemów niemające żadnych ogólnych, dobrych rozwiązań. W przypadku tych kwestii dostępny jest raczej nieuporządkowany zestaw kompromisów zestawiony z (prawie) tak samo nieuporządkowanym innym zestawem.

Projektanci oprogramowania zdobywają znakomite umiejętności związane z wyszukiwaniem w sieci rozwiązań bieżącego problemu. Jeśli np. muszą się dowiedzieć, jak skonfigurować konkretne narzędzie w używanym środowisku, eksperckie możliwości korzystania z wyszukiwarki Google pozwalają znaleźć odpowiedź.

Nie dotyczy to jednak architektów.

W przypadku architektów wiele problemów powoduje unikalne wyzwania, gdyż łączą one samo środowisko i okoliczności występujące w danej organizacji. Jakie są szanse na to, że ktoś miał do czynienia z takim samym scenariuszem, *a ponadto* opisał to na blogu lub we wpisie zamieszczonym w serwisie Stack Overflow?

Architekci mogą się zastanawiać, dlaczego dostępnych jest tak niewiele książek o architekturze w porównaniu z zagadnieniami technicznymi, takimi jak środowiska, interfejsy API itp. Architekci rzadko doświadczają typowych problemów, a ponadto muszą się ciągle borykać z podejmowaniem decyzji w zupełnie nowych sytuacjach. Z punktu widzenia architektów każdy problem to płatek śniegu. W wielu przypadkach trudna kwestia jest nowością nie tyle w skali konkretnej organizacji, ale raczej w skali globalnej. Nie ma żadnych książek lub materiałów z sesji konferencyjnych, które zapewnią rozwiązanie tych zagadnień!

Architekci nie powinni nieustannie szukać rozwiązań napotkanych problemów w postaci cudownego środka. Takie rozwiązania są obecnie równie rzadkie jak w 1986 r., gdy Fred Brooks ujął to w następujący sposób:

> *Nie istnieje żadne pojedyncze dokonanie, czy to w postaci technologii, czy techniki zarządzania, które samo w sobie stanowiłoby obietnicę wzrostu w ciągu dekady choćby o jeden rząd wielkości (dziesięciokrotnie) dotyczącego wydajności, niezawodności i prostoty.*
>
> — z książki *No Silver Bullet* Freda Brooksa

Ponieważ niemal każdy problem stawia nowe wyzwania, prawdziwa rola architekta objawia się w możliwości obiektywnego ustalania i oceniania zestawu kompromisów po obu stronach ważnej decyzji tak, aby była ona jak najlepsza. Autorzy nie dyskutują o „najlepszych praktykach" (ani w książce, ani w codziennym życiu), ponieważ słowo „najlepsze" wskazuje na to, że architekt był w stanie zmaksymalizować w ramach projektu wszystkie możliwe rywalizujące ze sobą czynniki. Zamiast tego autorzy mają poniższą żartobliwą radę.

Nie próbuj szukać *najlepszego* projektu w architekturze oprogramowania. Zamiast tego usiłuj uzyskać *najmniej niekorzystną* kombinację kompromisów.

Często najlepszy projekt, jaki architekt może utworzyć, jest najmniej niekorzystnym zbiorem kompromisów. Żadna pojedyncza cecha architektury nie wyróżnia się tak jak wtedy, gdy istnieje samodzielnie, ale powodzeniu projektu sprzyja równowaga wszystkich nakładających się na siebie cech.

Nasuwa się tutaj na myśl następujące pytanie: „Jak architekt może *znaleźć* najmniej niekorzystną kombinację kompromisów (i skutecznie ją udokumentować)?". W książce jest mowa głównie o podejmowaniu decyzji, co ma umożliwiać architektom lepsze decyzje przy konfrontacji z zupełnie nowymi sytuacjami.

Skąd określenie „trudne kwestie"?

Dlaczego książce nadano tytuł *Złożone zagadnienia architektury oprogramowania*? Tak naprawdę słowo „złożone" pełni w tytule podwójną funkcję. Po pierwsze, słowo to przywodzi na myśl słowo *trudne*. Architekci nieustannie zajmują się złożonymi problemami, z którymi dosłownie (i w przenośni) nikt wcześniej nie miał do czynienia. Obejmuje to liczne decyzje dotyczące technologii, które powodują długoterminowe konsekwencje mające wpływ na środowisko interpersonalne i polityczne, gdzie decyzja musi zostać podjęta.

Po drugie, słowo *trudne* (ang. *hard*) może się wiązać z zapewnianiem czegoś *trwałego*. Tak jak w przypadku rozdzielenia terminów anglojęzycznych *hardware* (sprzęt) i *software* (oprogramowanie), anglojęzyczne słowo *hard* powinno oznaczać, że coś zmienia się znacznie rzadziej, gdyż zapewnia fundament dla czegoś bardziej zmiennego, co identyfikuje anglojęzyczne słowo *soft* (miękkie). Podobnie architekci dyskutują na temat rozróżniania pojęcia *architektury* i *projektu*. Pierwsze z nich ma znaczenie strukturalne, natomiast drugie wskazuje coś znacznie łatwiej zmienianego. A zatem w książce jest mowa o fundamentalnych elementach architektury.

Sama definicja architektury oprogramowania przyczyniła się do wielogodzinnych, bezproduktywnych dyskusji prowadzonych przez związane z tym zagadnieniem osoby. Jedna z ulubionych, żartobliwych definicji brzmi następująco: „architektura oprogramowania to *rzecz*, którą później trudno zmienić". Właśnie tej *rzeczy* poświęcono książkę.

Ponadczasowa rada dotycząca architektury oprogramowania

Ekosystem projektowania oprogramowania nieustannie się zmienia i rozrasta w chaotyczny sposób. Zagadnienia, które kilka lat temu cieszyły się największą popularnością, zostały włączone do ekosystemu i znikły albo zostały zastąpione przez coś innego lub lepszego. Na przykład 10 lat temu dominującym wariantem architektury w przypadku dużych przedsiębiorstw była architektura zorientowana na usługi sterowana orkiestracją. Obecnie niemal nikt nie tworzy już niczego z wykorzystaniem tego wariantu architektury (z powodów, które zostaną przedstawione w dalszej części książki). Mikrousługi to dziś faworyzowany wariant w wielu systemach rozproszonych. Dlaczego doszło do takiej zmiany i w jaki sposób ona przebiegła?

Gdy architekci analizują konkretny wariant (a zwłaszcza dawniej używany), muszą wziąć pod uwagę istniejące ograniczenia prowadzące do tego, że dany wariant architektury staje się dominujący. W tamtym czasie wiele firm łączyło się, aby stać się *przedsiębiorstwami*. Takiej przemianie towarzyszyły wszystkie problemy integracyjne. Ponadto z punktu widzenia dużych firm oprogramowanie *open source* nie było realną opcją (często raczej z powodów natury politycznej niż technicznej). A zatem jako rozwiązanie architekci uwypuklali zasoby współużytkowane i scentralizowaną orkiestrację.

W tym czasie jednak oprogramowanie *open source* i system Linux stały się opłacalnymi opcjami, sprawiając, że systemy operacyjne były oferowane za darmo do zastosowań *komercyjnych*. Prawdziwy moment przełomowy nastąpił jednak, gdy system Linux stał się darmowy na poziomie *operacyjnym* wraz z pojawieniem się narzędzi, takich jak Puppet i Chef, które umożliwiły zespołom projektowym w sposób programowy zwiększyć wydajność swoich środowisk w ramach zautomatyzowanego procesu kompilowania. Po pojawieniu się takiej możliwości rozpoczęła się rewolucja architektoniczna oparta na mikrousługach oraz szybkie powstawanie infrastruktury kontenerów i narzędzi do orkiestracji, takiej jak Kubernetes.

Pokazuje to, że ekosystem projektowania oprogramowania powiększa i rozwija się na różne, całkowicie nieoczekiwane sposoby. Jedna nowa możliwość prowadzi do kolejnej, która niespodziewanie powoduje pojawienie się nowych możliwości. Z upływem czasu ekosystem zastępuje całkowicie sam siebie element po elemencie.

Objawia to stary jak świat problem dotyczący autorów książek traktujących ogólnie o technologii, a dokładniej o architekturze oprogramowania: jak napisać o czymś, co w bardzo krótkim czasie stanie się nieaktualne?

W książce nie koncentrujemy się na technologii lub innych szczegółach implementacji. Zamiast tego skupiamy się na tym, *jak* architekci podejmują decyzje, a także jak obiektywnie ocenić kompromisy, mając do czynienia z zupełnie nowymi sytuacjami. W celu zapewnienia szczegółów i kontekstu

korzystamy z przykładów i scenariuszy pochodzących z jednego okresu, ale podstawowe zasady koncentrują się na analizie kompromisów i podejmowaniu decyzji związanych z zajmowaniem się nowymi problemami.

Ważność danych w architekturze

Dane to cenna rzecz, która przetrwa dłużej niż same systemy.

