Od developera do architekta

Wielu programistów zastanawia się się nad następnym krokiem na ścieżce swojego rozwoju zawodowego – część chce awansować na seniora, ten i ów na menadżera, a niektórzy decydują się na stanowisko, w którego nazwie jest „architekt". Funkcja ta bywa w branży IT dość mętnie zdefiniowana, dziedzina jest uważana za naukę tajemną, a potencjalni kandydaci często mają mgliste pojęcie o tym, czym będą się zajmować. W tym artykule chciałbym rozwiać wątpliwości i mity otaczające architekturę oprogramowania i rolę architekta.

HISTORIA O TYM, JAK ZOSTAŁEM ARCHITEKTEM

Przyznam się szczerze: trochę z przypadku. Moja pierwsza po studiach praca w branży polegała mniej więcej na wstawianiu ifów do kodu spadkowego (ang. *legacy code*), którego najstarsze fragmenty zostały napisane, kiedy byłem jeszcze w podstawówce. Moje stanowisko, z przyczyn bodajże podatkowych, nazywało się „specjalista ds. rozwoju oprogramowania", ale wszyscy wołali na to po prostu „programista".

W tym wstawianiu ifów byłem naprawdę dobry, więc na rozmowie okresowej w 2006 roku mój przełożony zaproponował mi stanowisko starszego programisty. Odpowiedziałem wtedy, że „wolałbym zostać młodszym architektem" i jakoś malowniczo to uargumentowałem, ale w głowie miałem głównie myśl, że „architekt lepiej brzmi i lepiej wygląda w CV, robi to samo i do tego rządzi". Pojęcie o tym, czym naprawdę zajmują się architekci, miałem dość mgliste, jak wielu szeregowych pracowników branży IT. Rola architekta obrośnięta jest mitami, półprawdami i nieporozumieniami.

Przekonanie, któremu uległem – że architekt to taki senior developer na sterydach, jest jednym z najbardziej rozpowszechnionych. Jest w nim ziarno prawdy – są organizacje, które radzą sobie z problemem szklanego sufitu, czyniąc stanowisko (choć niekoniecznie rolę) architekta kolejnym szczeblem w karierze seniora. Dodatkowo wielu ludzi pełniących rolę architekta relatywnie dużo koduje, i są to rzekomo rzeczy przekraczające kompetencje przeciętnego developera. Z drugiej strony niektórzy architekci kodują mało lub wcale, co jest źródłem mitu, że architekt to taki trochę bardziej techniczny menadżer – rozumie może więcej niż przeciętny kierownik, ale programować i tak (już) nie potrafi.

W organizacjach, gdzie architektów jest niewielu, stanowisko to faktycznie może wydawać się prestiżowe, ale znam struktury, gdzie wszyscy są, przynajmniej nominalnie, architektami. Sam pracuję w zespole, który składa się z kilkunastu architektów. Nieprawdą jest, jakoby architekci co do zasady zarabiali lepiej – z mojego doświadczenia wynika, że płace są porównywalne do wynagrodzeń developerów na tym samym poziomie doświadczenia (przy czym zwykle nawet początkujący architekt już na wejściu jest seniorem). Prawdą za to jest, że architekt podejmuje decyzje – a w każdym razie tego się od niego oczekuje. Jakie decyzje? Oczywiście architektoniczne, ale żeby wyjaśnić, co to tak naprawdę znaczy, musimy najpierw zastanowić się nad jedną rzeczą.

CO TO JEST ARCHITEKTURA OPROGRAMOWANIA?

Okazuje się, że sprawa nie jest prosta, a architektura to trochę *scientia occulta*, nauka tajemna. Pamiętam, jak dawno temu, tuż przed obroną pracy magisterskiej, recenzent wziął mnie na stronę i przestrzegł: „proszę pod żadnym pozorem nie mówić >>architektura<<, bo to robi bardzo złe wrażenie, kiedy komisja kłóci się między sobą, co to właściwie jest". Przemysł IT od zawsze borykał się z precyzyjnym zdefiniowaniem tego pojęcia. Kiedy wyjaśniam laikom, czym się zajmuję w pracy, używam, trochę z lenistwa, metafory budowli. Mówię mianowicie, że oprogramowanie przypomina budynki, że składa się je z mniejszych elementów, płyt, cegiełek i bloczków, a architekt oprogramowania pełni podobną funkcję, jak „zwykły" architekt. Ta „budowlana" metafora jest, *nomen omen*, fundamentalnie błędna. Gmachy mają raczej tylko cechy statyczne – wymiary, masę, liczbę i umiejscowienie pięter, okien i drzwi itp. Oczywiście wysokie budynki kołyszą się z wiatrem, a budowle na obszarach nawiedzanych przez trzęsienia ziemi trzęsą się tak, żeby się nie zawalić, ale miażdżąca większość budynków oprócz stania nie robi nic ekscytującego. Oprogramowanie jest zupełnie inne, bo z założenia służy do tego, żeby *coś* robić. Z tego powodu lepsza byłaby metafora skrzydła – skrzydło ma zestaw cech statycznych, takich jak wymiary i masa, wykazuje dynamiczne zachowania – tzn. lata, i podobnie jak oprogramowanie, ma pewien zestaw „atrybutów jakościowych" – jak dobrze zachowuje się w locie, czy jest wodoodporne, trzyma ciepło itp. Metafora skrzydła jest jednak nieco zbyt ryzykowna, żeby używać jej na imprezach wśród nieznajomych. Na pytanie, kim jest w niej architekt, nie ma bezpiecznej odpowiedzi, bo mamy do wyboru „ślepą siłę ewolucji" albo „inteligentnego projektanta". Ryzykujemy, że kogoś możemy obrazić.

Inżynierowie na szczęście nie potrzebują i brzydzą się metaforami, zadowalając się tylko formalną definicją. W 2010 roku badacze z SEI (Software Engineering Institute), działającego w Carnegie Mellon University i zajmującego się m.in. teorią architektury oprogramowania, przeprowadzili badania, mające na celu znalezienie jej „jedynej słusznej" definicji. Przegląd literatury przedmiotu dostarczył około trzystu, a formularz na stronie instytutu mniej więcej ośmiuset. Nawet pobieżny ich przegląd pozwala spostrzec powtarzający się wzorzec: [architektura to] „fundamentalna organizacja systemu, uosabiana przez jego komponenty, relacje między nimi, i z otoczeniem [systemu], oraz zasady rządzące jego projektowaniem i ewolucją". Ta konkretna definicja pochodzi ze standardu, czy też

„rekomendowanej praktyki" ANSI/IEEE [1], ale powtarza się w „kanonicznych" książkach nt. tworzenia i dokumentowania architektury (na czele ze znakomitymi „Software Architecture in Practice" [2] i „Documenting Software Architectures" [3], autorstwa związanych z SEI Clementsa, Bassa i Kazmana). Typowe, chciałoby się powiedzieć „klasyczne" definicje architektury skupiają się na strukturze systemu, na który patrzymy z lotu ptaka i nie widzimy pojedynczych linii kodu, a raczej całe klasy, moduły, komponenty czy też pakiety. Co jednak począć ze wspomnianymi w standardzie ANSI/IEEE „projektowaniem i ewolucją"? Czyżby metaforą skrzydła była aż tak trafna, że dało się nią obrazić wszystkich, niezależnie od światopoglądu?

