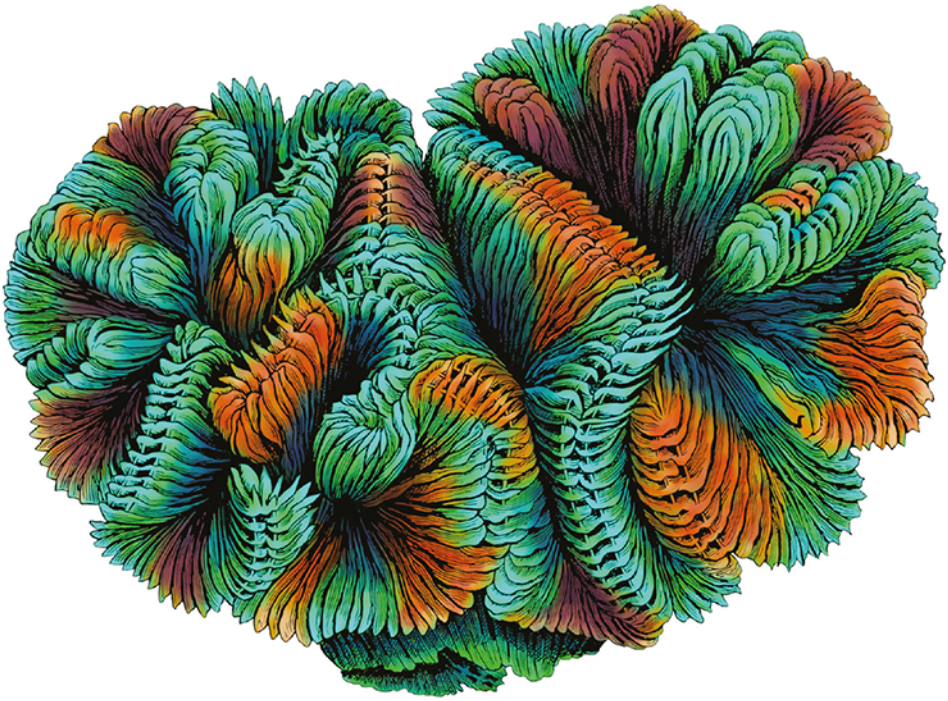
O'REILLY®

Wydanie II

Architektura ewolucyjna

Projektowanie oprogramowania i wsparcie zmian



Neal Ford, Rebecca Parsons,
Patrick Kua, Pramod Sadalage

Helion

Tytuł oryginału: Building Evolutionary Architectures: Automated Software Governance, 2nd Edition

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-289-0066-0

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/arche2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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 1.

Ewoluowanie architektury oprogramowania

Tworzenie starzejących się wdzięcznie i skutecznie systemów jest jednym z głównych wyzwań projektowania oprogramowania w ogólności, a szczególnie architektury oprogramowania. W tej książce zostały omówione dwa podstawowe aspekty budowania ewoluującego oprogramowania: wykorzystywanie skutecznych praktyk inżynieryjnych wywodzących się z ruchu oprogramowania zwinnego oraz planowanie architektury w sposób ułatwiający zmiany i zarządzanie.

Czytelnicy poznają aktualny stan wiedzy na temat zarządzania w sposób deterministyczny zmianami w architekturze, ujednolicający wcześniejsze próby zabezpieczania parametrów architektury, a także różne techniki poprawiające zdolność architektury do przemian bez jej niszczenia.

Wyzwania związane z ewoluującą architekturą

> **Gnicie bitów** (ang. *bit rot*), zwane także *gniciem oprogramowania*, *gniciem kodu*, *erozją oprogramowania*, *rozpadem oprogramowania* lub *entropią oprogramowania*, to stopniowe pogarszanie się jakości oprogramowania w miarę upływu czasu lub wydłużanie się jego czasu reakcji, prowadzące ostatecznie do jego zwiększonej awaryjności.

Zespoły od dawna próbują tworzyć oprogramowanie wysokiej jakości, które takie *pozostaje* w miarę upływu czasu, i tworzą różnorodne definicje odzwierciedlające trudności, z jakimi muszą się mierzyć, takie jak różne określenia powyższego pojęcia **gnicia bitów**. Podstawą toczonej przez nie walki są przynajmniej dwa czynniki: problemy z nadzorowaniem wszystkich elementów skomplikowanego oprogramowania oraz dynamiczna natura środowiska tworzenia oprogramowania.

Współczesne oprogramowanie składa się z tysięcy, a nawet milionów poszczególnych elementów, z których każdy może być zmieniany wzdłuż pewnego zbioru wymiarów. Każda z tych zmian ma przewidywalne, a czasami nieprzewidywalne skutki. Zespoły starające się ręcznie zarządzać tymi zmianami zostają w końcu przytłoczone samą liczbą elementów i narastających efektów ubocznych.

Już samo w sobie zarządzanie licznymi oddziaływaniami oprogramowania w statycznym środowisku byłoby straszne, ale nie mamy z takowym do czynienia. Środowisko inżynierii oprogramowania składa się ze wszystkich narzędzi, struktur, bibliotek i najlepszych rozwiązań, czyli nagromadzonego najnowszego w danym momencie stanu wiedzy z dziedziny tworzenia oprogramowania. W środowisku tym występuje równowaga — podobnie jak w układach biologicznych — która

jest zrozumiała dla programistów i w ramach której potrafią oni tworzyć produkty. Równowaga ta jest jednak dynamiczna — cały czas pojawiają się nowe elementy i zaburzają ją aż do pojawienia się nowego stanu równowagi. Wyobraź sobie osobę jadącą na rowerze jednokołowym i wiozącą paczki: jest to układ **dynamiczny**, ponieważ monocyklista musi balansować ciałem, żeby nie spaść z monocyklu, a także znajdujący się w **równowadze**, ponieważ osoba ta jest w stanie utrzymywać się na pojeździe. W środowisku inżynierii oprogramowania każda nowa innowacja/rozwiązanie może zaburzyć stan bieżący, co wymusza ustalenie nowej równowagi. Mówiąc w przenośni, dokładamy monocykliście kolejne pudełka, czym zmuszamy go do wkładania większego wysiłku w utrzymanie równowagi.