— Tim Berners-Lee

Dla wielu osób w przypadku architektury dane są wszystkim. Każde przedsiębiorstwo budujące dowolny system musi zajmować się danymi, które zwykle istnieją znacznie dłużej niż systemy lub architektura, a ponadto wymagają skrupulatnego przemyślenia i projektu. Dużo instynktów architektów, którzy przetwarzają dane, prowadzących do tworzenia silnie sprzężonych systemów powoduje jednak występowanie konfliktów w nowoczesnych architekturach rozproszonych. Na przykład architekci i administratorzy baz danych muszą zapewnić, że dane biznesowe nie zostaną naruszone w wyniku rozdzielenia systemów monolitycznych. Ponadto firmy nadal mają mieć możliwość uzyskania korzyści ze swoich danych niezależnie od zmienności architektury.

Mówi się, że *dane to najważniejszy zasób w firmie*. Firmy chcą w wartościowy sposób korzystać z posiadanych danych, dlatego poszukują nowych metod uwzględniania ich w procesie podejmowania decyzji. Każda część przedsiębiorstwa bazuje obecnie na danych, począwszy od obsługi istniejących klientów, a skończywszy na pozyskiwaniu nowych klientów, zwiększaniu poziomu utrzymania klientów, ulepszaniu produktów, przewidywaniu wartości sprzedaży oraz uwzględnianiu innych trendów. Taka zależność od danych oznacza, że cała architektura oprogramowania ma za zadanie udostępniać dane, zapewniając, że dostępne są odpowiednie dane, które mogą zostać wykorzystane przez wszystkie działy przedsiębiorstwa.

Autorzy zbudowali wiele systemów rozproszonych kilka dekad temu, gdy po raz pierwszy stały się one popularne. Pomimo tego podejmowanie decyzji w przypadku współczesnych mikrousług wydaje się dla nich trudniejsze. Naszym zamiarem jest stwierdzenie, dlaczego tak się dzieje. Ostatecznie zdaliśmy sobie sprawę z tego, że w początkach istnienia architektury rozproszonej dane były głównie utrzymywane w pojedynczej relacyjnej bazie danych. W przypadku jednak mikrousług oraz zgodności na poziomie filozoficznym z *kontekstem ograniczonym* (ang. *bounded context*) wywodzącym się z techniki projektowania ukierunkowanego na domenę (ang. *Domain-Driven Design*, *DDD*; *https://pl. wikipedia.org/wiki/Domain-driven_design*) w ramach metody ograniczenia zasięgu łączenia szczegółów implementacji dane wraz z transakcyjnością przeniesiono w obszar architektury. Wiele trudnych kwestii związanych z nowoczesną architekturą wynika z napięć między aspektami odnoszącymi się do danych a architekturą; omówieniem tych napięć zajmiemy się zarówno w części pierwszej, jak i drugiej książki.

Ważnym rozróżnieniem uwzględnionym w różnych rozdziałach jest oddzielenie danych *operacyjnych* i *analitycznych*.

Dane operacyjne

Dane wykorzystywane przez firmę do prowadzenia działalności; obejmują dane dotyczące sprzedaży, transakcji, magazynowania itp. Są to dane niezbędne firmie do funkcjonowania. Jeśli coś spowoduje uszkodzenie tych danych, organizacja nie może działać przez bardzo długi czas. Tego typu dane są definiowane w ramach *przetwarzania transakcji na bieżąco* (ang. *Online Transactional Processing*, *OLTP*), co zwykle wiąże się z wykonywaniem operacji wstawiania i aktualizowania danych w bazie danych oraz usuwania ich z niej.

Dane analityczne

Dane analityczne są używane przez danologów i innych analityków biznesowych w celu tworzenia przewidywań, trendów oraz innych typów informacji analityki biznesowej. Takie dane nie są zwykle transakcyjne i często nie są też relacyjne. Mogą one znajdować się w grafowej bazie danych lub obrazach danych w innym formacie niż ich oryginalna postać transakcyjna. Te dane nie są kluczowe z punktu widzenia codziennej działalności firmy, lecz raczej długoterminowych decyzji i celów strategicznych.

W książce będzie mowa o wpływie zarówno danych operacyjnych, jak i analitycznych.

Rekord decyzji dotyczących architektury

Jeden z najbardziej efektywnych sposobów dokumentowania decyzji odnoszących się do architektury polega na wykorzystaniu *rekordu decyzji dotyczących architektury* (ang. *Architectural Decision Records*, *ADR*; *https://adr.github.io/*). Rekord ADR był początkowo propagowany przez Michaela Nygarda w artykule na blogu (*https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions*), a później uzyskał status „zaadaptowanego" w serwisie Thoughtworks Technology Radar (*https://www.thoughtworks.com/radar/techniques/lightweight-architecture-decision-records*). Rekord ADR ma postać krótkiego pliku tekstowego (zwykle o długości od jednej do dwóch stron) opisującego konkretną decyzję dotyczącą architektury. Choć rekord ADR może zostać utworzony w postaci zwykłego tekstu, zazwyczaj jest generowany przy użyciu pewnego rodzaju formatu dokumentu tekstowego, takiego jak AsciiDoc (*https://asciidoc.org/*) lub Markdown (*https://www.markdownguide.org/*). Alternatywnie może też zostać utworzony za pomocą szablonu stron serwisu *wiki*. W naszej poprzedniej książce, *Podstawy architektury oprogramowania dla inżynierów* (wydawnictwo Helion), rekordowi ADR poświęcono cały rozdział.

Rekord ADR posłuży nam jako metoda dokumentowania różnych decyzji dotyczących architektury podejmowanych w książce. W przypadku każdej takiej decyzji będzie stosowany poniższy format rekordu ADR przy założeniu, że każdy rekord jest akceptowany.

Rekord ADR: Krótka fraza w postaci rzeczownika zawierająca decyzję dotyczącą architektury

Kontekst

W tej sekcji rekordu zostanie dodany krótki opis problemu liczący jedno lub dwa zdania, a także zostaną wyszczególnione alternatywne rozwiązania.

Decyzja

W tej sekcji zostanie podana decyzja dotycząca architektury oraz jej dokładne uzasadnienie.

Konsekwencje

W tej sekcji rekordu ADR zostaną opisane wszelkie konsekwencje wynikające z zastosowania decyzji, a także zostaną zaprezentowane kompromisy, które były rozważane.

W dodatku B znajduje się lista wszystkich rekordów ADR utworzonych w książce.

Dokumentowanie decyzji jest istotne dla architekta, ale zarządzanie właściwym wykorzystaniem decyzji to osobne zagadnienie. Na szczęście współczesne rozwiązania z dziedziny inżynierii umożliwiają zautomatyzowanie wielu typowych kwestii związanych z zarządzaniem dzięki zastosowaniu funkcji dopasowania architektury.

Funkcje dopasowania architektury

Jak po zidentyfikowaniu przez architekta relacji między komponentami i przedstawieniu tego w postaci projektu można zapewnić, że implementujący będą przestrzegać jego założeń? Bardziej ogólnie: w jaki sposób architekci zapewniają, że zdefiniowane przez nich zasady projektowe urzeczywistnią się w sytuacji, gdy inne osoby zajmują się ich implementacją?

Te pytania zaliczają się do kategorii o nazwie *zarządzanie architekturą*, która odnosi się do wszelkiego uporządkowanego nadzoru jednego lub większej liczby aspektów projektowania oprogramowania. Jako że w książce omawiana jest głównie struktura architektury, objaśniane jest, w jaki sposób w wielu miejscach zautomatyzować zasady projektowe i jakościowe za pośrednictwem funkcji dopasowania.

Dziedzina projektowania oprogramowania z upływem czasu powoli rozwijała się w kierunku adaptacji unikalnych rozwiązań inżynieryjnych. W początkach istnienia tej dziedziny w odniesieniu do rozwiązań dotyczących oprogramowania posługiwano się często metaforą z branży produkcyjnej, zarówno na dużą (np. proces projektowania Waterfall), jak i na małą skalę (rozwiązania integracyjne w projektach). Opracowanie na nowo z początkiem lat 90. praktyk inżynierii projektowania oprogramowania przez Kenta Becka i innych inżynierów w ramach projektu C3 o nazwie eXtreme Programming (XP) uwidoczniło istotność stopniowego uzyskiwania informacji zwrotnej i automatyzacji jako kluczowych czynników zapewniających produktywność w procesie projektowania oprogramowania. Wkrótce po nastaniu 2000 r. te same wnioski wykorzystano tam, gdzie krzyżują się ze sobą projektowanie oprogramowania i działania operacyjne. W ten sposób zapewniono nową rolę metodyki DevOps i zautomatyzowano wiele działań operacyjnych, które wcześniej realizowano ręcznie. Tak jak wcześniej, automatyzacja umożliwia zespołom szybszą pracę, gdyż nie muszą przejmować się rzeczami, które są zaburzane bez odpowiedniej informacji zwrotnej. A zatem *automatyzacja* i *informacja zwrotna* stały się centralnymi założeniami w przypadku efektywnego projektowania oprogramowania.

