1

Wprowadzenie do przywództwa w oprogramowaniu

Podczas tworzenia systemów oprogramowania naszym celem jest budowa systemów, które spełniają standardy jakości i zapewniają najwyższy zwrot z inwestycji (*return on investment*, ROI) w dłuższej perspektywie lub w określonym horyzoncie czasowym. To z kolei staje się celem architektury oprogramowania, która jest planem budowy systemów oprogramowania.

Tutaj ROI nie polega wyłącznie na byciu opłacalnym. Jeśli większe wydatki na produkt skutkują większymi przychodami, należy to uznać za dobry zwrot z inwestycji. Z drugiej strony, kiepski projekt wymaga później licznych zmian, przez co ostatecznie kosztuje znacznie więcej. Dobra architektura oprogramowania równoważy obie skrajności i maksymalizuje ROI.

Projekt architektoniczny obejmuje wiele rzeczy, na przykład znajdywanie odpowiednich abstrakcji, wybieranie funkcji do uwzględnienia, określanie zakresu każdej funkcjonalności, ustawianie parametrów jakości usługi (quality of service, QoS) czy ustalanie stopnia elastyczności, czasu i doświadczenia użytkownika.

Rola oceny sytuacji

Jako architekci oprogramowania uczymy się o abstrakcjach, stylach architektury i wzorcach. Badamy ich wady i zalety, stosowanie w konkretnych sytuacjach i sposoby ich komponowania ze świadomością problemów, negatywnych przykładów i przypadków użycia. Jednak wiele błędów popełnianych jest nie dlatego, że nie rozumiemy tych rzeczy. Większość błędów projektowych powstaje z braku oceny sytuacji, a nie z braku wiedzy.

W tym przypadku *ocena sytuacji* odnosi się do zdolności podejmowania przemyślanych decyzji lub wyciągania rozsądnych wniosków przy optymalizowaniu pod kątem najważniejszego rezultatu.

W ciągu 20 lat mojej pracy architekta systemów widziałem ten rezultat wielokrotnie. Oto typowe błędy, które znalazłem:

- Próba uwzględnienia zbyt wielu funkcji wymaganych przez użytkownika
- Uczynienie projektu zbyt elastycznym lub zbyt sztywnym, co wpływa na przyszłe zmiany
- Ograniczanie zakresu funkcjonalności, co znacząco wpływa na doświadczenia użytkownika (user experience, UX)
- Rozwiązywanie problemów bezużytecznych dla użytkownika końcowego
- Niewystarczające skupianie się na doświadczeniach użytkownika
- Mijanie się z harmonogramami dostaw

Większość z tych błędów popełniamy, ponieważ nie mamy wiedzy o przyszłości i o użytkownikach korzystających z systemu. Nie wiemy też, jak system będzie działał na granicy swoich możliwości. Tutaj dostrzegam potrzebę oceny sytuacji. Widzę wyzwania związane z przywództwem, a nie wyzwania techniczne!

Zastanówmy się, co przez to rozumiem.

Dla mnie przywództwo polega na zarządzaniu niepewnością, na porządkowaniu chaosu, dawaniu nadziei na lepszą przyszłość i dążeniu w kierunku tej przyszłości. Rozważmy następujący cytat:

Lider jest dilerem nadziei. – Napoleon Bonaparte

Nie oznacza to, że liderzy muszą być wszechwiedzący i zawsze wiedzieć, co przyniesie przyszłość, ale powinni mieć *wizję* przyszłości, powinni zarządzać niepewnością w sposób minimalizujący ryzyko. Liderzy powinni komunikować innym swoją wizję i jej realizację oraz prowadzić ich w kierunku tej wizji.

To samo stwierdzenie powtórzę z punktu widzenia architekta. Nie chodzi o to, że architekci oprogramowania muszą być wszechwiedzący i zawsze wiedzieć, jak system będzie używany i co powinien zawierać, ale powinni mieć wizję dotyczącą kompletnego rozwiązania. Powinni zarządzać niepewnością w sposób, który minimalizuje ryzyko. Liderzy powinni komunikować zespołowi swoją wizję i jej realizację oraz prowadzić go w kierunku budowy systemu i jego obsługi.

Nie twierdzę, że dla architekta wiedza nie jest ważna, gdyż jest. Jednak ocena sytuacji również odgrywa tu kluczową rolę. Niestety, o ile wiedza jest powszechna, o tyle ocena sytuacji już nie.