Architekci pod wieloma względami przypominają naszego nieszczęsnego monocyklistę, gdyż bez przerwy muszą balansować i dostosowywać się do zmieniających się warunków. Praktyki inżynieryjne stanowiące część techniki ciągłego dostarczania stanowią przykład takiego zaburzania równowagi: wcielenie uprzednio odizolowanych funkcji, takich jak operacje, do cyklu inżynierii oprogramowania pozwoliło ujrzeć znaczenie **zmiany** z zupełnie nowej perspektywy. Architekci pracujący w korporacjach nie mogą już polegać na statycznych, pięcioletnich planach, ponieważ w tym okresie cały wszechświat inżynierii oprogramowania będzie ewoluować, przez co każda długoterminowa decyzja może okazać się w końcu nieistotna.

Destrukcyjne zmiany są trudne do przewidzenia nawet dla doświadczonych praktyków. Wzrost zainteresowania narzędziami pozwalającymi na korzystanie z kontenerów, takimi jak Docker (*https://www.docker.com/*), stanowi przykład nieprzewidywalnych przemian w branży. Możemy jednak prześledzić wzrost popularności konteneryzacji jako szereg niewielkich, przyrostowych etapów. Dawno, dawno temu systemy operacyjne, aplikacje serwerowe i inne elementy infrastruktury były tworami komercyjnymi, wymagającymi licencji oraz olbrzymich wydatków. Wiele zaprojektowanych w owym czasie architektur było skoncentrowanych na wydajnym korzystaniu ze współdzielonych zasobów. Dystrybucje linuksowe stopniowo stawały się coraz przyjaźniejsze dla firm, co zredukowało koszty *monetarne* systemów operacyjnych do zera. Następnie pojawiły się rozwiązania z zakresu metody DevOps, na przykład zautomatyzowana aprowizacja maszyn poprzez takie narzędzia jak Puppet (*https://puppet.com/*) i Chef (*https://www.chef.io*) sprawiła, że dystrybucje Linuksa stały się *operacyjnie* darmowe. Gdy środowisko stało się darmowe i zostało spopularyzowane, kwestią czasu była konsolidacja wokół najczęściej używanych, przenośnych formatów: padło więc na narzędzie Docker. Konteneryzacja nie byłaby jednak możliwa bez wcześniejszych etapów ewolucyjnych prowadzących do tego rozwiązania.

Środowisko inżynierii oprogramowania nieustannie ewoluuje, co prowadzi do nowych rozwiązań architektonicznych. Wielu programistów podejrzewa, że klika architektów zaszywa się w wieży z kości słoniowej, aby tam decydować o kolejnym *Wielkim Projekcie*, ale proces ten jest znacznie bardziej naturalny. Nowe możliwości nieustannie pojawiają się w naszym środowisku i zapewniają sposoby łączenia się z już istniejącymi oraz nowymi funkcjami, co pozwala uzyskać jeszcze większe możliwości. Weźmy na przykład pod uwagę nową modę na architektury mikrousługowe. W miarę wzrostu popularności otwartych systemów operacyjnych w połączeniu z praktykami inżynieryjnymi ciągłego dostarczania oprogramowania wystarczająco wielu bystrych architektów nauczyło się budować skalowalne systemy, którym trzeba było nadać nazwę: tak właśnie powstały mikrousługi.

Dlaczego nie było mikrousług w 2000 roku?

Wyobraź sobie architekta, który za pomocą wehikułu czasu cofa się do 2000 roku i odwiedza kierownika operacyjnego z nowym pomysłem.

— Mam nową, świetną koncepcję architektury, umożliwiającą doskonałe rozdzielenie poszczególnych możliwości. Nazwałem ją **mikrousługami**. Zaprojektujemy każdą usługę wokół możliwości biznesowych i będziemy je utrzymywać rozdzielone.

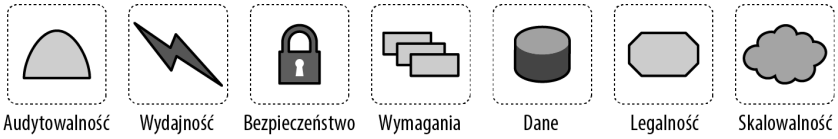
— Świetnie — odpowiada kierownik. — Czego pan będzie potrzebował?

— Około pięćdziesięciu nowych komputerów, oczywiście pięćdziesięciu nowych licencji na system operacyjny, a do tego dwudziestu kolejnych komputerów pełniących funkcję baz danych oraz licencji do nich. Kiedy mogę spodziewać się wszystkich tych rzeczy?

— Proszę wyjść.

Koncepcja mikrousług nawet w owym czasie mogła wydawać się kusząca, ale środowisko nie było jeszcze gotowe na ich obsługę.

Jednym z zadań architekta jest projektowanie strukturalne pozwalające rozwiązywać określone problemy: masz jakiś problem i postanawiasz, że rozwiąże go oprogramowanie. Rozważając projektowanie strukturalne, możemy podzielić je na dwa obszary: **domenę** (lub **wymagania**) i **parametry architektury**, co zostało zaprezentowane na rysunku 1.1.