„Nowoczesna" definicja architektury oprogramowania, którą chciałbym przytoczyć, pochodzi z wydanej w 2020 roku książki „Fundamentals of SW architecture" [4] autorstwa znakomitego duetu Neal Ford i Mark Richards. Definicja mówi, że na architekturę oprogramowania składa się struktura systemu, charakterystyki architektoniczne, które system musi wspierać, decyzje architektoniczne i zasady projektowania. Struktura dotyczy wzorca, czy też stylu architektonicznego, w którym system jest zaimplementowany – np. mikrojądra, architektury warstwowej czy opartej na usługach. Powiedzenie, że „w naszym systemie mamy architekturę opartą o mikroserwisy", nie daje pełnego obrazu – bo mówi tylko o strukturze. „Klasycznym" definicjom to wystarcza.

Definicja „nowoczesna" jest pełniejsza – na drugim miejscu wymienia „charakterystyki architektoniczne". Mówiąc inaczej, to „atrybuty jakościowe" czy po prostu „wymagania niefuncjonalne" – rzeczy takie jak skalowalność, bezpieczeństwo, modyfikowalność, wydajność itp. Wrócimy jeszcze do tego.

Kolejny element architektury to decyzje. Decyzje definiują zasady, według jakich system powinien być tworzony, formułują ograniczenia i instruują zespoły developerskie o tym, co jest, a co nie jest dozwolone (np. „warstwa prezentacji nie może bezpośrednio wywoływać funkcji z warstwy danych"). Oczywiście zasady czasem trzeba złamać – nazywa się to „rozbieżnością", albo z angielskiego „wariancją" (ang. *variance*). Dojrzałe organizacje mają (albo przynajmniej udają, że mają) modele, używane przez jakieś ciało decydujące o architekturze (albo przez Głównego Architekta), których celem jest formalizacja procesu poszukiwania rozbieżności (tzn. wyjątku) od konkretnej decyzji architektonicznej na podstawie uzasadnienia i analizy za i przeciw.

Ostatni element, zasady projektowania, to raczej wskazówki niż twarde reguły. Zasada projektowania (ang. *design principle*) może np. mówić, że „z przyczyn wydajnościowych preferujemy komunikację asynchroniczną" – twarda reguła raczej nie uwzględni scenariusza komunikacji między bytami w systemie, ale zasada projektowania daje wskazówki, które pozwalają developerowi wybrać protokół komunikacyjny odpowiedni do konkretnej sytuacji.

ARCHITEKTOWANIE, CZYLI ZA CO PŁACĄ ARCHITEKTOM?

Skoro już mamy pewne pojęcie, czym jest architektura, możemy wrócić do tego, czym zajmują się architekci. Praca architekta to proces podejmowania decyzji na podstawie celów biznesowych, wymagań i ograniczeń i komunikowania ich zespołowi. Oczywiście nie wszystkich decyzji – pisanie kodu ma to do siebie, że trudno jest to robić, nie tworząc ciągle jakichś zależności do *czegoś*, i praktycznie każda linijka wymaga jakiejś decyzji projektowej. No właśnie – projektowej, nie architektonicznej. Wyprodukowano tony papieru, żeby rozstrzygnąć, gdzie kończy się projektowanie (ang. *design*), a zaczyna architektura. Zapytanie Google o „architecture vs design in software engineering" da nam 10 różnych odpowiedzi. Najprościej powiedzieć, że architektura mówi o strukturze, w jakiej umieszczone są poszczególne komponenty systemu, a projektowanie zajmuje się tym, co jest wewnątrz każdego komponentu. Osobiście preferuję definicję Grady Boocha, twórcy UML (Unified Modelling Language). Mówi on mianowicie, że design to „(…) nazwana struktura lub zachowanie systemu (…) design reprezentuje więc jeden punkt w przestrzeni potencjalnych decyzji. Każda architektura to design, ale nie każdy design to architektura. Architektura reprezentuje *znaczące* decyzje, gdzie znaczenie jest mierzone kosztem zmiany". Jako architekci podejmujemy więc tylko *znaczące* decyzje: jakiego języka programowania użyć i czy będziemy budować monolit, czy może mikrousługi. Design decyduje o tym, jak to zrobić, a implementacja to spacje vs. tabulatory. Głównym celem istnienia architektury jest redukcja kosztu tworzenia oprogramowania i wprowadzania w nim zmian. Jest w software coś szczególnego, cecha, której nie ma żadna inna technologia, a co czyni problem wprowadzania zmian wyjątkowo nieznośnym. Otóż oprogramowanie nie zużywa się, nie starzeje i nie psuje. Fotel, na którym siedzę, pisząc te słowa, jest bardzo fajny i wygodny, ale mam go już kilka lat i widać na nim ich upływ, zwłaszcza w miejscach narażonych na kontakt z moimi *glutei maximi*. Za jakiś czas będę go musiał wymienić – przedmiot, który wybiorę, ktoś wcześniej zaprojektuje, wyprodukuje, zapakuje, zareklamuje, sprzeda i dostarczy. Taki tradycyjny cykl zużycia i wymiany nie zachodzi w przypadku oprogramowania. Kiedy technologia zostanie wdrożona i zaadoptowana, żyje wiecznie, kryjąc każdą błędną, czy po prostu zdezaktualizowaną, decyzję projektową. Praktycznie zawsze łatwiej, szybciej i taniej jest położyć na nieaktualny system kolejną warstwę software'owej szpachli, niż go przeprojektować i wymienić na nowy. Tak rosną nasze architektoniczne piramidy (w najlepszym przypadku – zwykle są to wieże babel, kule błota lub innej brązowej substancji).

Dodatkowo architektura ma za zadanie utrzymywanie developerów w ryzach. Programistów jest (nie zawsze dobrych) coraz więcej, średnia spada. Programowanie jest trudne. Świadomie lub nie, developerzy czasami piszą kiepski kod. Wpadkę może zaliczyć nawet wykształcony, doświadczony, profesjonalny i odpowiedzialny developer. W sytuacji kiedy wyrzeźbi jakiejś przecudnej urody kawałek programu, perfekcyjny płatek śniegu i pokaże go swojemu przełożonemu, usłyszy najpewniej coś w stylu „bardzo ładnie, to teraz dwa tuziny do czwartku i wiaderko do końca przyszłego tygodnia". Nie jesteśmy w stanie patrzeć każdemu na ręce – stąd decyzje architektoniczne, zasady projektowania i dobrze zdefiniowane interfejsy, między którymi zawsze można zamknąć niezbyt dobrze pachnący kod w nadziei, że kiedyś da się go przepisać.

