O'REILLY®

Helion

Tworzenie architektury oprogramowania

Wspieranie zespołów w podejmowaniu trafnych decyzji

Andrew Harmel-Law

Tytuł oryginału: Facilitating Software Architecture: Empowering Teams
to Make Architectural Decisions

Tłumaczenie: Piotr Pilch

ISBN: 978-83-289-2750-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/twarop
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: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Projektowanie architektury to podejmowanie decyzji

Na tym etapie trzeba zagłębić się w fundamentalny element, który stanowi podstawę każdego podejścia do architektury oprogramowania, czyli decyzje. W tym rozdziale jasno zdefiniujemy, czym są decyzje architektoniczne. Natomiast w rozdziale 3. zostanie omówiony proces podejmowania decyzji, zarówno indywidualnie, jak i w grupach.

Zanim zacznę, chciałbym, żebyś coś dla mnie zrobił. Zamknij oczy i wyobraź sobie, że podejmowana jest decyzja dotycząca architektury oprogramowania. Gdy otworzysz oczy, zastanów się nad następującymi kwestiami:

- Jakie obrazy przyszły Ci na myśl?
- Czy to Ty podejmowałeś decyzję, czy ktoś inny, a może kilka osób?
- Czy osoba podejmująca decyzję zajmowała wysokie stanowisko w organizacji i „narzucała" ją z góry? A może była niżej w hierarchii od Ciebie? Czy może znajdowała się na tym samym poziomie co Ty?
- W jakim wieku była ta osoba? Czy była starsza, młodsza czy w podobnym wieku? Czy w ogóle zwróciłeś na to uwagę?
- Jakie umiejętności techniczne te osoby posiadają? Czy są one takie same jak Twoje, a może większe lub mniejsze? Czy dotychczas one programowały? W jakim języku? Czy znasz ten język? Czy uruchamiały swój kod w środowisku produkcyjnym? Na jaką skalę?
- Czy te osoby mają takie samo wykształcenie co Ty? A co z ich perspektywą i doświadczeniem? Czy są one takie same jak Twoje? Jeśli nie, w jaki sposób się różnią?

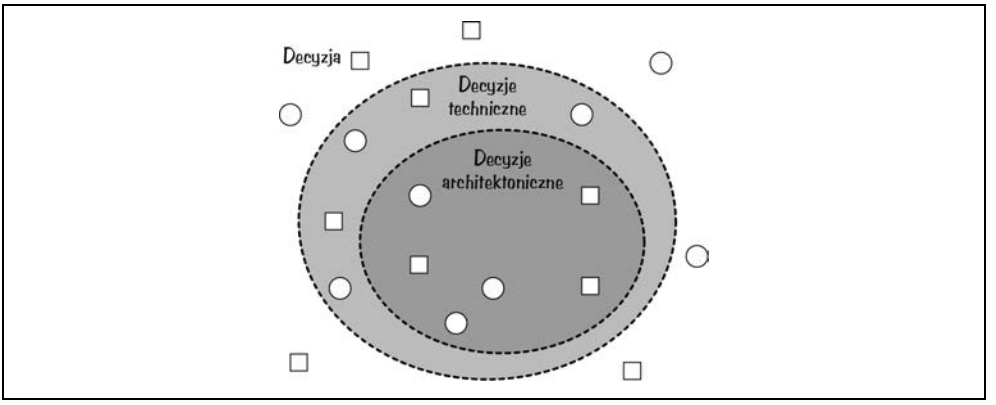
Miej w pamięci odpowiedzi na te pytania, czytając ten rozdział i odkrywając, co sprawia, że decyzja „architektoniczna" jest istotna. Po drodze zwrócę uwagę na powszechne założenia dotyczące decyzji, które mogą zaciemniać nasze myślenie. Możesz być zaskoczony tym, jak bardzo zmieni się Twoje obecne postrzeganie podejmowania decyzji dotyczącej architektury oprogramowania.

Decyzje są podstawą architektury oprogramowania

Niezależnie od wybranego podejścia praktyka architektury oprogramowania polega na podejmowaniu decyzji. Architektura oprogramowania jest pełna decyzji, tak wielu, że czasem możesz nawet nie zdawać sobie sprawy z ich istnienia. Podczas tworzenia oprogramowania musisz decydować o strukturze poszczególnych elementów, o tym, jak ma przebiegać proces ich tworzenia, itd. Musisz nawet zdecydować, czy Twój zespół powinien korzystać z zasobów zewnętrznych, takich jak biblioteka *open source*.

W tym kontekście architektury oprogramowania są sumą wszystkich decyzji architektonicznych podjętych w całym ich cyklu projektowym[1]. Wraz z pojawieniem się architektur ewolucyjnych[2] decyzje architektoniczne podejmowane są nieustannie przez cały okres funkcjonowania systemu w sposób ciągły i nieprzewidywalny. To ogromna liczba decyzji nawarstwiających się jedna na drugiej, a każda z nich stanowi utrwalony zapis struktur władzy i pętli sprzężenia zwrotnego (lub ich braku)[3].

W związku z tym kluczowe jest, aby wszyscy, zarówno osoby zajmujące się projektowaniem architektury, jak i programiści, dobrze rozumieli, czym *jest* decyzja w architekturze oprogramowania, a także *jak się ją podejmuje*. Takie zrozumienie sprzyja lepszemu zgraniu zespołów i pomaga wszystkim podejmować trafniejsze wybory. Na tej drodze jest kilka etapów, a pierwszym z nich jest jasne określenie, co sprawia, że dana decyzja jest architektoniczna, a nie tylko techniczna. Jak widać na rysunku 2.1, wszystkie decyzje architektoniczne są decyzjami technicznymi, lecz nie wszystkie decyzje techniczne są decyzjami architektonicznymi.



Rysunek 2.1. Wszystkie decyzje architektoniczne są decyzjami technicznymi, lecz nie wszystkie decyzje techniczne to decyzje architektoniczne

[1] Dla jasności: mam tutaj na myśli wyłącznie decyzje architektoniczne, a nie decyzje mające wpływ na architekturę. Nie oznacza to jednak, że te drugie nie wywierają istotnego wpływu.

[2] Zajrzyj do książki *Architektura ewolucyjna. Projektowanie oprogramowania i wsparcie zmian* (*https://helion.pl/ksiazki/architektura-ewolucyjna-projektowanie-oprogramowania-i-wsparcie-zmian-wydanie-ii-neal-ford-rebecca-parsons-patrick-kua-pramod-sa,arche2.htm#format/d*) napisanej przez Neala Forda, Rebeccę Parsons, Patricka Kua i Pramoda Sadalage.

[3] Temu tematowi poświęciłem wątek w serwisie Twitter: *Thoughts on the anthropology of software (power and freedom special edition)* (*https://x.com/al94781/status/1564288746228416513*).