Rysunek 1.1. Na pełen zakres architektury oprogramowania składają się wymagania i parametry architektury: wszystkie „-ości" oprogramowania

Wymagania zaprezentowane na rysunku 1.1 stanowią domenę problemu, który ma być rozwiązany przez oprogramowanie. Pozostałe elementy są różnie nazywane: **parametry architektury** (preferowany przez nas termin), **wymagania niefunkcjonalne**, **atrybuty jakościowe systemu**, **wymagania przekrojowe** i wiele innych. Bez względu na nazwę reprezentują one kluczowe możliwości wymagane do efektywnego projektu, zarówno w kontekście pierwotnego opublikowania, jak i długotrwałego utrzymywania. Na przykład takie parametry architektury jak **skala** i **wydajność** mogą określać rynkowe kryteria sukcesu, natomiast inne, takie jak **modułowość**, mają wpływ na **utrzymywalność** i **ewoluowalność**.

Wiele nazw parametrów architektury

Używamy w tej książce określenia **parametry architektury** (ang. *architecture characteristics)* w odniesieniu do kwestii projektowania niedomenowego. Jednakże wiele organizacji nazywa to pojęcie inaczej, między innymi: wymagania niefunkcjonalne, wymagania przekrojowe czy atrybuty jakości systemu. Wybór używanego przez Ciebie terminu nie ma większego znaczenia, nic się nie stanie, jeśli będziesz podstawiać stosowany przez siebie termin w dowolnym miejscu książki. Wymienione pojęcia są jednym i tym samym.

Oprogramowanie rzadko kiedy bywa statyczne — ewoluuje, gdy zespoły dodają nowe funkcje, punkty integracji oraz mnóstwo innych powszechnych zmian. Tym, czego potrzebują architekci, są mechanizmy chroniące parametry architektury, przypominające testy jednostkowe, ale skoncentrowane na tychże parametrach, które to zmieniają się w różnym tempie i czasami znajdują się pod wpływem sił innych niż domenowe. Na przykład decyzje techniczne w obrębie firmy mogą doprowadzić do niezależnej od rozwiązania domenowego zmiany bazy danych.

W tej książce opisujemy mechanizmy i techniki projektowe umożliwiające dodawanie takiej samej ciągłej gwarancji zarządzania architekturą, jaką wydajne zespoły mają w każdym innym aspekcie procesu inżynierii oprogramowania.

To właśnie w przypadku decyzji architektonicznych każdy wybór oferuje istotne kompromisy. W dalszej części książki, pisząc o roli **architekta**, mamy na myśli każdą osobę podejmującą decyzje architektoniczne, bez względu na jej tytuł w organizacji. Dodatkowo istotne decyzje architektoniczne praktycznie zawsze wymagają współpracy z innymi rolami.

Czy projekty zwinne wymagają architektury?

Jest to pytanie zadawane często przez osoby korzystające już od pewnego czasu z praktyk inżynierii zwinnej. Celem zwinności jest usunięcie *niepotrzebnych* kosztów, a nie niezbędnych etapów, takich jak projektowanie. Podobnie jak w wielu kwestiach architektonicznych, skala dyktuje rozmiary architektury. Skorzystajmy z analogii do budynku: gdybyśmy chcieli zbudować psią budę, nie potrzebowalibyśmy jakiejś wyszukanej architektury, lecz jedynie materiałów. Natomiast pięćdziesięciopiętrowy wieżowiec musi zostać zaprojektowany. Na podobnej zasadzie nie potrzebujemy architektury, jeśli potrzebujemy witryny internetowej śledzącej prostą bazę danych; możemy znaleźć materiały umożliwiające złożenie wszystkich elementów w całość. Musimy jednak ostrożnie rozważyć wiele kompromisów podczas projektowania wysoce skalowalnego i ogólnodostępnego serwisu, na przykład tłumnie odwiedzanego sklepu z biletami koncertowymi.

Zamiast zadawać pytanie *czy projekty zwinne wymagają architektury?* architekt powinien raczej zastanowić się, na jak wiele niepotrzebnego projektowania może sobie pozwolić podczas opracowywania zdolności do iterowania wczesnych wersji projektu w celu uzyskiwania bardziej optymalnych rozwiązań.

Architektura ewolucyjna

Zarówno wybór mechanizmów ewolucyjnych, jak i decyzje podejmowane przez architektów podczas projektowania oprogramowania mają swoje podłoże w następującej definicji:

> Architektura ewolucyjna wspiera *kierowane*, *przyrostowe* zmiany w *wielu różnych wymiarach*.

Definicja ta składa się z trzech części, które omówimy teraz bardziej szczegółowo.

Zmiana kierowana

Po doborze istotnych parametrów zespoły chcą *kierować* zmianami architektury, by ochronić jej atrybuty. W tym celu zapożyczymy z dziedziny obliczeń ewolucyjnych pojęcie **funkcji dopasowania** (ang. *fitness function*). Funkcją dopasowania nazywamy funkcję celu służącą do określania, czy potencjalne rozwiązanie projektowe będzie realizować założone zadania. W obliczeniach ewolucyjnych funkcja ta pozwala ocenić, czy dany algorytm uległ udoskonaleniu po upływie jakiegoś czasu. Inaczej mówiąc, funkcja dopasowania sprawdza, w jakim stopniu każdy wygenerowany wariant algorytmu jest „dopasowany" do definicji „dopasowanego" algorytmu wyznaczonej przez projektanta.

Architektura ewolucyjna stawia przed nami podobne wyzwanie — wraz z rozwojem architektury potrzebujemy mechanizmów określających wpływ zmian na istotne parametry architektury oraz zapobiegających degradacji tych parametrów w czasie. Opisywany tu odpowiednik funkcji dopasowania dotyczy różnorodnych mechanizmów wprowadzanych przez nas po to, aby zagwarantować, że architektura nie ulegnie zmianom w niepożądany sposób — należą do nich m.in. wskaźniki, testy i inne narzędzia weryfikacyjne. Gdy architekt wyznacza parametry architektury, które mają być chronione w trakcie ewolucji systemu, definiuje przynajmniej jedną funkcję dopasowania służącą do zabezpieczenia tych cech.

