helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

COM+. Kompendium programisty



Autor: Luke Hohmann
Tłumaczenie: Paweł Koronkiewicz
ISBN: 83-246-0110-4
Tytuł oryginału: Beyond Software Architecture: Creating and Sustaining Winning Solutions
Format: B5, stron: 320

Przygotuj projekt systemu informatycznego, który naprawdę spełni oczekiwania użytkowników

- Wybierz technologię, platformę sprzętową i model licencjonowania
- Zadbaj o funkcjonalność i łatwość rozbudowy systemu
- Zabezpiecz system przed piractwem, kradzieżą i utratą danych

Termin „architektura oprogramowania" kojarzy się zwykle z doborem języka programowania, wzajemnymi zależnościami między komponentami powstającego systemu informatycznego, wyborem platformy bazodanowej i zaplanowaniem innych elementów związanych wyłącznie z zagadnieniami technicznymi. Tymczasem w opisie architektury systemu nie wolno pomijać także innych kwestii: modelu licencjonowania, sposobu wdrażania i konserwacji systemu, a przede wszystkim jego użyteczności. Te pozornie niezwiązane z projektem elementy mogą mieć duży wpływ na powodzenie przedsięwzięcia, jakim jest stworzenie i sprzedaż oprogramowania. Odpowiednio przygotowany projekt systemu informatycznego powinien więc obejmować zarówno zagadnienia techniczne, jak i ekonomiczne.

Książka „Więcej niż architektura oprogramowania" to poradnik, dzięki któremu stworzenie odpowiedniej relacji między technologią a biznesem jest łatwiejsze, niż mogłoby się wydawać. Może się przydać zarówno menedżerowi, jak i programiście. Autor książki, doświadczony kierownik projektów i twórca oprogramowania, przedstawia związki między zagadnieniami technicznymi a innymi aspektami. Znajdziesz w niej opisy dobrych i skutecznych rozwiązań oraz zaczerpnięte z rynku przykłady planowania produkcji oprogramowania.

- Znaczenie architektury oprogramowania
- Zarządzanie oprogramowaniem jako produktem
- Modele licencjonowania
- Wykorzystywanie obcych technologii w projekcie
- Wdrażanie systemu
- Obsługa techniczna
- Dobór marki
- Funkcjonalność i łatwość obsługi
- Zabezpieczanie aplikacji

Sprawy z pozoru mało ważne często powodują największe problemy. Nie ignoruj ich. Pracuj nad projektem kompleksowo.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa Martina Fowlera 13
Przedmowa Guya Kawasaki 15
Wstęp 17

1.

Architektura oprogramowania 21

Definicja architektury oprogramowania 21
Inne spojrzenia na architekturę oprogramowania 22
Znaczenie architektury oprogramowania 24
Tworzenie architektury 27
Wzorce i architektura 28
Ewolucja i dojrzewanie architektury — funkcje a możliwości 29
Opieka nad architekturą 32
Ogólne zasady 36
Pełne zrozumienie architektury 38
Zespół 39
Podsumowanie rozdziału 40

2.

Oprogramowanie jako produkt 43

Czym jest zarządzanie produktem? 43
Znaczenie zarządzania produktem 44
Proces tworzenia produktu — do wersji 1.0 44
Wcale tak nie jest 50
Biznesplan 53
Proces tworzenia produktu — wersja n.n.n 53
Wspomaganie procesu tworzenia produktu 55
Zarządzanie produktem — najważniejsze pojęcia 57
Podsumowanie rozdziału 64

3.

Różnica między m-architekturą i t-architekturą 67

Jaki jest podział odpowiedzialności? 67
Siły działające na początku projektu 68
Praca w długim okresie i wyniki w krótkim okresie 72
Wizja przyszłości 73
Zarządzanie informacjami zwrotnymi 74
Budowanie przejrzystości 74
Jednomyślność działań 76
Diagramy kontekstowe i produkty docelowe 78
Podsumowanie rozdziału 79

4.

Symbioza modelu biznesowego i modelu licencjonowania 81

Typowe modele biznesowe oprogramowania 82
Prawa licencjobiorcy 92
Wpływ modelu biznesowego na t-architekturę 95
Stosowanie modeli licencjonowania 99
Dojrzałość rynku a model biznesowy 104
Podsumowanie rozdziału 106

5.

Korzystanie z technologii licencjonowanych 109

Licencjonowanie — zagrożenia i korzyści 109
Umowa 112
Niezgodność modeli biznesowych i negocjacje 117
Honorowanie umów licencyjnych 118
Włączanie technologii licencjonowanej 119
Licencjonowanie Open Source 119
Opłaty licencyjne 120
Ekonomika licencjonowania 122
Podsumowanie rozdziału 122

6.

Wieloplatformowość 125

Rzekome korzyści z wieloplatformowości 125
Uzasadnienie biznesowe wieloplatformowości 126
Tworzenie aplikacji wieloplatformowej 129
Macierz trudów pracy 131
Niebezpieczne obietnice 135
Podsumowanie rozdziału 135

7.

Architektura wdrożeniowa ... 137

Rodzaje architektury wdrożeniowej ... 137
Wpływ klienta ... 140
Wpływ rodzimej organizacji ... 142
Wybór architektury wdrożeniowej ... 145
Architektury wdrożeniowe i podział pracy ... 146
Urządzenia informatyczne typu IA ... 147
Wpływ na architekturę oprogramowania ... 147
Przyszłość oprogramowania powszechnego użytku ... 149
Podsumowanie rozdziału ... 149

8.

Integracja i rozbudowa ... 151

Kontrola w rękach klienta, czyli siła przewodnia ... 151
Architektura warstwowa — struktury logiczne ... 153
Projektowanie i implementacja architektury warstwowej ... 157
Integracja i rozbudowa warstw logiki biznesowej ... 159
Integrowanie i rozbudowa magazynu danych ... 164
Konsekwencje natury biznesowej ... 168
Interfejs API a kolejne wersje ... 174
Podsumowanie rozdziału ... 175

9.

Marka i elementy marki ... 177

Elementy marki ... 177
Marki produktów licencjonowanych ... 182
Dostosowywanie elementów marki ... 182
Zmiana elementów marki ... 183
Podsumowanie rozdziału ... 185

10.

Funkcjonalność („usability”) ... 187

Funkcjonalność = pieniądze ... 187
Modele myślowe, metafory i funkcjonalność ... 189
Wpływ t-architektury na interfejs użytkownika ... 190
Szybciej, wyżej, mocniej ... 196
Podsumowanie rozdziału ... 203

11.

Instalacja ... 205

Wyjmij z pudełka i rozpocznij pracę ... 205
Au! Może boleć ... 207
Instalacja a architektura ... 208

Jak instalować ... 210
Ostatnie szlify ... 214
Podsumowanie rozdziału ... 216

12.
Aktualizacja ... 219

Jak instalacja, tyle że gorsza ... 219
Mniej kłopotliwe uaktualnienie ... 222
Aktualizacje a dojrzałość rynku ... 225
Podsumowanie rozdziału ... 226

13.
Konfiguracja ... 229

Konfigurowalność — element funkcjonalności ... 229
Kontekst systemu ... 230
W trakcie inicjalizacji i w trakcie pracy ... 231
Ustawianie wartości ... 232
Ustawianie właściwej wartości ... 233
Ogólne zasady pracy z parametrami ... 234
Podsumowanie rozdziału ... 235

14.
Dzienniki ... 237

Chcę wiedzieć, co jest grane ... 238
Nie tylko fakty ... 239
Format i zarządzanie dziennikiem ... 241
Przetwarzanie danych dziennika ... 245
Usługi rejestrowania ... 245
Podsumowanie rozdziału ... 246

15.
Zarządzanie wersjami ... 249

Tak, to jest potrzebne ... 249
Nasz punkt wyjścia ... 250
Zarządzanie wersjami ... 251
Identyfikacja wersji ... 252
Oznaczenia SKU i numery seryjne ... 257
Zarządzanie wersjami a t-architektura ... 260
Podsumowanie rozdziału ... 262

16.
Zabezpieczenia ... 263

Wirusy, hakerzy i piraci ... 264
Zarządzanie cyfrową tożsamością ... 266
Bezpieczeństwo transakcji ... 269

Bezpieczeństwo oprogramowania ... 271
Bezpieczeństwo informacji ... 273
Ukrywanie algorytmów czy ukrywanie kluczy? ... 274
Tylne wejście ... 274
Bezpieczeństwo i m-architektura ... 275
Podsumowanie rozdziału ... 277

A

Release — lista kontrolna ... 281

Identyfikacja ... 281
Programowanie ... 281
Zapewnienie jakości ... 281
Publikacje techniczne ... 282
Produkt ... 282
Przekaz informacji — usługi ... 282
Przekaz informacji — sprzedaż i dystrybucja ... 282
Przekaz informacji — pomoc techniczna ... 283
Przygotowanie dystrybucji ... 283

B

Język wzorców strategicznego zarządzania produktem ... 285

Stosowanie wzorców ... 285
Prezentacja wyników ... 287
Mapa rynku ... 287
Wydarzenia i rytmy rynku ... 289
Mapa funkcji i korzyści ... 291
Plan rozwoju t-architektury ... 292

Bibliografia ... 295

Literatura ... 297

O autorze ... 301

Skorowidz ... 303

1

Architektura oprogramowania

Podstawą udanego rozwiązania jest architektura, która pozwala je stworzyć, a potem rozwijać. W tym rozdziale spróbuję określić, czym jest architektura oprogramowania, jak powstaje i jak ewoluuje. Jednocześnie zwrócę uwagę na wzorce architektury, ponadczasowe zasady jej projektowania i czynniki, które wpływają na jej kształt. Na końcu poruszę zagadnienie zależności między architekturą a zespołem, który tworzy ją i rozwija.

Definicja architektury oprogramowania

Architektura oprogramowania to złożone zagadnienie. Z tej złożoności wynika różnorodność definicji. Ich użyteczność zależy od przyjmowanej perspektywy. Oto definicja z mojej pierwszej książki, *Journey of the Software Professional*:

> Architektura systemu określa podstawową „strukturę" systemu (moduły wysokiego poziomu, które realizują główne funkcje systemu, zarządzanie i rozkład danych, rodzaj i charakter interfejsu użytkownika, platformę docelową itp.).