Fanatyczny zelota Agile™ powinien się, czytając poprzedni akapit, zagotować. „W moim zespole są tylko Developerzy, bardziej cenimy działający kod, niż szczegółową dokumentację, i w ogóle nie robimy projektowania przed implementacją". Zgadzam się, że robienie BDUF (Big Design Up Front) szczegółowego, kompletnego projektu przed

rozpoczęciem implementacji jest głupie. Brak projektowania przed implementacją jest jeszcze głupszy. Fakt, że w zespole nie ma roli architekta, nie oznacza, że nie są podejmowane decyzje architektoniczne – są, bezustannie, i powinniśmy się modlić, by były właściwe. Bez architektów możemy oczekiwać chaosu, aplikacji o architekturze Wielkiej Kuli Błota, bałaganu w kodzie, niespójnych podejść do rozwiązywania takich samych problemów, ignorowania wymagań niefunkcjonalnych, kłopotów z utrzymaniem i CV-driven design. Ostatnie pojęcie wymaga wyjaśnienia – posłużę się przykładem. W każdym sklepie z aplikacjami na telefon jest mnóstwo apek, które dostarczają jakiejś prostej usługi, np. są minutnikiem do jajek. Dość łatwo jest znaleźć taki wyspawany przez mikrofirmę minutnik, który zajmuje np. 200 MB, choć jest bardzo prościutki. Rzut oka na listę wykorzystywanego open source pozwala stwierdzić, że w aplikacji użyto czterech różnych frameworków do obsługi UI. Bliższa inspekcja wykazuje, że mikrofirma to czterech programistów – przypuszczalnie każdy chciał umieścić w swoim CV konkretny pakiet, żeby móc potem powiedzieć na kolejnej rozmowie kwalifikacyjnej: „Oczywiście, że znam ten framework, sam zaproponowałem, żebyśmy go użyli w naszej ostatniej aplikacji".

Agility, zwinność, to pogodzenie się z ciągłą zmianą, szybka reakcja, częste wydania, poszukiwanie i reakcja na informację zwrotną – *inspect and adapt*. To samo można powiedzieć o dobrej architekturze. Jedna z „Zasad Zwinnego Tworzenia Oprogramowania" mówi, że „ciągłe skupienie na technicznej doskonałości i dobrym projektowaniu zwiększa zwinność" [5]. Wszystkie dobre architektury są iteracyjne i „zwinne", czy się to komuś podoba, czy nie. Procesy i metodyki są w większości wzajemnie niezależne od architektury, punktem stycznym są wyłącznie wymagania. W znakomitym artykule [6] były pilot myśliwca zdefiniował zwinność jako zdolność do wykonywania pętli obserwacja-orientacja-decyzja-akcja (OODA) szybciej niż robi to nasz przeciwnik. To nasza kultura pracy, a nie narzędzia czy metodyki określają naszą szybkość wykonywania pętli OODA. Doba architektura umożliwia bycie zwinnym, a zwinność rzadko pojawia się w środowisku obojętnym na kwestie architektoniczne. Zwinność jest w praktyce kolejnym wymaganiem niefunkcjonalnym.

Oczywiście pozostaje pytanie, jak dużo powinniśmy mieć architektury, jeśli chcemy być i pozostać zwinni. Odpowiedź jest bardzo prosta – tyle, ile potrzeba. Mogę w tym miejscu polecić książkę George Fairbanksa „Just Enough Sofware Architecture" [7] (w wolnym tłumaczeniu „Tyle architektury, ile potrzeba"). Za rekomendację niech posłuży jedna z recenzji: „Nie koduję zbyt dobrze. Po przeczytaniu tej książki nadal nie koduję zbyt dobrze, ale teraz wiem dlaczego". Sięgając po tę książkę, nie trzeba mieć żadnego szczególnego doświadczenia – każdy programista może ją przeczytać w zasadzie z marszu.

SKĄD SIĘ BIORĄ DECYZJE?

Wróćmy jeszcze na chwilę do decyzji architektonicznych. Wpływ na nie ma kilka rzeczy. Po pierwsze, wymagania. Te funkcjonalne, które mówią, jak „system dostarcza swoją wartość biznesową", dostajemy w postaci historyjek lub przypadków użytkownika, scenariuszy, funkcjonalności itp. Osobiście nie jestem wielkim fanem powszechnie używanego terminu „wymagania niefunkcjonalne" – kadra zarządzająca ma tendencję do wyłączania się, kiedy słyszy słówko „niefuncjonalne". Brzmi ono bardzo podobnie do „niepotrzebne" oraz do „nie możemy tego sprzedać". Kiedy chcę brzmieć mądrze, architektonicznie i hermetycznie, mówię „charakterystyki architektoniczne" albo używam potworka „cross-cutting concerns". Kiedy chcę być dobrze zrozumiany, używam frazy „atrybuty jakościowe" – kierownictwo lubi słyszeć, że dbamy (jakoś) o jakość (czasem działa też zaklęcie „service-level agreement"). To są rzeczy takie jak zwinność, modyfikowalność, zarządzalność, bezpieczeństwo, wydajność itp. Po angielsku nazywa się to „-ities" (bo *agility*, *modifiability*, *manageability*, *security* itd.). Z *ities* są dwa problemy. Pierwszy to kwestia kontekstu – np. pojęcie modyfikowalności może mieć w zależności od niego różne znaczenia. Kontekst jest wszystkim i niekoniecznie będziemy w stanie odczytać go z samych wymagań. Drugie wyzwanie to liczba atrybutów jakościowych – np. w Wikipedii [8] można znaleźć około setkę. Zwykle staramy się zredukować złożoność problemu i zwracać uwagę tylko na niewielki podzbiór, a w jego ramach dodatkowo priorytetyzować. Po prostu bardzo rzadko da się zoptymalizować wszystkie atrybuty naraz. Różne wzorce, czy też style architektoniczne, w naturalny sposób wspierają różne charakterystyki architektoniczne, a są ich dziesiątki. Dodatkowo wymaganie wymaganiu nierówne – na architekturę wpływają tylko takie, które są „architektonicznie znaczące" (ASR – *Architecturally Significant Requirement*), przy czym w praktyce to architekt musi ustalić, które wymagania zasługują na miano ASRów, bo raczej nie zdarza się, żeby były po prostu wypisane od myślników. Większość „formalnych" definicji ASR jest niestety *ignotum per ignotum* (moje ulubione to „wymaganie jest znaczące architektonicznie, jeśli ma mierzalny wpływ na architekturę systemu"), więc musimy zdać się na własną ocenę. ASRy są zwykle „duże" albo nienegocjowalne, łamią jakieś założenia lub zasady czy też są po prostu trudnymi wyzwaniami technicznymi. Określenie, czy dana decyzja może być podjęta bez udziału architekta, także bywa niełatwe, a nawet decyzje „niearchitektoniczne" mogą wpływać na atrybuty jakościowe systemu. Na przykład wydajność może zależeć od tego, jak dużo jest komunikacji między komponentami, jak używane są współdzielone zasoby, jakie wybrano algorytmy – o tym wszystkim decyduje architekt, ale implementacja, której jakość wpływa znacząco na wydajność, należy do programistów. Kiedy na szkoleniach opowiadam o *maintainability* i pytam programistów, kto jest odpowiedzialny za dokumentację, odpowiadają chórem: „architekci", chociaż wszyscy przeczytali „Clean Code" [9] i rzekomo wiedzą, że kod to też dokumentacja i powinno się go dać czytać jak dobrą powieść.