W ujęciu historycznym część architektury była często postrzegana jako aktywność zarządcza i dopiero od niedawna architekci akceptują wprowadzanie zmian z poziomu architektury. Architektoniczne funkcje dopasowania pozwalają na podejmowanie decyzji w kontekście potrzeb organizacji i funkcji biznesowych przy jednoczesnym wprowadzaniu jawnych i testowalnych podstaw decyzyjnych. Architektura ewolucyjna nie jest nieograniczoną i nieodpowiedzialną techniką inżynierii oprogramowania, lecz metodą równoważącą potrzebę szybkich zmian z rygorem dotyczącym systemów i parametrów architektury. Funkcje dopasowania napędzają proces decyzyjny, nadając kierunek architekturze z jednoczesnym umożliwianiem zmian wymaganych do wspierania zmieniających się środowisk biznesowego i technologicznego.

Wykorzystujemy **funkcje dopasowania** do tworzenia wskazówek opisujących ewolucję architektur; omówimy je szczegółowo w rozdziale 2.

Zmiana przyrostowa

Zmiana przyrostowa (ang. *incremental change*) opisuje dwa aspekty architektury oprogramowania: sposób przyrostowego tworzenia oprogramowania i jego wdrażania przez zespoły.

W trakcie tworzenia oprogramowania architektura umożliwiająca niewielkie, przyrostowe zmiany ewoluuje w prostszy sposób, ponieważ programiści mają wpływ na mniejszy zakres zmian. W przypadku wdrażania zmiana przyrostowa dotyczy stopnia modułowości i rozprzężenia funkcji biznesowych, a także ich odwzorowania na architekturę. Czas na przykład.

Załóżmy, że firma Kapitalne Patenty, międzynarodowy sprzedawca widżetów, ma stronę katalogową bazującą na architekturze mikrousług i współczesnych technologiach inżynieryjnych. Jedną z cech tej strony jest możliwość oceniania różnych widżetów za pomocą systemu gwiazdek. Pozostałe usługi firmy Kapitalne Patenty również wymagają oceniania (obsługa klienta, czas dostarczenia produktu itd.), zatem wszystkie korzystają z systemu „gwiazdkowego". Pewnego dnia zespół odpowiedzialny za ten system wypuszcza nową wersję umożliwiającą przyznawanie połówek gwiazdek — niewielkie, ale istotne udoskonalenie. Pozostałe usługi nie wymagają korzystania z nowej wersji systemu oceniania, ale w celu zwiększenia wygody i tak stopniowo na nią przechodzą. W zakres technik DevOps stosowanych w przedsiębiorstwie Kapitalne Patenty wchodzi monitorowanie architektury nie tylko usług, lecz również połączeń pomiędzy nimi. Gdy grupa operacyjna zauważa, że nikt nie łączył się z określoną usługą w danym zakresie czasu, usługa ta zostaje automatycznie usunięta ze środowiska.

Jest to przykład zmiany przyrostowej na poziomie architektury: pierwotna usługa może działać wraz z jej nową wersją, dopóki wymagają jej pozostałe usługi. Zespoły mogą wprowadzać nowe zachowanie w stosownej chwili (lub w razie potrzeby), a stara wersja automatycznie zostaje przeniesiona do śmietnika.

Aby zmiany przyrostowe były skuteczne, wymagana jest koordynacja różnych rozwiązań w zakresie ciągłego dostarczania. Nie musimy w każdym przypadku wykorzystywać wszystkich dostępnych rozwiązań, ale często są one spotykane razem. Sposób wprowadzania zmian przyrostowych opisujemy w rozdziale 3.

Wielowymiarowość architektury

> Nie istnieją systemy odizolowane. Świat jest ciągły. Granice wokół systemu są wyznaczane w kontekście celu dyskusji.
>
> — Donella H. Meadows

Począwszy od starożytnych Greków fizyka była stopniowo rozwijana w celu analizowania Wszechświata jako zbioru o skończonej liczbie punktów, czego zwieńczeniem jest **mechanika klasyczna** (*https://pl.wikipedia.org/wiki/Mechanika_klasyczna*). Jednak pojawienie się precyzyjniejszych instrumentów badawczych i obserwacja bardziej złożonych zjawisk na początku XX wieku skierowały naszą uwagę na kwestię względności. Badacze zrozumieli, że to, co uprzednio uznawali za odizolowane zjawiska, w istocie stanowi sieć wzajemnych zależności. Począwszy od lat 90. ubiegłego wieku światli architekci zaczęli stopniowo dostrzegać wielowymiarowość architektury. Technika ciągłego dostarczania rozwinęła tę perspektywę również na kwestie operacyjne. Jednakże architekci oprogramowania często koncentrują się głównie na architekturze **technicznej** (wzajemnym dopasowaniu elementów oprogramowania), która stanowi zaledwie jedną z płaszczyzn projektu informatycznego. Jeśli architekci chcą tworzyć ewoluującą architekturę, muszą brać pod uwagę wszystkie

przeplatające się ze sobą elementy systemu, na które będą wpływać zmiany. Podobnie jak wiemy dzięki dokonaniom fizyków, że wszystko jest względne, tak architekci rozumieją, że projekt informatyczny składa się z wielu wymiarów.