Jest to definicja, która w dużej mierze pokrywa się z wieloma innymi, na przykład [Bass], [Larman] i [POSA]. Jednak brakuje w niej kilku ważnych elementów, takich jak wybór technologii i wymagania użytkowników systemu. Mój współpracownik, Myron Ahn, stworzył definicję architektury oprogramowania, którą przytaczam poniżej. Jest nieco bardziej rozbudowana i szersza niż ta, którą przedstawiłem wcześniej (2002, z rozmów):

> Architektura oprogramowania to suma znaczących modułów, procesów i danych systemu, ich struktury i wzajemnych zależności, tego, jak mogą być i jak będą rozbudowywane i modyfikowane, a także wykorzystywanych technologii. Wynikają z niej funkcje i możliwości systemu, może też służyć jako podstawa do jego implementacji lub modyfikacji.

Te dwie definicje można dalej rozbudowywać technicznie, ale to już nie zwiększyłoby znacząco ich wartości. *Architektura*, bardziej niż jakikolwiek inny aspekt systemu, dotyczy jego „widoku ogólnego". Aby naprawdę ją rozumieć, niezbędne jest spojrzenie z perspektywy całości.

Inne spojrzenia na architekturę oprogramowania

Choć powyższe definicje architektury oprogramowania są użyteczne, są zdecydowanie zbyt proste, aby uwzględnić pełen zbiór sił, które kształtują architekturę i, z drugiej strony, są kształtowane przez nią. Szczerze mówiąc, wątpię, aby jakakolwiek definicja architektury miała szansę uchwycić wszystko to, co uważamy za istotne. Aby uzasadnić to stwierdzenie, w tym podrozdziale przedstawię kilka zagadnień, których nie obejmują tradycyjne definicje architektury oprogramowania, a którym trudno odmówić znaczenia. W przeciwieństwie do wcześniejszych definicji, które koncentrują się na aspektach technicznych, są to zagadnienia bardziej związane z „czynnikiem ludzkim" i kwestiami natury biznesowej. Te również są częścią nierozerwalnie związanego z architekturą „spojrzenia całościowego".

Podsystemy a zależności

Doświadczenie w zarządzaniu rozproszonymi zespołami, których członkowie pracowali na wszystkich kontynentach Ziemi, nauczyło mnie, że ważnym kryterium w dekompozycji do podsystemów jest uzyskanie możliwie najprostszych zależności między współpracującymi organizacjami. Przez „proste" rozumiem takie, które nie utrudniają pracy osób tworzących system. Pracując jako konsultant, zauważyłem, że — w przeciwieństwie do uzasadnień natury technicznej — wiele związanych z architekturą decyzji dotyczących projektowania podsystemów to decyzje, których osią było dążenie do stworzenia jasnych i przejrzystych zależności między grupami programistów. Praktycznym skutkiem takich decyzji jest fakt, że pracę nad podsystemami rzadko dzieli się pomiędzy różnych wykonawców.

Podsystemy a motywacje i dążenia

Wiele książek poświęconych architekturze usuwa z procesu jej projektowania niemal cały bagaż związany z „czynnikiem ludzkim". Przykładowo, wzorce architektury są świetnym początkiem procesu projektowania. Jednak tworzenie architektury nie sprowadza się do przyjęcia pewnego rodzaju struktury początkowej, takiej jak wzorzec, i obiektywnego dopasowywania jej do potrzeb danej organizacji. Trzeba rozumieć i uwzględniać nadzieje, doświadczenia, marzenia, obawy i upodobania zespołu, który odpowiada za budowę systemu. Techniczna struktura architektury kształtuje strukturę społeczną (i odwrotnie), a towarzyszą temu ciągłe konflikty obiektywnych i subiektywnych przesłanek podejmowanych decyzji.

Aby nieco lepiej zrozumieć ten problem, przypomnijmy sobie sytuację, w której zdecydowanie zależało nam na pracy nad określonym aspektem architektury, bo wiedzieliśmy, że zrobimy to lepiej niż ktokolwiek inny (pewność siebie oparta na doświadczeniu); w której chcieliśmy lepiej poznać wykorzystywaną technologię (pragnienie); w której wierzyliśmy, że przyłożenie się do pracy pozwoli uzyskać premię lub awans (aspiracje), albo taką, w której obawialiśmy się, że nikt inny w zespole nie ma wystarczających umiejętności lub doświadczenia, aby rozwiązać problem *we właściwy sposób* (obawa, strach).

Projektowanie podsystemów a poczucie całości

Niewielu programistów chciałoby specjalizacji sprowadzającej ich pracę do jednego rodzaju działań — analizy, projektowania, pisania kodu czy poprawiania błędów. Dużo bardziej typowe jest dążenie do zaangażowania w pełen zakres czynności programisty: omawianie wymagań z klientem lub menedżerem produktu, opracowywanie i realizowanie artefaktów analizy i projektu, poprawianie błędów, optymalizowanie wydajności itd. Uważam, że jest to wyraz stosunkowo silnego dążenia do poczucia „kompletności" wykonywanej pracy. Dążenie to ma głębokie implikacje. Dobra decyzja projektowa to często decyzja, która pozwala je zaspokoić.

Pojęcie „kompletności" wykonywanej pracy jest różnie rozumiane przez różne osoby. Kierownictwo musi być dobrze zorientowane w tym, jak jest postrzegane przez dany zespół. W pewnej aplikacji typu klient-serwer rozważaliśmy kilka różnych konstrukcji klienta. Przytoczę tutaj dwie z nich, aby zilustrować naszą interpretację kompletności pracy i to, jak wpłynęła na podjęty wybór.

W pierwszej wersji jeden zespół odpowiadał za zagadnienia „związane z klientem", a drugi za „infrastrukturę". W drugiej wersji projektu każdy zespół przyjmował odpowiedzialność za komponenty z obu tych obszarów.

O ile pierwszy projekt mógł wydawać się nieco bardziej przejrzysty na papierze, zdecydowanie psuł morale zespołu infrastruktury, który skłaniał się ku opinii, że pozbawiłoby to ich pełnego uczestnictwa w procesie przygotowywania kolejnych wersji. Mówiąc ściślej, zespół ten tracił wówczas okazję do bezpośredniej pracy z osobami odpowiedzialnymi za kierowanie produktem, publikacjami technicznymi i potencjalnymi klientami. Należało to do jego obowiązków wcześniej i wszyscy chcieli, żeby tak zostało. W efekcie wybraliśmy drugą wersję architektury, ponieważ tylko ona pozwalała obu zespołom zachować to samo poczucie kompletności wykonywanej pracy, tak bardzo ważne dla programistów. Dla nich oznaczało ono, między innymi, kontakt z osobami kierującymi rozwojem produktu i udział w procesie przygotowywania kolejnych wersji.

Poddawanie się dobrej architekturze

Używam czasownika „poddawać się", gdy architekt lub zespół programistów odsuwa od siebie, tak daleko jak to możliwe, własne doświadczenia i wyobrażenia o tym, co jest poprawne i właściwe, pozwalając, aby ich pracą nad architekturą kierowały siły właściwe dziedzinie problemu. Niektórzy twierdzą, że nie jest to problemem i że oni lub ich zespół zawsze tworzą architekturą opartą wyłącznie na obiektywnej analizie problemu klienta i najlepszych metodach prowadzących do jego rozwiązania. Słowo-klucz to oczywiście *najlepsze*. Zdanie Czytelnika o tym, co jest najlepsze, może różnić się od mojego. Różnice tego rodzaju biorą się przede wszystkim z posiadanego zasobu doświadczeń. Nie wynikają bezpośrednio z dziedziny problemu (z wyjątkiem sytuacji, kiedy zdobywamy doświadczenia w obrębie danej dziedziny). Jednym z aspektów „poddawania się" dobrej architekturze jest ciągłe weryfikowanie, czy podejmowane decyzje mają na uwadze przede wszystkim potrzeby klienta, i zgoda na zmienianie tych decyzji, jeżeli weryfikacja taka będzie miała skutek negatywny.

Nie to ładne, co ładne, ale co się komu podoba!

Wszyscy możemy podać po kilka definicji udanych architektur oprogramowania. Podczas gdy firma może uważać, że system jest udany, bo wspiera produkcję lub sprzedaż dochodowego produktu, doglądający go programiści mogą w tym samym czasie załamywać ręce nad archaicznymi rozwiązaniami technicznymi. Analogicznie, wiele zgrabnych i eleganckich rozwiązań technicznych kończy jako mniejsze lub większe niepowodzenia. Co więcej, elegancja techniczna rozwiązań to pojęcie subiektywne. Dwóch najlepszych programistów mojego zespołu miało skrajne różnie podejścia do stosowania procedur przechowywanych w bazach danych. Jestem pewien, że każdy z nich potrafiłby stworzyć dobre rozwiązanie praktycznie każdego problemu związanego z bazami danych. Jestem równie pewien, że ich aplikacje byłyby różne, a każdy z nich potrafiłby uzasadnić podjęte decyzje, przedstawiając zarazem poprawną krytykę architektury zastosowanej przez drugiego. Niewielu programistów potrafi wyjść poza ograniczenia narzucane im przez własne rozumienie estetyki architektury.

Znaczenie architektury oprogramowania

Architektura oprogramowania jest ważna, ponieważ, gdy jest dobra, staje się czynnikiem decydującym o przyszłym powodzeniu projektu. Architektura wpływa na sukces przedsięwzięcia w różny sposób. Nie każdy z tych wpływów jest równie ważny, ale wszystkie mają swój początek w architekturze.

Trwałość

Większość architektur trwa dużo dłużej niż zespoły, które je tworzyły. Ogólne szacunki trwałości systemu lub architektury wskazują na okres od 12 do 30 lub więcej lat, podczas gdy typowy czas aktywnej pracy programisty nad tym samym systemem mieści się w granicach od 2 do 4 lat.

Stabilność

Wiele korzyści z dobrej architektury można wywieść właśnie od jej trwałości. Jedną z najważniejszych jest stabilność. Stabilność architektury sprzyja ograniczeniu ilości poważnych modyfikacji w trakcie rozbudowy zakresu funkcji systemu w kolejnych wersjach. W efekcie, maleje całkowity koszt implementacji. Stabilność to dla zespołu programistów ważna podstawa i znaczne ułatwienie. Pracę nad ulegającym ciągłym przemianom kodem zastępuje dzięki niej koncentracja na zmianach o największej wartości.

Stopień i natura zmian

Architektura wyznacza naturę zmian w systemie. Pewne zmiany wydają się proste. Inne są postrzegane jako trudne. Gdy prostota silnie koreluje z *pożądanym* zbiorem zmian zwiększających zadowolenie klienta lub pozwala nam dodawać funkcje przyciągające nowych klientów, zazwyczaj nie wzbraniamy się przed określeniem architektury przymiotnikiem „dobra”.

