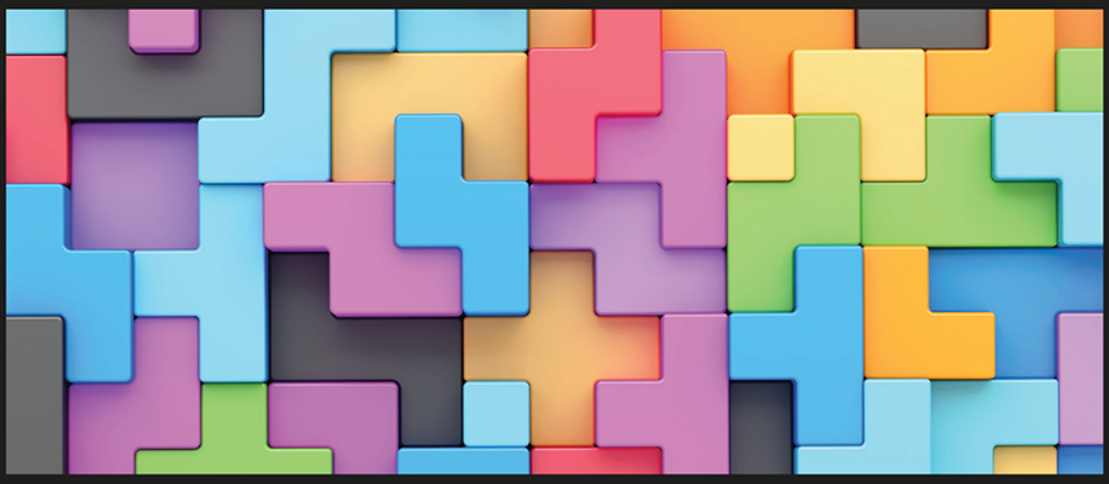
Helion

<packt>



WYDANIE II

Nie bój się ubrudzić rąk, tworząc czystą architekturę

Projektowanie aplikacji wysokiej jakości na przykładach w Javie

TOM HOMBERGS

Tytuł oryginału: Get Your Hands Dirty on Clean Architecture: Build 'clean' applications with code examples in Java, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1231-1

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/niebu2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/niebu2.zip

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

Printed in Poland.

Spis treści

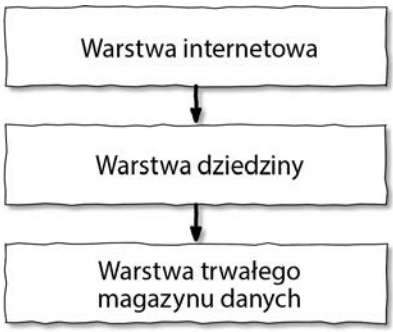
Rozdział 2

Na czym polega problem z warstwami?

Istnieje prawdopodobieństwo, że masz doświadczenie w tworzeniu (internetowych) aplikacji warstwowych. Być może takie rozwiązanie stosujesz także w obecnie realizowanym projekcie.

Praca z warstwami została nam zaszczepiona w trakcie zajęć, w samouczkach oraz w prezentacjach najlepszych praktyk. Jest polecana również w książkach[1].

Na rysunku 2.1 pokazałem ogólny schemat często spotykanej architektury składającej się z trzech warstw. Mamy więc **warstwę internetową**, która otrzymuje żądania i kieruje je do usługi działającej w **warstwie dziedziny**[2]. Usługa zawiera logikę biznesową i wywołuje komponenty z **warstwy trwałego magazynu danych** w celu pobierania bądź modyfikowania bieżącego stanu encji dziedziny w bazie danych.



Rysunek 2.1. Konwencjonalna architektura aplikacji internetowej składa się z warstwy internetowej, dziedziny i trwałego magazynu danych

Nie uwierzysz, ale warstwy są solidnym wzorcem architektury. Jeżeli zostaną użyte poprawnie, umożliwią opracowanie logiki dziedziny, która będzie niezależna od warstw internetowej i trwałego magazynu danych. Jeżeli zajdzie potrzeba, będzie można dokonać

[1] Warstwy jako wzorzec pojawiają się na przykład w książce Marka Richardsa *Software Architecture Patterns* wydanej przez O'Reilly w 2015 roku.