Aby architekci byli w stanie tworzyć ewoluowalne systemy informatyczne, muszą wykraczać poza architekturę techniczną. Przykładowo, jeśli w projekcie jest wykorzystywana relacyjna baza danych, to jej struktura i relacje pomiędzy poszczególnymi encjami będą także ewoluować. Nie chcemy również budować systemu tak, aby w trakcie ewolucji pojawiały się kolejne dziury w zabezpieczeniach. Są to przykłady *wymiarów* architektury — jej składowych dopasowanych do siebie, często w ortogonalny sposób. Niektóre wymiary wpisują się w tak zwane **zagadnienia architektoniczne** (ang. *architectural concerns*; lista wspomnianych wcześniej „-ości"), w rzeczywistości jednak pojęcie *wymiarów* jest szersze, gdyż obejmuje elementy tradycyjnie wykraczające poza zakres architektury technicznej. Każdy projekt zawiera wymiary, które rola architekta musi brać pod uwagę podczas analizowania procesu ewolucji. Poniżej prezentujemy niektóre powszechnie występujące wymiary wpływające na ewoluowalność współczesnych architektur oprogramowania:

Techniczny

Elementy implementowane w ramach architektury: struktury, biblioteki i implementowane języki.

Danych

Schematy bazodanowe, układy tabel, planowanie optymalizacji itd. Wymiarem tym zajmuje się najczęściej administrator bazodanowy.

Zabezpieczeń

Definiuje strategie bezpieczeństwa, wytyczne oraz narzędzia pomagające odkrywać braki zabezpieczeń.

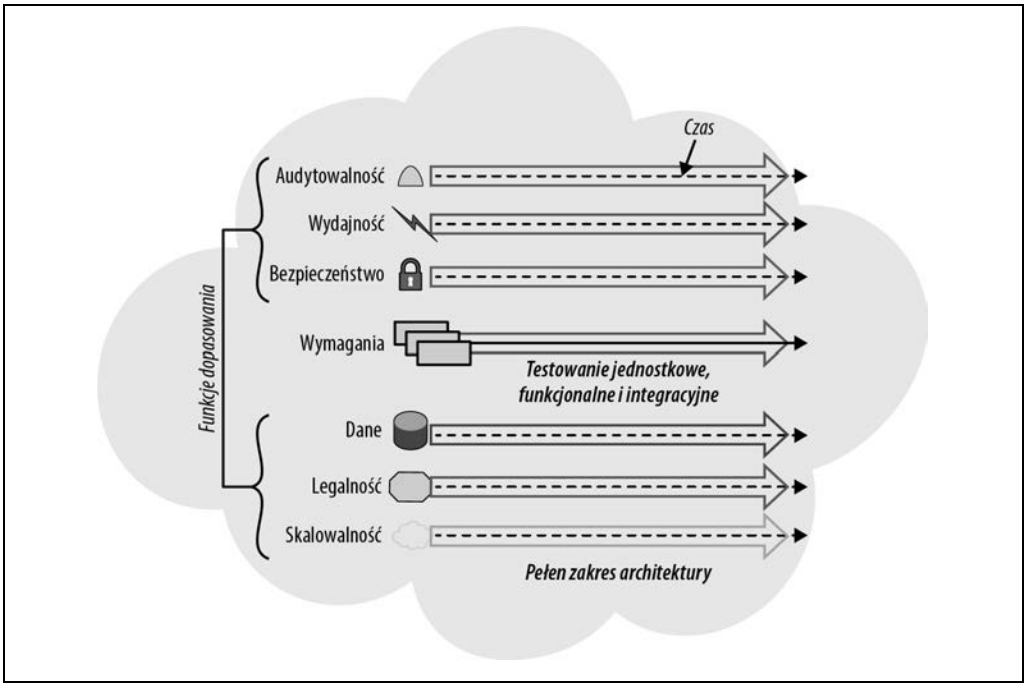
Operacyjny/systemowy

Dotyczy odwzorowania architektury na istniejącą fizyczną lub wirtualną infrastrukturę: serwery, klastry, switche, zasoby w chmurze itd.

Każdy z tych aspektów kształtuje określony *wymiar* architektury — celowe rozdzielenie składowych danej perspektywy. W naszym ujęciu wymiary architektoniczne zawierają tradycyjne parametry architektury („-ości"), a także każdą inną rolę stanowiącą część inżynierii oprogramowania. Tworzą one perspektywę architektury, którą chcemy zachować w miarę ewolucji naszego problemu i zmian otaczającego nas świata.

W kontekście wymiarów architektury istnieje mechanizm, za pomocą którego architekci mogą analizować ewoluowalność różnych architektur poprzez ocenę reakcji ważnego parametru na zmianę. Wraz ze wzrostem zależności systemów od ścierających się zagadnień (skalowalności, bezpieczeństwa, dystrybucji, transakcji itd.) architekci muszą rozszerzać obserwowane wymiary na projekty. Aby stworzyć ewoluowalny system, architekci muszą brać pod uwagę ewolucję tego systemu we wszystkich istotnych wymiarach.

Pełen zakres architektury projektu składa się z wymagań oprogramowania i pozostałych wymiarów. Możemy wykorzystać funkcje dopasowania do ochrony tych parametrów wraz z ewoluowaniem architektury i środowiska w czasie, co zostało ukazane na rysunku 1.2.



Rysunek 1.2. Każda architektura składająca się z wymiaru wymagań i pozostałych płaszczyzn jest chroniona przez funkcje dopasowania

Widzimy na rysunku 1.2, że architekci wyznaczyli **audytowalność**, **dane**, **bezpieczeństwo**, **wydajność**, **legalność** i **skalowalność** jako dodatkowe parametry architektury ważne dla danej aplikacji. Jako że wymagania biznesowe ewoluują w miarę upływu czasu, każdy z tych parametrów zawiera funkcję dopasowania chroniącą jego integralność.

