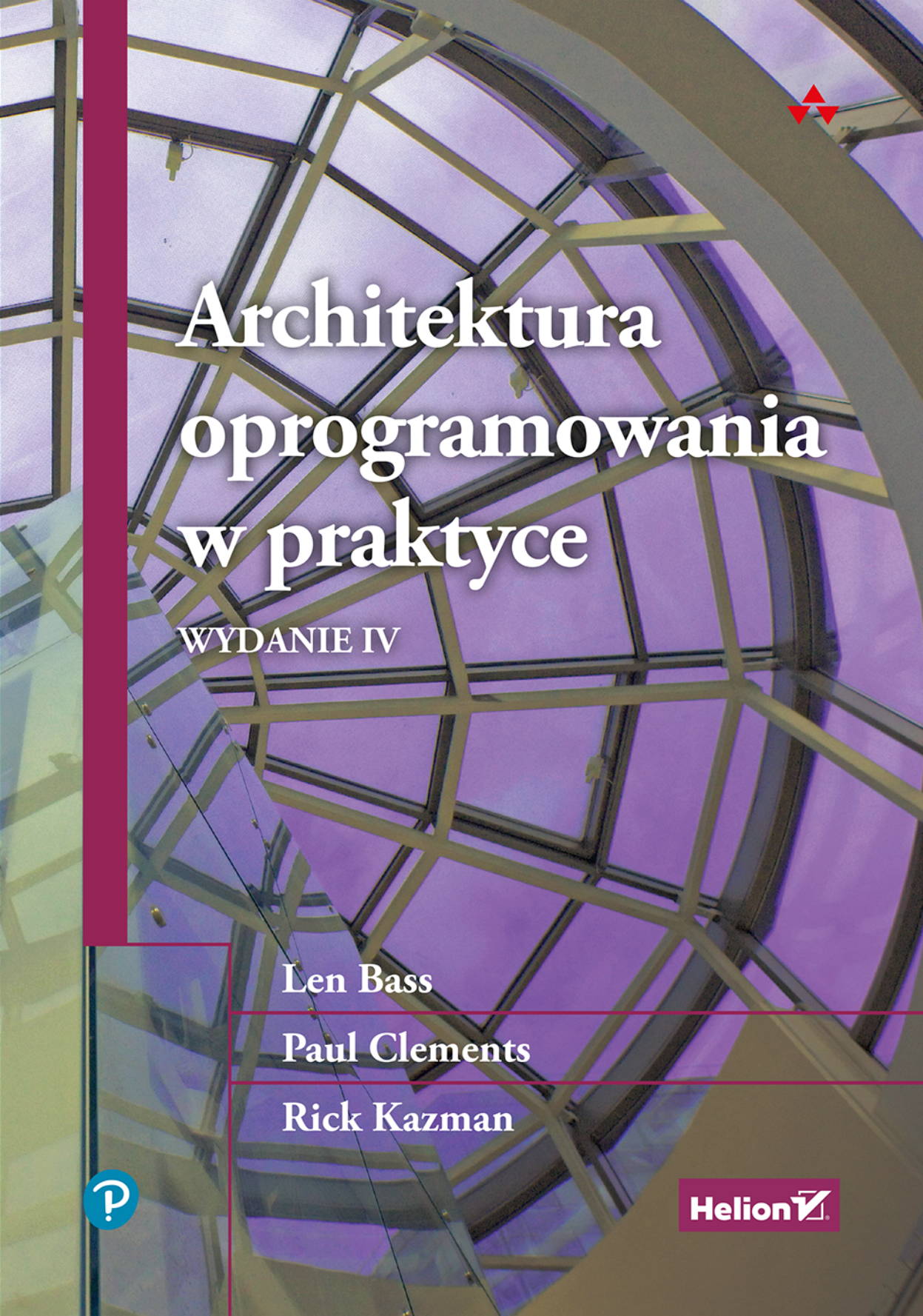

Architektura oprogramowania w praktyce

WYDANIE IV

Len Bass

Paul Clements

Rick Kazman

Helion

Tytuł oryginału: Software Architecture in Practice, 4th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-9052-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/aropw4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

Printed in Poland.

Spis treści

2

Dlaczego architektura oprogramowania jest istotna?

Ach, budować, budować!
To najszlachetniejsza ze wszystkich sztuk.
— Henry Wadsworth Longfellow

Skoro architektura jest odpowiedzią, jak brzmi pytanie?

W tym rozdziale koncentrujemy się na tym, dlaczego architektura jest istotna z technicznego punktu widzenia. Przyjrzymy się 13 najważniejszym powodom. Możesz je wykorzystać do uzasadnienia prac nad nową architekturą lub do analizy i modyfikowania istniejącej architektury systemu.

1. Architektura może albo utrudniać, albo ułatwiać uzyskanie głównych atrybutów jakościowych systemu.
2. Decyzje podjęte w trakcie tworzenia architektury umożliwiają wnioskowanie na temat zmian i zarządzanie nimi w trakcie ewoluowania systemu.
3. Analiza architektury umożliwia wczesne oszacowanie atrybutów jakościowych systemu.
4. Udokumentowana architektura ułatwia komunikację między interesariuszami.
5. Architektura jest nośnikiem najwcześniejszych, a tym samym najbardziej fundamentalnych i najtrudniejszych do zmiany decyzji projektowych.
6. Architektura określa zestaw ograniczeń dotyczących późniejszej implementacji.
7. Architektura wyznacza strukturę organizacji lub na odwrót.
8. Architektura może być podstawą do stopniowego rozwijania oprogramowania.
9. Architektura jest najważniejszym artefaktem, który umożliwia architektowi i menedżerowi projektu wnioskowanie na temat kosztów i harmonogramu prac.
10. Architekturę można stworzyć w formie uniwersalnego modelu wielokrotnego użytku, który stanowi istotę linii produktowej.

11. Programowanie oparte na architekturze wymaga skupienia się na łączeniu komponentów, a nie tylko na ich tworzeniu.
12. Przez ograniczenie alternatywnych projektów architektura kierunkuje kreatywność programistów, ponieważ zmniejsza złożoność projektu i systemu.
13. Architektura może być podstawą do szkolenia nowych członków zespołu.