Weźmy pod uwagę środowiska i sytuacje, które doprowadziły do przełomów w dziedzinie automatyzacji. W czasach przed pojawieniem się integracji ciągłej większość projektów oprogramowania obejmowała rozwlekłą fazę integracji. Każdy projektant miał pracować na pewnym poziomie

odizolowania od innych projektantów, a na koniec cały kod łączono w ramach fazy integracji. Ślady takiej praktyki nadal pozostają w narzędziach do kontroli wersji, które wymuszają rozgałęzianie i uniemożliwiają integrację ciągłą. Nie jest zaskoczeniem to, że istniała silna korelacja między wielkością projektu a złożonością fazy integracji. Pionierskie dokonanie zespołu projektu XP w postaci integracji ciągłej uwidoczniło wartość natychmiastowej i ciągłej informacji zwrotnej.

Podobnym torem przebiegła rewolucja dotycząca metodyki DevOps. To, że system Linux oraz inne oprogramowania *open source* stały się „wystarczająco dobre" dla przedsiębiorstw w połączeniu z pojawieniem się narzędzi umożliwiających definiowanie (w końcu) maszyn wirtualnych w sposób programowy, doprowadziło do sytuacji, w której personel działów operacyjnych uświadomił sobie, że można zautomatyzować tworzenie definicji maszyn oraz wykonywanie wielu innych powtarzających się zadań.

W obu przypadkach postępy w rozwoju technologii oraz zrozumienie zagadnień przyczyniły się do automatyzacji powtarzających się zadań, które były realizowane przez generującego duży koszt specjalistę. Opisuje to aktualny stan zarządzania architekturą w większości organizacji. Jeśli np. architekt wybierze konkretny wariant architektury lub środek komunikacji, w jaki sposób może zapewnić, że zostanie on poprawnie zaimplementowany przez projektanta? Kiedy jest to realizowane ręcznie, architekci sprawdzają kod albo być może tworzą komisje rewizyjne dotyczące architektury, które zajmują się oceną stanu zarządzania. Tak jak jednak w przypadku ręcznego konfigurowania komputerów w działach operacyjnych, istotne szczegóły z łatwością mogą przepaść w ramach pobieżnych weryfikacji.

Zastosowanie funkcji dopasowania

W książce *Architektura ewolucyjna. Projektowanie oprogramowania i wsparcie zmian* (wydawnictwo Helion) z 2017 r. napisanej przez Neala Forda, Rebeccę Parsons i Patricka Kua zdefiniowano pojęcie *funkcji dopasowania architektury*, czyli dowolnego mechanizmu dokonującego oceny integracji celów jakiejś cechy lub kombinacji cech architektury. Oto analiza tej definicji krok po kroku.

Dowolny mechanizm

Architekci mogą korzystać z szerokiej oferty narzędzi implementujących funkcje dopasowania. W książce zostanie zaprezentowanych wiele możliwości. Na przykład istnieją dedykowane biblioteki służące do testowania struktury architektury. Architekci mogą używać narzędzi monitorujących do sprawdzania cech operacyjnych architektury, takich jak wydajność lub skalowalność. Ponadto środowiska związane z inżynierią chaosu umożliwiają testowanie niezawodności i odporności.

Ocena integracji celów

Kluczowy czynnik sprzyjający zautomatyzowanemu zarządzaniu objawia się w postaci definicji celów dla cech architektury. Na przykład architekt nie może określić, że wymaga witryny internetowej o „dużej wydajności". Musi zapewnić wartość celu, która ma zostać zmierzona drogą testowania, monitorowania lub wykorzystania innej funkcji dopasowania.

Architekci muszą uważać na *złożone cechy architektury*, które nie mogą być w obiektywny sposób mierzone, lecz w rzeczywistości są złożeniami innych rzeczy umożliwiających pomiar.

Na przykład zwinność nie może zostać zmierzona, ale jeśli architekt zacznie dokonywać podziału obszernego pojęcia *zwinności*, celem zespołów będzie uzyskanie szybkiej i pewnej reakcji na zmianę w ekosystemie lub domenie. A zatem architekt może znaleźć cechy pozwalające na pomiar, które wpływają na zwinność, czyli możliwość wdrażania, możliwość testowania, czas cyklu itp. Często brak możliwości wykonania pomiaru dla cechy architektury wskazuje na jej zbyt niejednoznaczną definicję. Jeśli architekci dążą do uzyskania cech pozwalających na pomiar, będą mogli zautomatyzować stosowanie funkcji dopasowania.

Pewna cecha architektury lub kombinacji takich cech

Takie cechy opisują dwa zakresy funkcji dopasowania:

Atomowy

Atomowe funkcje dopasowania obsługują pojedynczą, wyizolowaną cechę architektury. Na przykład zasięgiem atomowym charakteryzuje się funkcja dopasowania, która sprawdza cykle komponentu w obrębie bazy kodu.

Holistyczny

Holistyczne funkcje dopasowania weryfikują poprawność kombinacji cech architektury. Element komplikujący w przypadku cech architektury daje efekt synergii, jaką one czasami ujawniają przy okazji innych cech architektury. Jeśli np. architekt zamierza zwiększyć poziom bezpieczeństwa, z dużym prawdopodobieństwem wpłynie to na wydajność. Podobnie kłócą się ze sobą czasami skalowalność i elastyczność. Obsługiwanie dużej liczby jednocześnie zalogowanych użytkowników może sprawić, że znacznie trudniejsze będzie radzenie sobie z nagłymi skokami obciążenia. Holistyczne funkcje dopasowania kontrolują kombinację nakładających się na siebie cech architektury, aby zapewnić, że ich sumaryczny efekt nie będzie mieć negatywnego wpływu na architekturę.

Architekt implementuje funkcje dopasowania w celu zapewnienia środków ochrony względem nieoczekiwanej zmiany dotyczącej cech architektury. W świecie projektowania oprogramowania opartego na metodyce Agile (zwinnej) projektanci stosują testy jednostkowe, testy funkcjonalne oraz testy akceptacyjne, żeby sprawdzić poprawność różnych wymiarów projektu *domeny*. Jak na razie jednak nie powstał żaden podobny mechanizm pozwalający weryfikować część projektu związaną z *cechami architektury*. Okazuje się, że rozdzielenie funkcji dopasowania i testów jednostkowych zapewnia architektom odpowiednią wytyczną w zakresie ustalania zasięgu. Funkcje dopasowania sprawdzają poprawność cech architektury, a nie kryteria domeny, z kolei ich przeciwieństwo stanowią testy jednostkowe . A zatem architekt może decydować o tym, czy niezbędna jest funkcja dopasowania, czy test jednostkowy, zadając następujące pytanie: „Czy do wykonania tego testu wymagana jest jakakolwiek wiedza na temat domeny?”. Jeśli odpowiedź brzmi „tak”, właściwy będzie test jednostkowy, test funkcjonalny lub test akceptacyjny. W przeciwnym razie konieczna jest funkcja dopasowania.

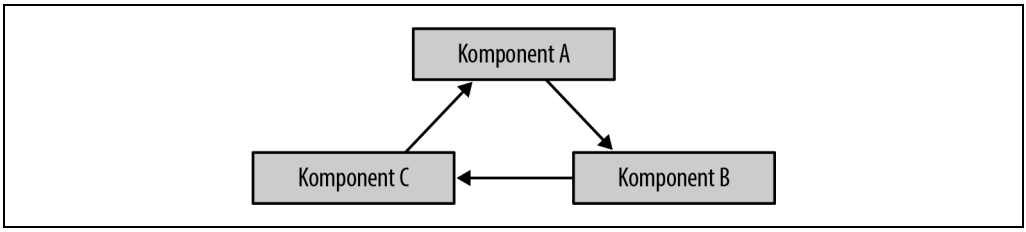
Gdy np. architekci dyskutują o *elastyczności*, mają na myśli możliwość zadziałania w celu poradzenia sobie z nagłym wzrostem liczby użytkowników. Zauważ, że architekt nie musi znać żadnych szczegółów dotyczących domeny, która może reprezentować witrynę do handlu elektronicznego, grę sieciową lub coś innego. Wynika Dzieje się tak, ponieważ *elastyczność* to kwestia

powiązana z architekturą, która znajduje się w zasięgu funkcji dopasowania. Z kolei gdy architekt wymagał sprawdzenia poprawności odpowiednich części adresu pocztowego, jest to uwzględniane w ramach tradycyjnego testu. Takie rozdzielenie nie jest oczywiście zupełnie binarne. Niektóre funkcje dopasowania będą mieć styczność z domeną, i odwrotnie, ale różne cele zapewniają odpowiedni sposób logicznego oddzielenia tych funkcji.

Aby zagadnienie stało się mniej abstrakcyjne, poniżej zaprezentowano kilka przykładów.

Typowym celem architekta jest utrzymanie dobrej, wewnętrznej integralności strukturalnej w bazie kodu. W przypadku wielu platform złowrogie siły stają jednak naprzeciw dobrym intencjom architekta. Jeśli np. kod jest tworzony w dowolnym popularnym środowisku projektowym Java lub .NET, jak tylko projektant odwoła się do klasy, która nie została jeszcze zaimportowana, środowisko IDE pomocnie wyświetla okno dialogowe z pytaniem o to, czy projektant chciałby automatycznie zaimportować odwołanie. Zdarza się to tak często, że większość programistów wyrabia w sobie nawyk zamykania okna dotyczącego automatycznego importowania, tak jakby służyło ono sprawdzeniu refleksu.