W jednej z aplikacji, nad którymi pracowałem, stworzyliśmy architekturę *pluginów*, które pozwalały rozbudowywać narzędzia analityczne operujące na różnych, zarządzanych przez system danych. Dodawanie nowego narzędzia było stosunkowo proste. Cecha ta była wartościowa, ponieważ głównym celem kierownictwa projektu było dodanie do aplikacji możliwie dużej liczby narzędzi.

Opłacalność

Dobra architektura to architektura opłacalna. Rozumiem przez to, że firma, która ją stworzyła, może pielęgnować ją, zachowując akceptowalny poziom kosztów. Jeżeli koszty dalszego utrzymywania architektury stają się zbyt wielkie, architektura zostaje porzucona.

Nie oznacza to, że opłacalna architektura jest architekturą elegancką czy ładną. Jedną z najbardziej dochodowych architektur w historii jest rodzina systemów operacyjnych Microsoft Windows. Mimo to krytycznych opinii na temat elegancji zastosowanych rozwiązań nie brakuje.

Warto zwrócić uwagę, że opłacalność określonych rozwiązań technicznych często niewiele ma wspólnego z samą architekturą. Firma Microsoft decydowała ogromną przewagą w obszarach marketingu, dystrybucji czy marki. Wszystko to miało duży udział w niezwykłej opłacalności Windows.

Nie usprawiedliwiam tutaj architektur źle zaprojektowanych i brzydkich, których budowa i utrzymanie kosztuje więcej niż to konieczne! W długim okresie, bazą dla opłacalnych rozwiązań są zdecydowanie architektury proste i eleganckie.

Struktura społeczna

Dobra architektura jest korzystna dla zespołu jej twórców. Pozwala w pełni wykorzystać jego silne strony, a niekiedy też ograniczyć wpływ słabych. Dobrym przykładem jest częsty brak umiejętności niezbędnych do właściwego korzystania z języków C i C++. Typowym wynikiem popełnianych wtedy błędów w zarządzaniu pamięcią są pozornie zupełnie przypadkowe błędy aplikacji. Dla zespołów, które nie wymagają niepowtarzalnych cech C lub C++, lepszym wyborem jest bezpieczniejszy język, jak Java, Visual Basic, Perl lub C#, wyręczający programistę w zarządzaniu pamięcią.

Raz zdefiniowana architektura wywiera silny wpływ na dalszą pracę zespołu i jego kształtowanie. Bez względu na to, jaki język wybierzemy, do podjętej decyzji dostosujemy dalsze, przede wszystkim związane z zatrudnianiem nowych programistów i szkoleniami. Ponieważ architektury trwają dłużej niż zespoły, skutki mogą dawać się we znaki przez lata (przypomnijmy sobie niewiarygodny wzrost zapotrzebowania na programistów języka COBOL w ostatnich trzech latach przed rokiem 2000, kiedy wciąż działające aplikacje musiały zostać dostosowane do pracy w nowym millenium).

Wyznaczone granice

W procesie projektowania architektury nieustannie podejmowane są decyzje o tym, co stanie się częścią systemu, a co nie. Wyznaczane w ten sposób granice, ich uwarunkowania i to, jak są traktowane, to kolejny ważny czynnik wpływający na powodzenie architektury. Związanych z nimi pytań nie brakuje. Czy pisać własną warstwę dostępu do bazy danych, czy wykorzystać gotowy produkt? Czy korzystać z serwera Open Source, czy komercyjnego? Który podzespół będzie odpowiedzialny za interfejs użytkownika? Dobre systemy mają ściśle określone granice, odpowiadające rzeczywistym potrzebom ich

Ilu programistów, żeby to wymienić?

Jedną z najtrudniejszych decyzji stojących przed starszymi członkami każdego zespołu programistów jest określenie właściwego momentu zastąpienia starej architektury nową.

Istotnych jest wiele czynników, takich jak koszt utrzymywania bieżącej architektury, wymagania klientów, możliwości spełnienia tych wymagań czy działania konkurencji. Wyznaczenie formalnego zestawu reguł, który pozwalałby określić, kiedy zmienić architekturę, jest oczywiście niemożliwe. Czynników do uwzględnienia jest zbyt wiele, a większość z nich jest niemierzalna. Poniżej wymieniam kilka prostych zasad postępowania, które sprawdziły się w mojej praktyce.

Jeżeli sądzisz, że nową architekturę może przygotować w czasie krótszym niż rok zespół programistów o połowę mniejszy od obecnego, możliwość przeprowadzenia takiej zmiany warto poważnie rozważyć. Mówiąc bardzo ogólnie, jest to sytuacja, w której zespół można podzielić na grupę zajmującą się nowym systemem i grupę odpowiedzialną za konserwację starego, bez wprowadzania zmian w strukturze kosztów.

Zasoby wykorzystywane w tworzeniu nowej architektury wykraczają zazwyczaj poza to, co może zapewnić bieżący zespół. Nie można się oszukiwać, tworzenie nowej architektury przez mniejszą liczbę osób to sytuacja wymuszająca radykalne zmiany w procesie budowy oprogramowania. Nie każdy zespół ma niezbędne umiejętności. Nie każdy zespół chce je zdobyć i nie każdemu to się udaje. Gotowość do zmiany w składzie zespołu jest często warunkiem powodzenia zmiany architektury systemu.

O ile niewielkie zmiany w składzie zespołu programistów są często wymagane, szaleństwem będzie próba wymiany całego składu. Im system starszy i bardziej złożony, tym istotniejsza jest rola współpracy z co najmniej jednym, a najlepiej kilkoma „weteranami". Zapewniają oni przede wszystkim wierne odwzorowanie zestawu funkcji starego systemu. W starym kodzie zawsze znajdą się zagadki, których rozwiązania znają tylko najstarsi programiści.

Oczywiście, wszystkie tradycyjne ostrzeżenia — „ostrożność przede wszystkim", „to nie będzie takie proste i zajmie więcej czasu, niż się spodziewamy" — pozostają aktualne. Tak, duże zmiany wymagają wyważonych decyzji, a większość okazuje się trudniejsza i trwa dłużej, niż planowano. Wciąż zapominamy o pełnych kosztach zmiany, które poza samym programowaniem obejmują przeróżne konsultacje, pisanie dokumentacji, szkolenia i aktualizacje. Pewną wskazówką może być uproszczona reguła, że nowa architektura, funkcjonalnie równoważna wcześniejszej, będzie kosztować co najmniej 20 procent *całkowitej* sumy wydanej na starą architekturę, *o ile programiści są doświadczeni, nie wystąpią zakłócenia w realizacji projektu, a sam proces budowy oprogramowania zostanie wydatnie poprawiony*. Jeżeli mamy do czynienia z czwartą iteracją systemu budowanego przez 5 lat, kosztem 23 milionów dolarów, plan stworzenia pierwszej wersji nowego systemu za mniej niż 3 do 5 milionów jest najprawdopodobniej zanadto optymistyczny. Koszt stworzenia nowej, funkcjonalnie równoważnej architektury to zazwyczaj od 40 do 60 procent całkowitych wydatków poniesionych na starą. Zależy on w dużej mierze od wielkości zespołu i złożoności wymienianego systemu.

użytkowników. Warunek ten nie zawsze jest spełniony, zwłaszcza w środowisku nowego rynku, gdy projektanci nie mogą liczyć na dużą pomoc i muszą tworzyć większość architektury od podstaw. Złe decyzje mogą zaprowadzić ich w ślepą uliczkę, z której bardzo trudno wyjść.

Trwała, niesprawiedliwa przewaga

Ten ostatni punkt podsumowuje poprzednie i jest zdecydowanie najważniejszy. *Bardzo dobra* architektura zapewnia trudną do powielenia, trwałą i opartą na solidnych, technicznych podstawach przewagę konkurencyjną. Jeżeli techniczne aspekty architektury są łatwe do powielenia, muszą istnieć inne czynniki wyróżniające system (na przykład lepsze wsparcie techniczne lub dystrybucja). Nawet wówczas jednak architektura pozostaje ważna — gdy jest dobra, ułatwi budowanie przewagi w oparciu o takie aspekty jak szybkość pracy lub zakres zastosowań.

Tworzenie architektury

Architektura oprogramowania (ang. *software architecture*) to na początku zbiór decyzji zespołu, który projektuje pierwszą wersję. Te początki, znaczone szkicami na tablicy i diagramami w notesie (lub na serwetce), reprezentują zamiary pierwszego zespołu programistów. Gdy system zostaje ukończony, jego budowa może w niczym nie przypominać wczesnych pomysłów. Z drugiej strony, retrospektywna analiza jest często jedyną drogą wyjaśnienia konstrukcji systemu. Jest niezbędna, aby powiązać to, co powstało, z pierwotnymi planami.

Pierwsza wersja architektury przypomina często małe dziecko: wymagana jest kompletność, ale brakuje dojrzałości i stabilności. Dopiero z czasem, w miarę używania oprogramowania i tworzenia kolejnych wersji, architektura nabiera tejże dojrzałości, a zarazem trwałości. Jednocześnie użytkownicy i twórcy budują swoją wiarę w możliwości systemu, uświadamiają sobie i rozwiązują problemy jego ograniczeń. Podstawą tego procesu jest rzetelne gromadzenie informacji zwrotnych i reagowanie na nie odpowiednimi, sprzyjającymi sukcesowi produktu, zmianami. Oczywiście, gdy brakuje informacji zwrotnych, niedojrzała architektura może taką pozostać, jej rozwój może ulec stagnacji. Decyduje zazwyczaj rynek.

Miałem szczęście uczestniczyć w tworzeniu kilku nowych architektur od podstaw. Były to doświadczenia o tyle ciekawe, że pozwoliły mi zauważyć, że faktyczny przebieg procesu projektowania i budowania nowej architektury odbiega w pewnym stopniu od zaleceń podręcznikowych. Mówią one między innymi, że gdy system jest jeszcze zupełną nowością i rozwiązywane problemy są jeszcze mało znane, projektanci powinni „badać architektury alternatywne w poszukiwaniu tej, która najlepiej sprawdza się w danym przypadku". Jest to bardzo rozsądna rada. Jest też zarazem mało przydatna.

Jest wiele przyczyn, które sprawiają, że przytoczone zalecenie jest mało użyteczne w praktyce. Najważniejsze wydają się trzy z nich. Każda jest w zupełności wystarczająca w roli czynnika, który powstrzymuje przed badaniem alternatyw. Zazwyczaj mamy do czynienia ze wszystkimi trzema jednocześnie. Każda wywiera swój wpływ, czasem w połączeniu z innymi czynnikami. Wszystkie prowadzą do nieco innego, bardziej pragmatycznego podejścia do budowania nowej architektury. Nie chodzi o to, że są to siły złe. Są to siły, które *są*.