[2] Dziedzina kontra biznes: w tej książce wymiennie używam pojęć „dziedzina" i „biznes". Warstwa dziedziny, inaczej warstwa biznesowa, to miejsce w kodzie źródłowym zawierające kod odpowiedzialny za rozwiązywanie problemów biznesowych. Mamy więc przeciwieństwo dla kodu rozwiązującego problemy techniczne, takie jak trwałe przechowywanie informacji w bazie danych lub przetwarzanie żądań internetowych.

zmiany technologii warstw internetowej i trwałego magazynu danych bez wpływania na logikę dziedziny. Ponadto możliwe będzie dodawanie nowych funkcjonalności bez wpływania na istniejące.

Zastosowanie dobrej architektury opartej na warstwach pozwala zachować dostępność wielu opcji i możliwość szybkiego dostosowania się do zmieniających się wymagań i czynników zewnętrznych (np. dostawca usług bazodanowych z dnia na dzień podwaja ich koszt). Dobra architektura oparta na warstwach będzie zapewniała łatwość obsługi technicznej rozwiązania.

Wobec tego w czym tkwi problem z warstwami?

Z mojego doświadczenia wynika, że architektura oparta na warstwach jest niezwykle wrażliwa na zmiany, co znacznie utrudnia jej późniejszą obsługę techniczną. Pozwala na wkradanie się złych zależności i powoduje, że wraz z upływem czasu wprowadzanie zmian w oprogramowaniu staje się coraz trudniejsze. Warstwy nie zapewniają wystarczająco sztywnych ram, aby utrzymać architekturę na właściwym torze. W zbyt dużym stopniu trzeba polegać na dyscyplinie i pracowitości człowieka, aby zapewnić łatwość obsługi technicznej.

To wszystko dokładniej wyjaśnię w kolejnych podrozdziałach.

Warstwy wspierają projekt oparty na bazie danych

Z definicji wynika, że podstawą konwencjonalnej architektury warstwowej jest baza danych. Działanie warstwy internetowej zależy od warstwy dziedziny, której z kolei zależnością jest warstwa trwałego magazynu danych, a więc baza danych. Wszystko inne jest zbudowane na bazie warstwy trwałego magazynu danych. Takie podejście jest problematyczne z kilku powodów.

Warto zrobić krok wstecz i zastanowić się nad tym, co próbujemy osiągnąć w przypadku niemalże każdej budowanej aplikacji. Przeważnie staramy się opracować model reguł lub „polityk" nadzorujących działalność biznesową, aby w ten sposób ułatwić użytkownikowi współdziałanie z nią.

Przede wszystkim podejmowane są próby modelowania sposobu działania, a nie stanu. Nie ulega wątpliwości, że stan jest ważnym aspektem każdej aplikacji. Jednak to sposób działania zmienia stan, a tym samym wpływa na działalność biznesową.

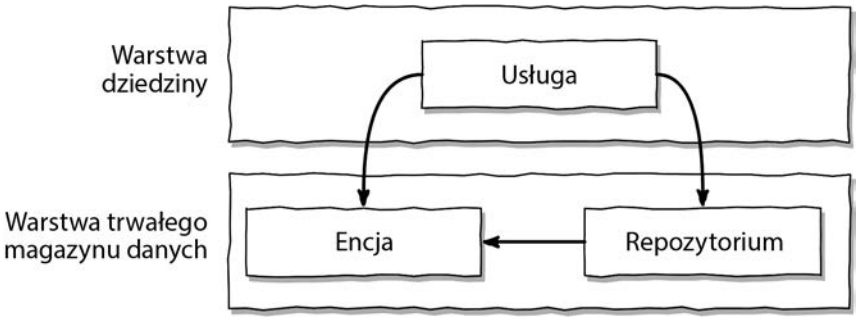
Dlaczego więc podstawą architektury staje się baza danych, a nie logika dziedziny?

Zastanów się nad ostatnimi rozwiązaniami, które zostały przez Ciebie zaimplementowane w dowolnej aplikacji. Czy praca rozpoczęła się od implementacji logiki dziedziny, czy od warstwy trwałego magazyn danych? Prawdopodobnie na początku została określona struktura bazy danych, a dopiero potem na tej podstawie rozpoczęły się prace nad zaimplementowaniem logiki dziedziny.