Arbitralne wzajemne importowanie klas lub komponentów zwiastuje jednak klęskę z punktu widzenia modułowości. Na rysunku 1.1 zilustrowano szczególnie szkodliwy antywzorzec, którego architekci zamierzają unikać.



Rysunek 1.1. Zależności cykliczne między komponentami

W przypadku tego antywzorca każdy komponent odwołuje się do czegoś w pozostałych komponentach. Użycie takiej sieci komponentów wpłynie destrukcyjnie na modułowość, gdyż projektant nie może ponownie wykorzystać jednego komponentu bez uwzględnienia też pozostałych. Jeśli te komponenty są połączone z innymi komponentami, oczywiście architektura coraz bardziej będzie zmierzać w stronę antywzorca Błotna bryła (ang. *Big Ball of Mud*; *https://pl.wikipedia.org/wiki/Antywzorzec_projektowy*). Jak architekci mogą zarządzać takim zachowaniem bez ciągłego zerkania na projektantów podejmujących zbyt szybko działania? Przeglądanie kodu jest pomocne, ale odbywa się zbyt późno w cyklu projektowym, aby było skuteczne. Jeżeli architekt pozwoli zespołowi projektowemu na intensywne importowanie w obrębie bazy kodu tydzień przed przejrzeniem kodu, w momencie rozpoczęcia tej operacji w bazie kodu będą już obecne poważne spustoszenia.

Rozwiązaniem jest utworzenie funkcji dopasowania pozwalającej uniknąć cykli komponentów (listing 1.1).

Listing 1.1. Funkcja dopasowania służąca do wykrywania cykli komponentów

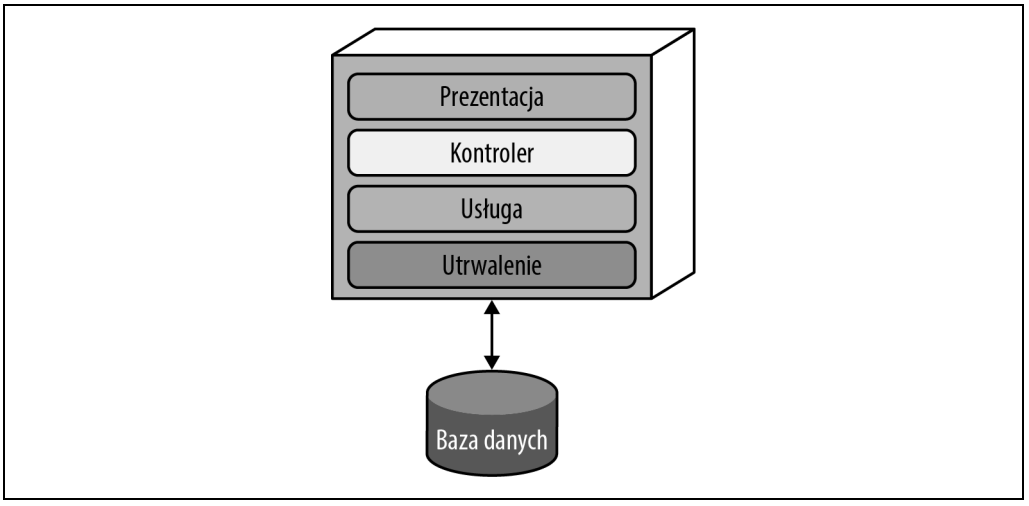
```
public class CycleTest {
  private JDepend jdepend;
```

```
    @BeforeEach
    void init() {
         jdepend = new JDepend();
         jdepend.addDirectory("/sciezka/projektu/persistence/classes");
         jdepend.addDirectory("/sciezka/projektu/web/classes");
         jdepend.addDirectory("/sciezka/projektu/thirdpartyjars");
    }

    @Test
    void testAllPackages() {
         Collection packages = jdepend.analyze();
         assertEquals("Cykle istnieją.", false, jdepend.containsCycles());
    }
}
```

W tym kodzie architekt stosuje narzędzie pomiarowe JDepend (*https://github.com/clarkware/jdepend*), aby sprawdzić zależności między pakietami. Narzędzie rozpoznaje strukturę pakietów Java i kończy test niepowodzeniem, jeśli istnieją jakiekolwiek cykle. Architekt może podłączyć taki test do procesu ciągłej kompilacji w projekcie i nie martwić się więcej przypadkowym wprowadzeniem cyklów przez działających impulsywnie projektantów. Jest to znakomity przykład funkcji dopasowania chroniącej raczej istotne niż nagłe praktyki związane z projektowaniem oprogramowania: to ważna kwestia dla architektów, a przy tym ma niewielki wpływ na codzienny proces tworzenia kodu.

Listing 1.1 prezentuje koncentrującą się na kodzie funkcję dopasowania, która jest bardzo niskopoziomowa. Dużo popularnych narzędzi oczyszczających kod (np. SonarQube; *https://www.sonarqube.org*) implementuje wiele typowych funkcji dopasowania w kompleksowy sposób. Architekci mogą jednak wymagać też sprawdzenia poprawności zarówno makrostruktury, jak i mikrostruktury architektury. W przypadku projektowania architektury warstwowej, takiej jak widoczna na rysunku 1.2, architekt definiuje warstwy w celu zapewnienia separacji zagadnień.



Rysunek 1.2. Tradycyjna architektura warstwowa

W jaki jednak sposób architekt może zapewnić, że projektanci będą respektować te warstwy? Niektórzy projektanci mogą nie uświadamiać sobie ważności wzorców, natomiast inni mogą przyjmować postawę „lepiej prosić o przebaczenie niż o zgodę" ze względu na jakąś nadrzędną, lokalną kwestię, taką jak wydajność. Zezwolenie jednak osobom implementującym na umniejszanie znaczenia powodów istnienia architektury długoterminowo wpłynie na nią niekorzystnie.

Narzędzie ArchUnit (*https://www.archunit.org*) umożliwia architektom rozwiązanie tego problemu z wykorzystaniem funkcji dopasowania zaprezentowanej na listingu 1.2.

Listing 1.2. Funkcja dopasowania narzędzia ArchUnit służąca do zarządzania warstwami

```
layeredArchitecture()
  .layer("Controller").definedBy("..kontroler..")
  .layer("Service").definedBy("..usługa..")
  .layer("Persistence").definedBy("..utrwalanie..")

  .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
  .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
  .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

W kodzie z listingu 1.2 architekt definiuje żądaną relację między warstwami i tworzy zarządzającą nią weryfikującą funkcję dopasowania. Umożliwia to architektowi ustanowienie zasad architektury poza diagramami oraz innych artefaktów informacyjnych, a także sprawdzanie ich na bieżąco.

W przypadku środowiska .NET podobne narzędzie NetArchTest (*https://github.com/BenMorris/NetArchTest*) pozwala na tworzenie podobnych testów dla tej platformy. Listing 1.3 prezentuje weryfikację warstw w kodzie języka C#.

Listing 1.3. Użycie narzędzia NetArchTest pod kątem zależności warstw