Pierwsza siła to charakterystyczny dla przedsięwzięć komercyjnych brak czasu na rzetelną weryfikację wielu ścieżek do rozwiązania tego samego problemu. Zasada „czas to pieniądz" sprawdza się zawsze, i o ile pierwsza wersja nie jest w oczywisty sposób katastrofą, będzie to niemal na pewno wersja, którą trzeba przedstawić klientowi. Odbiorca nie może czekać! Siła ta jest na tyle duża, że już przy kompletowaniu zespołu programistów dbamy o to, aby architekt pierwszego systemu dysponował wystarczającym doświadczeniem, aby podjąć właściwe decyzje bez badania alternatyw. Innymi słowy, zatrudniamy osobę, które spotkała się z alternatywami w poprzednich miejscach pracy.

Druga siła, wyraźnie mocniejsza niż pierwsza, to natura problemu i towarzyszących mu warunków, mocno zawężające dostępne opcje. Witryna WWW, która ma być szybka i obsługiwać wielu użytkowników, musi być oparta na względnie standardowej architekturze. Można stosować różne

Gdy już wiesz, że się nie uda…

Doświadczenia z różnymi architekturami ułatwiają rozpoznawanie błędnych decyzji jeszcze przed etapem implementacji. Można wtedy zrezygnować z architektury i odesłać programistów z powrotem do tablicy albo też zrezygnować z projektu w ogóle. Rezygnacja z projektu może wydawać się posunięciem drastycznym, ale jest w pewnych sytuacjach decyzją konieczną, zwłaszcza gdy złe decyzje znacząco zmieniają perspektywy ekonomiczne systemu.

W jednym ze znanych mi przedsięwzięć podjęto decyzję o zastosowaniu pełnej implementacji J2EE do rozwiązania problemu, z którym bez trudu poradziłoby sobie kilkadziesiąt skryptów w Perlu — o ile tylko zostałaby zmieniona druga część aplikacji. W tym klasycznym przypadku „projektowania kierowanego CV" szef zespołu technicznego, który przyjął rolę architekta, zdołał przekonać co ważniejsze osoby, że implementacja oparta na J2EE będzie najlepsza. Dodatkowo, na architekturze ciążyło kilka wyraźnie błędnych założeń. Najbardziej zapadł mi w pamięć pomysł przesyłania obrazów CD o rozmiarze do 600 MB pocztą elektroniczną, w postaci załączników! Ostatecznie doszedłem do przekonania, że nie ma w tej sytuacji niczego wartego ratowania i doprowadziłem do zamknięcia projektu. Było już po prostu za późno, aby przygotować i wprowadzić do sprzedaży planowany produkt.

systemy operacyjne, ale i tu obowiązują pewne kanony. Poszukiwanie nowych architektur wysokiego poziomu nie ma zazwyczaj wielkiej wartości, ale często warto zwrócić uwagę na inne rozwiązania w bardziej zawężonych obszarach projektu. U podstaw takich poszukiwań powinna leżeć rzetelna analiza zagrożeń stojących przed budowanym rozwiązaniem, a wybierane środki powinny być dobrze przemyślane.

Trzecia, najsilniejsza siła to problem z określeniem tego, czy dana architektura jest faktycznie dobra w danym zastosowaniu. Dopóki nie rozpocznie ona pracy w środowisku, dla którego została zaprojektowana, nie można mieć pewności, czy zastosowano właściwe rozwiązania. Zdobycie wiedzy na ten temat wymaga czasu, zazwyczaj liczonego w latach.

Wzorce i architektura

Główną konsekwencją działania wymienionych sił jest to, że tworzenie wstępnej wersji architektury musi opierać się na podejściu zdecydowanie pragmatycznym. Podejściu takiemu sprzyjają wzorce projektowe. Jest to próba uchwycenia dobrych rozwiązań powtarzalnych problemów w taki sposób, aby zgromadzoną wiedzę można było zastosować w nowych sytuacjach. Dobre wzorce znalazły zastosowanie w kilku czy kilkunastu systemach, najlepsze — w dziesiątkach i setkach rozwiązań. Ktoś, często więcej niż jedna osoba, zadał sobie trud, aby zbadać, przeanalizować i dokładnie opisać problem oraz sprawdzoną metodę jego rozwiązania. Dzięki temu inni mogą korzystać z doświadczeń zdobytych w pracy nad systemami, które zdały egzamin w praktyce. Wzorzec projektowy to rodzaj „wstępnie przetrawionej" wiedzy.

Wzorce architektury mają za zadanie uchwycić podstawowe organizacje struktury oprogramowania. Odnoszą się głównie do technicznych aspektów architektury wymienionych wcześniej w tym rozdziale i dostarczają opisów podsystemów oraz ich zakresów odpowiedzialności, a także interakcji między nimi. Koncentracja na ściśle określonej klasie problemu sprawia, że wzorzec architektury pomaga podjąć decyzję o tym, jaki rodzaj lub styl architektury jest właściwy.

Podejście pragmatyczne w tworzeniu architektury oprogramowania oznacza badanie wielu różnych udokumentowanych wzorców i wybór tego, który odnosi się do rozpatrywanej sytuacji. Jest to punkt wyjścia do dalszego kształtowania architektury pod kątem konkretnych potrzeb. Kształtowaniem architektury i podejmowanym przy tym decyzjom towarzyszą zasady, które opisuję w dalszej części tego rozdziału. Dla Czytelnika, który miał okazję poznać skatalogowane wzorce architektury, dalsza część tej książki może być istotnym uzupełnieniem, bo zwraca uwagę na elementy, których wzorce nie opisują. Przykładowo, każda architektura skorzysta na dopracowanej procedurze instalacji i aktualizacji czy rozsądnie zaprojektowanych plikach konfiguracyjnych. Celem nadrzędnym jest stosowanie wszystkich dobrych reguł, które prowadzą do rozwiązań udanych i trwałych.

Ewolucja i dojrzewanie architektury — funkcje a możliwości

Choć wiele uwagi poświęca się projektowaniu i wczesnym wersjom architektury, w rzeczywistości większość czasu spędzamy, pracując nad architekturą, która już istnieje. Ewolucja architektury może być dużo bardziej interesującym procesem niż tworzenie jej pierwszej wersji. To w trakcie tej ewolucji dowiadujemy się co jest dobre, a co złe, zwłaszcza gdy kierują nią informacje pozyskiwane bezpośrednio od klientów. Osobiście, jako największe wyzwania i przynoszące najwięcej satysfakcji osiągnięcia wspominam pracę z zespołami programistów, której celem było zmodyfikowanie istniejącej architektury w taki sposób, aby lepiej dopasować ją do potrzeb rynku i stworzyć warunki dla dalszego powodzenia produktu.

Procesem tym, na który składa się ewolucja i dojrzewanie architektury, kieruje przede wszystkim informacja od klientów, którzy faktycznie używają systemu. Wiele firm twierdzi, że nieustannie starają się o informacje zwrotne. Rzeczywistość jest jednak taka, że są to dane najbardziej cenione i najintensywniej pozyskiwane w okresie planowania następnej wersji systemu. Ta następna wersja jest definiowana pewnym zestawem **funkcji**, wymienianych (możliwie często) w materiałach marketingowych. O tym, czy dana funkcja może zostać włączona do systemu, decydują możliwości architektury. Właśnie zależności między pożądanymi funkcjami a możliwościami architektury kształtują ewolucję tej ostatniej.

Taka gra wzajemnych zależności jest o tyle ciekawa, że ma miejsce w środowisku nieustającej ewolucji technicznej. Innymi słowy, pozyskiwana od klienta informacja zwrotna nie zawsze odwołuje się do aspektów systemu, nad którym pracujemy. Jej pochodzenie często można wywieść od docierających do aktywnego użytkownika wiadomości o różnych, potencjalnie użytecznych technikach pracy. Źródłem takich danych nie musi być nawet klient, często dostarcza ich sam zespół programistów. Bez względu jednak na źródło czy kontekst informacji, praktycznym podejściem jest rozpatrywanie ewolucji i dojrzewania architektury w kategoriach funkcji *i* możliwości.

Funkcja (ang. *feature* lub *function*) definiuje to, co produkt robi lub powinien robić (a zarazem to, co zwraca uwagę klienta). Pełen zestaw wymaganych funkcji określa wymagania wobec produktu. Może być budowany, dokumentowany i modyfikowany wieloma metodami. Na wypadek, gdyby Czytelnik zastanawiał się, *czym* dokładnie jest funkcja w znaczeniu, do którego się tutaj odwołuję, oto sprawdzona definicja z *Exploring Requirements: Quality Before Design* [Weinberg i Gause, 1989]: „Aby sprawdzić, czy wymaganie jest [funkcją], umieść sformułowanie ‘Produkt ma…’ lub ‘Produkt powinien…’ przed opisem tego wymagania”. Podejście to prowadzi do zakwalifikowania jako funkcje bardzo zróżnicowanych wymagań. Oto kilka przykładów:

- *Obsługiwane platformy.* „Produkt ma pracować w systemach Solaris 2.8 i Windows XP."
- *Przypadki użycia.* „Produkt powinien umożliwiać zarejestrowanie nowego użytkownika."
- *Działanie.* „Produkt powinien dostarczać sygnał wybierania numeru w czasie nie dłuższym niż 100 milisekund od odebrania sygnału podniesienia słuchawki."

Zwróćmy uwagę, że gdy rozpatrujemy opisy funkcji, przypadki użycia mają znaczącą przewagę nad innymi postaciami dokumentacji, bo osadzają opisywaną funkcję w określonym kontekście. Najlepszym wyznacznikiem kontekstu są cele aktorów przypadku — ich zamiary. Gdy je znamy i gdy wiemy, że system pozwala realizować cele aktorów, dysponujemy danymi potrzebnymi kierownictwu produktu do stworzenia oferty informującej o strategiach sprzedaży, licencjonowania i cen.

Funkcje dobrze jest uporządkować. Dział marketingu określa ich priorytety. Zespół programistów powinien rozpoznać występujące między nimi zależności natury technicznej. Wynika to stąd, że funkcje są zazwyczaj powiązane — wprowadzenie jednej opiera się na wprowadzeniu drugiej. Jak pisałem wcześniej, ponieważ architektura systemu określa to, jak łatwe (lub jak trudne) będzie implementowanie przyszłych funkcji, dobrą architekturą będzie ta, w której implementowanie wymaganych funkcji jest proste.