Ale wymagania to nie wszystko. Kolejny element wpływający na decyzje architektoniczne to ograniczenia. Żyjemy w świecie realnym, tak jak nasz software. Prawdziwy świat ma ograniczenia – rzeczy takie jak czas, budżet, technologia, ludzie, umiejętności, polityka itp. Ograniczenia też trzeba czasem priorytetyzować. Ostatnia rzecz to zasady – pryncypia. Na wybór niektórych duży wpływ ma zespół – od konwencji kodowania i nazewniczej, przez podejście do testowania, aż do zasad review. O zasadach projektowania już wspominałem – dotyczą zwykle rzeczy takich jak komunikacja synchroniczna/asynchroniczna, *stateless*/*stateful*, anemiczny/bogaty model danych, modularność, warstwowość, separacja odpowiedzialności, bezpieczeństwo, obsługa błędów, logowanie itp. Dobre zrozumienie wymagań, atrybutów jakościowych, ograniczeń i pryncypiów to podstawa dobrej architektury.

JAK ZOSTAĆ ARCHITEKTEM I DLACZEGO PROGRAMIŚCI MAJĄ ŁATWIEJ?

Potencjalni architekci mają do wyboru dwie ścieżki: albo dadzą się awansować wewnątrz swojej organizacji, albo dadzą sobie podwyżkę i awans gdzie indziej. Architekt powinien być pracownikiem doświadczonym, zwykle potrzebne jest minimum 3-5 lat na stanowisku programisty. Powinni jednak pamiętać o jednym: wiele umiejętności, które pozwalają na zostanie architektem, niekoniecznie pomoże w utrzymaniu się na tym stanowisku (a już na pewno nie gwarantuje kolejnych awansów, podwyżek czy premii). „Architektowanie" niekoniecznie jest tym, czym się wydaje. Owszem, jest pełne wyzwań i daje dużo satysfakcji, ale nie polega na siedzeniu cały dzień i podejmowaniu decyzji. Nawiasem mówiąc, wśród młodych developerów pokutuje stereotyp architektów jako mężczyzn w średnim wieku, zamkniętych w wieżach z kości słoniowej, oderwanych od rzeczywistości i podejmujących bezsensowne decyzje. Jakoś obraz architektów pracujących z zespołem, pomagających mu odnieść sukces, a nie tylko decydujących o jakichś tam bibliotekach, słabo się sprzedaje.

Wielu z nas lubi programować. Tworzenie rzeczy jest fajne. Programowanie jest obiektywne – kod kompiluje się i działa albo nie, testy przechodzą albo nie, mamy nawet własne światła uliczne. Na poziomie historyjki użytkownika mamy kryteria akceptacji. Możemy mieć inną od kolegi opinię, jak coś powinno być zakodowane, ale w ostateczności zawsze można zakodować dwa razy i zmierzyć. Satysfakcja (lub frustracja) często jest niemal natychmiastowa – naciskamy guziczek i ciach, „All OK!". W porywach trzeba poczekać do końca sprintu, ew. na *release*. Jeśli nie pracujemy w trybie help deska, to kolejne zadania płyną strumieniem o mniej więcej stałym natężeniu. Implementujemy tyle a tyle story pointów na sprint. Są wahnięcia, ale nie dramatycznie duże. Praca opiera się na jasno (jasssne…) zdefiniowanych zadaniach, zadania bierzemy z backlogu. Raj.

Architektura nie ma zielonej lampki. Czerwonej zresztą też nie. Kiedy podejmujemy jakąś decyzję, zwykle mija trochę czasu, zanim zacznie przynosić owoce. „Trochę czasu" to nie 1-2 sprinty czy jeden *release*. To mogą być lata. Uczucie, że miało się rację lata temu, jest bardzo specjalne. Oczywiście czasem jest to tylko *schadenfreude*, jeśli podążono inną drogą niż rekomendacja architektoniczna. Musimy dokumentować podjęte decyzje: problem, założenia, ograniczenia, opcje – wady i zalety, co uchwalono, dlaczego, konsekwencje. Zapisać, złożyć w kapsule czasu, wykopać po latach. Mamy więcej autonomii, ale robota przychodzi do nas od czasu do czasu w sposób wybuchowy. Trzeba naprawdę dobrze umieć zarządzać swoim czasem. Zadania są czasem dość mgliste: „weź i zewaluuj *xyz*". Na pytanie „jak?" usłyszysz prawdopodobnie: „coś wymyślisz" oraz „zrób tak, żeby było dobrze". Niekiedy musimy sobie sami znaleźć pracę, ale zwykle mamy jej o wiele za dużo i musimy brutalnie priorytetyzować. Pamiętam, jak kiedyś poskarżyłem się koledze z zespołu, gdy nieopatrznie policzyłem, iloma tematami naraz się zajmuję. Popatrzył na mnie jak na wariata i zapytał: „Tylko trzynaście?". Więcej się nie żaliłem, jeszcze by ktoś pomyślał, że mam wolne przebiegi, a tymczasem pracy tyle, że nie ma czasu taczki załadować. Najgorsze, co się przydarza, to tzw. wrzutki: przychodzi ktoś i ze szczenięcą miną prosi o pomoc. Strzeżcie się rzeczy, których wasz przełożony nie chce, żebyście robili.

Z drugiej strony niektóre rzeczy są jak brokuły, nie musisz ich lubić, ale są dla ciebie dobre, bo na przykład pozwalają utrzymywać klientów w poczuciu szczęścia i że robisz dla nich dobre rzeczy.

CZY BĘDĘ JESZCZE KODOWAĆ? MODELOWANIE I EWALUACJA

Jedno z najczęściej pojawiających się w rozmowach z kandydatami na architektów pytań to „czy nadal będę kodować?". Odpowiadam zwykle wymijająco: „Będziesz robić inne rzeczy". Na przykład modelować. Architekci modelują, czyli rysują modele, takie diagramy z prostokątami i strzałkami. Podstawowe zadania modeli to edukowanie i komunikowanie, nie dajcie sobie wmówić, że na modelu można coś sprawdzić. Diagramy mają ogromną wadę – nie kompilują się, a papier, jak wiadomo, przyjmie wszystko. Sam obrazek nie wystarcza – architekt musi rozumieć, do kogo jest kierowany. Rysunek dla developera niekończenie będzie odpowiedni dla szefa działu. Modelujemy dla różnych odbiorców: użytkowników, developerów, testerów, innych architektów, specjalistów od bezpieczeństwa, mniej lub bardziej technicznych menadżerów różnych szczebli. Jesteśmy jednak, póki co, skazani na takie niedoskonałe narzędzia. Czytanie kodu jest fajne do czasu – owszem, może i dobrze się go czyta, ale nie wszyscy interesariusze mają na to czas, ochotę i umiejętności. Im więcej kodu, tym trudniej go ogarnąć – kiedy mamy do czynienia z systemem na kilka milionów linii w kilku różnych językach, nie mamy innego wyjścia, musimy jakoś zarządzać jego złożonością. Musimy go zdekomponować, żeby go zrozumieć i móc przewidzieć atrybuty jakościowe.