Co stanowi decyzję architektoniczną?

W *Słowniku Cambridge decyzję* zdefiniowano jako „wybór dokonywany w odniesieniu do czegoś po rozważeniu kilku możliwości”[4]. Czym zatem jest decyzja *architektoniczna*[5]?

Szybkie wyszukiwanie w internecie ujawni *wiele* definicji architektury oprogramowania, a także prawie tyle samo definicji decyzji architektonicznej. Spośród dostępnych definicji ta zaproponowana przez Michaela Nygarda we wpisie *Dokumentowanie decyzji architektonicznych* (*https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions*) na blogu z 2011 r. okazała się dla mnie najbardziej przydatna w ramach praktyki architektury oprogramowania, głównie dlatego, że jest prosta do zrozumienia i łatwa do zapamiętania.

Autor stwierdza: „[…] decyzje istotne pod względem architektonicznym[6]: *te, które wpływają na strukturę, charakterystyki niefunkcjonalne, zależności, interfejsy lub techniki konstrukcyjne*” [wyróżnienie moje]. Definicja ta wyszczególnia pięć kryteriów identyfikujących decyzję architektoniczną. Przyjrzyjmy się bliżej każdemu z nich.

Struktura

Przede wszystkim decyzje strukturalne są architektoniczne. Wydaje się to oczywiste, ponieważ odnosi się do organizacji części systemu oprogramowania, ich funkcji oraz sposobu, w jaki ze sobą współdziałają (lub nie).

Im bardziej architektura jest zdecentralizowana, tym wyraźniej widoczne są jej „części”, głównie dlatego, że interakcje między nimi zwykle odbywają się za pośrednictwem sieci (np. za pomocą protokołu HTTP lub kolejki komunikatów). Jednakże aspekt „części” strukturalnych ma zastosowanie również w innych stylach architektonicznych. „Części” to zarówno moduły w „modularnym monolicie”, jak i zbiór mikrousług. W obu przypadkach nadal istnieją połączenia między nimi, choć są one mniej widoczne w ramach wywołań funkcji środowiska wykonawczego.

Cechy wielofunkcyjne

W trakcie pracy nad dowolnym oprogramowaniem zauważysz, że nie da się uchwycić wszystkich cech (czy wymagań), używając wyłącznie formatu funkcjonalnego. Istnieją cechy, które muszą zapewniać Twoje systemy, a które nie pasują do standardowego formatu historyjki użytkownika[7]

[4] *Słownik Cambridge*, hasło *decision* (*https://dictionary.cambridge.org/dictionary/english/decision*), stan z 28 kwietnia 2023.

[5] Inne decyzje mogą dotyczyć produktów, finansów, zatrudniania, struktury organizacyjnej i personelu itp. Choć wszystkie one są ważne, zajmuję się tutaj wyłącznie decyzjami architektonicznymi.

[6] W dalszej części tego rozdziału powrócę do znaczenia istotnych decyzji architektonicznych. Na razie potraktuj definicję po prostu jako całość.

[7] Najlepsze wprowadzenie do historyjek użytkownika można znaleźć w rozdziale 6. książki *Mapowanie historyjek użytkownika* (*https://helion.pl/ksiazki/mapowanie-historyjek-uzytkownika-przepis-na-produkt-idealny-jeff-patton,maphiv.htm#format/e*) Jeffa Pattona i Roya McCrereya (Helion, 2016).

lub przypadku użycia[8] z pojedynczą funkcją. Niektóre wymagania po prostu nie definiują konkretnego przypadku czegoś. Co np. z wymaganiami dotyczącymi wydajności, bezpieczeństwa, przepisów, skalowalności, możliwości obsługi, użyteczności, odporności i kosztów eksploatacji?

Mógłbym tak długo wymieniać. Tego typu wymagania, zamiast określać zachowania różnych rzeczy, pozwalają ocenić działanie całego systemu.

Cechy te są również powszechnie określane jako „wymagania niefunkcjonalne" lub „atrybuty jakości". Oba terminy nie są jednak pozbawione problemów[9]. Nygard określa je mniej problematycznie jako „cechy niefunkcjonalne", ale wciąż pojawia się tu słowo „nie", które sugeruje brak czegoś. W książce będę używał terminu *cechy wielofunkcyjne* (ang. *cross-functional characteristics*), które są określane przez wymagania wielofunkcyjne CFR (ang. *Cross-Functional Requirements*). Uważam, że dokładniej oddaje to potrzeby, które zaspokoiłem, mające zastosowanie *w całym systemie*.

Decyzja, która jest odpowiedzią na tego typu potrzeby, również ma charakter architektoniczny. Architektura systemu przeznaczonego dla dziesięciu jednoczesnych użytkowników będzie się znacząco różniła od architektury systemu obsługującego równocześnie dziesiątki tysięcy użytkowników.

Zależności

Lubię traktować *zależności* jako elementy prowadzące interakcję z systemem, lecz takie, *nad którymi nie mamy kontroli*. Ten brak kontroli może wynikać z faktu, że dany system jest usługą dostarczaną przez zewnętrznego dostawcę. Może to być również platforma, na której działa ten system (np. w postaci kontenera zapewnianego przez środowisko typu Drupal, środowiska uruchomieniowego .NET Common Language Runtime lub systemu operacyjnego). Albo nawet może to być interfejs API oferowany przez zespół z innego działu tej samej firmy. Wszystkie te przypadki stwarzają ten sam problem, czyli niewielkie albo żadne możliwości kontroli nad ich harmonogramem prac i priorytetem zadań. W związku z tym trzeba się zastanowić nad sposobem łączenia zależności i komunikowania się z nimi. Decyzje, które z tego wynikają, mają charakter architektoniczny.

---

[8] Przed popularyzacją historyjek użytkownika przez programowanie ekstremalne powszechnie stosowano przypadki użycia. Po raz pierwszy zostały one zaproponowane przez Ivara Jacobsona na konferencji OOPSLA 1987 i stały się kluczowym elementem notacji diagramów UML oraz procesu RUP (*Rational Unified Process*). Przypadki użycia to bardziej rozbudowany, oparty na dokumentacji sposób zbierania wymagań użytkowników względem oprogramowania.

[9] Problem polega na tym, że niektóre z tych wymagań można uznać za funkcjonalne (np. niektóre wymagania związane z regulacjami). Sarah Taraporewalla, moja koleżanka z firmy Thoughtworks, ukuła anglojęzyczny termin *cross-functional* (*https://sarahtaraporewalla.com/agile/design/decade_of_cross_functional_requirements_cfrs*), aby nie tylko uniknąć tego problemu, ale także przeciwdziałać mylnemu przekonaniu ludzi co do tego, że przedrostek *non* („nie") w słowie *nonfunctional* („niefunkcjonalny") sugeruje, iż te wymagania są jakoś mniej istotne niż wymagania funkcjonalne. Obecnie oba terminy są stosowane w dużym stopniu wymiennie. FURPS+ to inny termin, popularny szczególnie wśród starszego pokolenia. To także nie jest idealne rozwiązanie (znak + to swego rodzaju unik), ale w tamtym czasie większość ludzi wiedziała, co on oznacza.