Nawet jeśli już nam wierzysz, że architektura jest ważna, i nie musisz 13 razy sobie tego powtarzać, potraktuj tych 13 punktów (wyznaczających strukturę tego rozdziału) jako 13 przydatnych sposobów na wykorzystanie architektury w projekcie lub na uzasadnienie nakładów na opracowanie architektury.

2.1. Utrudnianie albo ułatwianie uzyskania głównych atrybutów jakościowych systemu

Możliwość osiągnięcia pożądanych (lub wymaganych) atrybutów jakościowych w systemie w dużym stopniu zależy od jego architektury. Jeśli masz zapamiętać z tej książki jedną rzecz, niech będzie to właśnie to.

Ta zależność jest tak ważna, że poświęciliśmy całą część II tej książki na to, aby szczegółowo ją wyjaśnić. Na razie jako punkt wyjścia zapamiętaj następujące przykłady:

- Jeśli system musi mieć wysoką wydajność, trzeba zwrócić uwagę na zarządzanie związanym z czasem zachowaniem elementów, korzystaniem ze współużytkowanych zasobów oraz częstością i ilością komunikacji między elementami.
- Jeżeli istotna jest modyfikowalność, należy zwrócić uwagę na przydział zadań do elementów i ograniczenie interakcji (powiązań) między elementami, aby większość zmian w systemie wpływała tylko na niewielką liczbę elementów. W idealnym scenariuszu każda zmiana powinna dotyczyć tylko jednego elementu.
- Jeśli system musi być dobrze zabezpieczony, trzeba zarządzać komunikacją między elementami i chronić ją. Należy też kontrolować, które elementy mają dostęp do poszczególnych informacji. Możliwe, że trzeba będzie wprowadzić do architektury wyspecjalizowane elementy (na przykład mechanizmy autoryzacji), aby zbudować solidną granicę chroniącą przed atakami.
- Jeżeli system ma być zabezpieczony i bezpieczny, trzeba zaprojektować zabezpieczenia i mechanizmy odzyskiwania stanu.
- Jeśli uważasz, że powodzenie systemu zależy od skalowalności wydajności, musisz przeanalizować wykorzystanie zasobów i umożliwić wprowadzanie

zastępników o wyższych możliwościach. Należy też unikać zapisywania nieprzekraczalnych założeń lub limitów dotyczących zasobów.
- Jeżeli projekty wymagają stopniowego dodawania podzbiorów systemu, musisz zarządzać tym, jak komponenty używają siebie nawzajem.
- Jeśli chcesz, aby elementy systemu można było zastosować także w innych systemach, musisz ograniczyć powiązania między elementami, by po wyodrębnieniu jakiegoś elementu nie miał on zbyt wielu zależności od obecnego środowiska i pozostał użyteczny.

Strategie dotyczące tych i innych atrybutów jakościowych w dużej mierze są związane z architekturą. Jednak sama architektura nie gwarantuje funkcji ani atrybutów jakościowych wymaganych od systemu. Niskiej jakości projekt z dalszych etapów prac lub złe decyzje projektowe zawsze mogą zniweczyć poprawny projekt architektoniczny. Czasem mówimy (*najczęściej* dla żartu), że „co architektura dała, implementacja może odebrać". Jakość systemu zależy od decyzji podejmowanych na wszystkich etapach cyklu życia oprogramowania: od projektowania architektury po pisanie kodu i testy. Dlatego jakość nie wynika wyłącznie z projektu architektury, ale dbałość o nią zaczyna się właśnie od tego punktu.

2.2. Wnioskowanie na temat zmian i zarządzanie nimi

Jest to następstwo poprzedniego punktu.

Modyfikowalność, czyli łatwość wprowadzania zmian w systemie, jest atrybutem jakościowym, a tym samym dotyczą go uwagi z poprzedniego podrozdziału. Ten atrybut jest jednak tak ważny, że zasłużył na własne miejsce na „liście trzynastu". Ludzie zajmujący się rozwojem oprogramowania zaczynają rozumieć, że mniej więcej 80% łącznych kosztów związanych z typowym systemem oprogramowania jest generowanych *po* jego wdrożeniu. Większość systemów, nad jakimi pracujemy, znajduje się właśnie na tym etapie. Wielu programistów i projektantów oprogramowania *nigdy* nie zajmuje się tworzeniem nowych systemów. Te osoby muszą uwzględniać ograniczenia związane z istniejącą architekturą i obecnym kodem. Prawie wszystkie systemy oprogramowania zmieniają się w ciągu życia. Modyfikacje są konieczne, aby uwzględnić nowe funkcje, dostosować system do nowych środowisk, poprawić błędy itd. Jednak wprowadzanie zmian często jest trudne.

W każdej architekturze niezależnie od jej postaci można podzielić zmiany na trzy kategorie: lokalne, nielokalne i architektoniczne.

- Zmiany lokalne można wprowadzić przez zmodyfikowanie jednego elementu, na przykład dodanie reguły biznesowej do modułu wyznaczania cen.

- Zmiana nielokalna wymaga modyfikacji kilku elementów, ale z zachowaniem pierwotnego podejścia architektonicznego. Dzieje się tak na przykład, gdy trzeba utworzyć nową regułę biznesową w module wyznaczania cen, a następnie dodać do bazy danych nowe pola niezbędne dla tej reguły i wyświetlić wyniki zastosowania tej reguły w interfejsie użytkownika.
- Zmiana architektoniczna wpływa na sposób interakcji między elementami i często wymaga modyfikacji w wielu miejscach systemu, na przykład przejścia z modelu jednowątkowego na wielowątkowy.

Oczywiście najbardziej pożądane są zmiany lokalne, dlatego *skuteczna* architektura powinna sprawiać, że większość zmian jest lokalna, a tym samym łatwa do wprowadzenia. Zmiany nielokalne są niepożądane, ale mają tę zaletę, że przeważnie można je wprowadzać stopniowo w uporządkowany sposób. Można na przykład najpierw utworzyć nową regułę wyznaczania cen, a następnie wprowadzić modyfikacje potrzebne do zastosowania tej reguły.