```
// Klasy warstwy prezentacji nie powinny bezpośrednio odwoływać się do repozytoriów
var result = Types.InCurrentDomain()
  .That()
  .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
  .ShouldNot()
  .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
  .GetResult()
  .IsSuccessful;
```

Pojawiają się kolejne narzędzia związane z tym zagadnieniem, które cechują się coraz większym stopniem zaawansowania. W dalszym ciągu będziemy zwracać uwagę na wiele takich technik podczas ilustrowania funkcji dopasowania związanych z wieloma naszymi rozwiązaniami.

Kluczowe znaczenie ma znalezienie wyniku celu dla funkcji dopasowania. Termin *cel* nie oznacza jednak *statyczności*. Niektóre funkcje dopasowania będą zapewniać niekontekstowe wartości zwracane, takie jak prawda lub fałsz, albo wartość liczbową np. w postaci wartości progowej wydajności. Inne funkcje dopasowania (uznawane za *dynamiczne*) zwracają wartość opartą na jakimś kontekście. Gdy np. mierzona jest *skalowalność*, architekci ustalają liczbę jednoczesnych użytkowników, a ponadto przeważnie mierzą wydajność dla każdego z nich. Często architekci projektują systemy w taki sposób, że w przypadku wzrostu liczby użytkowników nieznacznie zmniejsza się (lecz nie załamuje całkowicie) wydajność przypadająca na jednego użytkownika. A zatem na potrzeby takich systemów architekci projektują wydajnościowe funkcje dopasowania, które uwzględniają liczbę jednoczesnych użytkowników. Dopóki pomiar dotyczący cechy architektury jest obiektywny, architekci mogą ją testować.

Choć większość funkcji dopasowania powinna zostać zautomatyzowana i działać w sposób ciągły, niektóre z nich z konieczności będą ręcznie obsługiwane. Tego rodzaju funkcja dopasowania wymaga do obsługi procesu sprawdzania poprawności interwencji specjalisty. Na przykład w przypadku systemów zawierających poufne informacje prawne w celu zapewnienia legalności prawnik może wymagać sprawdzenia zmian dotyczących kluczowych elementów. Nie może to zostać zautomatyzowane. Większość potoków procesu wdrażania obsługuje fazy realizowane ręcznie, co pozwala zespołom dostosować ręcznie wykonywane funkcje dopasowania. W idealnej sytuacji takie funkcje są uruchamiane tak często, jak to jest faktycznie możliwe. Proces sprawdzania poprawności, który nie może zostać rozpoczęty, nie pozwoli na sprawdzenie czegokolwiek. Zespoły wykonują funkcje dopasowania albo na żądanie (rzadko), albo w ramach strumienia działań integracji ciągłej (najczęstszy wariant). Aby w pełni skorzystać ze sprawdzania poprawności, takiego jak w przypadku funkcji dopasowania, ten proces powinien być nieustannie realizowany.

Ciągłość jest ważna, co zostało zilustrowane w przykładzie zarządzania na poziomie przedsiębiorstwa z wykorzystaniem funkcji dopasowania. Pod uwagę można wziąć następujący scenariusz: jak firma postąpi, gdy w jednym z używanych przez nią środowisk do projektowania lub w jednej z bibliotek zostanie wykryty atak „dnia zerowego" (ang. *zero-day exploit*)? W przypadku większości firm w takiej sytuacji eksperci od zabezpieczeń sprawdzają projekty w celu znalezienia wersji środowiska, która jest celem ataku, i upewniają się, że ją zaktualizowano. Ten proces jest jednak rzadko zautomatyzowany i bazuje na wielu ręcznie wykonywanych krokach. Nie jest to abstrakcyjne pytanie. Dokładnie taki scenariusz opisany poniżej w ramce „Włamanie do danych agencji Equifax" nastąpił w dużej instytucji finansowej. Tak jak wspomniano wcześniej przy omawianiu kwestii zarządzania architekturą, ręcznie realizowane procesy są podatne na błędy i sprzyjają niezwracaniu uwagi na szczegóły.

Włamanie do danych agencji Equifax

7 września 2017 r. duża agencja Equifax ze Stanów Zjednoczonych zajmująca się oceną wiarygodności kredytowej ogłosiła, że nastąpiło włamanie do jej danych. Ostatecznie w ramach analizy problemu stwierdzono, że wykorzystano atak dotyczący popularnego środowiska aplikacji internetowych Struts działającego w ekosystemie Java (Apache Struts w wersji CVE-2017-5638). Organizacja rozwijająca to środowisko zamieściła oświadczenie informujące o podatności na atak i opublikowała poprawkę 7 marca 2017 r. Departament Bezpieczeństwa Krajowego USA skontaktował się następnego dnia z agencją Equifax oraz podobnymi do niej firmami, ostrzegając o tym problemie. W efekcie te firmy 15 marca 2017 r. uruchomiły procesy przeszukiwania, które nie ujawniły wszystkich systemów, które padły ofiarą ataku. A zatem aż do 29 lipca 2017 r., gdy eksperci od zabezpieczeń agencji Equifax zidentyfikowali działalność hakerów prowadzącą do włamania do danych, w przypadku wielu starszych systemów nie została zastosowana ścieżka krytyczna.

Wyobraź sobie alternatywny świat, w którym każdy projekt uwzględnia potok wdrażania, a członkowie zespołu zajmującego się zabezpieczeniami dysponują „miejscem" w takim potoku poszczególnych zespołów, gdzie mogą wdrażać funkcje dopasowania. Będą one przeważnie dotyczyć mało ciekawych sprawdzeń pod kątem zabezpieczeń, takich jak uniemożliwienie projektantom zapisywania haseł w bazach danych oraz podobnych, zwykłych niezbędnych działań związanych z zarządzaniem. Gdy jednak wystąpi atak „dnia zerowego", zastosowanie wszędzie tego samego mechanizmu umożliwi zespołowi od zabezpieczeń umieszczenie testu w każdym projekcie, który sprawdza konkretne

środowisko oraz numer wersji. Jeśli zostanie znaleziona niebezpieczna wersja, proces kompilacji nie powiedzie się i zostaną powiadomieni członkowie tego zespołu. Zespoły konfigurują potoki wdrażania, by uświadomić firmie pojawienie się w ekosystemie (kod, baza danych, schemat, konfiguracja wdrażania i funkcje dopasowania) jakiejkolwiek zmiany. Pozwala to przedsiębiorstwom uzyskać powszechną automatyzację ważnych zadań związanych z zarządzaniem.

Funkcje dopasowania zapewniają architektom wiele korzyści, spośród których wyróżnia się zwłaszcza szansa ponownego pisania kodu! Jednym z powszechnych narzekań architektów jest to, że nie mają zbyt wiele okazji tworzenia kodu. Funkcje dopasowania to jednak często kod! Tworząc specyfikację architektury możliwą do uruchomienia, której poprawność może sprawdzić każdy w dowolnej chwili przez załadowanie kompilacji projektu, architekci muszą rozumieć system, a także jego bieżący stan rozwoju, co pokrywa się z głównym celem, czyli ciągłym śledzeniem kodu powiększającego się projektu.

Jakkolwiek duże byłyby jednak możliwości funkcji dopasowania, architekci powinni unikać przesadnego korzystania z nich. Nie powinni oni knuć intrygi, a potem wycofywać się do „żelaznej wieży", aby stworzyć niemożliwie złożony zestaw powiązanych ze sobą funkcji dopasowania, które tylko powodują frustrację projektantów i zespołów. Zamiast tego architekci powinni zapewnić możliwą do uruchomienia listę kontrolną *ważnych*, lecz nie *pilnych* zasad dotyczących projektów oprogramowania. Z powodu koncentrowania się w wielu projektach na kwestii pilności przy okazji zezwala się na pominięcie ważnych zasad. Jest to częsta przyczyna długu technicznego: „Wiemy, że jest to złe, ale powrócimy do tego później, aby to naprawić". „Później" nigdy nie następuje. Ustalając reguły dotyczące jakości kodu, struktury oraz innych zabezpieczeń przed negatywną skłonnością do sięgania po nieustannie działające funkcje dopasowania, architekci tworzą listę kontrolną jakości, której projektanci nie mogą pominąć.