Interfejsy

Interfejsy to druga strona zależności, która jest równie ważna, choć w inny sposób. W przeciwieństwie do zależności *interfejs* pojawia się, gdy Twoje oprogramowanie jest wykorzystywane przez inny podmiot. Udostępniając interfejs, stajesz się tym, od kogo inni zależą w realizacji swoich celów. Choć masz pełną kontrolę nad swoim harmonogramem, musisz być tutaj ostrożny. Nawet drobne zmiany w Twoich interfejsach API mogą zakłócić integracje użytkowników, zmuszając ich do dokonania modyfikacji. W efekcie decyzje wpływające na interfejs, który Twój system udostępnia innym, mają charakter architektoniczny.

Relacje między systemami mogą być dwukierunkowe

Czasami Twój i inny system będą wzajemnie zależne. Twój system będzie korzystał z interfejsu systemu, który też używa interfejsu po stronie Twojego systemu. Jeśli brzmi to skomplikowanie i wygląda na koszmar pod względem zarządzania, masz rację, gdyż tak właśnie jest. Wielokierunkowe relacje mogą szybko stać się „piekłem", ponieważ wszyscy muszą myśleć o swojej architekturze z obu perspektyw.

Techniki budowania

Wreszcie pojawiają się *techniki budowania*, co prawdopodobnie tłumaczy, dlaczego tak bardzo cenię definicję decyzji architektonicznych sformułowaną przez Nygarda. W książce *Release It!* (Pragmatic Bookshelf) był on jednym z pierwszych propagatorów projektowania architektury z myślą o „budowaniu i uruchamianiu"[10], dlatego nie dziwi, że w swojej definicji uwzględnił techniki budowania.

W ciągu ostatniej dekady lub w dłuższym czasie od początków każdej z rewolucji w dziedzinie oprogramowania, które opisałem w rozdziale 1., coraz więcej wniosków zostało uwzględnionych zarówno w narzędziach, jak i technikach tworzenia i publikowania oprogramowania. W konsekwencji zmieniających się metod budowania zmieniły się również same architektury naszych systemów.

Jak to? Pozwól, że wyjaśnię.

Był rok 2005, gdy po raz pierwszy zobaczyłem wpływ rozwiniętych technik budowania. Dołączyłem wtedy do projektu opartego na platformie J2EE[11], w którym kod był podzielony

[10] Powszechnie uznaje się, że koncepcję tę spopularyzował Werner Vogels, dyrektor ds. technicznych w firmie Amazon, w wywiadzie udzielonym w 2006 r. (*https://queue.acm.org/detail.cfm?id=1142065*) dla serwisu ACMQueue. Stwierdził on: „Tradycyjny model zakłada, że bierzesz swoje oprogramowanie, przerzucasz je przez mur oddzielający dział projektowy od działu operacyjnego, a następnie o nim zapominasz. W firmie Amazon tak nie jest. Jeśli budujesz oprogramowanie, korzystasz z niego. Podejście to zapewnia projektantom kontakt z codzienną obsługą operacyjną ich oprogramowania, a także z jego użytkownikami. Taka pętla informacji zwrotnej od klientów jest kluczowa dla poprawy jakości usługi".

[11] Projekt był realizowany dwie przecznice od drogi, przy której miała miejsce eksplozja w Buncefield (*https://en.wikipedia.org/wiki/Buncefield_fire#Explosion_and_fire*), dlatego z różnych powodów jest dobrze zapamiętany.

na moduły w sposób, który dziś nazwano by architekturą „heksagonalną", „cebuli" lub „portów i adapterów" (*https://web.archive.org/web/20071028094131/http:/alistair.cockburn.us/index.php/Hexagonal_architecture*). Oznaczało to, że istniały wyodrębnione pakiety domenowe, które nie były zależne od standardowych bibliotek platformy Java EE. Dlaczego to zrobiliśmy? Z myślą o możliwości testowania. W tamtych czasach uruchomienie serwera J2EE lokalnie na maszynie projektowej w celu przeprowadzenia zautomatyzowanych testów mogło zająć nawet 25 minut. Większość „elementów inteligencji" znajdowała się w tych pakietach domenowych. Właśnie te elementy chcieliśmy testować najbardziej szczegółowo, a żeby to osiągnąć, potrzebowaliśmy szybko działających testów. Izolując te pakiety, mogliśmy uruchamiać testy kluczowych elementów bez konieczności aktywowania całej powiązanej infrastruktury. I to się sprawdziło. Zajmując się funkcjonalnością głównego komponentu, mogliśmy pracować, korzystając ze stylu projektowania Red Green Refactor opartego na testach z pętlą informacji zwrotnych trwającą 10 – 15 sekund. Szybko i bezpiecznie.

Od 2005 r. inne praktyki programowania ekstremalnego (*http://www.extremeprogramming.org*) stały się standardem (w szczególności ciągłe wdrażanie[12]), realizując korzyści wynikające z „nowej fizyki". Obecnie jesteśmy w stanie niezwykle szybko i często dostarczać kod do środowiska produkcyjnego i udostępniać go klientom. Zalety tego podejścia są dobrze znane. Wspomniałem o nich podczas omawiania pięciu rewolucji w rozdziale 1. Kluczowe jest to, że aby móc ciągle wdrażać oprogramowanie, trzeba stosować określone techniki budowania, ponadto niezbędne jest dokonanie określonych wyborów dotyczących architektury. Na przykład użycie takiego narzędzia jak LaunchDarkly (*https://launchdarkly.com*) do zarządzania przełącznikami wersji nigdy nie będzie opcją widoczną dla użytkowników (można uznać za porażkę sytuację, w której kiedykolwiek zauważą jej obecność), ale ma wpływ na Twój kod.

Stosowanie przełączników opcji, ciągłego wdrażania, a co za tym idzie: potoków i wielu innych rozwiązań, zalicza się do kategorii technik budowania. Związane z tym decyzje mają charakter architektoniczny, ponieważ wpływają na sposób, w jaki kształtujemy nasze oprogramowanie.

Czy powinienem przejmować się umiejscowieniem platform, języków, środowisk i bibliotek?

Choć można dyskutować, czy wybór platformy, języka, środowisk i bibliotek najlepiej pasuje do kryteriów struktury lub technik budowy, są one niewątpliwie istotne, nieważne gdzie je umieścimy w pięciu kryteriach decyzji architektonicznych Nygarda. W większości przypadków można argumentować za obiema opcjami. W niektórych sytuacjach mogą one nawet wchodzić w zakres cech wielofunkcyjnych. Niezależnie od tego, jak zdecydujemy się je sklasyfikować, wybory dotyczące platform, języków, środowisk i bibliotek z pewnością spełniają definicję decyzji architektonicznych.