Decydowanie o tym, kiedy zmiany są niezbędne, ustalanie, jakie ścieżki ich wprowadzania są narażone na najmniejsze ryzyko, ocena skutków proponowanych modyfikacji i wyznaczanie sekwencji oraz priorytetów zmian wymaga dobrego zrozumienia relacji, wydajności i działania elementów systemu oprogramowania. Wszystkie te zadania pojawiają się w opisie stanowiska architekta. Wnioskowanie na temat architektury i jej analizowanie może zapewnić wiedzę niezbędną do podejmowania decyzji o przewidywanych zmianach. Jeśli pominiesz ten etap i nie zadbasz o utrzymanie koncepcyjnej integralności architektury, prawie na pewno doprowadzisz do narastania *długu architektonicznego*. Ten temat omawiamy w rozdziale 23.

2.3. Prognozowanie atrybutów jakościowych systemu

Ten punkt wynika z dwóch wcześniejszych: architektura nie tylko wpływa na atrybuty jakościowe systemu, ale też robi to w przewidywalny sposób.

Może się to wydawać oczywiste, ale nie dla każdego takie jest. Wtedy projektowanie architektury polega na podejmowaniu serii w dużym stopniu losowych decyzji projektowych, po czym system jest budowany i następuje sprawdzenie atrybutów jakościowych w nadziei na powodzenie. Ups, system nie jest wystarczająco szybki lub okazał się zatrważająco podatny na ataki? Zabieramy się do pracy.

Na szczęście *możliwe* jest prognozowanie atrybutów jakościowych systemu wyłącznie na podstawie oceny jego architektury. Jeśli wiesz, że określone decyzje architektoniczne prowadzą do uzyskania danych atrybutów jakościowych, możesz podjąć te decyzje i słusznie oczekiwać nagrody w postaci tych atrybutów. Po zakończeniu prac,

w trakcie oceny architektury, możesz stwierdzić, czy podjęte zostały odpowiednie decyzje, i z wysokim prawdopodobieństwem przewidzieć, czy architektura będzie miała określone cechy.

Ten i poprzedni punkt razem oznaczają, że architektura w dużym stopniu determinuje atrybuty jakościowe systemu i, co jeszcze lepsze, wiadomo, w jaki sposób to robi, dzięki czemu wiadomo też, jak z jej użyciem osiągnąć pożądany efekt.

Nawet jeśli nie stosujesz analitycznego modelowania atrybutów jakościowych (czasem takie modelowanie jest konieczne, aby zagwarantować, że architektura zapewni określone korzyści), zasada oceny decyzji na podstawie ich wpływu na atrybuty jakościowe jest nieoceniona choćby dlatego, że pozwala wcześnie wykryć potencjalne problemy.

2.4. Komunikacja między interesariuszami

W rozdziale 1. napisaliśmy, że architektura jest abstrakcją. Wspomnieliśmy też, że jest przydatna, ponieważ reprezentuje uproszczony model całego systemu, który — w odróżnieniu od nieskończonej listy szczegółów dotyczących systemu — możesz utrzymywać w umyśle. To samo dotyczy innych członków zespołu. Architektura reprezentuje wspólną abstrakcję systemu, którą większość interesariuszy (a nawet każdy z nich) może posługiwać się jako podstawą do wzajemnego zrozumienia, w negocjacjach, do znajdowania konsensusu i do komunikacji. Architektura, a przynajmniej jej części, jest wystarczająco abstrakcyjna, aby większość osób bez wiedzy technicznej potrafiła zrozumieć ją na potrzebnym im poziomie, zwłaszcza z pewną pomocą ze strony architekta. Jednocześnie tę abstrakcję można rozbudować do postaci odpowiednio szczegółowych specyfikacji technicznych pomocnych w trakcie implementowania, integrowania, testowania i wdrażania systemu.

Każdy interesariusz związany z systemem oprogramowania (klient, użytkownik, menedżer projektu, koder, tester itd.) jest zainteresowany innymi cechami systemu wynikającymi z architektury. Oto przykład:

- dla użytkownika ważne jest, aby system był szybki, niezawodny i dostępny wtedy, gdy jest potrzebny;
- dla klienta (który płaci za system) liczy się, aby architekturę można było zaimplementować zgodnie z harmonogramem i budżetem;
- dla menedżera istotne jest (obok kosztów i harmonogramu), aby architektura umożliwiała zespołom w dużym stopniu niezależną pracę oraz interakcje w zdyscyplinowany i kontrolowany sposób;
- architekt zajmuje się strategiami pozwalającymi osiągnąć wszystkie wymienione cele.

Architektura zapewnia wspólny język, w którym różne kwestie można wyrazić, negocjować i rozwiązywać na poziomie dostępnym intelektualnie nawet w przypadku dużych, złożonych systemów. Bez takiego języka trudno jest wystarczająco dobrze zrozumieć duże systemy, aby móc na wczesnych etapach prac podejmować decyzje wpływające zarówno na jakość, jak i przydatność rozwiązania. Analiza architektoniczna, jak pokazujemy w rozdziale 21., zarówno zależy od tego poziomu komunikacji, jak i ją ułatwia.

W rozdziale 22., poświęconym dokumentacji architektury, szczegółowo omawiamy interesariuszy i ich potrzeby.

„Co się stanie, jeśli wcisnę ten przycisk?" — architektura jako narzędzie do komunikacji między interesariuszami

Przegląd projektu ciągnął się w nieskończoność. Finansowane przez rząd prace były opóźnione, a koszty przekroczyły planowany budżet. Projekt był na tyle duży, że te zaniedbania przyciągnęły uwagę amerykańskiego Kongresu. Obecnie rząd nadrabiał wcześniejsze zaległości, planując niekończącą się sesję przeglądową z udziałem wszystkich stron. Kontraktor został niedawno wykupiony, co też nie pomagało w pracach. Było popołudnie drugiego dnia przeglądu i zgodnie z planem miała zostać zaprezentowana architektura oprogramowania. Młody architekt, uczeń głównego architekta systemu, odważnie wyjaśniał, że architektura oprogramowania tego rozbudowanego systemu pozwoli spełnić bardzo ambitne wymogi dotyczące pracy w czasie rzeczywistym w środowisku rozproszonym przy wysokim stopniu niezawodności. Architekt miał dobrą prezentację i solidną architekturę, poprawną i sensowną. Jednak słuchacze, około 30 przedstawicieli rządu pełniących różne funkcje w zarządzie i przy nadzorze tego problematycznego projektu, byli zmęczeni. Niektórzy z nich zastanawiali się nawet nad zajęciem się handlem nieruchomościami, aby tylko uniknąć następnego z długiej serii przeglądów „zróbmy to wreszcie tak, jak należy".