Takie podejście wydaje się rozsądne w przypadku konwencjonalnej architektury opartej na warstwach, ponieważ pozwala wykorzystać naturalny przepływ zależności. Natomiast z biznesowego punktu widzenia zupełnie nie ma sensu. Logika dziedziny powinna zostać utworzona jako pierwsza, zanim przystąpisz do budowania czegokolwiek innego. Konieczne jest ustalenie, czy reguły biznesowe zostały poprawnie zrozumiane. Dopiero po upewnieniu się, że budowana jest właściwa logika dziedziny, można przystąpić do tworzenia na jej podstawie warstw internetowej i trwałego magazynu danych.

Siłą napędową w tego rodzaju architekturze opartej na bazie danych jest użycie frameworka zapewniającego obsługę **mapowania obiektowo-relacyjnego** (ang. *object-relational mapping*, ORM). Nie zrozum mnie źle — lubię te frameworki i regularnie z nich korzystam. Jeżeli jednak połączymy framework typu ORM i architekturę opartą na warstwach, wówczas bardzo łatwo można doprowadzić do pomieszania reguł biznesowych z aspektami trwałego magazynu danych.

Zwykle mamy zarządzane przez ORM encje będące częścią warstwy trwałego magazynu danych (zobacz rysunek 2.2). Ponieważ warstwa może uzyskać dostęp do warstw znajdujących się poniżej, warstwa dziedziny może mieć dostęp do tych encji. A jeśli wolno jest z ich korzystać, na pewnym etapie warstwa dziedziny użyje tych encji.



Rysunek 2.2. Przykład użycia encji bazy danych w warstwie dziedziny prowadzi do silnego jej powiązania z warstwą trwałego magazynu danych

W taki sposób powstaje silne powiązanie między warstwami dziedziny i trwałego magazynu danych. Nasze usługi biznesowe używają modelu trwałego magazynu danych jako modelu biznesowego i muszą zajmować się nie tylko logiką dziedziny, ale również wczytywaniem wczesnym i z opóźnieniem, transakcjami bazy danych, opróżnianiem buforów oraz podobnymi zadaniami porządkowymi[3].

Kod odpowiedzialny za obsługę trwałego magazynu danych jest praktycznie połączony z kodem dziedziny i dlatego bardzo trudne okazuje się zmodyfikowanie jednego bez ruszania drugiego. Mamy tutaj przeciwieństwo elastyczności i zachowania otwartych opcji, co powinno być celem naszej architektury.

[3] W swojej nowatorskiej książce *Refaktoryzacja. Ulepszanie struktury istniejącego kodu. Wydanie II* (Helion, 2018) Martin Fowler określa ten symptom mianem „rozbieżnych zmian": konieczność zmiany pozornie nie związanych ze sobą fragmentów kodu w celu zaimplementowania danej funkcjonalności. Jest to sytuacja, która powinna skłonić do refaktoryzacji.

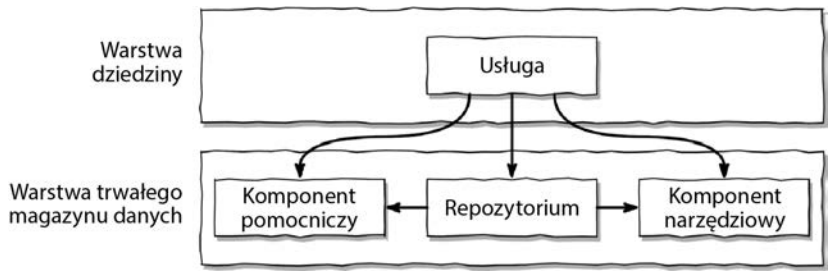
Warstwy są podatne na skróty

W przypadku konwencjonalnej architektury warstwowej jedyną globalną regułą jest to, że z poziomu określonej warstwy można uzyskać dostęp tylko do komponentów znajdujących się w tej samej warstwie bądź w warstwie poniżej. Wprawdzie mogą być jeszcze inne reguły uzgodnione przez zespół programistyczny i stosowanie części z nich może być egzekwowane przez narzędzia, ale mimo to architektura warstwowa sama w sobie nie narzuca tych reguł.