[12] Aby uzyskać pełne informacje na ten temat, zajrzyj do książki *Continuous Deployment* (O'Reilly, 2024, *https://helion.pl/ksiazki/continuous-deployment-valentina-servile,e_4098.htm*) Valentiny Servile.

Przykłady decyzji architektonicznych i niearchitektonicznych

Jak wspomniano, decyzje architektoniczne muszą spełniać co najmniej jedno z pięciu kryteriów: powinny wpływać na strukturę, cechy wielofunkcyjne, zależności, interfejsy i/lub techniki budowania. Pora przećwiczyć dokonywanie takich ustaleń, analizując poniższe listy, które zawierają decyzje architektoniczne i niearchitektoniczne.

Decyzje architektoniczne

Struktura

Jak podzielić moduł i jakie będą nowe granice?

Które zespoły będą odpowiedzialne za poszczególne usługi (choć jest to decyzja dotycząca organizacji i personelu, powszechnie uznaje się obecnie, że ma ona bezpośredni wpływ na ostateczną strukturę)?

Cechy wielofunkcyjne

Format, jakiego wszyscy będą używać do zapisywania swoich dzienników, a także miejsce, do którego będą one wysyłane w celu gromadzenia.

Sposób skalowania poziomego (albo nie, gdyż dostępne są też inne formy skalowania) różnych aspektów swojego systemu.

Zależności

Jak dokonać integracji z nowym interfejsem API zewnętrznego dostawcy?

Interfejsy

Jak udostępniać funkcjonalność lub dane za pośrednictwem interfejsu API albo kolejki komunikatów (zarówno innemu zespołowi, z którym współpracujesz, jak i w przyszłości nieznanemu zewnętrznemu dostawcy)?

Techniki budowania

Jakiego środowiska użyć do testowania kontraktów?

Jak korzystać z przełączników wersji?

Zależności, prawdopodobna struktura oraz całkiem możliwe cechy wielofunkcyjne

Aktualizacja do nowej wersji głównej używanego środowiska[13].

Zależności i najbardziej prawdopodobne cechy wielofunkcyjne

Wybór w celu użycia podstawowego typu wirtualnych zasobów obliczeniowych dostawcy usług chmurowych.

[13] Zdaję sobie sprawę z tego, że mogłeś mieć zupełnie inne doświadczenia. Wynika to z nieprzestrzegania *zasad wersjonowania semantycznego* (*https://semver.org*). Na samej górze strony tej specyfikacji stwierdzono, że *główne* wersje wchodzą w grę wtedy, gdy środowisko wprowadza niezgodne zmiany w interfejsie API.

Decyzje niearchitektoniczne

Cechy, które nie są wielofunkcyjne

Jakie parametry strojenia przekazać procedurze odzyskiwania pamięci używanej maszyny wirtualnej? Można się spodziewać, że będą one zmieniane w trakcie użytkowania systemu. Chociaż można zaprojektować system tak, aby zapewnić dużą elastyczność w tym obszarze, zmiana tych ustawień nie jest decyzją architektoniczną.

Nie ma znaczenia to, czy skalowanie poziomu obejmuje 5, 7 czy 30 instancji części systemu (jest to po prostu efekt uboczny skalowania systemu).

Nie są to interfejsy

Który segment klientów wybierzesz spośród zestawu dostępnych opcji (jest to decyzja biznesowa/produktowa)?

Nie są to techniki budowania

Z jakiego środowiska projektowego IDE powinni skorzystać projektanci (nie powinno to mieć wpływu na architekturę systemu; jest to indywidualna decyzja projektanta)?

Podejmowanie decyzji dotyczącej tego, czy projektanci będą pracować w parach w przypadku danej historyjki użytkownika (jest to decyzja dotycząca praktyki zespołowej; zespół może jednak zdecydować się na częstsze działanie w parach w przypadku historyjek użytkownika o większym znaczeniu architektonicznym).

Jaką konwencję nazewniczą chcesz zastosować względem folderów i plików (nie powinno to mieć wpływu na strukturę systemu)?

Nie są to zależności ani techniki budowania

Wybór biblioteki do testów jednostkowych, która zostanie użyta (sposób projektowania architektury systemu pod kątem testów jednostkowych prawdopodobnie nie zmieni się, jeśli zmienisz konkretną bibliotekę do tych testów, choć one same niewątpliwie ulegną zmianie).

Wybór narzędzia do testowania wydajności to także kwestia zewnętrzna względem systemu (choć można zaprojektować architekturę w sposób ułatwiający testowanie wydajności, sam wybór oprogramowania do generowania obciążenia nie jest decyzją architektoniczną).

Kto podejmuje te decyzje architektoniczne?

Przed przejściem dalej wróćmy do naszego ćwiczenia wizualizacyjnego. Kogo na początku rozdziału wyobraziłeś sobie jako osobę podejmującą decyzje architektoniczne? Prawdopodobnie w przypadku większości z nich dostrzegłeś architekta lub głównego inżyniera. Jednakże w odniesieniu do innych decyzji, takich jak wybór środowiska do testów kontraktów, tak naprawdę mogła to być osoba z zespołu projektowego (np. inżynier ds. zapewnienia jakości). Zwróć na to uwagę. Decyzje architektoniczne mogą być podejmowane przez każdego, a nie tylko przez osoby z tytułem „architekta" w nazwie stanowiska.

Pora zająć się doprecyzowaniem kryteriów decyzji architektonicznych, koncentrując się na tych *najbardziej istotnych*.

Istotne decyzje architektoniczne

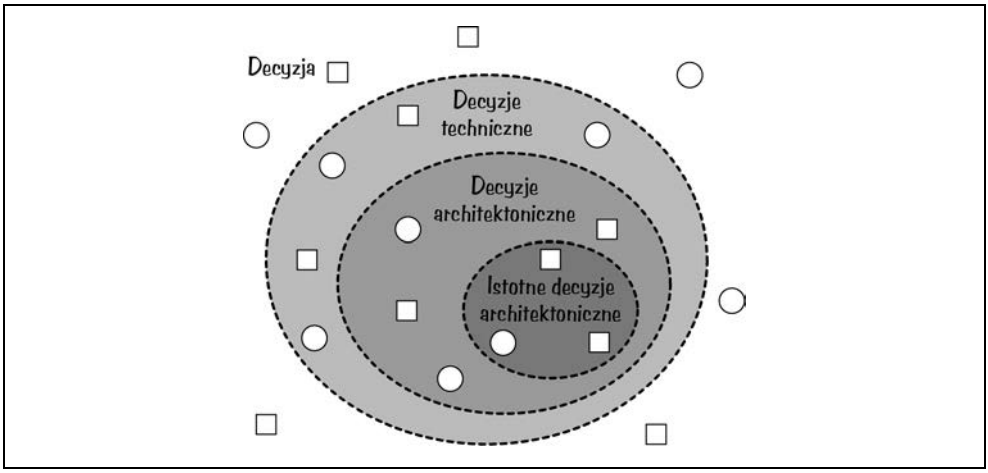
Podczas tworzenia, dostarczania, uruchamiania i rozwijania systemów oprogramowania decyzje architektoniczne podejmowane są nieustannie. Nie wszystkie jednak mają taki sam wpływ.