Pojawił się slajd, na którym za pomocą półformalnej notacji z polami i liniami pokazane były główne elementy oprogramowania w perspektywie czasu wykonania. Wszystkie nazwy były akronimami, niezrozumiałymi bez wyjaśnień, które młody architekt zaprezentował. Linie przedstawiały przepływ danych, przepływ komunikatów i synchronizację procesów. Architekt wyjaśnił, że dla elementów zastosowano nadmiarowość. „W przypadku awarii" — zaczął, pokazując jedną z linii wskaźnikiem laserowym — „uruchamiany jest mechanizm wznawiania pracy zgodnie z tą ścieżką...".

„A co się stanie, jeśli wciśnięty zostanie przycisk wyboru trybu?" — przerwał jeden ze słuchaczy. Był to przedstawiciel rządu, reprezentujący użytkowników systemu.

„Przepraszam bardzo, nie rozumiem pytania" — odpowiedział architekt.

„Przycisk wyboru trybu" — powtórzył słuchacz. „Co się stanie, jeśli go wciśniemy?".

„Hmm, to uruchomi zdarzenie w sterowniku urządzenia, w tym miejscu" — rozpoczął architekt, pokazując element wskaźnikiem. „System wczyta zawartość rejestru i zinterpretuje kod zdarzenia. Jeśli dotyczy ono wyboru trybu, to, no cóż, przesłany zostanie sygnał do tablicy, która z kolei przekaże go obiektom subskrybującym to zdarzenie...".

„Nie, nie. Chodzi mi o to, co system *zrobi*" — przerwał autor pytania. „Czy zresetuje ekrany? I co się stanie, jeśli stanie się to w trakcie rekonfigurowania systemu?".

Architekt wyglądał na nieco zaskoczonego i wyłączył wskaźnik. Nie było to pytanie z zakresu architektury, ale ponieważ prezentujący był architektem i dlatego dobrze znał wymogi dotyczące systemu, znał odpowiedź. „Jeśli wiersz poleceń znajduje się w trybie konfiguracji, ekrany zostaną zresetowane" — powiedział. „W przeciwnym razie w konsoli sterowania pojawi się komunikat o błędzie, a sygnał zostanie zignorowany". Architekt ponownie włączył wskaźnik. „Wróćmy teraz do mechanizmu wznawiania pracy, o którym mówiłem...".

„Hmm, tak się zastanawiam" — powiedział przedstawiciel użytkowników. „Ponieważ z wykresu wynika, że konsola przesyła sygnały do modułu lokalizowania celu".

„A co *powinno* się dziać?" — spytał inny słuchacz autora pierwotnego pytania. „Czy naprawdę chciałbyś, żeby użytkownik otrzymywał dane o trybie w trakcie rekonfiguracji?". Przez następnych 45 minut architekt obserwował, jak słuchacze zabierali jego czas, dyskutując na temat właściwego działania systemu w różnych rzadko występujących stanach. Były to bardzo ważne rozmowy, które jednak należało przeprowadzić na etapie formułowania wymogów, co z niewiadomych powodów nie nastąpiło.

Ta dyskusja nie dotyczyła architektury, jednak została wywołana przez architekturę i jej graficzną reprezentację. Naturalne jest myślenie o architekturze jak o podstawie do komunikacji między niektórymi interesariuszami, nie tylko architektami i programistami. Na przykład menedżerowie mogą na podstawie architektury tworzyć zespoły i przydzielać im zasoby. Ale użytkownicy? Przecież architektura jest dla nich niewidoczna. Dlaczego mieliby traktować ją jak narzędzie do zrozumienia systemu?

Tak się jednak dzieje. W omawianym przykładzie autor pytania przez dwa dni oglądał perspektywy ilustrujące funkcje, operacje, interfejsy użytkownika i testy. Jednak dopiero pierwszy slajd przedstawiający architekturę sprawił, że słuchacz, choć zmęczony i czekający tylko na powrót do domu, zdał sobie sprawę,

że czegoś nie zrozumiał. Uczestnictwo w wielu przeglądach architektury przekonało mnie, że spojrzenie na system z nowego punktu widzenia pobudza umysł i rodzi nowe pytania. Dla użytkowników to architektura jest często tym nowym sposobem, a zadawane pytania dotyczą działania systemu. W trakcie pamiętnego dla mnie ćwiczenia z oceny architektury, które miało miejsce kilka lat temu, przedstawiciele użytkowników byli znacznie bardziej zainteresowani tym, co system będzie robić, niż tym, jak to zrobi. To zupełnie naturalne. Wcześniej kontaktowali się z producentem tylko za pośrednictwem marketingowców. Architekt był pierwszym prawdziwym ekspertem od systemu, z którym mogli porozmawiać, i nie wahali się wykorzystać tej okazji.

Naturalnie staranna i dokładna specyfikacja wymogów ułatwiłaby pracę, ale z różnych powodów nie zawsze jest tworzona lub dostępna. Gdy jej brak, specyfikacja architektury często zachęca do zadawania pytań i zwiększa jasność sytuacji. Prawdopodobnie rozsądniejsze jest zaakceptowanie tego faktu niż przeciwdziałanie mu.

Czasem przegląd architektury ujawnia też nieuzasadnione wymogi, których przydatność można ponownie ocenić. Przegląd tego rodzaju, w trakcie którego podkreślana jest synergia między wymogami a architekturą, umożliwiłby młodemu architektowi z naszej historii wyjście z impasu, ponieważ możliwe byłoby poświęcenie całej sesji na omówienie informacji tego typu. Ponadto przedstawiciel użytkowników uniknąłby skrępowania wynikającego z zadania pytania w najwyraźniej niewłaściwym momencie.