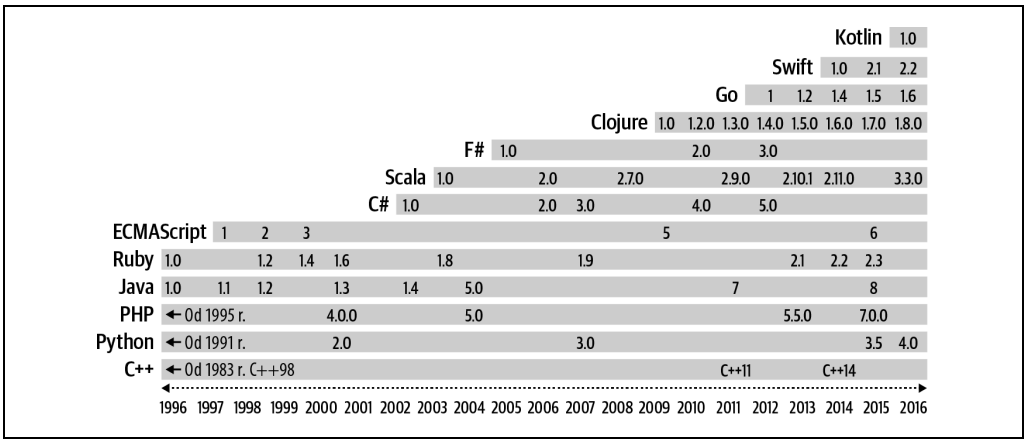
Podkreślamy znaczenie holistycznego ujęcia architektury, mamy jednak świadomość, że znaczna część ewolucji architektury wiąże się z jej wzorcami technicznymi oraz zagadnieniami pokrewnymi, takimi jak sprzężenie czy spójność. W rozdziale 5. analizujemy wpływ sprzężenia architektury technicznej na ewolucyjność, natomiast rozdział 6. poświęcamy tematyce konsekwencji sprzężenia danych.

Sprzężenie dotyczy nie tylko elementów strukturalnych projektu informatycznego. Ostatnio wiele firm informatycznych odkryło wpływ struktury zespołu na tak zaskakujące elementy, jak architektura. Omówimy wszystkie aspekty sprzężenia w oprogramowaniu, jednak wpływ zespołu występuje tak często, że nie możemy go zignorować.

Architektura ewolucyjna pomaga odpowiedzieć na dwa popularne pytania zadawane przez architektów we współczesnym środowisku inżynierii oprogramowania: *jak można planować coś długoterminowo, gdy wszystko zmienia się bez przerwy?* oraz *w jaki sposób możemy po stworzeniu architektury zabezpieczyć ją przed degradacją?*. Zastanówmy się dokładniej nad tymi pytaniami.

Jak można planować coś długoterminowo, gdy wszystko zmienia się bez przerwy?

Używane przez nas platformy programistyczne ilustrują taką ciągłą ewolucję. Nowsze wersje języka programowania oferują lepsze interfejsy programowania aplikacji (API), poprawiające elastyczność lub stosowalność wobec nowych typów problemów; z kolei nowsze języki programowania wprowadzają odmienny paradygmat i inny zestaw konstruktów. Przykładowo, język Java został wprowadzony jako zamiennik języka C++ w celu ułatwienia pisania kodu sieciowego i rozwiązania problemów z zarządzaniem pamięcią. Jeśli spojrzymy 20 lat wstecz, zauważymy, że w przypadku wielu języków ich interfejsy API są do teraz rozwijane, a nowsze języki programowania regularnie są dostosowywane do rozwiązywania nowych problemów. Ewolucja języków programowania została zaprezentowana na rysunku 1.3.



Rysunek 1.3. Ewolucja popularnych języków programowania

Bez względu na aspekt inżynierii oprogramowania — platformę programistyczną, język programowania, środowisko operacyjne, trwałe technologie, rozwiązania rozproszone itd. — oczekujemy ciągłych zmian. Chociaż nie potrafimy przewidzieć momentu pojawienia się zmian w krajobrazie technicznym lub domenowym ani czy będą one trwałe, to wiemy, że są nieuniknione. W rezultacie powinniśmy tworzyć architekturę systemów z myślą o tych zmianach.

Jeżeli środowisko ulega ciągłym zmianom w nieprzewidziany sposób i nie można założyć, kiedy będą się pojawiać, to jaką mamy *alternatywę* dla ustalonych planów? Architekci korporacyjni i pozostali programiści muszą nauczyć się sztuki adaptacji. Jeden z powodów sporządzania planów długoterminowych był natury finansowej; wprowadzanie zmian w oprogramowaniu było kosztowne. Jednak współczesne rozwiązania inżynieryjne obalają to założenie, gdyż obecnie wprowadzanie zmian jest tańsze z powodu automatyzacji części procesów oraz pojawienia się nowych technik, takich jak DevOps.

Wraz z upływem lat wielu bystrych programistów uświadomiło sobie, że niektóre części stworzonych przez nich systemów były trudniejsze do zmodyfikowania od innych. Dlatego **architektura oprogramowania** jest zdefiniowana jako „elementy, które trudno zmieniać w późniejszym terminie".

Taka wygodna definicja oddzieliła składniki, które można modyfikować bez większego wysiłku, od wymagających naprawdę trudnych zmian. Niestety definicja ta zaprowadziła w ślepy zaułek: założenie, że zmiany będą trudne, staje się samospełniającą się przepowiednią.