Widziałem wiele dobrych książek i artykułów na temat architektury oprogramowania: książki Boba Martina, książki Gregora Hohpe czy blogi Martina Fowlera to tylko niektóre z nich. Jednak skupiają się one głównie na wiedzy, a nieco mniej na ocenie sytuacji.

Widziałem również wiele dobrych książek na temat przywództwa: *The Hard Things About Hard Things* autorstwa Bena Horowitza, *Trillion Dollar Coach* autorstwa Erica Schmidta et al., *Team of Teams: New Rules of Engagement for a Complex World* autorstwa Stanleya McChrystala, *Good Strategy, Bad Strategy* autorstwa Richarda Rumelta czy chociażby książki Jocko Willinka[1]. Omawiają one ocenę sytuacji, ale tylko na poziomie ogólnym, a nie na poziomie technicznym. Istnieje luka między dobrym przywództwem i dobrą architektoniczną oceną oprogramowania.

Cel tej książki

Ta książka omawia lukę między przywództwem i architektoniczną oceną oprogramowania. Opisuje ona przywództwo w oprogramowaniu i przedstawia najlepszy sposób jego wykorzystania w czasie budowy naszych systemów. Jak wspomniałem, moje doświadczenie pokazuje, że wiele z naszych błędów architektonicznych wynika z luki między wiedzą a oceną sytuacji.

To nie jest książka o tym, jak należy zarządzać zespołem. Nie jest to również książka o zarządzaniu inżynierskim czy o zasobach ludzkich i sposobie budowania zespołu. Nie jest to także książka o strategii. Co więcej, ta książka nie opisuje sposobu tworzenia wizji. Musimy mieć już pewną wizję – naszą własną, bądź też podzielaną przez naszych współzałożycieli lub członków zarządu. Choć na temat wizji napisano już wiele książek, nie jestem pewien, czy da się ją wystarczająco wyjaśnić.

To jest książka techniczna, więc dotyczy oceny technicznej. Wyjaśnia ona zasady i pojęcia, które moim zdaniem starszy architekt musi dogłębnie zrozumieć, a także wyjaśnia sposób stosowania tych zasad do zarządzania niepewnością. Jest to książka o tym,

1 Ben Horowitz, *Najtrudniejsze w tym, co trudne* (Helion, *2016*); Eric Schmidt et at., *Coach wart bilion dolarów* (Onepress, *2020*); Stanley McChrystal, *Zespół zespołów: Nowe zasady podejmowania działań w złożonym świecie* (OSMpower, *2023*); Richard P. Rumelt, *Dobra strategia, zła strategia* (MT Biznes, *2013*)

jak liderzy techniczni/architekci powinni myśleć i jak powinni nadzorować swój produkt poprzez zarządzanie niepewnością.

Przykładowo jedna z tez tej książki zakłada, że należy głęboko rozważać, ale powoli implementować. Innym przykładem jest to, że liderzy muszą zdefiniować zakres, biorąc na siebie ciężar niepewności bez przekazywania go współpracownikom. Pytania i zasady omawiane w tej książce pomagają nam zarządzać niepewnością i zapewniają pewne ramy do podejmowania decyzji.

Czy ta książka będzie przydatna dla kogoś, kto nie dowodzi? Myślę, że tak. Ludzie podążają za każdym, kto radzi sobie z niepewnością w celu zapewnienia dalszego rozwoju. Dobrzy architekci zaczynają odgrywać tę rolę na wiele lat przed objęciem tego stanowiska. Im większą mamy wiedzę, tym większa będzie nasza szansa, gdy zdecydujemy się przewodzić. Przejmijmy inicjatywę, pomóżmy naszemu liderowi i spełnijmy oczekiwania. Zaczniemy dostrzegać, że zdobywamy coraz więcej. Stanowiska pojawią się wkrótce.

Jeśli uważamy, że ktoś odgrywa tę rolę lepiej od nas, obserwujmy go, wypytujmy i uczmy się od niego. W takim przypadku, korzystając z tego, co omawiane jest w tej książce, możemy na wiele sposobów pomóc takiemu liderowi. Wkrótce nadejdzie nasza kolej.