Modele mogą być zapisane z użyciem formalnej notacji. Notacje nie- lub półformalne są jak Święte Księgi Objawień – poddają się wielorakiej interpretacji. Formalizmy są pozbawione tej wady, ale odbiorcy muszą je znać. Czasem mogą myśleć, że je znają – autor może rozumieć niuans między pustym a zamalowanym rombem w UML, odbiorca już niekoniecznie. Czasem możemy jednak chcieć użyć formalizmu, żeby uniknąć niepotrzebnych dyskusji i sprzeciwów. Widziałem kiedyś Poważny Dokument Architektoniczny używający notacji Z. Wstęp był napisany w tonie: „Klękajcie Narody! Oto jest dokument w Notacji Z. Prawdopodobnie jesteście za głupi, żeby pojąć genialną myśl w nim zawartą". Diagramy są często kolorowe – działa to wprost wybornie, gdy wydrukuje się je w odcieniach szarości. Można też napotkać kogoś takiego, jak ja, tzn. złośliwego daltonistę, który nie omieszka wytknąć, że użycie kolorów to jawna dyskryminacja i kwalifikuje się do zbadania przez korporacyjną komisję do spraw etyki. Niezależnie jednak od tego, jak formalny i barwny jest model, to komunikat jest najważniejszy, a nie narzędzia.

Nacisk położony na modelowanie przez niektóre organizacje może być czasem błogosławieństwem – modele pomagają w edukacji i transferze wiedzy, ale również w archeologii. Zakładając oczywiście, że są dokładne i aktualne – brak diagramu jest zwykle lepszy od diagramu, który jest, ale łże.

No dobrze, ale czy będę jeszcze kodować? Będziesz ewaluować! Oceniać rozwiązania, biblioteki, technologie, frameworki, pakiety *open source*. Przy ewaluacji trzeba zadać sobie mnóstwo pytań, zaczynając od „co to robi, co jest w tym ważne, co jest ważne dla mnie i jak dobre jest w tym, co robi?". Potem jest już z górki. Zaczynamy od dokumentacji: czy jest w ogóle jakaś? Czy jest zrozumiała i aktualna?

A jak wygląda kod? Czy ten projekt w ogóle żyje, kiedy był ostatni commit? A może po prostu ten produkt jest dojrzały? Jesteśmy na wersji 0.0.4 czy 4.5? Jak często wychodzą wersje minor i major? Czy mój projekt nadąży za tempem zmian? Jak często zmiany coś psują? Czy nowe wersje są stabilne, czy standardem jest stado patchy po każdej? Jak wygląda community? Czy jakiś większy gracz wspiera projekt? Jeśli to produkt komercyjny, to kto go produkuje, czy firma ma solidne podstawy, czy będziemy w stanie się dogadać? Czy używa tego jeszcze ktoś inny? Czy można kupić wsparcie, treningi, zrobić własne? Czy to się pojawia w CV, czy można zatrudnić ludzi, którzy już to znają? Czy to pasuje do mojej organizacji, czy nie będzie jakichś oporów politycznych? Jak wygląda licencja? Czy sposób, w jaki będziemy tego używać, nie będzie jej łamał, albo spowoduje, że nasz know-how będzie musiał być ujawniony? Czy w razie czego można się odforkować? Czy nie lepiej napisać coś samemu? Co sądzi o tym zespół?

Ludzie potrafią decyzję, że będziemy używać jakiegoś tam pakietu XYZ, potraktować bardzo personalnie. Developerzy generalnie miewają opinie. Często dość zdecydowane. Stawiają opór. W środowisku panuje strach przed używaniem i uczeniem się starych rzeczy. Można też wpaść w syndrom „not invented here" – jest to choroba pożerająca zasoby, ale czasem warto ją złapać. Weźmy na przykład framework do komunikacji wewnątrz naszego systemu. Możemy wziąć coś z półki, albo wręcz za darmo, tzn. *open source*. Kod pisze się i testuje sam, płacimy tylko za integrację (i za prawnika, który sprawdzi, czy używamy pakietu zgodnie z licencją). Jesteśmy jednak zależni od community. Jeśli okręt nagle postanowi popłynąć innym kursem, w szczególności prosto na górę lodową, to możemy wsiąść na inny statek (dużo kosztuje), odforkować się i pożeglować gdzie indziej (czyli piszemy i testujemy sami, na dodatek musimy się dzielić naszym *de facto proprietary* kodem, bo pozostaje on *open source*), albo napisać swój własny framework. Nad rozwiązaniami stworzonymi *in-house* mamy pełną kontrolę, możemy je zoptymalizować pod swoje potrzeby i nie musimy się z nikim nimi dzielić. Tylko niestety trzeba za nie zapłacić. Chyba że jesteśmy Googlem, wtedy nie musimy patrzeć na koszty, możemy wszystko pisać od zera, robić z tego *open source* i reklamować jako nowy standard.

To wszystko wpędza architektów w stan nazywany *analysis paralysis*. Można go leczyć – w moim zespole służy do tego szklana kula, architekci wyższych poziomów używają też podobno obserwacji lotu ptaków, kształtu dziur w serze i układu gwiazd. A tak na poważnie – trzeba mieć świadomość faktu, że każda taka analiza zawsze będzie mocno subiektywna. Możemy oczywiście dołożyć wszelkich starań, wylistować kryteria, nadać im wagi, ale niestety nie ma żadnych uniwersalnych, magicznych formuł. Każda przyzwoita odpowiedź na interesujące pytanie zaczyna się przecież od „to zależy". Możemy się zrelaksować, czerpiąc satysfakcję z przekonania, że niezależnie od tego, jaką decyzję podejmiemy, będzie ona błędna – wystarczy odpowiednio długo poczekać. Decyzja zapada na podstawie wiedzy, jaką mamy w danym momencie. Pierwsze Prawo Architektury mówi, że każda decyzja zawiera jakiś kompromis – jeśli go nie widzisz, to jeszcze go nie znalazłeś/aś.

No dobrze, ale czy będę nadal kodować? Cóż, przy okazji ewaluacji warto zakasać rękawy i trochę się pobrudzić. Będziesz więc kodować prototypy, eksperymenty, *proofs-of-concept*. Będziesz dużo czytać, robiąc review. Najprawdopodobniej będziesz pisać dużo mniej kodu produkcyjnego. Jako architekt nie musisz być mistrzem kodowania, programistą-gwiazdą rocka, ale musisz być mistrzem-konstruktorem, twórcą, budowniczym. No i musisz widzieć „duży obrazek", a to wymaga podniesienia głowy znad ekranu i odsunięcie się nieco od kodu.

To w takim razie jak dużo będę kodować? To zależy od roli architekta w twojej organizacji.