Dlatego jeśli zachodzi potrzeba uzyskania dostępu do określonej warstwy znajdującej się powyżej naszej, komponent można po prostu przesunąć w dół warstwy i uzyskać do niego dostęp. Problem został rozwiązany. Jednorazowe działanie tego typu może wydawać się w porządku. Jeżeli jednak zrobisz tak raz, za drugim razem będzie to łatwiejsze. A skoro ktoś mógł tak zrobić, ja również mogę, prawda?

Wcale nie twierdzę, że programiści łatwo decydują się na tego rodzaju drogę na skróty. Jeżeli jednak istnieje pewna droga, ktoś może nią pójść, zwłaszcza kiedy wisi nad nim widmo nieuchronnie zbliżającego się terminu. Gdy coś zostało zrobione raz, znacznie zwiększa się prawdopodobieństwo, że zostanie zrobione po raz kolejny. Mamy tutaj do czynienia z efektem psychologicznym nazywanym **teorią wybitej szyby** — do tego zagadnienia jeszcze powrócę w rozdziale 11.

Na przestrzeni lat opracowywania i późniejszej obsługi technicznej oprogramowania warstwa trwałego magazynu danych może wyglądać tak jak na rysunku 2.3.



Rysunek 2.3. Ponieważ każda warstwa może uzyskać dostęp do warstwy trwałego magazynu danych, istnieje tendencja do jej rozrastania się wraz z upływem czasu

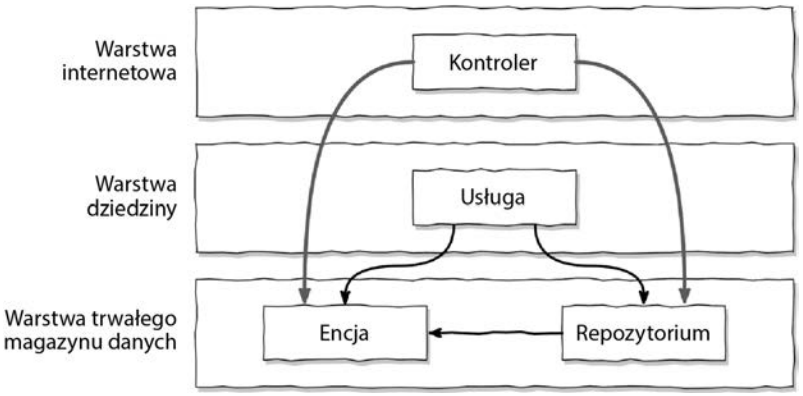
Warstwa trwałego magazynu danych (bądź w bardziej ogólnych kategoriach warstwa najniższa) rozrasta się, ponieważ trafiają do niej komponenty pochodzące z innych warstw. Doskonałymi kandydatami są tutaj wszelkie komponenty pomocnicze i narzędziowe, nie należą one bowiem do żadnej konkretnej warstwy.

Tak więc jeśli chcesz uniemożliwić stosowanie *drogi na skróty* w danej architekturze, warstwy nie będą najlepszym rozwiązaniem, przynajmniej nie bez użycia mechanizmu pozwalającego wymuszać dodatkowe reguły architekturalne. Określenie *wymuszanie* nie oznacza tutaj działania ze strony starszego programisty przeprowadzającego przegląd kodu, lecz raczej rozwiązania zautomatyzowane, które w przypadku niespełnienia wymagań doprowadzą do awarii.

Warstwy utrudniają testowanie

W przypadku architektury warstwowej powszechne jest pomijanie warstw. Dostęp do warstwy trwałego magazynu danych jest uzyskiwany bezpośrednio z poziomu warstwy internetowej, ponieważ przeprowadzana jest jedynie operacja na pojedynczym polu encji i w związku z tym nie ma konieczności sięgania po warstwę dziedziny, prawda?

Na rysunku 2.4 pokazałem pominięcie warstwy dziedziny oraz uzyskanie dostępu do warstwy trwałego magazynu danych bezpośrednio z poziomu warstwy internetowej.