— PCC

2.5. Wczesne decyzje projektowe

Architektura oprogramowania jest odzwierciedleniem najwcześniejszych decyzji projektowych dotyczących systemu. Te wczesne wytyczne mają olbrzymi wpływ na późniejszy rozwój systemu, jego wdrażanie i konserwację. Jest to także najwcześniejszy moment, w którym można przeanalizować istotne decyzje projektowe dotyczące systemu.

Każdy projekt w każdej dziedzinie można postrzegać jak sekwencję decyzji. W procesie malowania obrazu artysta jeszcze przed rozpoczęciem prac decyduje się na materiał płótna i narzędzia do malowania: farbę olejną, akwarele, kredki. W momencie rozpoczynania malowania podejmuje inne decyzje: gdzie narysować pierwszą linię, jakiej grubości, o jakim kształcie? Wszystkie te wczesne decyzje projektowe mają istotny wpływ na ostateczny wygląd obrazu. Każda z tych decyzji ogranicza też listę późniejszych wyborów. Każda pojedyncza decyzja może się wydawać mało istotna,

jednak zwłaszcza wczesne wybory mają nieproporcjonalnie duże znaczenie, ponieważ wpływają na tak wiele dalszych kroków i ograniczają ich zakres.

Tak samo jest z projektowaniem architektury. Ten proces też można traktować jako zestaw decyzji. Zmiana wczesnych decyzji wywołuje efekt domina, ponieważ wymaga modyfikacji także innych wyborów. To prawda, czasem konieczna jest refaktoryzacja lub zmiana projektu architektury, jednak nie należy pochopnie podejmować takich decyzji, gdyż efekt domina może łatwo przekształcić się w efekt lawiny.

Czego dotyczą wczesne decyzje projektowe ujęte w architekturze oprogramowania? Rozważ następujące pytania:

- Czy system będzie działać w jednym procesorze, czy będzie rozproszony między wiele procesorów?
- Czy oprogramowanie będzie warstwowe? Jeśli tak, to ile będzie warstw? Co będzie robić każda z nich?
- Czy komponenty będą komunikować się synchronicznie, czy asynchronicznie? Będą komunikować się ze sobą przez przekazywanie sterowania, przekazywanie danych czy z użyciem obu tych technik?
- Czy informacje przepływające przez system będą szyfrowane?
- Jaki system operacyjny będzie używany?
- Jaki protokół komunikacji wybierzesz?

Wyobraź sobie, jakim koszmarem byłaby zmiana dowolnej z tych lub wielu innych podobnych decyzji. Tego rodzaju wybory wpływają na kształt struktur architektury i interakcji między nimi.

2.6. Ograniczenia dotyczące implementacji

Jeśli chcesz, aby implementacja była zgodna z architekturą, trzeba dostosować ją do decyzji projektowych reprezentowanych w tej architekturze. Konieczne jest utworzenie zestawu elementów przedstawionych w architekturze, elementy muszą komunikować się między sobą w sposób opisany w architekturze, a każdy element musi wykonywać swoje obowiązki zgodnie z architekturą. Każdy opis z architektury jest ograniczeniem dla autora implementacji.

Autorzy elementów muszą najpierw dobrze poznać specyfikacje poszczególnych elementów, jednak czasem nie znają kompromisów narzucanych przez architekturę. Architektura (lub architekt) ograniczają autorów implementacji w taki sposób, aby przestrzegać tych kompromisów. Oto klasyczny przykład: architekt przydziela budżet wydajności do fragmentów oprogramowania związanych z większą funkcją. Jeśli każda

jednostka oprogramowania zmieści się w budżecie, cała transakcja będzie spełniać wymogi dotyczące wydajności. Autorzy implementacji poszczególnych fragmentów mogą nie znać całego budżetu; wiedzą tylko, ile wynosi ich budżet.

Z kolei architekci nie muszą być ekspertami od wszystkich aspektów projektowania algorytmów lub szczegółów języka programowania, choć oczywiście powinni mieć wystarczającą wiedzę, aby nie zaprojektować czegoś, co trudno będzie zbudować. Architekci odpowiadają za podejmowanie, analizowanie i zapewnianie przestrzegania decyzji oraz kompromisów architektonicznych.

2.7. Wpływ na strukturę organizacji

Architektura opisuje strukturę rozwijanego systemu, a ponadto ta struktura ma odzwierciedlenie w strukturze prac (a czasem nawet w strukturze całej organizacji). Standardowa metoda podziału prac w dużym projekcie polega na przydzielaniu różnym grupom porcji systemu do zbudowania. W architekturze ta struktura podziału prac ma postać struktury przydziału prac opisanej w rozdziale 1. Ponieważ architektura reprezentuje najbardziej ogólny podział systemu, zwykle jest używana jako punkt wyjścia do tworzenia struktury podziału prac. Z kolei struktura podziału prac wpływa na jednostki planowania, harmonogram i budżet, kanały komunikacji między zespołami, zarządzanie konfiguracją i układ systemu plików, plany i procedury integracji oraz testów, a nawet szczegóły takie jak struktura intranetu używanego w projekcie lub usadowienie osób na firmowym pikniku. Zespoły komunikują się między sobą na podstawie specyfikacji interfejsów rozwijanych elementów. Także konserwacja uruchomionego systemu odzwierciedla strukturę oprogramowania. Zespoły są tworzone w taki sposób, aby zajmować się konserwacją określonych elementów architektury: bazy danych, reguł biznesowych, interfejsu użytkownika, sterowników urządzeń itd.

Efektem ubocznym utworzenia struktury podziału prac jest „zamrożenie" niektórych aspektów architektury oprogramowania. Grupa odpowiedzialna za jeden z podsystemów może być przeciwna rozdzieleniu jej zadań między inne grupy. Jeśli te zadania zostały formalnie opisane w kontrakcie, zmiana obowiązków może być kosztowna lub nawet prowadzić do sporów prawnych.

Dlatego po uzgodnieniu architektury wprowadzanie w niej istotnych zmian staje się bardzo kosztowne (z przyczyn biznesowych i związanych z zarządzaniem). Jest to jeden z wielu argumentów za tym, aby architekturę dużego systemu przeanalizować przed podjęciem ostatecznych decyzji.