Kilka lat temu pewni innowacyjni architekci oprogramowania powrócili do problemu „elementów, które trudno zmieniać w późniejszym terminie": a może wbudować zmienność *w* architekturę? Innymi słowy, jeśli *swoboda zmian* stanie się fundamentalną zasadą architektury, to same zmiany przestaną być trudne. Wprowadzenie ewoluowalności do architektury oznacza pojawienie się zupełnie nowego zestawu zachowań, przez co zostaje ponownie zaburzona dynamiczna równowaga.

Nawet jeśli środowisko nie ulega zmianom, to co ze stopniową erozją parametrów architektury? Architekci projektują architekturę, ale później wystawiają ją na działanie zagmatwanego środowiska *implementowania* elementów w ramach tej architektury. W jaki sposób architekci chronią istotne, zdefiniowane przez siebie składniki?

W jaki sposób możemy po stworzeniu architektury zabezpieczyć ją przed degradacją?

Niefortunny rozpad, często zwany **gniciem bitów** (ang. *bit rot*), występuje w wielu organizacjach. Architekci dobierają określone wzorce architektoniczne w celu uwzględniania wymogów biznesowych i różnych „-ości", ale nieraz parametry te przypadkowo zanikają w miarę upływu czasu. Przykładowo, jeśli architekt zaprojektował wielowarstwową architekturę, zawierającą na szczycie warstwę prezentacji, na spodzie warstwę trwałości i kilka warstw pośrednich, programiści pracujący nad systemem raportowania często proszą o bezpośredni dostęp do warstwy trwałości z warstwy prezentacji, z pominięciem warstw pośrednich, gdyż uzyskują w ten sposób lepszą wydajność. Architekci wstawiają warstwy po to, aby odizolować zmiany. Programiści pomijają te warstwy, przez co zwiększają sprzężenie i niweczą cel ich obecności.

W jaki sposób architekci, po zdefiniowaniu istotnych parametrów architektury, mogą *chronić* je przed erozją? Dodanie **ewoluowalności** jako parametru architektury ma na celu ochronę pozostałych jej cech w miarę ewolucji systemu. Jeśli na przykład architekt zaprojektował architekturę pod względem skalowalności, to z pewnością nie chce, aby ten parametr zaniknął wraz z rozwojem układu. Zatem **ewoluowalność** jest metaparametrem, otoczką architektury chroniącą wszystkie pozostałe jej parametry.

Mechanizm architektury ewolucyjnej pokrywa się w znacznym stopniu z problemami i celami zarządzania architekturą: zdefiniowanymi zasadami projektowania, jakości, bezpieczeństwa i innych kwestii jakościowych. Ta książka ukazuje wiele sposobów, na jakie architektura ewolucyjna umożliwia automatyzację zarządzania architekturą.

Dlaczego ewolucyjna?

Jednym z częściej spotykanych pytań dotyczących architektury ewolucyjnej jest pochodzenie jej nazwy: dlaczego jest ona **ewolucyjna**, a nie jakaś inna? Można wymienić znacznie więcej określeń, jak choćby **przyrostowa**, **ciągła**, **zwinna**, **reaktywna** czy **wschodząca** (**emergentna**). Ale żadna z tych nazw nie oddaje w pełni istoty omawianej architektury. Podana przez nas definicja architektury ewolucyjnej zawiera dwa kluczowe parametry: przyrostowa i kierowana.

Pojęcia **ciągła**, **zwinna** i **wschodząca** ukazują koncepcję zmiany w czasie, co jest oczywiście kluczowym aspektem architektury ewolucyjnej, ale żadne z nich nie definiuje *sposobu, w jaki* te zmiany następują, ani jaki może być jej pożądany stan końcowy. Każdy z tych terminów sugeruje obecność zmiennego środowiska, ale żaden nie wskazuje, jak powinna wyglądać taka architektura. W naszej definicji za tę część odpowiada wyraz **kierowana** — odzwierciedla on architekturę, jaką chcemy uzyskać — nasz cel końcowy.

Wybraliśmy wyraz **ewolucyjna**, a nie **przystosowawcza**, ponieważ interesują nas architektury przechodzące przez fundamentalne zmiany ewolucyjne, a nie takie, które są modyfikowane i dostosowywane do stopniowo coraz mniej zrozumiałej, przypadkowej złożoności. **Przystosowywanie** sugeruje poszukiwanie sposobu zmuszania czegoś do pracy bez względu na jakość lub rozmiar rozwiązania. Aby tworzyć prawdziwie ewoluujące architektury, architekci muszą wprowadzać rzeczywiste zmiany, a nie prowizoryczne rozwiązania. Wróćmy na chwilę do naszej metafory biologicznej: **ewolucyjność** dotyczy procesu tworzenia układu dostosowanego do pełnienia określonej funkcji oraz zdolnego do przetrwania w zmiennym środowisku, w którym przyszło mu istnieć. Systemy mogą być pojedynczymi przystosowaniami, ale jako architekci powinniśmy troszczyć się o całokształt ewoluującego układu.

Korzystne może być także porównywanie architektury ewolucyjnej z **projektowaniem emergentnym** (ang. *emergent design*) oraz zastanowienie się, dlaczego nie ma czegoś takiego jak „architektura emergentna". Jednym z popularniejszych błędnych przekonań na temat zwinnej inżynierii oprogramowania jest domniemany brak architektury: „Zacznijmy po prostu pisać kod, a architektura wyłoni się sama". Zależy to jednak od złożoności problemu. Wyobraź sobie budynek fizyczny. Jeżeli chcesz zbudować psią budę, nie potrzebujesz architektury, wystarczy, że pójdziesz do sklepu budowlanego, kupisz deski i pozbijasz je gwoździami. Jeśli jednak musisz postawić pięćdziesięciopiętrowy wieżowiec, architektura zdecydowanie okaże się nieodzowna! Podobnie sytuacja wygląda w przypadku prostego systemu katalogowego dla małej liczby użytkowników: prawdopodobnie nie będziesz musiał wiele planować. Jeśli jednak projektujesz cały system oprogramowania mający rygorystyczne wymagania wydajności dla znacznej liczby użytkowników, to planowanie jest niezbędne! Celem architektury zwinnej jest nie *brak* architektury, lecz *brak bezużytecznej* architektury; pomija procesy biurokratyczne niedodające żadnej wartości do procesu inżynierii oprogramowania.