Rysunek 2.4. Pomijanie warstwy dziedziny prowadzi do rozproszenia logiki dziedziny w bazie kodu

Także tu na początku takie rozwiązanie wydaje się w porządku, ale jeśli będzie zdarzało się częściej (a za każdym kolejnym razem będzie łatwiej się na to zdecydować), będzie się wiązało z dwiema poważnymi wadami.

Pierwsza: logika warstwy dziedziny jest implementowana w warstwie internetowej, nawet w przypadku przeprowadzania operacji na pojedynczym polu. Co się stanie w sytuacji, gdy w przyszłości zajdzie potrzeba rozbudowy danego przypadku użycia? Wówczas prawdopodobnie kolejna parta logiki dziedziny zostanie dodana do warstwy internetowej, co doprowadzi do dalszego mieszania kodu i rozproszenia na wszystkich warstwach ważnej logiki dziedziny.

Druga: w testach jednostkowych warstwy internetowej trzeba będzie zajmować się nie tylko zależnościami warstwy dziedziny, ale również zależnościami warstwy trwałego magazynu danych. Jeżeli zdecydujesz się na użycie imitacji w testach, będzie to oznaczało konieczność tworzenia imitacji dla obu warstw. To z kolei prowadzi do zwiększenia poziomu złożoności testów. A skomplikowane przygotowywanie testów jest pierwszym krokiem do rezygnacji z nich ze względu na brak czasu na ich przeprowadzanie. Gdy komponent internetowy będzie się rozrastał wraz z upływem czasu, nastąpi kumulacja wielu zależności dla różnych komponentów trwałego magazynu danych, co jeszcze bardziej skomplikuje testy. W pewnym momencie więcej czasu trzeba będzie poświęcić na zrozumienie zależności i tworzenie imitacji niż na rzeczywiste tworzenie kodu testującego.

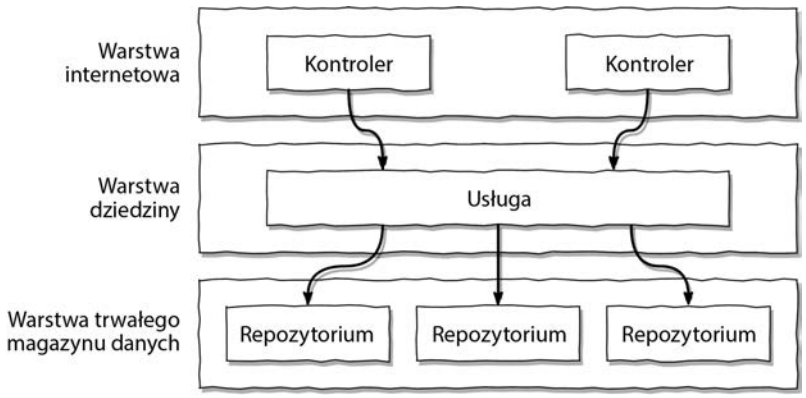
Warstwy ukrywają przypadki użycia

Programiści lubią tworzyć nowy kod do implementacji zupełnie nowych przypadków użycia. Jednak zwykle więcej czasu poświęca się na zmianę istniejącego kodu niż na tworzenie zupełnie nowego. To dotyczy nie tylko wzbudzających lęk starszych projektów, w których przypadku pracuje się z kodem tworzonym dekadami, ale również zupełnie nowych projektów, w których zostały zaimplementowane początkowe przypadki użycia.

Skoro często szukamy właściwego miejsca na dodanie bądź zmianę funkcjonalności, czy architektura nie powinna ułatwiać szybkiego poruszania się po bazie kodu? W jaki sposób może w tym pomóc architektura oparta na warstwach?

Jak już wcześniej wyjaśniłem, w przypadku architektury warstwowej bardzo łatwo można doprowadzić do sytuacji, w której logika dziedziny będzie rozrzucona po różnych warstwach. Może znajdować się w warstwie internetowej w przypadku pominięcia logiki dziedziny dla „łatwego" przypadku użycia. Może również znajdować się w warstwie trwałego magazynu danych, jeśli określony komponent zostanie przeniesiony do dołu, aby stał się dostępny z poziomu warstw dziedziny i trwałego magazynu danych. To powoduje, że naprawdę trudne staje się znalezienie właściwego miejsca na dodanie nowej funkcjonalności.