ARCHITEKT NA KAŻDĄ OKAZJĘ

Tak jak SW developer, projektant-programista czy nieśmiertelny „specjalista ds. rozwoju oprogramowania" (mam niejasne wrażenie, że rozpowszechnienie i żywotność tej nazwy ma coś wspólnego z pozyskiwaniem unijnych funduszy na działalność badawczo-rozwojową) może znaczyć różne rzeczy w różnych organizacjach, tak i architekci występują w wielu kolorach, wzorach, smakach i zapachach, a taka sama nazwa stanowiska może znaczyć kompletnie różne rzeczy w różnych organizacjach. Podział, który przedstawię, jest pewną aproksymacją tego, co można napotkać w dużych organizacjach.

Architekt komponentu lub aplikacji (czasem nazywa się *solution architect*) jest odpowiedzialny za architekturę wewnętrzną konkretnego komponentu lub aplikacji. Zwykle siedzi z zespołem projektowym, koduje relatywnie dużo i jest trochę takim senior developerem na sterydach. Większość ludzi zaczyna swoją ścieżkę kariery architektonicznej właśnie w tym miejscu. Piętro wyżej siedzi *portfolio architect*, odpowiedzialny za zbiór kilku komponentów, pracując z ich architektami, ale jest zwykle angażowany wcześniej w czasie życia projektu, pracuje więcej z zarządzającymi stroną biznesową przedsięwzięcia, zajmuje się wizją, mapami drogowymi i pomaga w łagodzeniu tarć między często sprzecznymi interesami poszczególnych komponentów. System architect patrzy na system całościowo, jako na połączenie software i infrastruktury, zajmując się wysokopoziomową strukturą obu. Co ciekawe, taki tytuł to zaklęcie-lep na rekruterów. Kiedy po którymś z kolei porządkowaniu nazw stanowisk zostałem (bez zmiany zespołu i zakresu obowiązków) system architectem, nagle rozwiązał się worek z ofertami pracy. Architekt domenowy zajmuje się jakąś konkretną technologią albo funkcjonalnością i ma zadania nieco bardziej „fundamentalne" – takie jak opracowywanie standardów, implementacji referencyjnych, dobrych praktyk, ewaluacją nowych technologii, PoCami itp. – portfolio architect mu pomaga. Discipline architect skupia się na jakimś konkretnym aspekcie technologii lub domenie biznesowej (standardowy przykład to architekt ds. bezpieczeństwa) i jest czymś w rodzaju konsultanta w różnych projektach w całej organizacji. Prawie na samej górze siedzi enterprise architect – to jest „bardzo senior", pracujący nad wizjami, mapami drogowymi i wzorcami dla całych organizacji, programów czy portfolio. Komunikuje się z wyższymi kadrami zarządzającymi i na co dzień ma zwykle niewiele do czynienia z pojedynczymi projektami. Na szczycie siedzi sobie ktoś, czyje stanowisko ma szumną nazwę w stylu „VP of architecture" – to osoba dużo mniej techniczna, skupia się raczej na zadaniach organizacyjnych, zarządzaniem budżetami i rozwojem (ego) swoich pracowników. Te role nie są wykute w kamieniu, architekt może się spodziewać, że będzie poproszony/a o zakładanie różnych kapeluszy, w zależności od bieżących potrzeb organizacji. No i oczywiście to są tylko nazwy, mogą być różne, a konkretny zakres obowiązków jest specyficzny dla danej struktury.

Skoro więc architekci kodują mniej, niewiele albo w ogóle, to czy to nie jest ktoś taki jak menadżer projektu – niby wciąż IT, ale już nie inżynier? Czy to nie jest nietechniczna ścieżka kariery? Odpowiedź na to pytanie jest jednoznaczna: nie. Z założenia architekt ma mieć szerokie umiejętności techniczne. Dobrzy architekci są (czy też byli) z reguły dobrymi developerami – ludzie, którzy projektują software, muszą rozumieć technologię, bo każda ich decyzja musi być wynikiem kompromisu. Muszą wiedzieć, czy to, co budują, będzie działać. Muszą też umieć identyfikować i mitygować ryzyka techniczne – zapewnić, że architektura „działa", dbać o jakość, wprowadzać i pilnować standardów, zasad, pryncypiów. Tworzyć strategię techniczną, wizję, mapę drogową. Ale architekt to coś więcej niż „osoba techniczna". Kiedy w domu szwankuje kanalizacja, nie wołamy architekta, tylko technika – hydraulika. Tak samo jest z software'm – kiedy program nie działa, naprawiają go w pierwszej kolejności programiści. Rola architekta jest nie tylko techniczna – oczekuje się znajomości domeny biznesowej, komunikacji w górę i w dół organizacji oraz bycia liderem. To rola dyktuje, w jakim stopniu architekt jest techniczny. Taki już nasz los – musimy mieć i dbać o umiejętności miękkie.

O SOFTSKILLACH SŁÓW PARĘ (SETEK)

Pierwsza z umiejętności to przywództwo. Niestety, zwykle nie sprowadza się to do mówienia ludziom, żeby ciężko pracowali, nie wykonując samemu żadnej rzeczywistej roboty. Architekt ma być liderem – technicznym – przez cały czas życia oprogramowania. Katalizatorem zmian, który „sprzedaje" nowe pomysły. Każdy zespół potrzebuje lidera, by można było wprowadzić kontrolę w miejsce chaosu. Oczywiście powstaje pytanie, ile potrzebujemy kontroli, bo różne zespoły oczekują różnych stylów przywództwa. Znalezienie balansu między ograniczeniem a wskazywaniem drogi nie jest łatwe.

Na pewno jako architekt będziesz się musiał/a komunikować więcej i niekończenie będziesz to lubić. Komunikacja jest absolutnie kluczowa. Na początku trzeba wybrać odpowiedni kanał – mail, chat, telefon, spotkanie itp. To odbiorca dyktuje metodę komunikacji – to, że twoim ulubionym sposobem kontaktu są gołębie pocztowe, niekoniecznie musi oznaczać, że wszyscy je kochają. Czasami warto mieć ślad – zapis, nagranie, minutki, czasami nie. Gramatyka ma znaczenie, ton ma znaczenie, trzeba bardzo uważać na żarty. W środowisku międzynarodowym jest trudniej. Z jednej strony mało kogo obchodzi, że angielski nie jest twoim językiem ojczystym. Z drugiej – nikogo nie obchodzi, że znasz go doskonale. Masz się ograniczyć do 1500 słów i trzech czasów na krzyż. Musisz pamiętać o takich rzeczach, jak różnice międzykulturowe, strefy czasowe, język neutralny płciowo. Po prostu pole minowe. W skrócie: albo pogodzisz się i dostosujesz do kultury firmy, albo sczeźniesz.