Książka czerpie przykłady z kilku wzorów do naśladowania, które obrazują przywództwo techniczne. W szczególności wymieniam dwa: Kelly Johnson, który zaprojektował samoloty U-*2* i Blackbird SR-*71*, oraz znani wszystkim bracia Orville i Wilbur Wright. Ci liderzy zaprezentowali pewnego rodzaju dogłębną kontrolę techniczną, która umożliwiła stworzenie systemów uznawanych za niemożliwe, i to przy wykorzystaniu ograniczonych zasobów. Owszem, wielu jest liderów w oprogramowaniu, jak choćby Jeff Dean z Google, którego darzę podobnym szacunkiem, ale są oni współcześni, a ich metody nie zostały jeszcze opublikowane w formie książek. Ten fakt zmusza mnie do wyboru wcześniej wspomnianych wzorów dla tej książki.

Jak wszyscy wiemy, bracia Wright zbudowali pierwszy napędzany samolot pozwalający na ciągły i kontrolowany lot. Nie mieli wyższego wykształcenia i byli właścicielami sklepu rowerowego. Rywalizowali z dobrze finansowanymi profesjonalistami, a mimo to udało im się wygrać. Owszem, przed braćmi Wright istniało wiele projektów samolotów, ale to oni jako pierwsi dobrali prawidłowo wszystkie parametry dla projektu. Bracia Wright wykazali się świetną zdolnością oceny sytuacji budując szybowiec, ucząc się nim sterować, ucząc się go usprawniać, a następnie dodając do niego śmigła i silniki, które stopniowo przekształciły ich szybowiec w samolot. Ustalenie właściwych parametrów

to lekcja, z której wszyscy możemy skorzystać. Bracia Wright mieli intuicyjne wyczucie projektu, które nabywali przez lata. Później osiągnęli zarówno sławę, jak i fortunę.

Kelly Johnson był projektantem samolotów Lockheed U-*2*, SR-*71* Blackbird oraz 40 innych modeli. Blackbird jest niewidoczny dla radaru, jest w stanie wyprzedzać pociski, a w czasie swojej ponad 20-letniej służby nigdy nie został zestrzelony. Był to pierwszy samolot produkcyjny, który przekroczył prędkość Mach 3 (trzykrotność prędkości dźwięku). Kelly zbudował także pierwszy myśliwiec zdolny do rozwinięcia prędkości Mach 2 i pierwszy myśliwiec, który przekroczył prędkość 644 kilometrów na godzinę. Jego samolot U-*2* osiągał i utrzymywał pułap 20 000 metrów[2]. Kelly potrafił postawić sobie niemożliwy do osiągnięcia cel, podzielić go na wykonalne zadania, wymagać doskonałości, a potem sprawić, że to wszystko zadziała. Był znany z tego, że kończył projekty wcześniej i poniżej ustalonego budżetu, zwracając rządowi pieniądze. Jego szef ponoć rzekł kiedyś: „Ten przeklęty Szwed naprawdę widzi powietrze", odnosząc się w ten sposób do intuicyjnego wyczucia projektu przez Kelly'ego[3].

Pojęcia architektura i projekt są często używane zamiennie, przy czym projekt to pełny i szczegółowy plan (np. diagramy klas i diagramy sekwencji), zaś architektura to ogólny widok koncepcyjny (np. widok komponentów i diagram sekwencji na poziomie komponentu). My skupiamy się na widoku ogólnym, dlatego też w książce posługujemy się terminem architektura.

Do wskazania tematów, na których skupiamy się w tej książce, możemy wykorzystać trzy warstwy architektury ze szkieletu The Open Group Architecture Framework (TOGAF):

- Warstwa architektury biznesowej szkicuje operacje biznesowe i pokazuje, jak różne komponenty współpracują ze sobą, aby napędzać działanie biznesu.
- Warstwa architektury systemów informacyjnych dzieli się na dwie części: architekturę danych i architekturę aplikacji. Architektura danych koncentruje się na kategoryzowaniu różnych typów danych i podkreślaniu ich powiązań. Z kolei architektura aplikacji identyfikuje unikalne części systemu, takie jak usługi, i wyjaśnia ich interakcję w obrębie systemu.
- Warstwa architektury technologicznej opisuje konkretne wybrane technologie. Obejmują one takie elementy, jak standardy oprogramowania, używane pakiety oprogramowania, sprzęt, sieci i drobne szczegóły dotyczące bezpieczeństwa.