Choć decyzje podejmowane przez architektów *powinny być* istotne z punktu widzenia architektury, ważne jest zrozumienie, że projektanci również mogą podejmować znaczące decyzje architektoniczne. Oznacza to, że nie można oceniać tego, czy decyzja zasługuje na uwagę, wyłącznie na podstawie tego, kto ją podjął lub ile czasu zajęło jej podjęcie.

Doprecyzujmy pięć kryteriów podejmowania decyzji architektonicznych pozwalających uznać je za *istotne* z punktu widzenia architektury.

Co sprawia, że decyzja architektoniczna jest „istotna"?

Tak jak nie wszystkie decyzje techniczne są architektoniczne, tak nie wszystkie decyzje architektoniczne mają jednakowe znaczenie lub wpływ. Zilustrowano to na rysunku 2.2.



Rysunek 2.2. Nie wszystkie decyzje architektoniczne są „znaczące"

Możesz określić, czy decyzja architektoniczna jest istotna, jeśli w wyróżniający się sposób zawiera bardziej skonkretyzowane wersje wcześniejszych kryteriów, czyli strukturę, cechy wielofunkcyjne, zależności, interfejsy lub techniki budowania. Stosunkowo łatwo jest zidentyfikować znaczącą decyzję architektoniczną na podstawie trzech spośród tych kryteriów (zależności, interfejsy i techniki budowania), ponieważ skala ich wpływu jest przewidywalna. Zacznijmy więc od nich.

Twoja decyzja architektoniczna jest istotna, jeśli uwzględnisz:

- Zupełnie nową zależność lub (w przypadku istniejącej zależności) aktualizację do wersji głównej tego samego dostawcy albo całkowitą zmianę dostawcy.

- Nowy interfejs API w miejsce posiadanego interfejsu lub wprowadzenie w nim dowolnej zmiany, która wymagałaby aktualizacji do wersji głównej z myślą o konsumentach[14].
- Nową technikę budowania w ramach środowiska produkcyjnego (np. wdrożenia kanarkowe lub niebiesko-zielone) lub jako etap w potoku.

Trudniej jest zidentyfikować istotną decyzję architektoniczną na podstawie pozostałych dwóch kryteriów (struktura i cechy wielofunkcyjne), ponieważ skala ich wpływu jest trudniejsza do określenia ilościowego. Znaczące decyzje architektoniczne związane ze strukturą dotyczą zwykle albo *umiejscowienia funkcji* (gdzie znajduje się jakaś logika lub odpowiedzialność za coś), albo decyzji o rozpoczęciu (lub zaprzestaniu) stosowania nowego wzorca projektowego lub wzorca architektonicznego wyższego poziomu.

Wzorce projektowe

Mówiąc o *wzorcu*, mam na myśli powszechnie znane i wielokrotnie używane rozwiązania problemów często spotykanych w strukturze oprogramowania. Termin *wzorzec projektowy* (ang. *design pattern*) odnosi się do koncepcji, która zyskała szeroką popularność w środowisku projektantów oprogramowania po wydaniu w 1994 r. książki *Wzorce projektowe* (Helion, *https://helion.pl/ksiazki/wzorce-projektowe-elementy-oprogramowania-obiektowego-wielokrotnego-uzytku-erich-gamma-richard-helm-ralph-johnson-john-vli,wzoevv.htm#format/d*) Ericha Gammy, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa. Kolejne prace rozwinęły ten temat, np. seria książek *Pattern-Oriented Software Architecture* (Wiley), książka *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe* (Helion, *https://helion.pl/ksiazki/architektura-systemow-zarzadzania-przedsiebiorstwem-wzorce-projektowe-martin-fowler,szabko.htm#format/d*) Martina Fowlera, książka *Domain-Driven Design. Zapanuj nad złożonym systemem informatycznym* (Helion, *https://helion.pl/ksiazki/domain-driven-design-zapanuj-nad-zlozonym-systemem-informatycznym-eric-evans,dddzap.htm#format/e*) napisana przez Erica Evansa, książka *Enterprise Integration Patterns* (Addison-Wesley) Gregora Hohpe'a oraz książka *Cloud Architecture Patterns* (O'Reilly) Billa Wildera. Wszystkie te książki wprowadziły nie tylko nowe wzorce projektowe, ale także wzorce architektoniczne. Gorąco polecam zapoznanie się z nimi, choć nie ma potrzeby uczenia się ich wszystkich na pamięć. Ja sam z pewnością tego nie zrobiłem.

Decyzja architektoniczna jest zatem istotna (w sensie strukturalnym), jeśli wiąże się z umieszczeniem kluczowej logiki lub odpowiedzialności w konkretnym miejscu albo z rozpoczęciem lub zaprzestaniem stosowania jednego lub więcej uznanych wzorców projektowych opisanych w książkach wymienionych w powyższej ramce.

Na koniec docieramy do najtrudniejszego do zdefiniowania spośród pięciu kryteriów. Mowa o decyzjach wpływających na cechy wielofunkcyjne. Wspomniałem wcześniej, że pojęcie *cech wielofunkcyjnych* obejmuje szeroki i zróżnicowany zbiór niejasnych koncepcji, lecz to nie dlatego trudno jest określić, co sprawia, że dana cecha wielofunkcyjna jest istotna. Trudność w ich

[14] *Wersjonowanie semantyczne* (*https://semver.org*) pozwala we właściwy sposób ustalać, kiedy coś powinno być traktowane jako zmiana dotycząca głównej wersji.

sformułowaniu wynika z tego, że powiązane z nimi wymagania wielofunkcyjne CFR są kiepsko zdefiniowane. Gdy uda się doprecyzować te wymagania, łatwiej będzie zdefiniować cechy wielofunkcyjne i znacznie prostsze stanie się ustalenie ich znaczenia.

Jak zatem uzyskać przejrzystość definicji w przypadku wymagań CFR? Przewidywane wymagania należy sformułować dokładnie tak samo jak wszystkie inne wymagania, czyli przez wyraźne określenie ich zakresu, wyjaśnienie ich wartości i zdefiniowanie ich kryteriów akceptacji.