2.8. Umożliwianie stopniowego rozwijania systemu

Po zdefiniowaniu architektura może być podstawą do stopniowego rozwijania systemu. Pierwszym etapem może być szkieletowy system, w którym dostępna jest przynajmniej część infrastruktury (odpowiedzialna za to, jak elementy są inicjowane, jak się komunikują, współużytkują dane, uzyskują dostęp do zasobów, informują o błędach, rejestrują operacje itd.), ale wiele funkcji aplikacji z systemu jeszcze nie działa.

Tworzenie infrastruktury i rozwijanie funkcji aplikacji mogą się odbywać jednocześnie. Zaprojektuj i zbuduj niewielką część infrastruktury potrzebną do obsługi niewielkiej porcji funkcji, a następnie powtarzaj te kroki do momentu ukończenia systemu.

Wiele systemów jest budowanych w formie szkieletowej, którą można rozszerzać za pomocą wtyczek, pakietów lub rozszerzeń. Przykładami są: język R, edytor Visual Studio Code i większość przeglądarek internetowych. Rozszerzenia zapewniają dodatkowe funkcje obok tych dostępnych w wersji szkieletowej. To podejście ułatwia proces rozwoju, ponieważ gwarantuje, że system można uruchomić już na wczesnych etapach cyklu życia produktu. Wartość systemu rośnie wraz z dodawaniem rozszerzeń lub zastępowaniem wczesnych wersji określonych części oprogramowania ich bardziej kompletnymi odpowiednikami. Zdarza się, że części są wersjami o niskiej wartości lub prototypami ostatecznych funkcji. W innych sytuacjach tworzone są *zastępniki*, które konsumują i produkują dane z określoną szybkością, wykonując niewiele innych operacji. Pozwala to między innymi wykryć problemy z wydajnością (i inne) na wczesnych etapach cyklu życia produktu.

To podejście stało się znane na początku XXI wieku dzięki pomysłom Alistaira Cockburna i jego „chodzącego szkieletu". Ostatnio zaczęły je stosować osoby posługujące się strategią MVP (ang. *minimum viable product*; jest to podstawowa wersja produktu o minimalnej funkcjonalności) do ograniczania ryzyka.

Jedną z korzyści płynących ze stopniowego rozwoju jest ograniczenie ryzyka w projekcie. Jeśli architektura dotyczy rodziny powiązanych systemów, infrastrukturę można wykorzystać w całej tej rodzinie, co zmniejsza koszty prac nad poszczególnymi systemami.

2.9. Szacowanie kosztów i harmonogramu prac

Szacunki kosztów i harmonogramu prac są ważnym narzędziem dla menedżera projektu. Pomagają pozyskać niezbędne zasoby, a także monitorować postęp prac nad projektem. Jednym z zadań architekta jest pomoc menedżerowi projektu w oszacowaniu

kosztów i harmonogramu na wczesnych etapach cyklu życia projektu. Szacunki generowane metodą „od ogółu do szczegółu" są przydatne do wyznaczania celów i planowania budżetu, jednak szacunki kosztów otrzymane metodą „od szczegółu do ogółu" na podstawie zrozumienia fragmentów systemu są zwykle bardziej precyzyjne niż te oparte na ogólnej wiedzy o systemie.

Wcześniej wspomnieliśmy, że struktura organizacyjna i struktura podziału prac w projekcie są prawie zawsze oparte na architekturze. Każdy zespół i każda osoba odpowiedzialna za jednostkę pracy potrafi dokładniej niż menedżer projektu przedstawić szacunki dotyczące swojego fragmentu i czuje większą odpowiedzialność za to, aby rzeczywiste wyniki się z nimi pokrywały. Jednak najlepsze szacunki kosztów i harmonogramu prac są wynikiem połączenia podejścia top-down (stosowanego przez architekta i menedżera projektu) z podejściem bottom-up (stosowanym przez programistów). Dyskusje i negocjacje prowadzone w takim procesie skutkują znacznie precyzyjniejszymi szacunkami niż stosowanie któregokolwiek z obu podejść z osobna.

Korzystne jest, jeśli wymogi dotyczące systemu zostały już sprawdzone i zatwierdzone. Im większa jest wiedza na temat zakresu prac, tym dokładniejsze będą szacunki kosztów i harmonogramu.

W rozdziale 24. omawiamy wykorzystanie architektury do zarządzania projektem.

2.10. Uniwersalny model wielokrotnego użytku

Im wcześniej w cyklu życia fragmenty oprogramowania są ponownie używane, tym większe może to przynieść korzyści. Choć ponowne użycie kodu jest korzystne, ponowne użycie architektury pozwala znacznie ułatwić rozwój systemów, którym stawiane są podobne wymagania. Gdy decyzje architektoniczne można ponownie zastosować w wielu systemach, wszystkie wcześniej opisane konsekwencje wczesnego dokonywania wyborów będą dotyczyć także tych systemów.

Linia lub rodzina produktów to zestaw systemów budowanych za pomocą tego samego zbioru współużytkowanych zasobów: komponentów oprogramowania, dokumentów z wymogami, przypadków testowych itd. Najważniejszym z tych zasobów jest architektura zaprojektowana tak, aby zaspokajać potrzeby całej rodziny produktów. Architekci linii produktów wybierają architekturę (lub rodzinę ściśle powiązanych architektur), która będzie przydatna dla wszystkich planowanych produktów. Taka architektura określa, co jest stałe dla wszystkich produktów z rodziny, a co może się zmieniać.

Linie produktów są wartościowym sposobem rozwijania wielu systemów, który w praktyce daje olbrzymie korzyści w czasie do wprowadzenia produktu na rynek,

kosztach, produktywności i jakości produktów. Architektura jest istotą tego podejścia. Podobnie jak inne inwestycje kapitałowe, tak i architektury linii produktów stają się współużytkowanymi zasobami organizacji.

2.11. Architektura umożliwia dodawanie niezależnie rozwijanych elementów

We wcześniejszych modelach rozwoju oprogramowania najważniejszą aktywnością było *programowanie*, a postęp był mierzony w wierszach kodu. W rozwoju opartym na architekturze często koncentrujemy się na *łączeniu elementów*, które nieraz są tworzone osobno, a nawet niezależnie od siebie. Takie łączenie jest możliwe, ponieważ architektura określa elementy, które można włączyć do systemu. Architektura ogranicza możliwe zastępniki (lub dodatki) na podstawie ich interakcji ze środowiskiem, przejmowania i zwalniania kontroli, konsumowanych i produkowanych danych, dostępu do danych i protokołów używanych do komunikacji i współużytkowania zasobów. Więcej na ten temat piszemy w rozdziale 15.