2 *http://pl.wikipedia.org/wiki/Lockheed_U-2*
3 *http://pl.wikipedia.org/wiki/Clarence_Johnson*

Głównym tematem tej książki jest architektura systemów informacyjnych. Niemniej jednak w niektórych miejscach możemy omawiać architekturę technologiczną, gdy wybór technologii wpływa znacząco na naszą dyskusję.

Związek pomiędzy architekturą biznesową i architekturą systemów informacyjnych jest bardziej złożony. Projekt architektury systemów informacyjnych uzależniony jest w dużej mierze od różnych czynników biznesowych. Te czynniki obejmują nie tylko architekturę biznesową, ale także inne elementy, takie jak harmonogramy projektów, umiejętności zespołu i wyzwania ze strony konkurentów. Chociaż w architekturze biznesowej w takich szkieletach jak TOGAF te czynniki nie są zwykle uwzględniane, wpływają one na implementację architektury i strategiczny kurs organizacji. Razem określa się je mianem „kontekstu biznesowego". Głównym wyzwaniem w architekturze systemów jest uwzględnienie tego kontekstu biznesowego podczas podejmowania decyzji technologicznych. Zapewnienie tego jest odpowiedzialnością, która spoczywa na przywództwie. Kluczowym celem pięciu pytań omawianych w tej książce jest upewnienie się, że utrzymujemy ten kontekst biznesowy.

Istnieją dwa główne podejścia do architektury systemu:

- Podejście kaskadowe
- Podejście zwinne

Podejście kaskadowe (waterfall) opiera się na założeniu, że możliwe jest wcześniejsze szczegółowe określenie pełnych wymagań systemu. Tym samym podejście to sugeruje dokładne zaplanowanie, po którym następuje wykonanie. Przykład takiego podejścia możemy zobaczyć w modelu projektowania architektury (architecture design model, ADM) w TOGAF, który pokazuje, jak możemy dokładnie uchwycić wymagania i opracować je. Ponadto takie grupy jak Object Management Group (OMG) i International Organization for Standardization (ISO) oferują standardy wspierające podobne modele koncepcyjne.

Z kolei podejścia iteracyjne, takie jak podejście zwinne (agile), koncentrują się na szybkim wydawaniu wersji i współpracy z użytkownikami w celu udoskonalenia wymagań i skonstruowania systemu, który może przynieść użytkownikowi rzeczywiste korzyści.

Porównując ze sobą te dwa podejścia skłaniam się ku podejściu zwinnemu. Podjęto wysiłki, aby połączyć funkcje iteracyjne z modelami, takimi jak TOGAF ADM, ale w praktyce utrzymanie szybkiego tempa wymaganego w przypadku modeli iteracyjnych często okazuje się zbyt skomplikowane (zwykle iteracje trwają od jednego do dwóch tygodni). W przypadku dużych organizacji i złożonych projektów bardziej centralne

planowanie mogłoby być uzasadnione, przy czym nawet mając za sobą współpracę z setkami przedsiębiorstw, w tym z firmami z listy Fortune 500, nie widziałem jeszcze, aby taki projekt był w stanie dostarczyć wyjątkowych rezultatów.

Wiele procesów oprogramowania (takich jak TOGAF ADM), standardów i architektur referencyjnych opiera się na modelu kaskadowym i ma na celu precyzyjne uchwycenie wymagań. Chociaż z TOGAF, OMG i ISO można wyciągnąć cenne wnioski, działają one przy założeniu, że wymagania można w dużej mierze zdefiniować wcześniej i będą one ulegać jedynie stopniowym zmianom. Jednak moim zdaniem nasze doświadczenia pokazały, że jest inaczej. Popieram bardziej interaktywne lub zwinne podejście, w którym wymagania są prostsze i nieformalne oraz stale ulepszane w krótkich iteracjach przy wykorzystaniu informacji pozyskanych od użytkowników.

Na szerszym poziomie projektowania preferuję podział systemu na luźno powiązane podsystemy (każdy potencjalnie wchodzący w interakcję z użytkownikami i dostarczający jakąś wartość), definiowanie API między nimi, a następnie niezależne operowanie nimi z nadzorem ogólnym w celu połączenia kropek.