Być może nie jest to zaskakujące, ale dobrym sposobem na zrobienie tego jest wykorzystanie standardowej formy historyjki użytkownika. Zacznijmy od wartości:

PRZYKŁAD

„Odgrywając rolę [ROLA]…, oczekuję działania [DZIAŁANIE]…, aby zapewnić wartość [WARTOŚĆ]"

Jako następne występują kryteria akceptacji:

PRZYKŁAD

„Biorąc pod uwagę okoliczności [OKOLICZNOŚCI]…, gdy wystąpi zdarzenie [ZDARZENIE]…, uzyska się wynik [WYNIK]"

Gdy już poświęcisz czas na osiągnięcie tego poziomu szczegółowości, masz realną szansę na stwierdzenie, czy coś jest istotne (zamieszczając znacznie bardziej szczegółowe omówienie w rozdziale 8., powrócę w nim do tego sposobu uzyskiwania wymagań funkcjonalnych CFR, ale na razie tyle wystarczy).

Decyzja architektoniczna jest istotna, jeśli zmienia naszą zdolność do spełnienia co najmniej jednego wymagania CFR. Przyjmijmy następujące wymaganie CFR:

PRZYKŁAD

Jako klient

Chcę otrzymywać wyniki wyszukiwania w ciągu 500 milisekund

Aby móc szybko znaleźć to, czego szukam

Mając na uwadze to, że witryny dotyczą normalne poziomy obciążenia żądaniami wyszukiwania

Po wysłaniu przez klienta żądania

System zwraca w 99% przypadków poprawne wyniki wyszukiwania w czasie krótszym niż 500 milisekund

Jeśli podejmuję decyzję, która może zagrozić spełnieniu kryteriów akceptacji, jest ona istotna.

Podsumowując: decyzję architektoniczną można uznać za istotną, gdy spełnia ona co najmniej jedno z pięciu kryteriów decyzji architektonicznej.

Czego nie należy brać pod uwagę przy ocenie znaczenia z punktu widzenia architektury?

Pora spojrzeć na drugą stronę medalu: czego *nie powinniśmy* brać pod uwagę przy ocenie znaczenia z punktu widzenia architektury?

Porównajmy ze sobą dwa przykłady.

Po pierwsze, wyobraź sobie sytuację, w której projektant pracuje nad kodem i zauważa, że jedna z zależności jest nieaktualna, ale dotyczy to jedynie braku wersji zawierającej poprawkę błędu. Decyduje się ją zaktualizować, uruchamiając skrypt kompilacji. Nie zdaje sobie jednak sprawy, że ta operacja przypadkowo powoduje jednoczesne zaktualizowanie wielu innych zależności przechodnich. W rezultacie jedna z tych zależności zmienia swoją licencję oprogramowania *open source* z licencji MIT (którą dział bezpieczeństwa informacji akceptuje) na licencję GPL 3.0 (nie jest ona akceptowana ze względu na jej tzw. wirusową naturę)[15].

Po drugie, wyobraź sobie architekta, który przeprowadza długi, szczegółowy i technicznie skomplikowany proces analizy decyzji o przeniesieniu kluczowej części systemu z relacyjnej bazy danych do bazy dokumentów (konkretnie z bazy MySQL do bazy MongoDB). W trakcie tego procesu angażuje wielu ekspertów, przeprowadza pewne testy i prowadzi szeroko zakrojone badania, a nawet organizuje pojedynek „wypieków[16]" między dostawcami obu rozwiązań, aby sprawdzić, jak ich produkty radzą sobie przy realistycznych wolumenach danych i profilach obciążenia.

Jakie wnioski można wyciągnąć z tych dwóch przykładów? Pojawiają się trzy wnioski.

Pierwszy wniosek, do którego nawiązywałem przez cały czas, jest taki, że każdy może podejmować decyzje dotyczące czegoś o znaczeniu architektonicznym. Choć najczęściej robią to architekci, nie są oni jedynym źródłem decyzji. Projektanci także mogą je podejmować. Nie można określić, czy decyzja jest istotna, bazując jedynie na tym, kto był w nią zaangażowany. W związku z tym oba te przykłady mają znaczenie architektoniczne niezależnie od tego, kto brał w nich udział (motyw wskazujący, że każdy może podejmować decyzje, jest kluczowy dla tej książki i będzie pojawiał się wielokrotnie).

Drugi wniosek jest taki, że czas poświęcony na proces decyzyjny nie ma znaczenia. Długotrwały proces wyboru nowej bazy danych, która ma zastąpić istniejącą bazę MySQL, jest nieistotny. Bardzo możliwe, że w innych okolicznościach ta sama decyzja mogłaby zostać podjęta w ciągu kilku sekund przez dyrektora ds. technologii lub głównego architekta, którzy opieraliby się wyłącznie na swoich osobistych uprzedzeniach lub preferencjach. Podobnie natychmiastowość „decyzji" o użyciu zależności objętej licencją GPL nie byłaby ani mniej, ani bardziej

[15] W tym przypadku unikanie licencji GPL 3.0, a nawet wszystkich „wirusowych" licencji *open source*, powinno być ujęte jako wymaganie CFR i w idealnej sytuacji testowane w ramach potoku.

[16] „Wypieki" (ang. *bake-off*) to potoczne określenie konkursu pomiędzy firmami o zdobycie kontraktu. Termin ten stał się znany dzięki brytyjskiemu programowi telewizyjnemu *The Great British Bake-Off* (znanemu w Ameryce Północnej jako *The Great British Baking Show*, co bywa mylące).

znacząca, gdyby poddano ją kilkudniowej analizie. Oba te przykłady są istotne z punktu widzenia architektury niezależnie od tego, ile czasu zajęło podjęcie decyzji.

Trzeci wniosek mówi o tym, że decyzja nie musi być celowa, aby miała znaczący wpływ na architekturę. Przypomnijmy sobie ponownie przypadkowe dodanie zależności z licencją GPL. Zmiana ta nie była planowana. Łatwo mogłaby umknąć uwadze, ponieważ nic by nas o niej nie poinformowało. Kompilacja nadal by działała, a kod wykonywałby swoje zadanie.

Wszystkie te elementy są nieistotne w kontekście znaczenia decyzji. Oba przykłady są architektonicznie znaczące niezależnie od tego, czy osoby odpowiedzialne za decyzje były świadome ich wagi.

Znaczenie architektury w kontekście wdrożonego oprogramowania

Dysponujemy zatem kryteriami istotności, czyli kluczowymi decyzjami dotyczącymi struktury, cech wielofunkcyjnych, zależności, interfejsów i technik budowania. Wiadomo też dobrze, jakie czynniki są nieistotne: kto podejmuje decyzje, jak długo to trwa i czy proces był świadomy, czy nie. Ważne jest jednak, aby zwrócić uwagę na jeszcze jeden aspekt, który kryje się za tym wszystkim. W dzisiejszym zrewolucjonizowanym świecie decyzje są istotne tylko wtedy, gdy weźmiemy pod uwagę ich związek z wdrożonym oprogramowaniem: albo w odniesieniu do ścieżki prowadzącej system do środowiska produkcyjnego, albo jego cech, gdy już tam trafi.