Ale na tym nie koniec. Architektura warstwowa nie narzuca reguł związanych z „szerokością" usług dziedziny. Wraz z upływem czasu często prowadzi to do powstania niezwykle szerokich usług, które będą służyły do obsługi wielu przypadków użycia (zobacz rysunek 2.5).



Rysunek 2.5. „Szerokie" usługi znacznie utrudniają znalezienie w bazie kodu określonych przypadków użycia

Szeroka usługa może mieć wiele zależności w warstwie trwałego magazynu danych, a ponadto będzie od niej zależało działanie wielu komponentów w warstwie internetowej. W takim przypadku trudne staje się nie tylko przetestowanie usługi, ale również znalezienie kodu dotyczącego przypadku użycia, nad którym chcesz pracować.

O ile łatwiej byłoby, gdyby istniały wysoce wyspecjalizowane, wąskie usługi dziedziny służące do pojedynczych celów. Zamiast szukać w klasie `UserService` przypadku użycia na przykład związanego z rejestracją użytkownika, można byłoby po prostu otworzyć klasę `RegisterUserService` i od razu przystąpić do pracy.

Warstwy utrudniają pracę równoległą

Kierownictwo zwykle oczekuje, że tworzenie oprogramowania zakończy się w określonym terminie. Oczekuje również, że zmieścimy się w określonym budżecie, ale nie będę w tym miejscu komplikował omawianego przykładu.

Pomijam również fakt, że w trakcie swojej kariery inżyniera oprogramowania jeszcze nigdy nie spotkałem się z „ukończeniem" pracy nad oprogramowaniem; „ukończenie" oprogramowania w określonym terminie przeważnie oznacza, że musi nad nim pracować wiele osób.

Prawdopodobnie znasz sławny wniosek z książki *Legendarny osobomiesiąc*, nawet jeśli nie miałeś okazji się z nią zapoznać: *dodawanie pracowników do opóźnionego projektu tylko zwiększa opóźnienie*[4].

W pewnym stopniu to odnosi się również do projektów polegających na tworzeniu oprogramowania, które nie są (jeszcze) opóźnione. Nie można oczekiwać, że grupa 50 programistów będzie pracowała pięciokrotnie szybciej niż mniejszy zespół składający się z 10 programistów. Jeżeli zajmują się oni ogromną aplikacją i można ich podzielić na mniejsze podzespoły jednocześnie pracujące nad różnymi fragmentami aplikacji, wówczas takie podejście może się sprawdzić. Natomiast w większości przypadków będą oni po prostu wchodzić sobie w drogę.

Jednak w rozsądnych granicach można oczekiwać, że większa liczba osób pracujących nad projektem przyśpieszy jego realizację. Kierownictwo ma prawo tego oczekiwać.

Aby spełnić to oczekiwanie, *architektura musi umożliwiać pracę równoległą*. To wcale nie jest takie łatwe. A na dodatek architektura warstwowa w tym nie pomaga.

Wyobraź sobie dodawanie nowych przypadków użycia do aplikacji. Nad projektem pracuje troje programistów. Pierwszy może zajmować się dodawaniem niezbędnych funkcjonalności w warstwie internetowej, drugi pracuje w warstwie dziedziny, trzeci zaś w warstwie trwałego magazynu danych, prawda?

No cóż, to zwykle tak nie działa w przypadku architektury warstwowej. Skoro wszystko zostaje zbudowane na bazie warstwy trwałego magazynu danych, to ona musi zostać opracowana jako pierwsza. Następnie trzeba się zająć warstwą dziedziny, a dopiero na końcu warstwą internetową. Dlatego w danej chwili nad konkretną funkcjonalnością może pracować tylko jeden programista.

Już słyszę, jak mówisz: „Przecież programiści mogą najpierw zdefiniować interfejsy. Następnie każdy z nich może pracować z tymi interfejsami, bez konieczności odwoływania się do rzeczywistej implementacji".

Oczywiście istnieje taka możliwość, ale jedynie w przypadku, gdy nie doszło do połączenia logiki dziedziny i trwałego magazynu danych (o czym już wcześniej wspomniałem), co uniemożliwiałoby oddzielną pracę nad poszczególnymi aspektami.