Chociaż architekci mogą odnaleźć korzyści w TOGAF i podobnych modelach, waham się, czy polecić je w tej książce. Jest to opinia oparta na moich doświadczeniach.

W tej książce skupiamy się na podejściach zwinnych. Zanim jednak zagłębimy się w jej treść, warto zapoznać się z typowymi rolami w projekcie oprogramowania. Menedżerowie produktu, z pomocą interesariuszy biznesowych, projektantów UX i architektów, decydują o tym, co należy zbudować. Architekci współpracują z kierownikami technicznymi i zespołem w celu zbudowania produktu. Z kolei menedżer produktu współpracuje ze wszystkimi, aby zapewnić wymaganą jakość (wymagając doskonałości).

Zakres roli architekta może się zmieniać w zależności od miejsca wykonywania pracy. Na przykład, w startupie architekci mogą zajmować się zarządzaniem produktem i decydować o tym, jakie funkcje mają zostać zbudowane, podczas gdy w dużej firmie architekci mogą być odcięci od specyfikacji wymagań. Jednak w erze, w której odchodzimy od podejścia kaskadowego na rzecz iteracyjnego, zwinnego podejścia do tworzenia oprogramowania, obowiązki są współdzielone, a role te łączą się ze sobą. Przykładowo uważam, że architekci powinni współpracować z menedżerami produktu przy podejmowaniu decyzji o tym, które funkcje uwzględnić i kiedy należy to zrobić, a także przy definiowaniu UX, wymagając doskonałości od zespołu.

Ta książka podzielona jest na cztery części. Choć skupia się ona na ocenie sytuacji, wiedza jest równie ważna. Części II i III zagłębiają się w tematy dotyczące wiedzy, jednocześnie badając, jak możemy ocenić jej wykorzystanie.

Część I: Wprowadzenie

W części I rozdział 2. omawia architekturę oprogramowania, niepewność i ocenę sytuacji. Identyfikuje on pięć pytań i siedem zasad, które pomagają nam radzić sobie z niepewnością.

Tych pięć pytań to:

- Kiedy jest najlepszy czas na wprowadzenie produktu na rynek?
- Jaki jest poziom umiejętności zespołu?
- Jaka jest wrażliwość naszego systemu na wydajność?
- Kiedy możemy przepisać system?
- Jakie są trudne problemy?

Do wspomnianych siedmiu zasad należą:

- Opieraj wszystko na doświadczeniach użytkownika
- Używaj iteracyjnej strategii cienkich plastrów
- W każdej iteracji dodawaj największą wartość przy najmniejszym wysiłku w celu wsparcia większej liczby użytkowników
- Podejmuj decyzje i absorbuj ryzyko
- Projektuj dogłębnie rzeczy, które trudno zmienić, ale implementuj je powoli
- Wyeliminuj niewiadome i wyciągaj wnioski z dowodów, pracując nad trudnymi problemami wcześnie i równolegle
- Poznaj kompromisy między spójnością i elastycznością w architekturze oprogramowania

Jak zobaczymy w rozdziale 2., tych pięć pytań i siedem zasad dotyczy każdego z typowych błędów architektonicznych, które często popełniamy.

Część II: Podstawowe informacje

W części II zagłębiamy się w dwa obszary związane z wydajnością i UX, które moim zdaniem nie są zbyt dobrze rozumiane przez wielu architektów. Pierwszym z nich jest wydajność systemu, która decyduje o tym, co jest, a co nie jest wykonalne w naszych architekturach. Drugim jest UX, który często decyduje o przyjęciu systemu przez użytkowników, a tym samym o dalszym losie systemu.

Rozdział 3. jest bardziej szczegółowy i techniczny w porównaniu do pozostałych rozdziałów. Te szczegóły są kluczowe i nie są szeroko omawiane w innych książkach. Jeśli ktoś początkowo poszukuje jedynie szerszego zrozumienia, może pokrótce przejrzeć rozdział 3., ale zalecam ponowne zapoznanie się z nim, aby ostatecznie uchwycić wszelkie subtelniejsze punkty.

Rozdział 4. podkreśla znaczenie zasad UX i przekonuje nas, abyśmy już na wczesnym etapie umieścili w zespole ekspertów z zakresu UX i słuchali udzielanych przez nich rad. Chcę dodatkowo podkreślić znaczenie UX dla API, konfiguracji i rozszerzeń.