Kilka lat temu w znakomitej książce *The Checklist Manifesto* autorstwa Atula Gawande (wydawnictwo Picador) zwrócono uwagę na stosowanie list kontrolnych przez profesjonalistów, takich jak chirurdzy i piloci samolotów, a także przez przedstawicieli innych dziedzin, którzy powszechnie posługują się takimi listami (czasami z powodu wymogów prawnych) w swojej pracy. Dzieje się tak nie dlatego, że te osoby nie znają się na swojej profesji albo są szczególnie zapominalskie. Gdy profesjonaliści ciągle realizują to samo zadanie, łatwe staje się skompromitowanie w przypadku przypadkowego pominięcia go. Zapobiegają temu listy kontrolne. Funkcje dopasowania reprezentują listę kontrolną istotnych zasad definiowanych przez architektów i są stosowane jako część procesu kompilacji w celu zapewnienia, że projektanci przypadkowo (lub celowo z powodu czynników zewnętrznych, takich jak presja dotrzymania zaplanowanych terminów) pominą je.

Funkcje dopasowania są używane w książce, gdy nadarza się okazja zilustrowania zarządzania rozwiązaniem dotyczącym architektury, a także wstępnym projektem.

Architektura a projekt: utrzymywanie definicji w prostej postaci

Stały obszar zabiegów architektów stanowi zadbanie o to, aby *architektura* i *projekt* pozostały osobnymi, lecz powiązanymi ze sobą zakresami aktywności. Choć naszym celem nie jest brnięcie

w nigdy niekończącą się dyskusję dotyczącą takiego rozróżnienia, w książce z kilku powodów dążymy do zdecydowanego pozostania po stronie *architektury*.

Po pierwsze, aby podejmować skuteczne decyzje, architekci muszą rozumieć podstawowe zasady architektury. Na przykład zanim architekci wprowadzą szczegóły implementacji przy decydowaniu się między *synchronicznym* a *asynchronicznym* wariantem komunikacji, muszą stawić czoła pewnej liczbie związanych z tym kompromisów. W książce *Podstawy architektury oprogramowania dla inżynierów* autorzy zdefiniowali drugie prawo architektury oprogramowania: określenie *dlaczego* jest ważniejsze niż *jak*. Choć ostatecznie architekci muszą zrozumieć, jak implementować rozwiązania, najpierw niezbędne jest, aby wiedzieli, dlaczego jedna opcja wyboru zapewnia lepsze kompromisy niż inna.

Po drugie, koncentrując się na zagadnieniach związanych z architekturą, można uniknąć ich licznych implementacji. Architekci mogą implementować komunikację asynchroniczną na różne sposoby. W innym miejscu zajmiemy się tym, dlaczego architekt miałby wybrać wariant komunikacji asynchronicznej i pozostawić szczegóły implementacji.

Po trzecie, jeśli zaczęto by podążać ścieżką implementowania wszystkich prezentowanych różnorodnych opcji, byłaby to najbardziej obszerna książka, jaką dotąd napisano. Skoncentrowanie się na zasadach architektury pozwala zachować wszystko w najbardziej ogólnej postaci, jaka jest możliwa.

Aby w jak największym możliwym stopniu osadzić zagadnienia w architekturze, w przypadku kluczowych pojęć używane są najprostsze możliwe definicje. Na przykład pojęciu *sprzężenia* powiązanemu z architekturą można poświęcić całe książki (i tak jest). W związku z tym posługujemy się następującymi, przesadnie uproszczonymi definicjami:

Usługa

> Kolokwialnie mówiąc, *usługa* to spójna kolekcja funkcjonalności wdrażanych jako niezależny kod wykonywalny. Większość pojęć omawianych w odniesieniu do usług dotyczy ogólnie architektur rozproszonych, a dokładniej architektur mikrousług.
>
> Z punktu widzenia terminów zdefiniowanych w rozdziale 2. *usługa* jest częścią kwantu architektury, który obejmuje dodatkowe definicje zarówno statycznego, jak i dynamicznego sprzężenia między usługami i innymi kwantami.

Sprzężenie

> Dwa artefakty (uwzględniające usługi) są sprzężone, gdy zmiana w jednym z nich może wymagać dokonania zmiany w drugim w celu utrzymania poprawnej funkcjonalności.

Komponent

> Komponent to blok konstrukcyjny architektury aplikacji, który realizuje pewnego rodzaju funkcję biznesową lub infrastrukturalną manifestowaną zwykle za pośrednictwem struktury pakietu (język Java), przestrzeni nazw (język C#) lub fizycznego grupowania plików z kodem źródłowym w obrębie wybranego typu struktury katalogów. Na przykład komponent historii zamówień może być implementowany z wykorzystaniem zestawu plików klas zlokalizowanych w przestrzeni nazw `app.business.order.history`.

Komunikacja synchroniczna

Dwa artefakty komunikują się ze sobą synchronicznie, jeśli przed dalszym kontynuowaniem działań obiekt wywołujący musi czekać na odpowiedź.

Komunikacja asynchroniczna

Dwa artefakty komunikują się ze sobą asynchronicznie, gdy przed kontynuowaniem działań obiekt wywołujący nie czeka na odpowiedź. Opcjonalnie obiekt wywołujący może być powiadamiany po zrealizowaniu żądania przez obiekt odbierający za pośrednictwem osobnego kanału.

Koordynacja z orkiestracją

Przepływ informacji podlega orkiestracji, jeśli uwzględnia usługę, której podstawowym zadaniem jest koordynacja przepływu.

Koordynacja z choreografią

Przepływ informacji podlega choreografii, gdy jest pozbawiony orkiestracji. W tym przypadku usługi w przepływie dzielą się odpowiedzialnością za jego koordynację.

Atomowość

Przepływ informacji jest *atomowy*, jeśli wszystkie jego części cały czas utrzymują spójny stan. Przeciwieństwo tego reprezentuje spektrum *ostatecznej spójności* omówione w rozdziale 6.

Kontrakt

Termin *kontrakt* jest używany ogólnie do definiowania interfejsu między dwiema częściami oprogramowania, co może obejmować wywołania funkcji lub metod, zdalne wywołania architektury integracji, zależności itd. Zawsze tam, gdzie są łączone dwa elementy oprogramowania, stosowany jest kontrakt.

Architektura oprogramowania jest z założenia abstrakcyjna: nie można stwierdzić, z jakiej unikalnej kombinacji platform, technologii, oprogramowania komercyjnego oraz innej, oszałamiającej grupy możliwości korzystają Czytelnicy. Wyjątkiem jest tutaj tylko to, że żadne dwie z tych kombinacji nie są dokładnie takie same. Omawiamy wiele abstrakcyjnych pojęć, ale w celu ich skonkretyzowania musimy je poprzeć pewnymi szczegółami implementacji. W związku z tym niezbędny jest problem, w przypadku którego zostaną zilustrowane zagadnienia architektury. To prowadzi nas do kwestii zespołu operatorów systemu.

Wprowadzenie do sagi zespołu operatorów systemu

Saga

Długa historia heroicznego wyczynu.

— Słownik języka angielskiego Oxford

W książce zostanie zaprezentowanych kilka sag zarówno w znaczeniu dosłownym, jak i przenośnym. Architekci wykorzystali termin *saga* do opisywania zachowania transakcyjnego w architekturach rozproszonych (szczegółowo będzie o tym mowa w rozdziale 12.). Dyskusje dotyczące architektury

stają się jednak zwykle abstrakcyjne, a zwłaszcza wtedy, gdy rozważa się takie abstrakcyjne problemy jak trudne kwestie architektury. Aby ułatwić rozwiązanie tego problemu i zapewnić jakiś rzeczywisty kontekst dla omawianych rozwiązań, zdecydowaliśmy się na posłużenie się sagą poświęconą *zespołowi operatorów systemu*.