Może się to wydać arbitralne, dlatego pozwól, że wyjaśnię. Przypomnij sobie przykład projektanta dodającego nową zależność objętą licencją GPL, która trafiła do środowiska produkcyjnego. Wyobraź sobie teraz, że zanim projektant zaktualizował zależności, dodał do potoku (odpowiedzialnego za kompilację, weryfikację poprawności i wdrażanie) skaner zależności. Skaner wykryłby naruszenie licencji i spowodowałby niepowodzenie kompilacji. Zależność przechodnia nigdy nie trafiłaby do środowiska produkcyjnego. Ta świadoma i istotna decyzja projektanta sprawiła, że „przypadkowa" decyzja nie była znacząca. Z kolei istotna z punktu widzenia architektury decyzja o dodaniu skanera i skonfigurowaniu go w celu wykrywania zależności przechodnich z licencją GPL była znacząca.

Co jednak będzie, jeśli jeszcze niczego nie dostarczono do środowiska produkcyjnego? W takiej sytuacji powinno się zwrócić uwagę na decyzje, które aktywnie utrudniają wprowadzenie Twojego oprogramowania do tego środowiska. Te decyzje są znaczące w sensie negatywnym.

Piszę o tym, ponieważ dopóki Twoje decyzje architektoniczne nie znajdą odzwierciedlenia we wdrożonym oprogramowaniu, nie są to jeszcze prawdziwe decyzje, a jedynie *aspiracje, a kod to tylko zbiór schematów*. W takich okolicznościach poważnie zalecam jak najszybsze wprowadzenie swojej architektury do środowiska produkcyjnego. Zawsze istnieje sposób, by to zrobić, poza *wyjątkowo trudnymi okolicznościami*, a 99,99% z nas nie staje przed takimi wyzwaniami (zachęcam do ciągłego zadawania sobie następującego pytania: „Jak można to wdrożyć już teraz?" zamiast pytania: „Dlaczego nie mogę tego wdrożyć?"). Jeśli potrzebujesz nowych lub innych decyzji, by to osiągnąć, tak właśnie postąp. Na tym etapie możesz zdać sobie sprawę, że niektóre z Twoich kluczowych decyzji architektonicznych faktycznie stoją na przeszkodzie. Jeżeli nie spróbujesz uzyskać dostępu do środowiska produkcyjnego, nie będziesz miał żadnych możliwości poznania tego negatywnego aspektu.

Czy przypominasz sobie sytuację, o której wcześniej wspomniałem: zespół projektowy zastosował techniki projektowania opartego na domenie (DDD — *Domain-Driven Design*) do określenia struktury kodu w taki sposób, że większość można było testować lokalnie bez uruchamiania serwera J2EE? Korzyść ta objęła również potoki. Gdyby podjęto inną decyzję, a testowanie zarówno lokalnie, jak i w potoku byłoby utrudnione, uznano by to za znaczące (znacząco złe, ale wciąż znaczące). Późniejsza decyzja projektantów o wykorzystaniu architektury heksagonalnej umożliwiającej lokalne testowanie, które było również możliwe do przeprowadzenia w potokach, okazała się kluczowa dla wdrożenia kodu w środowisku produkcyjnym.

Przed umieszczeniem kodu w tym środowisku oczekiwane korzyści z obecności serwera J2EE w środowisku uruchomieniowym pozostawały jedynie w sferze nadziei i nie były jeszcze zrealizowane. Jednocześnie ich negatywny wpływ na etapie kompilacji był łagodzony.

Dlatego uważam, że kryteria technik budowania są równie ważne jak wszystkie pozostałe kryteria znaczących decyzji architektonicznych. Techniki te zapewniają, że temu ważnemu aspektowi poświęca się tyle samo uwagi, co pozostałym czterem kryteriom. Służy to zrównoważeniu, przynajmniej częściowo, tendencji innych kryteriów do pomijania środowiska wykonawczego.

Decyzje dotyczące produktów i przydziału zadań zespołom

Aby utrzymać koncentrację w tym rozdziale, celowo pominąłem następujące dwa źródła wpływu na decyzje architektoniczne:

- Zarządzanie produktem/działalność biznesowa.
- Określenie, kto decyduje o przydziale zespołów do zadań.

Nie ulega wątpliwości, że decyzje dotyczące produktu mogą bezpośrednio wpływać na architekturę oprogramowania. To samo można powiedzieć o decyzjach związanych z personelem lub organizacją pracy zespołów.

Aby uniknąć nieporozumień, traktuj te decyzje wpływające na architekturę oprogramowania *tak, jakby były decyzjami architektonicznymi*. Podchodź do nich w taki sam sposób jak do wszystkich innych decyzji architektonicznych. Metody ich ułatwienia, które przedstawię, są w pełni możliwe do zastosowania. Menedżer produktu może podejmować decyzje mające wpływ na architekturę, ale w idealnym świecie będzie świadomy tego faktu i zaangażuje odpowiednie osoby, gdy to nastąpi.

Przykłady istotnych decyzji architektonicznych

Istotne decyzje architektoniczne mają duży wpływ, ponieważ spełniają co najmniej jedno z pięciu rozszerzonych kryteriów: decydują o strukturze, cechach wielofunkcyjnych, zależnościach, interfejsach i/lub technikach budowania. Znajdują one też odzwierciedlenie w kodzie.

Na zakończenie przyjrzyjmy się kilku dodatkowym przykładom decyzji. Tym razem wszystkie dotyczą architektury, ale tylko te z pierwszej listy są istotne, natomiast decyzje z drugiej listy nie mają takiego znaczenia. Warto zastanowić się nad decyzjami, w których ostatnio brałeś udział. Zadaj sobie pytanie: czy były to decyzje architektoniczne, czy jedynie techniczne? A jeśli architektoniczne, to czy były znaczące?

Istotne decyzje architektoniczne

Struktura

Projektanci w zespole decydują się na refaktoryzację kodu w celu wyodrębnienia mikrofrontendu.

Cechy wielofunkcyjne

Projektant infrastruktury dodaje bramę API przed usługami publicznymi. Nie jest włączony żaden mechanizm uwierzytelniania, ograniczanie liczby żądań ani inne opcje interwencji. Na razie jest to proste przekazywanie danych.

Zależności i prawdopodobne cechy wielofunkcyjne

Komitet ds. architektury podejmuje decyzję o zmianie podstawowego typu wirtualnych zasobów obliczeniowych dostawcy usług chmurowych, z których firma będzie korzystać.