[4] Frederick P. Brooks Jr., *Legendarny osobomiesiąc. Opowieści o inżynierii oprogramowania. Wydanie II*, Helion, Gliwice 2019.

Jeżeli w bazie kodu masz szerokie usługi, równoczesna praca nad *różnymi* funkcjonalnościami okaże się jeszcze trudniejsza. Praca nad poszczególnymi przypadkami zastosowania może prowadzić do jednoczesnego edytowania tej samej usługi, co z kolei może spowodować powstanie konfliktów w trakcie dołączania kodu, a tym samym wprowadzić potencjalne regresje.

W jaki sposób może to pomóc w tworzeniu oprogramowania łatwego w późniejszej obsłudze technicznej?

Jeżeli masz doświadczenie w tworzeniu architektur warstwowych w przeszłości, prawdopodobnie jesteś w stanie powiązać niektóre z kwestii omówionych w tym rozdziale, a może nawet wymienić kolejne.

Jeśli rozwiązanie będzie zaimplementowane poprawnie i będą zastosowane w nim pewne reguły, to architektura oparta na warstwach może zapewniać łatwą obsługę techniczną, a wprowadzanie w niej zmian bądź dodawanie kolejnych funkcjonalności do bazy kodu będzie odbywało się z łatwością.

Jednak praktyka dowodzi, że w przypadku architektury warstwowej może pojawić się wiele problemów. Brak dobrej samodyscypliny będzie wraz z upływem czasu prowadził do degradacji architektury i jej coraz trudniejszej obsługi technicznej. Poziom samodyscypliny zazwyczaj zmniejsza się z powodu rotacji pracowników, a także wskutek narzucania zespołowi programistycznemu kolejnych terminów przez menedżera.

O pułapkach związanych z architekturą warstwową warto pamiętać w trakcie następnej dyskusji dotyczącej zastosowania skrótów i zdecydować się na zbudowanie rozwiązania umożliwiającego jego łatwiejszą obsługę techniczną, czy to z wykorzystaniem architektury warstwowej, czy w innym stylu.

Tworzenie czystych aplikacji na przykładzie rozwiązań w Javie

Wyobraź sobie sytuację: chcesz napisać oprogramowanie, które nie tylko spełni oczekiwania klienta, ale również będzie przykładem eleganckiej i czystej architektury. Na drodze do tego celu z pewnością napotkasz przeszkody, takie jak nierealny termin czy niedziałające API zewnętrznego dostawcy. Nie będzie wyjścia, czas zmusi Cię do pójścia na skróty i Twoja architektura wkrótce straci swoją elegancką strukturę. Aby tego uniknąć, musisz przejąć nad nią kontrolę.

Dzięki tej książce zorientujesz się, że utrzymanie kontroli nad architekturą w dużej mierze zależy od zastosowanego stylu architektonicznego. Zrozumiesz też wady konwencjonalnej architektury warstwowej i zapoznasz się z zaletami stylów koncentrujących się na dziedzinie, takich jak architektura heksagonalna. Dowiesz się także, jak można ją wyrazić w kodzie źródłowym. Poznasz szczegóły różnych strategii mapowania między warstwami architektury heksagonalnej, a ponadto prześledzisz, jak różne elementy architekturalne łączą się w jedną aplikację. Bazując na stylu architektury heksagonalnej, nauczysz się tworzyć intuicyjne w późniejszej obsłudze technicznej aplikacje internetowe. Szybko się przekonasz, że wiedza zdobyta w trakcie lektury pozwoli Ci na tworzenie wysokojakościowych aplikacji, które przetrwają próbę czasu.

Najciekawsze zagadnienia:

- niedoskonałości związane z architekturą warstwową
- egzekwowanie granic architektury
- wpływ stosowania skrótów na debet techniczny
- korzystanie z poszczególnych stylów architekturalnych
- struktura kodu a architektura
- testy sprawdzające wszystkie elementy architektury

Tom Hombergs jest inżynierem oprogramowania, pasjonatem kodu i autorem. Stara się upraszczać zarówno kod, jak i tekst. Uważa, że lektura artykułów, książek i dokumentacji technicznej powinna być przyjemnością. Obecnie pracuje w firmie Atlassian w Sydney.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1231-1
9 788328 912311
Cena: 49,90 zł
packt