Gotowe komercyjne komponenty, oprogramowanie otwartoźródłowe, publicznie dostępne aplikacje i usługi sieciowe to przykłady niezależnie rozwijanych elementów. Złożoność i powszechność procesu integrowania wielu niezależnie zbudowanych jednostek w systemie doprowadziły do powstania całej grupy narzędzi programowych, takich jak Apache Ant, Apache Maven, MSBuild i Jenkins.

W przypadku oprogramowania korzyści mogą być następujące:

- skrócenie czasu do wprowadzenia produktu na rynek (łatwiej powinno być zastosować czyjeś gotowe rozwiązanie, niż zbudować własne),
- wyższa niezawodność (w powszechnie używanym oprogramowaniu większość błędów powinna już być wyeliminowana),
- obniżenie kosztów (dostawca oprogramowania może przerzucić koszty prac rozwojowych na wielu klientów),
- elastyczność (jeśli element, który chcesz kupić, nie jest wysoce wyspecjalizowany, często jest dostępny z kilku źródeł, co poprawia Twoją pozycję negocjacyjną).

Otwarty system charakteryzuje się tym, że definiuje zestaw standardów dla elementów oprogramowania. Te standardy określają, jak elementy się zachowują, jak wchodzą ze sobą w interakcje, jak współużytkują dane itd. Otwarty system ma umożliwiać wielu dostawcom tworzenie potrzebnych elementów, a nawet zachęcać ich do tego. Pozwala to uniknąć uzależnienia się od producenta, kiedy tylko jeden producent

potrafi udostępnić dany element, przez co może dyktować wyższe ceny. Otwarte systemy są oparte na architekturze określającej elementy i interakcje między nimi.

2.12. Ograniczanie listy możliwości w projekcie

Po zebraniu różnych przydatnych rozwiązań architektonicznych staje się jasne, że choć elementy oprogramowania można łączyć ze sobą na niemal nieskończenie wiele sposobów, korzystne może być dobrowolne ograniczenie się do stosunkowo niewielkiej liczby elementów i interakcji między nimi. Dzięki temu minimalizujemy złożoność procesu projektowania rozwijanego systemu.

Inżynier oprogramowania nie jest *artystą*, dla którego najważniejsze są kreatywność i swoboda. W inżynierii ważna jest dyscyplina, która po części bierze się z *ograniczania* listy możliwości do sprawdzonych rozwiązań. Przykładowe sprawdzone rozwiązania projektowe to taktyki i wzorce, które szczegółowo omawiamy w części II tej książki. Ponowne wykorzystanie gotowych elementów to następne podejście pozwalające ograniczyć listę możliwości w projekcie.

Ograniczenie listy możliwości w projekcie do sprawdzonych rozwiązań może przynieść następujące korzyści:

- wyższy poziom ponownego wykorzystania elementów,
- bardziej standardowe i prostsze projekty, które są łatwiejsze do zrozumienia i przekazania, a także dają bardziej przewidywalne skutki,
- łatwiejsze analizy z wyższym poziomem pewności co do ich poprawności,
- krótszy czas dokonywania wyborów,
- lepsze współdziałanie elementów.

Bezprecedensowe projekty są ryzykowne. Sprawdzone projekty są, no cóż, sprawdzone. Nie chcemy przez to powiedzieć, że projekt oprogramowania nigdy nie może być innowacyjny ani nie powinien obejmować nowych i ekscytujących rozwiązań. Jest to do przyjęcia. Jednak rozwiązań nie należy wymyślać tylko po to, aby były nowatorskie. Nowinek należy szukać, jeśli istniejące rozwiązania nie wystarczają do rozwiązania danego problemu.

Cechy oprogramowania wynikają z wyboru taktyk lub wzorców architektonicznych. Taktyki i wzorce pożądane w określonym problemie powinny zwiększać jakość wynikowego rozwiązania projektowego, na przykład ułatwiać wybór między sprzecznymi ograniczeniami projektowymi, zwiększać wiedzę na temat słabo zrozumianych kontekstów projektu i pomagać ujawnić niespójne aspekty wymagań. Taktyki i wzorce architektoniczne omawiamy w części II.

2.13. Podstawa do szkoleń

Architektura, włącznie z opisem interakcji między elementami w celu wykonywania oczekiwanych zadań, może stanowić pierwsze wprowadzenie do systemu dla nowych uczestników prac nad projektem. To podkreśla nasze słowa na temat tego, że ważnym zastosowaniem architektury oprogramowania jest ułatwianie komunikacji między różnymi interesariuszami i zachęcanie do niej. Architektura stanowi wspólny punkt odniesienia dla wszystkich zainteresowanych.

Perspektywy modułów doskonale nadają się do prezentowania struktury projektu: kto czym się zajmuje, które zespoły są przydzielone do poszczególnych części systemu itd. Perspektywy komponenty-złącza są świetnym wyborem do wyjaśniania, jak system powinien działać i wykonywać swoje zadania. Perspektywy alokacji informują nowych członków projektu o tym, jakie miejsce przydzielone im elementy zajmują w środowisku rozwojowym lub wdrożeniowym projektu.

2.14. Podsumowanie

Architektura oprogramowania jest ważna z wielu powodów technicznych i nietechnicznych. Nasza 13-punktowa lista obejmuje następujące korzyści:

1. Architektura może albo utrudniać, albo ułatwiać uzyskanie głównych atrybutów jakościowych systemu.
2. Decyzje podjęte w trakcie tworzenia architektury umożliwiają wnioskowanie na temat zmian i zarządzanie nimi w trakcie ewoluowania systemu.
3. Analiza architektury umożliwia wczesne oszacowanie atrybutów jakościowych systemu.
4. Udokumentowana architektura ułatwia komunikację między interesariuszami.
5. Architektura jest nośnikiem najwcześniejszych, a tym samym najbardziej fundamentalnych i najtrudniejszych do zmiany decyzji projektowych.
6. Architektura określa zestaw ograniczeń dotyczących późniejszej implementacji.
7. Architektura wyznacza strukturę organizacji lub na odwrót.
8. Architektura może być podstawą do stopniowego rozwijania oprogramowania.
9. Architektura jest najważniejszym artefaktem, który umożliwia architektowi i menedżerowi projektu wnioskowanie na temat kosztów i harmonogramu prac.
10. Architekturę można stworzyć w formie uniwersalnego modelu wielokrotnego użytku, który stanowi istotę linii produktowej.