Możliwość (ang. *capability*) architektury to jej zdolność do zapewnienia obsługi pewnego zakresu funkcji. Znaczenie możliwości systemu daje się we znaki, gdy dział marketingu regularnie otrzymuje informacje, że pewna klasa funkcji (albo też zbiór funkcji pozornie niepowiązanych, a w rzeczywistości bliskich sobie w implementacji) jest trudna lub *niemożliwa* do zaimplementowania w obrębie stosowanej architektury. W ramce opisuję kilka przykładów takich sytuacji.

Zależności między dojrzałością i ewolucją architektury zmieniają się w czasie i wraz z kolejnymi wersjami systemu. Raczej nie zdarza się, żeby programiści zaimplementowali wszystkie czy nawet większość funkcji już w pierwszej wersji, zwłaszcza gdy menedżer produktu potrafi przedstawić jasną i przejrzystą specyfikację tych funkcji. Problem implementacji dalszych funkcji powraca bardzo szybko. Ponieważ nie ma jeszcze wielu klientów, którzy mogliby dostarczyć informacji zwrotnej, zespół pracuje nad pozostałymi funkcjami i ich ukończenie nie sprawia dużego problemu. Cała praca koncentruje się wtedy na wprowadzeniu pełnego zestawu cech produktu. W tym okresie niewiele czasu i uwagi poświęca się zmienianiu architektury. Jest to czas, w którym pierwotna architektura nabiera dojrzałości. Oczywiście, pewne zmiany następują, mają jednak zazwyczaj dość ograniczony charakter.

Kiedy system jest w ciągłym użyciu przez trzy lub więcej kolejnych wersji, albo przez dwa lub więcej lat, jego początkowe funkcje, przewidziane przez twórców, wyczerpują swój potencjał i menedżer produktu potrzebuje w procesie planowania dalszej pracy coraz większych ilości informacji od klientów. Te informacje zwrotne, mówiące o nowych funkcjach albo znaczących modyfikacjach funkcji istniejących, wyznaczają najczęściej początek ewolucji architektury, zmuszają bowiem programistów do stworzenia w niej niezbędnych możliwości. Proste implementowanie funkcji nie jest już wystarczające.

Oczywiście, ewolucję architektury nie zawsze napędzają potrzeby zgłaszane przez klientów. Firma, która proaktywnie kieruje rozwojem swoich produktów, stale poszukuje nowych technologii i metod pozwalających uzyskać przewagę konkurencyjną. Praktyka włączania do ewoluującej architektury najważniejszych z nowych technologii może zapewnić, że taka przewaga będzie względnie trwała. Piszę o tym szerzej w rozdziale 3. W dodatku B przedstawiam język wzorców, który jest jedną z metod porządkowania takiego procesu.

Dojrzewanie i ewolucja architektury mają charakter cykliczny. Każda faza cyklu wykorzystuje osiągnięcia poprzedniej. W dojrzałym produkcie rzadko wprowadza się nowe możliwości, a tylko praktyka pozwala poznać ich rzeczywistą wartość. Dzięki informacjom zwrotnym, nowe możliwości stają się coraz bardziej dojrzałe. W wyjątkowo dobrze ukierunkowanej architekturze możemy spotkać się z tym, że pewne możliwości są usuwane.

Możliwości architektury czy dynamika funkcji

W jednym z systemów, nad którymi pracowałem, użytkownicy mogli kooperatywnie porządkować wspólny zbiór dokumentów w foldery i przeszukiwać je. Regularnie napływały też nowe dokumenty z zewnątrz. Jedna z grup wymagań dotyczyła przechowywania i wykonywania określonych zapytań, operujących na nowych dokumentach. Druga grupa wymagań mówiła, że gdy użytkownik zmodyfikuje zawartość folderu, inny użytkownik otrzyma wiadomość e-mail z opisem wprowadzonych zmian. Obie grupy wymagały *mechanizmu powiadamiania*, czyli specyficznej możliwości architektury. Do czasu zaimplementowania tej możliwości żadna z wymaganych funkcji nie mogła zostać zapewniona.

Głównym rynkiem docelowym systemu były firmy z listy Fortune 2000. Pierwotny model biznesowy opierał się na licencjach bezterminowych lub rocznych, udzielanych konkretnemu użytkownikowi lub na określoną liczbę jednoczesnych użytkowników. O ile jest to model popularny w przypadku oprogramowania dla przedsiębiorstw, uniemożliwiał sprzedaż produktu firmom prawniczym, które w rozliczeniach wymagają śledzenia użycia systemu. Dzięki współpracy z kierownictwem produktu, zdaliśmy sobie sprawę, że moglibyśmy otworzyć się na nowy segment rynku, o ile tylko zapewniona byłaby obsługa specyficznego dla niego modelu biznesowego. Niestety, zdefiniowanie nowego modelu biznesowego było znacznie łatwiejsze niż jego implementacja, która wymagała istotnych zmian w możliwościach wykorzystywanej architektury. W miejsce prostego zliczania jednoczesnych użytkowników (lub rozpoznawania ich), niezbędne było wprowadzenie kilku nowych możliwości, takich jak rejestrowanie pojedynczych zdarzeń w zabezpieczonych plikach dziennika, które potem byłyby przetwarzane przez system księgowy. Dopóki to nie było zapewnione, dostęp do nowego segmentu rynku pozostawał zamknięty.

Inny system, z którym miałem do czynienia, odpowiadał za tworzenie próbnych wersji oprogramowania. Wersje próbne to potężne narzędzie marketingowe, powstające w drodze modyfikacji gotowego produktu przy użyciu specjalnych narzędzi, zapewniających jego trwałą ochronę. Klienci naszego systemu prosili o zwiększenie swobody w doborze ograniczeń użytkowania. Niestety, tego pozornie prostego wymagania nie przewidywała wykorzystywana architektura. Musiałem więc rozpocząć wprowadzanie w niej poważnych zmian, koniecznych do wyposażenia systemu w nowe możliwości.

Typowym przykładem różnicy między funkcjami a możliwościami jest sytuacja, w której dział marketingu żąda, aby interfejs użytkownika został zlokalizowany i był dostępny w wielu wersjach językowych. Każdy język może być postrzegany jako osobna funkcja. Odpowiada to często zamierzeniom marketingowym. Jednak dopóki wykorzystywana architektura nie posiada możliwości związanych z internacjonalizacją produktu, obsługa wielu języków łatwo może stać się koszmarem prześladującym całą organizację. Można też przytoczyć wiele innych przykładów, zwłaszcza w obrębie oprogramowania dla przedsiębiorstw: przepływ pracy, elastyczne reguły sprawdzania poprawności, rosnące zapotrzebowanie na dostosowywanie aplikacji i inne.

Jest jedna szczególna sytuacja, gdy cykl dojrzewania i ewolucji musi zostać złamany, a w pracach zespołu dominują rozważania na temat możliwości, a nie funkcji. Dzieje się tak, gdy trzeba przeprowadzić całkowite przeprojektowanie (przepisanie) systemu. Może to wynikać z różnych przyczyn, ale najbardziej typową jest fakt, że system nie dysponuje wymaganymi możliwościami, a dodanie ich jest zbyt kosztowne w kategoriach czasu lub zasobów. W takiej sytuacji bardzo ważne jest, aby architekt (lub zespół architektów) systemu bardzo dobrze znał i rozumiał brakujące możliwości techniczne. Wszyscy uczestnicy procesu muszą mieć pewność, że wkrótce rozpoczną pracę, która będzie miała za podstawę nowy, lepszy fundament.

Motywacja do zmiany konstrukcji (przepisania) systemu — brak wymaganych możliwości — ma często swoje źródło w bardzo szybkim prezentowaniu rosnącej bazie użytkowników nowych funkcji. Mówiąc prościej, szybki sukces może być ojcem przyszłej porażki. Niezbędne jest sprawne reagowanie na zmieniającą się sytuację.

Degradacja architektury rozpoczyna się w bardzo prosty sposób. Gdy nacisk rynku na wprowadzanie nowych funkcji jest duży, a brakuje niezbędnych do ich implementacji możliwości systemu, skądinąd dobry i rozsądny menedżer może ulec pokusie nakłonienia programistów do zaimplementowania wymaganych funkcji bez towarzyszących im zmian w zakresie możliwości systemu. Uzasadnienia takiego działania nadchodzą ze wszystkich kierunków. Konkurencja może zwrócić uwagę ważnego klienta. Klient może odmówić aktualizacji do następnej wersji. Takie sytuacje pozbawiają zespół niezbędnych przychodów. Decyzje o dalszym rozwoju systemu nie są proste.

Niemal pewnym skutkiem takiego dobrze uzasadnionego działania jest to, że przyszły koszt pielęgnacji systemu i wprowadzania nowych funkcji będzie wysoki. Dodatkowo, ponieważ wciąż brakuje istotnych możliwości architektury, każda nowa, zależna od nich funkcja będzie równie kłopotliwa. Co gorsza, wpływ tego rodzaju sytuacji na morale zespołu prawdopodobnie zmniejszy efektywność jego pracy, co jeszcze pogłębi problem, zwłaszcza gdy rzekoma „presja rynku" okaże się mniejsza niż oczekiwano.

Wynikiem takiego implementowania nowych funkcji jest zaciąganie „długu technicznego", który musi zostać spłacony w przyszłości[1]. Kwotę zadłużenia wyznacza koszt związany z poprawną implementacją wymaganej możliwości systemu. Ta kwota wcześniej czy później musi zostać spłacona. Oprocentowanie to dodatkowe obciążenie utrzymywaniem w systemie funkcji, której nie zapewnia stosowana architektura. Oprocentowanie rośnie wraz z kolejnymi wersjami systemu, a także z nadchodzącymi żądaniami nowych funkcji. Gdy sytuacja staje się już bardzo trudna — a należy tego oczekiwać — jedynym rozwiązaniem może być wyrzucenie całej architektury „do kosza" i rozpoczęcie pracy od nowa. Jest to sytuacja, w której obsługa zadłużenia okazuje się zbyt kosztowna.

Gdy czasem mamy do czynienia z krytycznym „oknem rynku" (ang. *market window*, okres, gdy produkt ma największe szanse trafić na zainteresowanego klienta) albo po prostu musimy zaimplementować daną funkcję, bo wymaga tego ważny klient, bardzo często okazuje się, że jest to przede wszystkim iluzja narzucana przez niecierpliwe kierownictwo, nadmiernie przejęte poszukiwaniem szybkich zysków.