W każdym rozdziale ta saga służy to zilustrowania technik i kompromisów prezentowanych w książce. Choć w wielu książkach poświęconych architekturze oprogramowania są omawiane nowe dokonania z dziedziny projektowania, wiele prawdziwych problemów istnieje wewnątrz stosowanych systemów. A zatem prezentowana przez nas historia rozpoczyna się od naświetlonej tutaj architektury, z którą ma do czynienia zespół operatorów systemu.

Penultimate Electronics to gigant z branży elektroniki mający wiele sklepów detalicznych w całym kraju. Gdy klienci kupują komputery, telewizory, sprzęt stereo oraz inne urządzenia elektroniczne, mogą się zdecydować na nabycie planu wsparcia technicznego. Po pojawieniu się problemów do miejsca zamieszkania (lub miejsca pracy) klienta przyjeżdżają eksperci zajmujący się technologią po stronie klienta (zespół operatorów systemu), aby usunąć problemy z urządzeniem elektronicznym.

Oto cztery podstawowe typy użytkowników aplikacji zgłoszeniowej zespołu operatorów systemu:

Administrator

Zajmuje się użytkownikami wewnętrznymi systemu, co obejmuje listę ekspertów oraz powiązany z nimi zakres umiejętności, lokalizację i dostępność. Administrator zarządza też całością przetwarzania rozliczeń dla klientów korzystających z systemu, a także utrzymuje statyczne dane referencyjne (dotyczące np. wspieranych produktów, par nazwa-wartość istniejących w systemie itp.).

Klient

Rejestruje się w celu uzyskania usługi zespołu operatorów systemu i utrzymuje swój profil, umowy dotyczące wsparcia oraz informacje na potrzeby rozliczeń. Klienci generują w systemie zgłoszenia problemów, a także wypełniają ankiety po zakończeniu prac.

Ekspert zespołu operatorów systemu

Ekspertom są przydzielane zgłoszenia problemów, na podstawie których usuwają problemy. Korzystają oni również z bazy wiedzy, aby znaleźć rozwiązania problemów u klienta i wprowadzić uwagi dotyczące napraw.

Menedżer

Monitoruje działania związane ze zgłoszeniami problemów oraz otrzymuje raporty operacyjne i analityczne dotyczące całego systemu obsługi zgłoszeń problemów używanego przez zespół operatorów.

Przepływ informacji bez zgłoszeń

Przepływy informacji bez zgłoszeń obejmują te działania realizowane przez administratorów, menedżerów i klientów, które nie są związane ze zgłoszeniem problemu. Takie przepływy informacji są opisywane w następujący sposób:

1. Eksperci zespołu operatorów systemu są dodawani przez administratora i utrzymywani przez niego w systemie. Administrator wprowadza informacje dotyczące ich lokalizacji, dostępności i zakresu umiejętności.
2. Klienci rejestrują się w systemie zespołu operatorów i mają dostęp do wielu planów wsparcia zależnie od zakupionych produktów.
3. Klienci są automatycznie rozliczani co miesiąc przy użyciu danych o karcie kredytowej podanych w ich profilu. Klienci mogą wyświetlić historię rozliczeń oraz wyciągi w obrębie całego systemu.
4. Menedżerowie generują żądania dotyczące różnych raportów operacyjnych i analitycznych oraz otrzymują je. Raporty obejmują raporty finansowe, raporty z informacją o wydajności ekspertów oraz raporty zgłoszeń.

Przepływ informacji ze zgłoszeniami

Przepływ informacji ze zgłoszeniami jest inicjowany w momencie wprowadzenia przez klienta do systemu zgłoszenia problemu, a kończy się, gdy klient wypełni ankietę po dokonaniu naprawy. Taki przepływ informacji jest opisywany w następujący sposób:

1. Klienci, którzy nabyli plan wsparcia, wprowadzają zgłoszenie problemu za pośrednictwem witryny internetowej zespołu operatorów systemu.
2. Po wprowadzeniu zgłoszenia problemu system ustala, jaki ekspert zespołu operatorów systemu byłby najbardziej odpowiedni do wykonania zadania. System bazuje na zakresie umiejętności, bieżącej lokalizacji, obszarze działania i dostępności eksperta.
3. Po przydzieleniu zgłoszenie problemu jest wysyłane do dedykowanej, niestandardowej aplikacji zainstalowanej na urządzeniu przenośnym eksperta zespołu operatorów systemu. Ekspert jest też powiadamiany za pomocą wiadomości tekstowej o otrzymaniu nowego zgłoszenia problemu.
4. Klient o tym, że ekspert jest w drodze do niego, zostaje powiadomiony przy użyciu wiadomości tekstowej SMS lub wiadomości pocztowej (zależnie od preferowanej formy kontaktu podanej w profilu).
5. Ekspert używa aplikacji na swoim telefonie do pobrania informacji o zgłoszeniu i adresu. Ekspert zespołu operatorów systemu korzysta też z bazy wiedzy za pośrednictwem aplikacji mobilnej, aby dowiedzieć się, co w przeszłości zostało wykonane w celu usunięcia problemu.
6. Po poradzeniu sobie z problemem ekspert oznacza zgłoszenie jako zrealizowane. Może on następnie dodać informacje o problemie i poprawić w bazie wiedzy dotyczący go wpis.
7. Po otrzymaniu powiadomienia o zakończeniu realizacji zgłoszenia system wysyła do klienta wiadomość pocztową z odnośnikiem do ankiety, którą klient następnie wypełnia.
8. System odbiera od klienta wypełnioną ankietę i zapisuje jej zawartość.

Zły scenariusz

W ostatnim czasie aplikacja obsługująca zgłoszenia problemów dla zespołu operatorów systemu nie sprawowała się zbyt dobrze. Używany obecnie system zgłoszeń problemów to duża, monolityczna aplikacja zaprojektowana wiele lat temu. Klienci narzekają, że konsultanci nigdy nie pojawiają się u nich — przyczyną są zagubione zgłoszenia — a często przyjeżdża niewłaściwy konsultant w celu naprawienia czegoś, na czym się zupełnie nie zna. Klienci narzekali również na to, że system nie zawsze umożliwia wprowadzanie nowych zgłoszeń problemów.

W przypadku takiego dużego monolitu zmiana jest trudna i ryzykowna. Wprowadzanie zmiany każdorazowo trwa zbyt długo i zwykle na skutek tego coś innego przestaje działać. Z powodu problemów z niezawodnością system zespołu operatorów często przestaje reagować lub zawiesza się. W efekcie cała funkcjonalność aplikacji jest niedostępna z dowolnego miejsca przez czas wynoszący od pięciu minut do dwóch godzin — tyle zajmuje identyfikowanie problemu i ponowne uruchamianie aplikacji.

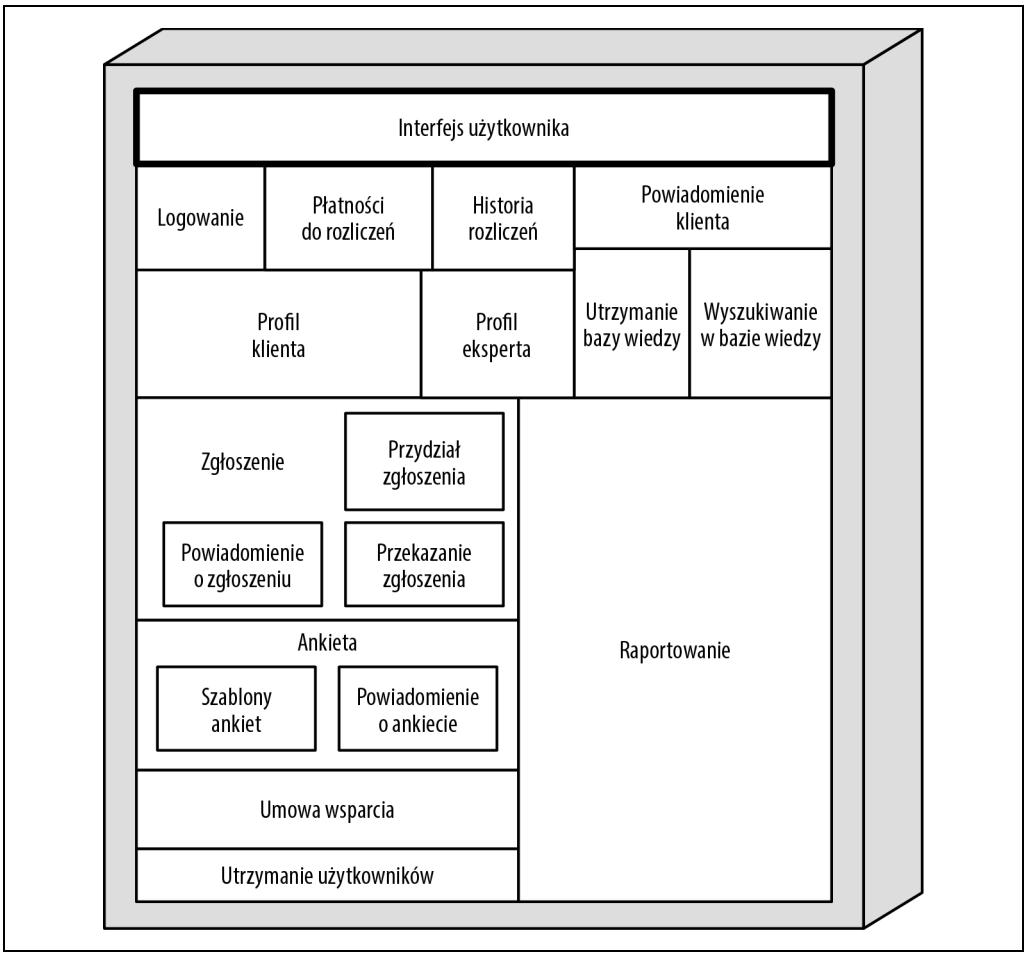
Jeśli coś nie zostanie wkrótce zrobione, firma Penultimate Electronics będzie zmuszona zrezygnować z bardzo lukratywnych biznesowych umów wsparcia i zwolnienia wszystkich tworzących zespół operatorów systemu administratorów, ekspertów i menedżerów, a także informatyków z działu projektowania z uwzględnieniem architektów.

Komponenty architektury aplikacji zespołu operatorów systemu

System monolityczny aplikacji zespołu operatorów systemu zajmuje się zarządzaniem zgłoszeniami, generowaniem raportów operacyjnych, rejestrowaniem klientów i rozliczeniami, jak również ogólnymi funkcjami administracyjnymi, takimi jak utrzymanie użytkowników, logowanie oraz utrzymanie profili i informacji o umiejętnościach ekspertów. Na rysunku 1.3 i w powiązanej z nim tabeli 1.1 zilustrowano i opisano komponenty istniejącej aplikacji monolitycznej (łańcuch `ss.` w identyfikatorze przestrzeni nazw określa kontekst aplikacji zespołu operatorów systemu).

Tabela 1.1. Istniejące komponenty aplikacji zespołu operatorów systemu

Komponent	Przestrzeń nazw	Zakres odpowiedzialności
Logowanie	`ss.login`	Logowanie klientów i użytkowników wewnętrznych oraz logika zabezpieczeń
Płatności do rozliczeń	`ss.billing.payment`	Comiesięczne rozliczenia klientów oraz informacje o ich kartach kredytowych