11. Programowanie oparte na architekturze wymaga skupienia się na łączeniu komponentów, a nie tylko na ich tworzeniu.
12. Przez ograniczenie alternatywnych projektów architektura kierunkuje kreatywność programistów, ponieważ zmniejsza złożoność projektu i systemu.
13. Architektura może być podstawą do szkolenia nowych członków zespołu.

2.15. Literatura

W książce *The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise* Gregora Hohpego znajdziesz opis wyjątkowych możliwości, jakie architekci mają w zakresie interakcji z osobami zajmującymi różne stanowiska w firmie i poza nią oraz ułatwiania komunikacji między interesariuszami [Hohpe 20].

Praprzodkiem prac na temat architektury i organizacji jest [Conway 68]. Zgodnie z prawem Conwaya „organizacje, które projektują systemy […] są ograniczone do tworzenia projektów będących kopiami struktur komunikacji w tej organizacji".

Koncepcja „chodzącego szkieletu" została opisana przez Cockburna w książce *Agile Software Development: The Cooperative Game* [Cockburn 06].

Dobrym przykładem standardu dla architektur otwartych systemów jest AUTOSAR. Został on opracowany na potrzeby branży motoryzacyjnej (*https://www.autosar.org/*).

Kompletne omówienie tworzenia linii produktów informatycznych znajdziesz w [Clements 16]. Inżynieria linii produktów na podstawie cech to nowe, oparte na automatyzacji podejście do budowania linii produktów, które rozszerza inżynierię oprogramowania do poziomu systemów. Dobre podsumowanie tego zagadnienia znajdziesz w [INCOSE 19].

2.16. Pytania do dyskusji

1. Jeśli masz zapamiętać tylko jedną rzecz z tej książki, jest nią… Dodatkowe punkty przyznaj sobie za niepodglądanie.
2. Dla każdego z 13 opisanych w tym rozdziale powodów, dla których architektura jest ważna, przyjmij postawę przekorną: opisz okoliczności, w jakich architektura nie jest konieczna, aby uzyskać pożądane wyniki. Uzasadnij swoje zdanie. Postaraj się wymyślić inne okoliczności dla każdego z 13 powodów.
3. W tym rozdziale argumentujemy, że architektura zapewnia szereg konkretnych korzyści. Jak można zmierzyć w określonym projekcie korzyści związane z każdym z 13 punktów?

4. Załóżmy, że chcesz wprowadzić w swojej firmie podejście oparte na architekturze. Zarząd jest otwarty na propozycję, ale chce wiedzieć, jaki będzie zwrot z inwestycji. Jak na to odpowiesz?
5. Uporządkuj według ważności 13 powodów z tego rozdziału. Zastosuj kryteria, które są dla Ciebie istotne. Odpowiedź uzasadnij. Jeśli możesz wybrać tylko dwa lub trzy powody, aby zarekomendować zastosowanie architektury w projekcie, które z nich podasz i dlaczego?

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Wydawać by się mogło, że gruntowna wiedza o architekturze oprogramowania jest zbędna: istnieją infrastruktury chmurowe, mikrousługi, wzorce projektowe, po które wystarczy tylko sięgnąć. Niestety, to nie jest prawda. Konieczność ciągłego dostosowywania i rozbudowywania oprogramowania, a także rosnąca złożoność systemów sprawiają, że wiedza architekta wciąż jest niezbędna. Jeśli architekt nie wykona swojej pracy dobrze, gotowy system będzie trudny do zrozumienia, modyfikowania i debugowania. Stanie się ciężarem dla firmy.

Ta książka to kompleksowy przewodnik po tworzeniu architektury nowoczesnego oprogramowania. Pomoże Ci zrozumieć, czym jest architektura oprogramowania i jak nią zarządzać w zdyscyplinowany i skuteczny sposób. Poznasz zasady przekształcania architektury w celu rozwiązywania nowych problemów i tworzenia architektury wielokrotnego użytku. Otrzymasz praktyczne wskazówki, zapoznasz się też z metodami stosowanymi przez ekspertów i ze sprawdzonymi modelami. Nauczysz się radzić sobie z coraz większymi wymaganiami i coraz wyższym poziomem abstrakcji dynamicznie zmieniających się systemów. Znajdziesz tu również informacje na temat optymalizowania za pomocą architektury najważniejszych atrybutów jakościowych systemów.

Najciekawsze zagadnienia:

- wpływ architektury na środowisko techniczne, cykle życia projektów i profile biznesowe
- optymalizacja jakości za pomocą architektury
- architektura dla rozwiązań mobilnych, chmury, uczenia maszynowego i przetwarzania kwantowego
- tworzenie architektury uwzględniającej wydajność energetyczną i bezpieczeństwo
- zarządzanie długiem architektonicznym
- właściwa architektura a rozwój organizacji

Len Bass jest wykładowcą i autorem uznanych książek. Od ponad 50 lat zajmuje się rozwojem oprogramowania. Obecnie wykłada podejście DevOps na uczelni Carnegie Mellon.

Dr Paul Clements zajmuje się inżynierią linii produktów systemowych. Pracował nad wbudowanymi systemami czasu rzeczywistego w U.S. Naval Research Laboratory w Waszyngtonie. Autor blisko 100 prac z dziedziny inżynierii oprogramowania.

Rick Kazman jest badaczem. Zajmuje się architekturą oprogramowania i ekonomią inżynierii oprogramowania. Współtworzył cenione metody i narzędzia służące do analizowania architektury. Jest przewodniczącym IEEE TAC i członkiem ICSE Steering Committee.

Upewnij się, że znasz budowę wszystkiego, co chcesz przedstawić

Leonardo da Vinci

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9052-2

Cena: 99,00 zł

Software Engineering Institute | Carnegie Mellon

informit.com/sei