Komunikacja to także spotkania, zwane też po staropolsku *mitingami*, a mitingi to kolejne pole minowe. Są bardzo kosztowne w organizacji – co prawda pandemia nauczyła nas, że podróże są w zasadzie zbędne, ale wideokonferencje też potrafią kosztować krocie, jeśli zsumujemy stawki godzinowe uczestników. Gapienie się godzinami w twarze wszystkich obecnych naraz, i do tego w swoją, odbitą w lustrze, powoduje zmęczenie, które nawet dorobiło się naukowej nazwy – *zoom fatigue*. Czasem chcemy spotkania nagrywać, żeby był dowód, czasem nie chcemy, żeby mieć możliwość wiarygodnego zaprzeczenia – nie wiem, nie pamiętam, nie było mnie tam. Czasem nie mamy wyjścia. Jako architekt musisz chodzić na wiele spotkań, o wiele więcej niż kiedykolwiek wcześniej, dwa-trzy w tym samym slocie w kalendarzu nie są niczym osobliwym. Jeśli twój kalendarz nie wygląda jak Tetris, to coś źle robisz. Będziesz prosić swojego przełożonego o priorytety, ale czasem wszystko jest priorytetem. Możesz poprosić o plan spotkania, agendę, cel i nie chodzić na te, które go nie mają. Chyba że nie możesz. Wolno ci zapytać „po co mnie potrzebujecie", chyba że nie wolno. O roli architekta na mitingach za chwilę, na razie, w wielkim skrócie: to wszystko jest całkowicie normalne, mitingi są dla ludzi, nie dla ciebie. Przygotuj się, odrób zadanie domowe, bądź obecny, nie rozpraszaj się. Eufemizmem na „nie uważałem/am" jest „byłem/am na mute". Uważaj na stan mute. Potrzebujesz dużej, czerwonej lampki, która daje opartą na hardware pewność, że cię nie słychać. Zwłaszcza kiedy podczas telekonferencji naradzasz się z lokalnymi uczestnikami. Lepiej zawsze zakładać, że mute nie działa. Komunikacja równoległa z uczestnikami spotkania jest bardzo ważna, można namawiać do zadawania „dobrych" pytań, grać w dobrego i złego architekta i generalnie sterować przebiegiem z tylnego siedzenia. Trzeba czytać atmosferę spotkania, rozumieć strukturę władzy. Szczególnie należy uważać na tych, którzy po prostu siedzą cicho. Być może robią tylko sztuczny tłok, ale mogą też pełnić służbę na sonarze.

Służba na sonarze to jedna z trzech najczęstszych roli architekta na mitingu. Jak sama nazwa wskazuje, prowadzi się wtedy nasłuch i wskazane jest zachowanie ciszy. Czasami kończy się na tym – na koniec spotkania rzucamy zwyczajowe „na razie". Czasem, po namierzeniu celu, trzeba odpalić torpedę. Druga rola to kocyk bezpieczeństwa – „na tym spotkaniu potrzebujemy architekta", nie po coś konkretnego, tylko żeby był. Na koniec trzeba powiedzieć, że na ten moment nie masz uwag. Właściwe rozpoznanie typu spotkania pozwala na nadganianie innej roboty w trakcie. Niewłaściwe kończy się „byłem na mute". Ostatnia rola to prowadzenie spotkań.

Prowadzenie spotkań to między innymi prezentowanie, bo architektura to opowieść. Opowiadamy opowieść developerom, żeby mogli ją pojąć, pokochać, zaimplementować i przekazać dalej. Opowiadamy ją „górze" organizacji, żeby nasze pomysły dostały finansowanie. Twoje pomysły, choćby były genialne, nic nie znaczą, jeśli nie potrafisz ich sprzedać. Przygotuj się na prezentację, która będzie trwała 20, 10 i 5 minut (bo ktoś zamarudził na spotkaniu i kończy się czas). Miej w zanadrzu *elevator pitch* – coś, co spróbujesz wcisnąć Bardzo Ważnej Osobie spotkanej w windzie na 30 sekund. Musisz wiedzieć, gdzie w twojej organizacji są influencerzy i nauczyć się na nich wpływać. To, co i w jaki sposób prezentujesz, ma ogromne znaczenie. Niektóre elementy są oczywiste – np. trzeba zapomnieć o RTFM, bo z reguły będziesz prezentować tą samą rzecz wiele razy *ad nauseam*; albo należy porzucić techniczny żargon – wprowadza tylko niepotrzebne bariery. Część rzeczy jest jednak trochę wbrew intuicji. Na przykład styl prezentacji powszechny w twojej firmie może nie być tym, którego cię nauczono, może być niedobry, zły, beznadziejny, brzydki – ale będzie lepiej dla ciebie i twoich pomysłów, jeśli o tym zapomnisz i popsujesz (albo wręcz ogłupisz) swoją prezentację, bo „tak to tutaj robimy".

Przy prezentacji musisz używać wiarygodnych źródeł, ale też polegać na wiarygodnych sojusznikach. Każdy interesariusz to potencjalny sojusznik. Musisz mieć świadomość nie tylko ich istnienia, ale

także tego, jak cię postrzegają, co mówią, kiedy cię nie ma. Brzmi to banalnie, ale to twoja reputacja przemawia za ciebie pod twoją nieobecność. Interesariusze powinni wiedzieć, co dla nich robisz i że jest to dobre, ale pamiętaj, że generałów nie interesują poczynania sierżantów. Jak sama nazwa wskazuje, interesariusze mają interesy, walczą o władzę, żeby je realizować. Stąpaj ostrożnie, bo dzisiejszy wiceprezes ds. czegoś tam wczoraj mógł być architektem, który podjął decyzję, która dzisiaj boli, ale wczoraj mogła mieć racjonalne oparcie. To, co tobie wydaje się być błędem czy problemem, może wyglądać inaczej z innego punktu widzenia. Spędzanie czasu na tego typu przemyśleniach wpędzi cię w chorobę. Technologia jest prosta, ludzie niekoniecznie, a kultura organizacyjna na pewno nie. A'propos kultury – co jest ważne i cenione w twojej firmie? Stabilność? Innowacyjność? Koszt? Czy przypadkiem ty nie jesteś kosztem?

Relacje międzyludzkie są ważne dla twojej pracy: kogo znasz, do kogo możesz dotrzeć, kogo możesz poprosić o pomoc, kto zna ciebie. Twoja sieć powiązań wymaga kultywacji – oczywistości rodem z podręcznika „jak zjednywać sobie ludzi" (daj im odczuć swoje zainteresowanie, niech czują się docenieni, odnoś się do nich z szacunkiem i uprzejmością itd.). Nie wahaj się prosić o pomoc, ale też oferują ją – żyjemy, by służyć innym, zgłaszaj się na ochotnika, dbaj o widoczność swojej pracy, ale też przyznawaj się do błędów.

Trzy rady na koniec. Po pierwsze, wybieraj bitwy, które chcesz toczyć. Jeśli chcesz umierać za każdy pagórek i każdą piędź ziemi, prawdopodobnie tracisz czas. Po drugie, percepcja jest wszystkim – nie licz wyłącznie na siłę swoich pomysłów, ale również swojej reputacji. Po trzecie, dostosuj się do kultury organizacji. Nie zmienisz jej, to nie jest coś, co robią architekci.

I najważniejsze: architekt nie powinien się bać powiedzieć „nie", „nie wiem" i „to był mój błąd".

JAK BYĆ ARCHITEKTEM (I PO CO)?