Gdy człowiek staje się dojrzalszy, jego obawa przed niepotrzebnym zadłużaniem się rośnie. Z długu nie można zrezygnować, można go tylko spłacić. Gdy dany zestaw wymaganych funkcji wymaga znaczących, nowych lub zmodyfikowanych, możliwości systemu, kiedykolwiek to możliwe, należy unikać zaciągania długu. Gdy rozważamy implementowanie nowych możliwości, należy pamiętać o amerykańskim powiedzeniu „nie musisz płacić od razu, możesz zapłacić później, z odsetkami i karami za spóźnienia".

Opieka nad architekturą

Poza dojrzewaniem i ewoluowaniem, procesami, którymi kierują planowane funkcje i niezbędne do ich implementacji możliwości, architektura wymaga też stałej opieki. Pod tym względem przypomina starannie zaprojektowany ogród. Jeżeli go nie doglądamy, szybko traci początkową dyscyplinę i zarasta, a rolę chwastów pełni martwy kod. Jeżeli będziemy go przez dłuższy czas ignorować, jedynym rozwiązaniem mogą okazać się bardzo poważne — i bardzo kosztowne — naprawy. Opieka nad

[1] O ile mi wiadomo, pierwszą osobą, która użyła sformułowania „dług techniczny", jest Ward Cunningham.

Zmniejszanie entropii — spłacanie długu technicznego po każdej wersji

W wirze walki o zakończenie kolejnego etapu pracy pewne kompromisy dotyczące jakości kodu i implementacji są konieczne. Nie ma przy tym znaczenia jak bardzo zespół programistów ceni sobie idee Agile Development. Aby prace nad produktem zakończyć, trzeba skoncentrować się na ich zakończeniu. To niemal zawsze oznacza kompromisy (hakowanie, brzydki kod i tym podobne rzeczy).

Bez względu na przyczyny takich kompromisów, dopóki nie zostaną one usunięte, jakość kodu źródłowego będzie ulegać degradacji. Po zakończeniu pracy nad pierwszą, lub kolejną, wersją produktu, pewną ilość czasu *trzeba* poświęcić na korygowanie kodu źródłowego. Nazywam to „redukcją entropii" (*ER*, ang. *entropy reduction*). Proces ten przynosi ważkie korzyści. Kilka z nich wymieniam poniżej (w dość przypadkowej kolejności):

- Zapewnia, że jakości kodu źródłowego poświęca się wystarczająco dużo uwagi. Programiści dobrze znają znaczenie jakości kodu na najniższych jego poziomach. Dla firmy oznacza to redukcję ryzyka związanego z kodem, którego dalsza pielęgnacja i rozwój są niemożliwe lub bardzo trudne.
- Zapewnia wysokie morale zespołu. Dobrzy programiści krzywią się, gdy muszą iść na kompromis „tylko po to", żeby produkt był gotowy w terminie. Zrobią, co do nich należy, ale nigdy tego w pełni nie zaakceptują. Jeżeli nie pozwolimy im wrócić do wcześniejszej prowizorki i posprzątać, w przyszłości temat jakości stanie się dla nich „martwy". Gdy ich wrażliwość na jakość zginie, umrze też produkt.
- Znacznie poprawia możliwości pielęgnowania i rozbudowy systemu.
- Pozwala zespołowi dobrze poznać określone grupy zachowań systemu i rozpoznać obszary wartościowych zmian.
- Pozwala poświęcić stosowną uwagę wszelkim niespójnościom związanym z takimi aspektami jak standardy pisania kodu.

Redukcja entropii *nie oznacza* wprowadzania dużych zmian w architekturze czy wprowadzania nowych funkcji. Jej celem jest poprawienie wewnętrznej struktury systemu z zachowaniem jego zewnętrznych zachowań.

Redukcję entropii planuje się zazwyczaj bezpośrednio po dostarczeniu kolejnej wersji, w większości przypadków średnio co 9 do 12 miesięcy. W każdym takim cyklu należy dbać o systematyczne refaktoryzowanie kodu, odpowiednio do wyłaniających się potrzeb. Redukcja entropii to najczęściej refaktoryzacje, które zostały odłożone na później ze względu na ich rozmiar, złożoność lub goniące terminy.

Ważne jest, aby inicjowaniu sesji ER towarzyszyło ustalenie stosownych reguł i ogólnych zasad. Najważniejszą z nich jest ta, że w trakcie redukowania entropii nie wprowadza się nowych funkcji czy możliwości. Aby ułatwić trzymanie się tej zasady, warto zadbać o chwilową izolację kierownictwa firmy od programistów. Etap ER powinien być etapem „tylko dla programistów".

Niezbędne są oczywiście testy funkcjonalne (jeżeli istnieją). *Nie wolno* prezentować nowej wersji produktu, po ER, bez pełnych, kompletnych testów regresyjnych.

Stosunek programistów do ER bywa różny. Jeżeli jest to dla nich zwykły element pracy, nie ma większych problemów. Osoby zanadto entuzjastyczne mogą próbować wykorzystać ten etap do zmiany architektury. Bądź czujny! Zespoły, które utraciły poczucie jakości i zapał do kształtowania architektury w wyniku złego zarządzania, mogą wymagać stosowania prawdziwego przymusu („Wiem, że cała ta historia z ER może wydawać się dziwna, ale po prostu w taki sposób

pracuję, a skoro wcześniejsze podejście nie prowadzi do zadowalających wyników, spróbujemy czegoś nowego").

Przed rozpoczęciem właściwych zmian zespół powinien przygotować i wspólnie przejrzeć plan, w którym zestawione są konkretne moduły i (lub) funkcje, które zostaną poprawione. Jest to o tyle ważne, że niekiedy proponowane zmiany są zbyt rozległe, aby wprowadzić je w ciągu jednego lub dwóch tygodni, a to jest czas, który zazwyczaj przeznaczam na sesję ER. Warto też zwrócić uwagę na kilka dobrych praktyk. Ich wartość zawsze znajdowała potwierdzenie w moich doświadczeniach.

- *Korzystanie ze znaczników w kodzie.* Znaczniki umieszczane w kodzie mogą posłużyć jako sposób opisywania fragmentów, które powinny zostać skorygowane w trakcie przyszłej sesji ER. Zespoły, z którymi pracowałem, korzystały z haseł TODO (do zrobienia), REDO (do zmiany), a także HACK (wytrych). Wyszukiwanie takich znaczników nie sprawia kłopotu.
- *Wyznaczanie rytmu.* Rytm kolejnych wersji jest dla mnie bardzo ważny. Zespoły osiągające dobre wyniki wypracowują przyjemne, dobre tempo, towarzyszące im przez większość czasu. Przypomina ono szybkie spacery — spokojne w swoim zrównoważeniu, a zarazem miłe. W końcowym okresie pracy nad nową wersją, gdy dotrzymanie terminu staje się sprawą palącą, programiści pracują odpowiednio do potrzeb, nawet jeżeli oznacza to 80-godzinny tydzień pracy. Jest to jak końcowy sprint długodystansowego biegacza. Później zespół musi dojść do siebie. Po oddaniu nowej wersji przychodzi czas odpoczynku lub pracy nad ER (która *nie powinna* być dużym wyzwaniem intelektualnym).
- *Ograniczanie ER pewnym limitem czasu.* Najlepiej sprawdza się okres jednego lub dwóch tygodni. Kiedyś spróbowałem wydłużyć go do trzech tygodni, ale wtedy było to już zbyt trudne do kontrolowania — programiści zaczynali dodawać nowe funkcje. Jeżeli okazuje się, że efektywna sesja ER musi trwać cztery tygodnie lub dłużej, nie pozostaje nic innego jak spojrzeć w lustro i powiedzieć sobie „Ta architektura umarła".
- *Nieprezentowanie wyników sesji ER.* Wynikiem procedur zmniejszania entropii mogą być zmiany w całym kodzie. Sam fakt, że praca zautomatyzowanych testów jednostkowych nie została zakłócona, nie oznacza, że można przedstawić kolejną wersję bez pełnych testów regresyjnych.
- *Dbałość o język.* Używałem terminów „dług techniczny" i „redukcja entropii", mając na myśli zasadniczo jedną i tę samą rzecz. Różne osoby różnie odbierają takie określenia. Zanim zaczniemy stosować pojęcie „redukcji entropii", sprawdźmy, czy sprawdza się ono w danym zespole. Jeżeli nie, rozważmy inne, jak „spłacanie długu technicznego" albo „poprawianie architektury".
- *Wykorzystywanie ER do podkreślania znaczenia innych, ważnych wartości.* Sesje ER można wykorzystać do wpajania wielu innych zasad pracy, takich jak utrzymywanie jakości kodu, terminowość wykonywania zadań, odpowiedzialne zarządzanie („pozwól sobie na mały dług techniczny teraz, a będziesz miał czas, żeby spłacić go po oddaniu wersji"). Wraz z pierwszą sesją można wprowadzić pewne artefakty kultury, takie jak „gra w chaos" lub zabawka (symbol).
- *Zobowiązania muszą mieć pokrycie.* Jeżeli obiecujemy sesję ER, musimy dotrzymać tej obietnicy. Zdarzyło mi się niemal całkowicie stracić wiarygodność w sytuacji, kiedy szef firmy zażyczył sobie, aby odłożyć sesję ER i zająć się implementacją nowej funkcji, oczekiwanej przez ważnego klienta. Niezbędne były długie negocjacje i kilka gorących spotkań, ale ostatecznie sesja ER została przeprowadzona jeszcze przed wprowadzeniem funkcji. Warto zawczasu zabezpieczyć się przed takimi kłopotami i zadbać o to, aby przełożony docenił znaczenie etapu redukcji entropii.

architekturą nie oznacza dodawania nowych funkcji czy możliwości. Jest dbaniem o utrzymanie ich elegancji i ładu.

Poniżej wymieniam kilka sił, które mają duże znaczenie dla kształtowania architektury. Będę powracał do nich jeszcze w dalszych rozdziałach.

Nadążanie za rozwojem

Każda złożona aplikacja współdziała z wieloma różnymi technologiami. Nadążanie za ich rozwojem i zmianami w miarę ewolucji produktu jest o tyle ważne, że pozwala uniknąć kosztownego przebudowywania architektury. Zmiany w technologiach mają często kluczowe znaczenie dla możliwości dostarczenia użytkownikom ważnych funkcji. Wygrywamy przy tym podwójnie. Nie skarży się też dział marketingu, bo dzięki temu hasło „nowe, lepsze" nabiera treści.