Część III: Projekt systemu

W części III skupiamy się na sposobach budowania systemu lub aplikacji. Omawiamy dwa poziomy: poziom makro, gdzie łączymy usługi w jedną spójną architekturę, oraz poziom mikro, gdzie uczymy się budować dobre usługi.

W tej części, tam gdzie to możliwe, wyjaśniamy domyślny wybór architektoniczny, który sprawdza się w większości przypadków, ale też bardziej złożone wybory, omawiając przy tym sposób wyboru właściwej architektury dla naszej firmy. Ta dyskusja obejmuje antywzorce i często popełniane błędy. W tej części mówimy o następujących ideach technicznych, które są moim zdaniem kluczowe:

- Makroarchitekturę opisano w:

  Rozdziale 5: Wprowadzenie

  Rozdziale 6: Koordynacja

  Rozdziale 7: Zachowywanie spójności stanu

  Rozdziale 8: Obsługiwanie bezpieczeństwa

  Rozdziale 9: Obsługiwanie wysokiej dostępności i skali

  Rozdziale 10: rozważania na temat mikrousług

- Tworzenie dobrej, użytecznej usługi opisuje rozdział 11.
- Budowanie stabilnych systemów opisuje rozdział 12.

Rozważaniom dotyczącym mikrousług poświęciłem osobny rozdział zamiast rozpraszać je w rozdziałach 6, *7* i 8. Uważam, że takie podejście pozwoli łatwiej uchwycić tę koncepcję w formie połączonej całości niż w przypadku prezentowania tych części osobno.

Każdą decyzję techniczną wyjaśniam w oparciu o pięć wspomnianych pytań i siedem zasad.

Część IV: Łączenie wszystkiego w całość

Część IV zawiera tylko rozdział 13., który wyjaśnia, jak to wszystko łączy się ze sobą. W tym rozdziale skupiamy się na ustanowieniu szybkiego cyklu informacji zwrotnej, usuwając wszystko to, co spowalnia deweloperów w kończeniu iteracji, pozyskiwaniu informacji zwrotnych i uczeniu się. W tym rozdziale zachęcamy liderów, aby upewnili się, że deweloperzy mogą wydajnie wykonywać swoją pracę, a przy tym zaangażowali się w rozwiązywanie wszelkich problemów, które spowalniają deweloperów.

2

Systemy, projekt i architektura

Czym jest architektura oprogramowania?

Architektura oprogramowania to plan budowy systemu oprogramowania. Ten plan obejmuje zwykle dwie rzeczy: zdefiniowanie systemu jako pewnego zestawu komponentów i określenie sposobu, w jaki te komponenty ze sobą współpracują. W złożonych systemach ta dekompozycja ma postać rekurencyjnego procesu, w ramach którego architekt dzieli każdy komponent na mniejsze komponenty i określa ich zachowanie.

Istnieją dobre plany i złe plany. To samo dotyczy architektury oprogramowania. Jakie są cele dobrej architektury oprogramowania?

Nadrzędnym celem systemów oprogramowania (a więc i architektury oprogramowania) jest budowanie systemów spełniających standardy jakości i zapewniających najwyższy zwrot z inwestycji w dłuższej perspektywie lub w określonym horyzoncie czasowym. Idealna jest tutaj *dłuższa perspektywa*. Na przykład, jeśli nie zainwestujemy długoterminowo i zbudujemy kiepski projekt, użytkownicy będą niezadowoleni, przez co ostatecznie utracimy z nich przychody lub wydamy zbyt dużo starając się ich uszczęśliwić. Tanio w krótkim okresie to często drożej na dłuższą metę. Płacimy teraz lub płacimy później. Z drugiej strony dodanie nowej krytycznej funkcji, a więc wydanie większej ilości pieniędzy, może nam przynieść większe przychody, zwiększając w ten sposób ROI.

Projektowanie architektury komplikują trzy rodzaje niepewności. Pierwszy, to gdy tylko częściowo rozumiemy naszych użytkowników i ich oczekiwania. Drugi, to gdy mamy ograniczone pojęcie o tym, jak zachowują się nasze systemy, zwłaszcza w złożonych i nowych sytuacjach. Trzeci, to gdy nie zauważamy, że wraz z ewolucją przypadków użycia i użytkowników zmieniają się ich wymagania. Dlatego chcemy, żeby