Kolejnym czynnikiem komplikującym architekturę oprogramowania jest występowanie różnych podstawowych stopni złożoności, z myślą o których architekci muszą projektować oprogramowanie. Podczas szacowania kompromisów często nie występuje wybór między *prostym* a *skomplikowanym* systemem, lecz między systemami, które są skomplikowane na różne sposoby. Innymi

słowy każdy system ma niepowtarzalny zbiór kryteriów sukcesu. Zajmujemy się takimi stylami architektury jak mikrousługi, gdzie każdy styl stanowi punkt odniesienia dla złożonego systemu rozwijającego się w niepowtarzalny sposób.

Na drodze analogii, jeśli architekt buduje bardzo prosty system, może sobie pozwolić na bagatelizowanie kwestii architektury. Jednak zaawansowane systemy wymagają świadomego planowania oraz jakiegoś punktu początkowego. **Emergencja** sugeruje, że możesz zacząć od niczego, natomiast architektura zapewnia szkielet bądź strukturę dla wszystkich pozostałych elementów systemu; aby móc zacząć, potrzebny jest jakiś zalążek.

Koncepcja **wyłaniania** sugeruje także, że zespoły mogą powoli doprowadzać do osiągnięcia idealnego rozwiązania architektonicznego. Jednak podobnie jak w przypadku architektury budynku, nie istnieje idealna architektura, lecz jedynie różne sposoby, na jakie architekci mogą radzić sobie z kompromisami. Architekci mogą udanie implementować większość problemów w szerokiej gamie różnych stylów architektonicznych. Jednak niektóre z tych rozwiązań mogą być lepiej dopasowane oraz oferować mniej przeszkód i mniej obejść problemów.

Kluczem do architektury ewolucyjnej jest równowaga między stopniem ustrukturyzowania i zarządzania niezbędnym do wspierania celów długoterminowych a niepotrzebnymi formalnościami i tarciami.

Podsumowanie

Przydatne systemy oprogramowania nie są statyczne. Muszą rozwijać się i zmieniać w miarę zmieniania się domeny problemowej i ewoluowania środowiska, wprowadzając nowe możliwości i poziomy złożoności. Architekci i programiści mogą skutecznie rozwijać systemy oprogramowania, ale muszą zarówno zrozumieć umożliwiające to niezbędne praktyki inżynieryjne, jak i określić najlepsze sposoby tworzenia struktur architektury ułatwiających zmianę.

Zadaniem architektów jest również zarządzanie projektowanym przez nich oprogramowaniem, a także wieloma praktykami projektowymi użytymi do jego stworzenia. Na szczęście omawiane przez nas narzędzia ułatwiające ewoluowanie oprogramowania zawierają również mechanizmy automatyzujące istotne czynności zarządzania oprogramowaniem. W następnym rozdziale przyjrzymy się dokładniej umożliwiającym to mechanizmom.

Naucz się postrzegać architekturę systemową jako plastyczny wyzwalacz.

Sam Newman, autor książki *Budowanie mikrousług*

Jeszcze kilka lat temu koncepcja ewoluowania architektury była uznawana za zbyt odważną. Uważano, że powinna ona pozostawać elementem niezmiennym w czasie. Jednak rzeczywistość udowadnia, że systemy muszą ewoluować, aby spełniać wymogi użytkowników i odzwierciedlać zmiany w dynamicznym środowisku tworzenia oprogramowania. Innymi słowy, konieczne się staje budowanie architektur ewolucyjnych.

Dzięki tej książce dowiesz się, w jaki sposób uczynić architekturę oprogramowania wystarczająco plastyczną, aby mogła odzwierciedlać zachodzące zmiany biznesowe i technologiczne. W nowym wydaniu rozbudowano pojęcia zmiany kierowanej i przyrostowej, a także przedstawiono najnowsze techniki dotyczące funkcji dopasowania, automatycznego zarządzania architekturą i danych ewolucyjnych. Zaprezentowano praktyki inżynieryjne umożliwiające ewoluowanie systemów oprogramowania, jak również podejścia strukturalne, w tym zasady projektowe, które ułatwiają zarządzanie tą ewolucją. Opisano ponadto, w jaki sposób zasady i praktyki architektury ewolucyjnej wiążą się z różnymi elementami procesu tworzenia oprogramowania.

Najciekawsze zagadnienia:

- mechanika architektury ewolucyjnej
- zarządzanie projektami oprogramowania i ich ewolucją
- style architektoniczne i zasady projektowania
- sprzęganie i wielokrotne użycie
- łączenie praktyk inżynieryjnych z kwestiami strukturalnymi

Neal Ford jest architektem oprogramowania w firmie konsultingowej Thoughtworks.

Dr Rebecca Parsons jest dyrektorką do spraw technicznych w firmie Thoughtworks.

Patrick Kua jest liderem technologicznym z ponad dwudziestoletnim doświadczeniem.

Pramod Sadalage specjalizuje się w projektach aplikacji i ewolucyjnych baz danych.

Poznaj techniki umożliwiające tworzenie architektur oprogramowania na tyle zwinnych, aby dotrzymywały kroku ciągłym zmianom.

Mark Richards, developertoarchitect.com

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 978-83-289-0066-0

9 788328 900660

Cena: 67,00 zł