Dług techniczny

Programiści nieustannie stają przed problemem pogodzenia terminowości pracy nad bieżącą wersją ze stosowaniem rozwiązań, które sprawdzą się w dłuższym okresie. Dobre zespoły wiedzą, kiedy i jak stosować kompromisy, niezbędne do dotrzymania zobowiązań. Mówiąc prościej, dobry zespół wie, kiedy goniący termin uzasadnia zastosowanie brzydkiego „hakowania". Problem takich ścieżek na skróty nie jest problemem bieżącej wersji (bo ta właśnie może ich potrzebować). Dotyczy wersji przyszłych, kiedy wcześniejszy *kompromis* przeradza się w kosztowny problem.

Takie kompromisy to jeszcze jedna odmiana długu technicznego, podobna do tego, który zaciągamy przy implementowaniu funkcji, pomijając implementowanie odpowiednich możliwości architektury. Właściwa opieka nad architekturą to po części spłacanie tego długu.

Znane błędy

Każda złożona aplikacja zawiera pewne błędy, przypominające trudne do wyplenienia chwasty. Pozostawienie ich w ogrodzie, zwłaszcza gdy są na tyle duże, że rzucają się w oczy, może odbić się niekorzystnie na dalszej pracy zespołu. Dajmy programistom trochę czasu na usunięcie błędów, o których już wiedzą, a na pewno będą bardziej zadowoleni ze swojej pracy, a i architektura na tym skorzysta. Zaszczepimy zarazem zasadę zmian pozytywnych — każda zmiana powinna prowadzić do lepszego ogólnego stanu systemu.

Zgodność komponentów licencjonowanych

Złożone aplikacje korzystają z krytycznych dla ich działania komponentów różnych producentów. Zgodnie z zasadą nadążania za rozwojem, dotrzymanie kroku postępowi wymaga aktualizowania tych komponentów. Oczywiście, czasem nowa technologia może nie być potrzebna i lepiej skierować wysiłki w innych kierunkach. Konieczne jest jednak utrzymywanie czujności — jeżeli będziemy wstrzymywać aktualizacje zbyt długo, pojawią się problemy ze zgodnością z najnowszymi wersjami.

Pamiętajmy o tym, żeby zapoznać się z każdym uaktualnieniem. W pewnym momencie nowa wersja i zmiana architektury mogą być potrzebne przede wszystkim po to, aby utrzymać zgodność. Szersze omówienie tego tematu przedstawię w rozdziale 5.

Zachęcam Czytelnika, aby uzupełnił powyższą listę o dalsze kategorie. Należy jednak uważać, aby nie mylić radykalnych zmian w zestawie funkcji, wymagających podobnie radykalnych zmian w architekturze, ze zmianami podobnymi do opisanych powyżej. Skalowanie aplikacji stosowanej w jednym dziale firmy i przeznaczonej do obsługi 100 użytkowników do poziomu aplikacji korporacyjnej dla 1000 użytkowników albo dostosowanie aplikacji, której model biznesowy opiera się na licencjach rocznych, do pobierania opłat za transakcje *nie jest* zmianą, o jakiej tu piszę! Takie zmiany wymagają zupełnie nowych możliwości i znaczących modyfikacji architektury.

Ogólne zasady

Tworzenie, dojrzewanie, ewolucja i pielęgnowanie architektury wymagają podejmowania przez architektów i programistów wielu różnych decyzji. Ich waga jest dość zróżnicowana. Niektóre są proste i nie mają większego wpływu na architekturę. Inne prowadzą do głębokich zmian. Każdy programista, a tym bardziej architekt oprogramowania, musi stale doskonalić umiejętność rozpoznawania lepszych rozwiązań. Oznacza to, oczywiście, że musi dysponować pewnymi kryteriami oceny alternatyw i odpowiednio je stosować. Poniżej wymieniam kilka zasad projektowania, które wytrzymały próbę czasu.

Hermetyzacja

Architekturę organizuje się w odrębne i względnie niezależne części, stosując zasadę ukrywania ich implementacji. Dobrym przykładem jest system rozliczeń telefonicznych. Jedna z części tego systemu (centrala) odpowiada za ustanawianie, zarządzanie i śledzenie połączeń telefonicznych. Prowadzi to do generowania szczegółowych danych transakcji, opisujących poszczególne rozmowy (kto do kogo dzwonił, jak długo rozmawiał itp.). Te dane są przekazywane do różnych systemów bilingowych, które wykonują złożone obliczenia prowadzące do określenia wysokości opłat w oparciu o umowę między operatorem telefonicznym a klientem.

Wewnątrz każdego z tych dużych, zahermetyzowanych systemów występują dalsze poziomy hermetyzacji. Przykładowo, system bilingowy będzie miał zapewne kilka osobnych modułów do obliczania podstawowej wartości rachunku, odliczania zniżek, doliczania podatków i tak dalej.

Interfejsy

Interakcje między podsystemami rozbudowanej architektury są ściśle zdefiniowane. W idealnym przypadku charakter tej definicji pozwala w dużej mierze zachować jej niezmienność przez cały okres użytkowania aplikacji. Jedną z dróg prowadzących do tego celu jest operowanie abstrakcjami, które stoją ponad konkretną implementacją. Programowanie oparte na takich abstrakcjach ułatwia wprowadzanie modyfikacji, gdy okazuje się, że implementacja wymaga zmiany.

Wyobraźmy sobie program, który początkowo został zaprojektowany tak, aby zapisywać wyniki pracy do pliku. W miejsce programowania bezpośrednio pod kątem interfejsu pliku, piszemy procedury

wyjściowe, dostosowując je do abstrakcyjnego interfejsu, w wielu językach określanego mianem strumienia wyjściowego. Dzięki temu program może kierować dane wyjściowe do strumienia plikowego, standardowego, przechowywanego w pamięci strumienia znakowego lub gdziekolwiek indziej, o ile tylko zachowana jest zgodność z interfejsem. Dotyczy to w równej mierze przypadków wyprowadzania danych wyjściowych w sposób pierwotnie nieprzewidywany najważniejszym zagadnieniem jest zdefiniowanie ogólnego interfejsu. Aby to zrobić, niezbędne. Oczywiście, jest odpowiednie doświadczenie.

Innym obszarem, w którym zasada stosowania stabilnych interfejsów mocno wpływa na konstrukcję systemu, jest izolacja tych aspektów, po których oczekujemy największych zmian. Przykładem mogą być takie interfejsy programowania jak ODBC. Stosując ODBC, programista izoluje system od typowej zmiany. Wymiana systemu bazodanowego na inny nie sprawia wówczas problemu, o ile tylko jest to system oparty na SQL. Z drugiej strony, nie można zapominać, że taka elastyczność ma swoją cenę — rezygnujemy z tego, co oferują interfejsy programowania specyficzne dla różnych produktów.

Słabe powiązania

Terminy **słabe powiązania** (ang. *loose coupling*) i **silne powiązania** (ang. *tight coupling*) odnoszą się do ilości powiązań między elementami systemu. Ogólnie, elementy słabo powiązane łatwiej analizować, testować, ponownie wykorzystywać i pielęgnować, bo mogą być odizolowane od innych części systemu. Słabe powiązania ułatwiają też równoległe implementowanie komponentów. Warto zwrócić uwagę, że stosowanie dwóch wymienionych wcześniej zasad sprzyja słabym powiązaniom.

Właściwa ziarnistość

Jednym z głównych wyzwań związanych z zasadą słabych powiązań jest ziarnistość. Przez **ziarnistość** (ang. *granularity*) rozumiem ilość pracy, jaką wykonuje jeden składnik. Komponenty luźno powiązane mogą być łatwe w analizie, testowaniu, ponownym wykorzystaniu i pielęgnacji, gdy są izolowane, ale jeżeli ziarnistość jest zbyt duża, budowanie systemu będzie *trudniejsze*, ponieważ aby osiągnąć użyteczny w praktyce efekt, trzeba połączyć wiele z nich. Właściwy poziom ziarnistości zależy od wykonywanych przez składniki zadań.

Wysoka kohezja

Kohezja (ang. *cohesion*) określa, jak bardzo są ze sobą związane różne czynności wykonywane przez jeden składnik lub ich grupę. Składnik o wysokiej kohezji to taki, którego elementy łączy duży stopień pokrewieństwa. Najwyższą kohezję mają składniki, których elementy służą realizacji jednego zadania.

Parametryzacja

Składniki mogą być hermetyzowane, ale nie oznacza to, że wykonują swoją pracę bez żadnych parametrów czy konfiguracji. Najbardziej efektywne komponenty to takie, które wykonują właściwą ilość pracy w oparciu o dobrze dobraną liczbę i rodzaje parametrów. W wielu platformach i architekturach

rozszerzeń stosuje się wyszukaną formę parametryzacji określaną nazwą „odwrócenie sterowania" — mamy z nią do czynienia wtedy, gdy jeden składnik przekazuje sterowanie do innego.

Opóźnianie

Praca architekta oprogramowania i programisty często prowadzi do sytuacji, w której trzeba podjąć trudną decyzję, a brakuje czynników przekonujących, że ten czy inny wybór będzie dobry. Problem ten może mieć podłoże techniczne, jak przy wybieraniu biblioteki, gdy nie wiemy, który z produktów realizujących te same funkcje sprawdzi się najlepiej. Może też mieć podłoże czysto biznesowe, jak wtedy, gdy trwają pertraktacje z dwoma dostawcami, mające na celu ustalenie najlepszych warunków licencjonowania.

Opóźnianie takich decyzji tak długo, jak to możliwe, zwiększa szansę, że gdy w końcu zapadną, będą faktycznie dobre. Oczywiście, nie można odkładać decyzji w nieskończoność. Wówczas z pomocą przychodzą wymienione w tym podrozdziale zasady, bo dzięki nim skutki każdego wyboru pozostają wyraźnie oddzielone od całości systemu.

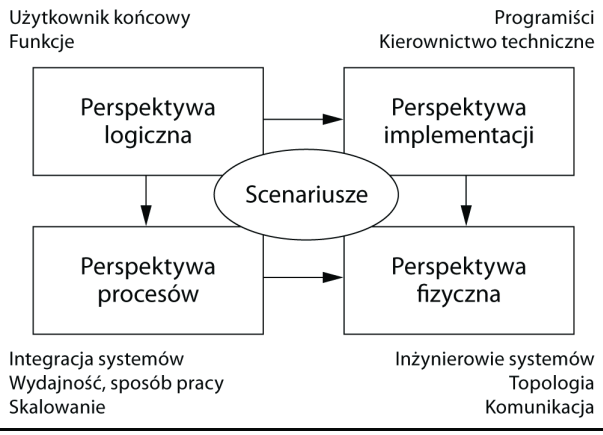
Pełne zrozumienie architektury

Powróćmy do przedstawionych wcześniej definicji architektury oprogramowania. Ważnym elementem pracy zespołu programistów jest to, aby dysponował on metodami identyfikowania, opisywania, wymiany informacji i modyfikowania architektury, które nie opierają się wyłącznie na kodzie źródłowym. Aby faktycznie pracować na poziomie *całościowym*, musi istnieć jeden lub kilka modeli architektury. Możliwości jest wiele. W mojej praktyce sprawdzało się co najmniej kilka z nich. W ostatnim czasie stosuję model Rational 4+1. Łączy on kilka z najbardziej praktycznych modeli w spójny i wygodny system (choć na pewno warto, aby Czytelnik znał też inne systemy).