Interfejsy

Architekt aktualizuje interfejs API REST, aby usunąć opcjonalny parametr, który mógł być wykorzystywany przez niektórych użytkowników.

Techniki budowania

Osoba zapewniająca jakość oprogramowania decyduje, jakie będzie używane środowisko do testowania kontraktów.

Decyzje nieistotne z punktu widzenia architektury

Nieistotna struktura

Projektant dokonuje refaktoryzacji klasy, aby wyodrębnić metodę prywatną.

Nieistotne cechy wielofunkcyjne

Projektant przeprowadza refaktoryzację kodu bez zmiany jego struktury, co eliminuje wąskie gardło i pozwala na podwojenie przepustowości (zauważ, że gdyby podjęto odwrotną decyzję, która w konsekwencji spowodowałaby brak możliwości spełnienia wymagania CFR, sytuacja zostałaby uznana za istotną).

Nieistotne zależności

Projektant porządkuje skrypt kompilacji, usuwając szereg zależności bez zmiany funkcjonalności systemu.

Dokonujesz aktualizacji do nowej wersji dodatkowej używanego już środowiska (jeśli faktycznie jest to zmiana dotycząca tej wersji, powinna być ona niezauważalna dla całego kodu, który z niej korzysta).

Nieistotne interfejsy

Architekt aktualizuje stronę *wiki*, usuwając opcjonalny parametr z interfejsu API po uprzednim sprawdzeniu, że żaden z jego użytkowników z niego nie korzysta.

Nieistotne techniki budowania

Projektant zajmujący się infrastrukturą wdraża portal dla projektantów, który umożliwia wszystkim zespołom udostępnianie swoich usług i interfejsów API.

Nie wszystkie decyzje mające wpływ na architekturę są na początku decyzjami „architektonicznymi”

Zaznajamiając się z przykładami decyzji w tym rozdziale, być może nie zgodziłeś się z moim sposobem ich kategoryzacji. Warto wiedzieć, że (1) w pełni uznaję wpływ decyzji nietechnicznych na architekturę oraz (2) zdecydowanie zgadzam się z tym, że niektóre decyzje techniczne, choć niekoniecznie o znaczeniu architektonicznym, powinny przejść przez pewnego rodzaju formalny (lub przynajmniej świadomy) proces decyzyjny.

Podsumowanie

Dlaczego poświęciłem cały rozdział na podkreślenie kwestii znaczenia architektury? Po pierwsze, skuteczne praktykowanie architektury w świecie po rewolucjach oznacza, że podejmuje się znacznie więcej decyzji. W obliczu tego wszystkiego ważne jest rozpoznanie, które decyzje są architektoniczne, a także które z nich są znaczące. Jest to absolutnie niezbędne do skoncentrowania się na kwestii czasu, nakładu pracy i zasobów.

Po drugie, ważne jest, aby docenić wpływ projektantów na architekturę działającego systemu, niezależnie czy są tego świadomi, czy nie. Wbrew wszelkim kryteriom istotności projektanci mogą (i faktycznie to robią) podejmować decyzje istotne z punktu widzenia architektury, co należy postrzegać w pozytywny sposób.

Jeśli nasze architektury składają się wyłącznie z decyzji, trzeba wiedzieć, na które z nich zwrócić szczególną uwagę. Podkreślałem również, że choć obecność decyzji we wdrożonym kodzie nie stanowi ścisłego kryterium znaczenia architektonicznego, decyzje utrudniające wczesne i częste wdrażanie są istotne *w negatywnym sensie*. Do momentu zainicjowania danej decyzji przypuszczamy jedynie, czy jest ona przydatna. Przestań zgadywać, zacznij wdrażać i wyciągaj wnioski z informacji zwrotnych.

Zanim przejdziesz dalej, przypomnij sobie ćwiczenie wizualizacyjne, które wykonałeś na początku rozdziału. Gdy zamknąłeś oczy i wyobraziłeś sobie, kto podejmuje decyzje architektoniczne (istotne lub nie) w Twojej firmie, czy ujrzałeś jedną osobę, czyli architekta? A może wyobraziłeś sobie grupę ludzi?

W rozdziale 3. zostanie rozwinięte to podejście i udzielona odpowiedź na następujące dwa pytania: „Jakie są standardowe metody podejmowania decyzji na dużą skalę?” oraz „Jakie są cechy »idealnego« zdecentralizowanego procesu decyzyjnego opartego na informacji zwrotnej?”.

Oto książka pełna praktycznej mądrości, trafiająca w sedno tego, czym jest architektura.

Grady Booch, IBM Fellow

Rola architekta oprogramowania się zmienia. W miarę jak systemy stają się coraz bardziej złożone, tradycyjny model działania architekta przestaje wystarczać. Zadań jest zbyt wiele, a ich zaniedbanie w końcu doprowadzi do punktu krytycznego. Kluczowe staje się współdziałanie architekta z zespołem projektowym — tylko wtedy możliwe jest tworzenie trwałej, elastycznej i efektywnej architektury.

Tę książkę doceni każdy, kto praktykuje architekturę w ramach i na rzecz zespołu. Dzięki niej zrozumiesz wady scentralizowanych praktyk architektury w rozproszonym świecie. Poznasz też kluczowe aspekty w zdecentralizowanego podejścia do architektury opartego na informacji zwrotnej i zasady jego wdrażania takiego podejścia. Znajdziesz tu omówienie czynników wpływających na efektywność decyzji architektonicznych, a także ich społeczny wymiar — dobra architetura bowiem to nie tylko struktura kodu, ale również jakość współpracy. Dzięki opisanym metodom rozwiniesz sposób myślenia, który pozwala każdemu w zespole praktykować architekturę i budować lepsze systemy.

Najważniejsze zagadnienia:

- jak się zmienia nowoczesny proces dostarczania oprogramowania
- metodologia łączenia architektury oprogramowania z jego rozwojem
- współzależność decyzji, architektury i informacji zwrotnej z działających systemów
- wprowadzanie praktyk maksymalizacji korzyści i minimalizacji ryzyka
- dostrajanie podejścia do architektury, umiejętności osób w zespole i kultury firmy

Najlepsza architektura oprogramowania ewoluuje dzięki zaangażowaniu wszystkich.

Martin Fowler, główny badacz w Thoughtworks

Andrew Harmel-Law jest głównym specjalistą technicznym w firmie Thoughtworks. Zajmuje się projektowaniem opartym na domenach, modernizacją na dużą skalę i transformacjami organizacyjnymi. Pasjonuje się oprogramowaniem *open source*, dzieli się swoją wiedzą w ramach szkoleń online, artykułów, publicznych prelekcji i mentoringu.

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 978-83-289-2750-6

9 788328 927506

Cena: 119,00 zł