Technologia zmienia się bezustannie. Nikt nie wie, co będzie na topie jutro, ale na pewno coś innego niż jest dziś. Chcemy mieć nowe rzeczy i chcemy się ich uczyć, ale nie co dwa miesiące i nie co roku – a decyzje architektoniczne mają tendencję nabierać znaczenia po latach. Boimy się nowych rzeczy, bo siedzenie na krwawiącym ostrzu technologii oznacza, że będziemy krwawić. Jeszcze bardziej boimy się uczyć i używać rzeczy starych. Nowe rzeczy mają tendencję do psucia rzeczy starych. Bycie pionierem nie jest łatwe. Pionierów łatwo rozpoznać po strzałach wystających im z pleców.

Jak więc rozpoznać, co z nowych rzeczy jest dla nas dobre i przydatne? Musimy cały czas eksperymentować i próbować świeżych rozwiązań, pamiętając, że w pewnym momencie trzeba będzie się na coś zdecydować. Przypominam Pierwsze Prawo Architektury Oprogramowania: każda decyzja zawiera jakiś kompromis – jeśli myślisz, że gdzieś go nie ma, to znaczy, że analiza problemu była zbyt pobieżna. Czasem kompromisy wydają się nam paskudne, ale nie po to pracujemy w przemyśle IT, by robić rzeczy piękne. Rzeczy piękne robią artyści, my jesteśmy rzemieślnikami i robimy rzeczy, które się sprzedadzą klientom. Przy podejmowaniu decyzji ważna jest strategia, a nie nadzieja. Nadzieja jest dobra przy rewolucjach i rebeliach. Rebeliantów łatwo rozpoznać po strzałach wystających im zewsząd.

Architekt musi być na bieżąco z tym, co się dzieje w jego dziedzinie. Wybór metody jest sprawą indywidualną – jedni czytają blogi, inni śledzą sieci społecznościowe, jeszcze inni słuchają podcastów. Niezależnie od tego, co działa w twoim przypadku, musisz wyrobić sobie nawyk – raz w tygodniu, góra raz w miesiącu poświęć kilka godzin na sprawdzenie, co nowego wymyślono. Zapomnisz mnóstwo rzeczy – musisz mieć fantastyczne archiwum, notatki, mindmapy – cokolwiek, co u ciebie działa. Ja na przykład jestem starym człowiekiem, piszę piórem i zapisuję nim jeden notes Moleskine formatu L rocznie. Twoja uwaga jest trochę jak ziemia – nie robią jej więcej i jej wartość bezustannie wzrasta. Musisz więc uprawiać brutalną selekcję wszystkiego, a i tak prawie wszystko cię ominie. Oszczędzaj swoją uwagę jak tylko się da.

Zostając architektem, nadal będziesz rozwijać swoje kompetencje techniczne, ale już nie tak bardzo i nie tak szybko jak wcześniej. Skoncentrujesz się bardziej na wywieraniu wpływu, promowaniu nowych idei i przewodzeniu większym zespołom w większych projektach. To niekoniecznie jest to, czego młodzi adepci sztuki architektowania spodziewają się, podejmując wyzwanie – praca okazuje się dużo mniej inżynierska niż myśleli. Ale to świetna robota. Powodzenia.

Bibliografia

[1] Std 1471-2000 – IEEE Recommended Practice for Architectural Description for Software-Intensive Systems https://ieeexplore.ieee.org/document/875998
[2] Len Bass, Paul Clements, Rick Kazman: „Software Architecture in Practice", Addison-Wesley Professional 2012
[3] Paul Clements et al.: „Documenting Software Architectures: Views and Beyond", Addison-Wesley Professional 2010
[4] Mark Richars, Neal Ford: „Fundamentals of Software Architecture: An Engineering Approach", O'Reilly Media 2020
[5] https://agilemanifesto.org/iso/pl/principles.html
[6] https://www.researchgate.net/publication/224643527_What_lessons_can_the_agile_community_learn_from_a_Maverick_fighter_pilot
[7] George H. Fairbanks, „Just Enough Software Architecture: A Risk-Driven Approach", Marshall & Brainerd 2010
[8] https://en.wikipedia.org/wiki/List_of_system_quality_attributes
[9] Robert C. Martin: „Clean Code: A Handbook of Agile Software Craftsmanship", Pearson 2008

MACIEJ NORBERCIAK

maciej.norberciak@nokia.com

Od początku kariery w przemyśle zajmuje się systemami wbudowanymi w telekomunikacji, od ośmiu lat jako architekt w Nokii. Doktor nauk technicznych w dziedzinie sztucznej inteligencji. Mentor, mówca, trener. W wolnych chwilach lubi się uczyć nowych rzeczy, głównie języków obcych, i jeździć na rowerze.

KRZYWE BÉZIERA | HISTORIA JEDNEGO REFAKTORA | DESERIALIZACJA W PHP

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

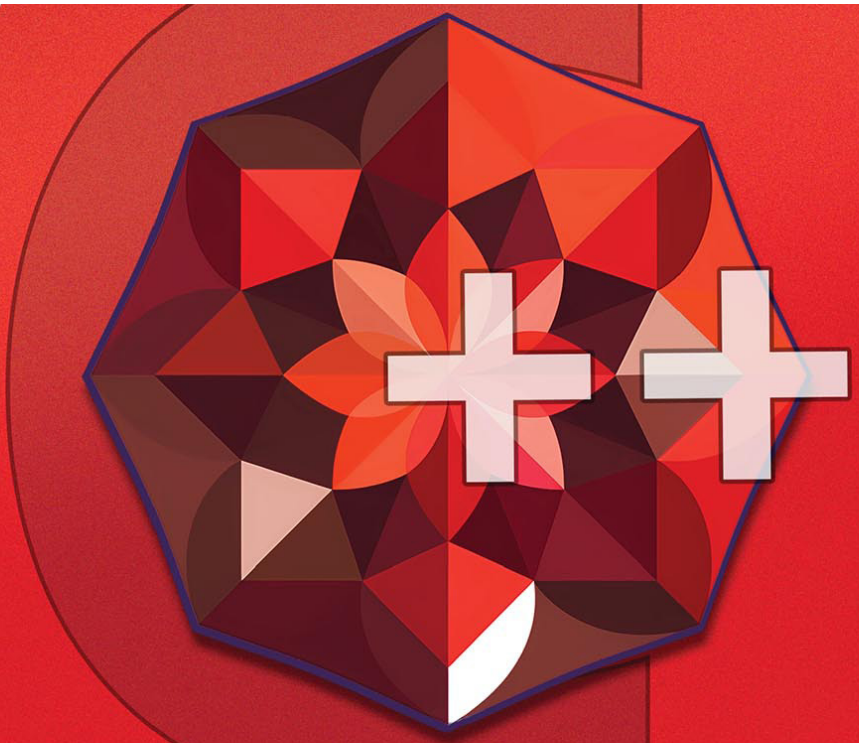
programista

3/2023 (108)

Cena 28,90 zł (w tym VAT 8%)

WYWOŁYWANIE KODU NATYWNEGO W C++ Z JĘZYKA RUBY

Aktualny numer już w sprzedaży



ISSN 2084-9400

03

9 772084 940305

Gdzie te dane? O zachowaniu spójności z Transactional Outbox Pattern

GPU Audio – od teorii do praktyki

MySQL Shell plugin dla Visual Studio Code

Jak AutoML zmienia sposób postrzegania uczenia maszynowego?