Historia rozliczeń	`ss.billing.history`	Historia płatności i wyciągi z wcześniejszymi rozliczeniami
Powiadomienie klienta	`ss.customer.notification`	Powiadomienia klientów o rozliczeniach oraz ogólne informacje
Profil klienta	`ss.customer.profile`	Utrzymanie profilu klientów oraz ich rejestrowanie
Profil eksperta	`ss.expert.profile`	Utrzymanie profilu ekspertów (nazwisko, lokalizacja, umiejętności itp.)
Utrzymanie bazy wiedzy	`ss.kb.maintenance`	Utrzymanie i wyświetlanie pozycji w bazie wiedzy



Rysunek 1.3. Komponenty w istniejącej aplikacji zespołu operatorów systemu

Wymienione komponenty będą używane w kolejnych rozdziałach do zilustrowania różnych technik i kompromisów mających zastosowanie podczas dzielenia aplikacji w ramach architektur rozproszonych.

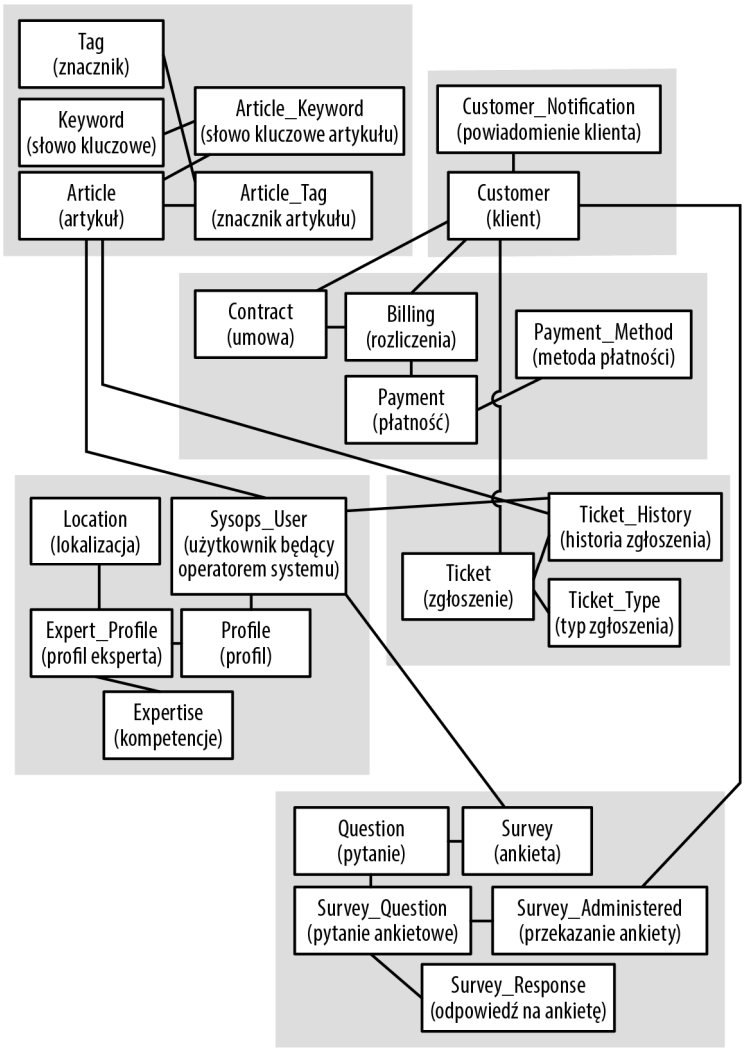
Model danych aplikacji zespołu operatorów systemu

Aplikacja zespołu operatorów systemu ze swoimi różnymi komponentami podanymi w tabeli 1.1 korzysta z pojedynczego schematu w bazie danych, aby udostępniać wszystkie swoje tabele i powiązany z nimi kod bazy danych. Baza służy do utrzymania klientów, użytkowników, umów, rozliczeń, płatności, bazy wiedzy oraz ankiet klientów. Tabele wyszczególniono w tabeli 1.2, a diagram związków encji zilustrowano na rysunku 1.4.

Tabela 1.1. Istniejące komponenty aplikacji zespołu operatorów systemu (ciąg dalszy)

Komponent	Przestrzeń nazw	Zakres odpowiedzialności
Wyszukiwanie w bazie wiedzy	`ss.kb.search`	Mechanizm zapytań do wyszukiwań w bazie wiedzy
Raportowanie	`ss.reporting`	Całość raportowania (uwzględnia ekspertów, zgłoszenia i dane finansowe)
Zgłoszenie	`ss.ticket`	Tworzenie, utrzymanie i realizacja zgłoszeń oraz wspólny kod
Przydział zgłoszenia	`ss.ticket.assign`	Znajdowanie eksperta i przydzielanie zgłoszenia
Powiadomienie o zgłoszeniu	`ss.ticket.notify`	Powiadomienie klienta o tym, że ekspert jedzie do niego
Przekazanie zgłoszenia	`ss.ticket.route`	Wysyłanie zgłoszenia do aplikacji na urządzeniu przenośnym eksperta
Umowa wsparcia	`ss.supportcontract`	Umowy wsparcia klientów oraz produkty objęte planem
Ankieta	`ss.survey`	Utrzymanie ankiet oraz zbieranie i zapisywanie podanych w nich danych
Powiadomienie o ankiecie	`ss.survey.notify`	Wysyłanie do klienta wiadomości pocztowej dotyczącej ankiety
Szablony ankiet	`ss.survey.templates`	Utrzymanie różnych ankiet zależnych od typu usługi
Utrzymanie użytkowników	`ss.users`	Utrzymanie użytkowników i ról wewnętrznych

Model danych aplikacji zespołu operatorów systemu to standardowy model trzeciej postaci normalnej uwzględniający tylko kilka procedur składowanych lub wyzwalaczy. Istnieje w nim jednak spora liczba widoków, które są głównie używane przez komponent raportowania. Gdy zespół zajmujący się architekturą będzie podejmował próbę podzielenia aplikacji i będzie zmierzał w stronę architektury rozproszonej, konieczna będzie współpraca z zespołem odpowiedzialnym za bazę danych, aby zostały zrealizowane zadania na poziomie bazy. Przedstawiona konfiguracja tabel i widoków bazy danych będzie używana w książce do omawiania różnych technik i kompromisów umożliwiających podzielenie bazy danych.



Tabele w bazie danych aplikacji operatorów systemu
(*aby zachować zwięzłość, pominięto kolumny i widoki*)

Rysunek 1.4. Model danych istniejącej aplikacji zespołu operatorów systemu

Tabela 1.2. Istniejące tabele bazy danych aplikacji zespołu operatorów systemu

Tabela	Zakres odpowiedzialności
`Customer`	Osoby wymagające wsparcia zespołu operatorów systemu
`Customer_Notification`	Preferencje dotyczące powiadamiania klientów
`Survey`	Ankieta dotycząca zadowolenia klienta po zapewnieniu mu wsparcia
`Question`	Pytania ankietowe
`Survey_Question`	Pytanie przypisane do ankiety
`Survey_Administered`	Pytania ankietowe przekazane klientowi
`Survey_Response`	Odpowiedź klienta na ankietę
`Billing`	Informacje rozliczeniowe na potrzeby umowy wsparcia
`Contract`	Umowa dotycząca wsparcia zawarta między daną osobą a zespołem operatorów systemu
`Payment_Method`	Obsługiwane metody płatności
`Payment`	Płatności realizowane na potrzeby rozliczeń
`SysOps_User`	Różni użytkownicy w zespole operatorów systemu
`Profile`	Informacje profilowe użytkowników w zespole operatorów systemu
`Expert_Profile`	Profile ekspertów
`Expertise`	Informacje o różnych kompetencjach w zespole operatorów systemu
`Location`	Lokalizacje obsługiwane przez eksperta
`Article`	Artykuły w bazie wiedzy
`Tag`	Znaczniki dotyczące artykułów
`Keyword`	Słowo kluczowe powiązane z artykułem
`Article_Tag`	Znaczniki skojarzone z artykułami
`Article_Keyword`	Połączona tabela słów kluczowych i artykułów
`Ticket`	Zgłoszenia dotyczące wsparcia wygenerowane przez klientów
`Ticket_Type`	Różne typy zgłoszeń
`Ticket_History`	Historia zgłoszeń dotyczących wsparcia

Skorowidz

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ż

Architekt musi być prorokiem...

Frank Lloyd Wright

W epoce infrastruktur chmurowych, mikrousług czy wysublimowanych wzorców projektowych architekt oprogramowania musi sobie radzić z trudnym zadaniem, jakim jest wybór odpowiednich rozwiązań. Będą one potem szczegółowo testowane podczas pracy w środowisku produkcyjnym, a także przy późniejszym dostosowywaniu i rozbudowywaniu oprogramowania. Tymczasem w wypadku architektury złożonych systemów nie ma łatwych kompromisów. Konieczne jest bardzo wnikliwe i krytyczne przemyślenie każdej decyzji projektowej, i to na możliwie najwcześniejszym etapie pracy.

Ta książka powinna zostać przestudiowana przez każdego architekta nowoczesnych systemów rozproszonych. Jej celem jest pokazanie metod rozwiązywania trudnych problemów związanych z projektowaniem takiego oprogramowania. W krytyczny i wszechstronny sposób omówiono w niej najważniejsze problemy utrudniające podejmowanie dobrych decyzji projektowych. Zaprezentowano najskuteczniejsze strategie doboru optymalnej architektury. Na jasnych przykładach pokazano, w jaki sposób należy przystąpić do analizy założeń projektowych — począwszy od określenia „ziarnistości" usług, przepływów informacji i orkiestracji, poprzez eliminację sprzężenia kontraktów i określenie nadzoru nad transakcjami rozproszonymi, a skończywszy na metodach optymalizowania właściwości operacyjnych, takich jak skalowalność, elastyczność i wydajność.

Najciekawsze zagadnienia:

- analiza kompromisów i dokumentowanie decyzji
- podejmowanie decyzji dotyczących „ziarnistości" usług
- złożoność procesu przekształcania aplikacji monolitycznych
- eliminacja sprzężeń kontraktów wiążących usługi
- obsługa danych w architekturze o dużym stopniu rozproszenia
- wzorce zarządzania przepływami informacji i transakcjami

Neal Ford jest architektem oprogramowania. Pracuje dla firmy Thoughtworks — lidera konsultantów zajmujących się technologiami.

Mark Richards zdobył praktyczne doświadczenie z dziedziny projektowania i implementowania mikrousług, systemów rozproszonych i systemów w architekturze zorientowanej na usługi.

Pramod Sadalage specjalizuje się w projektach aplikacji i ewolucyjnych baz danych, architekturze danych i bazach danych NoSQL.

Zhamak Dehghani zajmuje się systemami w architekturze rozproszonej i technologiami w fazie rozwoju.

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 978-83-283-9527-5

9 788328 395275

Cena: 99,00 zł