Model Rational 4+1 opiera się przede wszystkim na potrzebach uczestników procesu wytwarzania oprogramowania i zaleca stosowanie czterech głównych perspektyw architektury, przedstawionych na rysunku 1.1. Philippe Krutchen, twórca modelu, definiuje te perspektywy następująco:

- **Perspektywa logiczna.** Perspektywa logiczna to *statyczny* układ relacji pomiędzy obiektami i innymi jednostkami wykorzystywanymi w trakcie pracy nad systemem. Może to oznaczać dwie lub więcej różnych reprezentacji, z których podstawowe to model *abstrakcyjny* i realizacja tego modelu w schemacie bazy danych.
- **Perspektywa procesów.** Opisuje aspekty związane z pracą współbieżną i synchronizacją.
- **Perspektywa fizyczna.** Opisuje odwzorowanie oprogramowania na poziomie sprzętowym i obejmuje rozmieszczenie składników służących osiągnięciu takich celów jak ciągłość pracy, niezawodność, odporność na uszkodzenia i wydajność.
- **Perspektywa implementacji.** Opisuje statyczną organizację oprogramowania w środowisku programowania.

Cztery wymienione widoki mają służyć projektantom oprogramowania do organizowania podejmowanych decyzji dotyczących architektury systemu. Aby ułatwić jej zrozumienie, każdy widok może być opisany w kontekście kilku podstawowych przypadków użycia (które wspólnie tworzą perspektywę „+1").



RYSUNEK 1.1.
Model Rational 4+1

W rzeczywistości, podejście to powinno być raczej określane jako „*n*+1", ponieważ nic nie stoi na przeszkodzie, aby architekt użył innych perspektyw. Potrzeba korzystania z wielu perspektyw jest zawsze podkreślana jako bardzo istotna dla poprawności modelowania. O ile mi jednak wiadomo, *nikt* nie proponuje dogmatycznego stosowania *trzech* lub *pięciu*. Dużo ważniejsze jest dobre zrozumienie wartości i cech każdej perspektywy i korzystanie z niej odpowiednio do potrzeb.

Zastrzeżenie to nie powinno być zaskakujące, bo koncepcja stosowania wielu modeli złożonego systemu jest dość popularna. Zdarzało mi się budować i przebudowywać domy — nauczyło mnie to dokładnego studiowania wszystkich modeli dostarczonych przez architekta: od planu ogólnego działki, poprzez plany instalacji elektrycznej, oświetleniowej i poszczególnych pięter, po plan instalacji hydraulicznej. Wykwalifikowani wykonawcy, mimo swojej specjalizacji, potrzebują wszystkich tych modeli. Z kolei ja, jako odbiorca, chciałbym mieć pewność, że zbudowany dom będzie dokładnie taki, jakiego oczekuję.

Znaczenie planów i różnych perspektyw architektury polega na tym, że dostarczają zespołowi artefaktów, które pozostają względnie stałe. Mówiąc ogólnie, elementy architektury, których opisywanie jest warte czasu, energii i pieniędzy, dotyczą przede wszystkim stosunkowo stabilnych aspektów problemu, dziedziny problemu, modelu biznesowego (patrz rozdział 4.) i wyznaczanych przez nie, charakterystycznych aspektów realizacji technicznej.

Zespół

Procesów tworzenia, rozwijania i ewolucji architektury nie można oddzielić od zagadnień związanych z pracującym nad kolejnymi wersjami zespołem. W tym podrozdziale przedstawię tylko kilka uwag dotyczących zależności między architekturą a zespołem. Pełniejsze omówienie można znaleźć w innej mojej książce, *Journey of the Software Professional* [Hohmann, 1996].

Naturalnym początkiem rozważań na temat zespołu będzie grupa programistów, która jako pierwsza zaczęła pracę z systemem. Jako jego twórcy, podejmują oni decyzje projektowe, opierając się głównie na swoim doświadczeniu. Podczas gdy późniejsze zespoły muszą, w ten czy inny sposób, radzić sobie z decyzjami podjętymi przez pierwszy, twórcy systemu nie stykają się z tym problemem (jest to zresztą powód, dla którego programiści wolą pracować z nowymi systemami). Jest to przyczyna powodująca, że każda nowa architektura, nawet jeżeli jest oparta na pewnym wzorcu, w dużej mierze odbija doświadczenie jej twórców. Liczba osób tworzących początkowy zespół i relacje między nimi

mają bardzo duży wpływ na pierwotną wersję oprogramowania. Mówiąc najprościej, nie można oddzielić zespołu od tworzonej przez niego architektury.

Praktyka pokazuje, że w tworzeniu pierwszej wersji architektury powinien uczestniczyć możliwie niewielki zespół programistów. Pozwala to zminimalizować obciążenie zagadnieniami związanymi z komunikacją i uzyskać wysoką kohezję. W miarę definiowania, konstruowania i retrospektywnej oceny podsystemów, następuje naturalny podział zakresów odpowiedzialności odpowiednio do umiejętności poszczególnych członków zespołu. Ogólnych zaleceniem jest ograniczenie wielkości początkowego zespołu do od trzech do dziesięciu programistów. Osobiście polecałbym dalsze ograniczenie tej liczby do trzech – pięciu osób. Jednej z nich powinna zostać przydzielona rola architekta. Jest to ważne dla sprawnej pracy zespołu (pełne omówienie roli architekta przedstawiam w rozdziale 3.).

Po stworzeniu pierwszej wersji architektury zespół zazwyczaj rozrasta się, co jest wynikiem dużego zapotrzebowania na nowe funkcje i możliwości. Naturalną drogą rozbudowywania zespołu jest zachowanie podziałów wprowadzonych wcześniej. Przykładowo, do pracy nad interfejsem użytkownika, początkowo wykonywanej przez jedną osobę, można w kolejnej wersji przydzielić trzy. Podobnie, grupa zajmująca się bazą danych może rozrosnąć się do dwóch osób. W ten sposób podzespoły powstają niemalże spontanicznie, a proces ten sprzyja utrzymaniu początkowych założeń dotyczących architektury. Zaletą takiego modelu jest to, że członek początkowego zespołu, który staje się członkiem podzespołu, zachowuje orientację w ogólnej konstrukcji systemu i może przekazywać tę wiedzę nowym programistom.

Proces powiększania zespołu trwa aż do czasu, gdy zarówno zespół, jak i architektura uzyskują stabilność. Czasem granicę może wyznaczać kierownictwo projektu. Niektóre firmy z zasady ograniczają liczbę członków zespołu do pewnej z góry narzuconej liczby, aby zachować możliwość płynnej, otwartej i efektywnej komunikacji. Inne pozwalają na nieograniczone zwiększanie liczebności zespołów, odpowiednio do zgłaszanych potrzeb. Pracowałem jako konsultant w jednym z projektów ministerstwa obrony, gdzie zespół liczył ponad 150 programistów C++, podzielonych na około dziesięć grup, zajmujących się różnymi podsystemami. Choć był to stosunkowo duży zespół, pozostawał on niezwykle efektywny — rozmiary i struktura były ściśle dostosowane do wymagań stwarzanych przez rozwiązywany problem.

Najważniejsze w treści tego podrozdziału jest stwierdzenie, że zespół i architektura są zawsze bardzo ściśle ze sobą związane. Zarówno zespół wpływa na architekturę, jak i architektura wpływa na pracę programistów.

❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑

Podsumowanie rozdziału

- Architektura dotyczy przede wszystkim *całościowego* obrazu oprogramowania.
- Struktura zespołu i struktura systemu są ze sobą nierozerwalnie związane.
- Architektura dotyczy zagadnień technicznych i nietechnicznych.
- Podstawowe czynniki, które sprawiają, że architektura warunkuje powodzenie projektu to:
  - Trwałość — typowa architektura istnieje dłużej niż zespół, który ją stworzył.
  - Stabilność — stabilna architektura jest niezbędnym fundamentem implementowanych funkcji i uzyskiwanych profitów.
  - Przewaga konkurencyjna — dobra architektura to trwała przewaga konkurencyjna.
  - Doskonałym początkiem nowego oprogramowania mogą być znane wzorce architektury.

- Ewolucja architektury to zmiany zakresu implementowanych funkcji i możliwości. Funkcja jest tym, co produkt może lub powinien robić. Możliwość to dostosowanie architektury do implementowania pewnego zestawu funkcji.
- Architektura, podobnie jak ogród, wymaga pielęgnacji.
- Należy uważać na osoby, które mają się za architektów po stworzeniu jednej tylko wersji systemu.

Warto sprawdzić

- ❑ Czy zależności między składnikami podsystemu są przejrzyście określone?
- ❑ Czy każda osoba w zespole pracuje nad podsystemem, który uważa za interesujący?
- ❑ Czy każda osoba w zespole pracuje w sposób uznawany przez wszystkich za dobry dla produktywności grupy?
- ❑ Czy architektura jest dochodowa?
- ❑ Czy wiemy, co jest najważniejsze w bieżącej wersji systemu: ewolucja czy nowy poziom dojrzałości architektury?
- ❑ Czy zdajemy sobie sprawę z poziomu długu technicznego zaciągniętego i zaciąganego w trakcie pracy nad systemem? Czy istnieją standardy oznaczania takiego długu (czy na przykład umieszczamy w kodzie specjalne znaczniki wskazujące miejsca do poprawienia)?
- ❑ Czy zachowujemy właściwy poziom zgodności z wykorzystywanymi składnikami licencjonowanymi (patrz rozdział 5.)?
- ❑ Czy architekt jawnie określił zasady, którymi kierował się przy podejmowanych decyzjach projektowych?
- ❑ Czy zespół ma właściwą liczebność, aby osiągnąć cele? Nie może być zbyt duży ani zbyt mały.

Warto spróbować

(1) Jedną z możliwości podsumowania mojego podejścia do architektury oprogramowania jest stwierdzenie, że postrzegam architekturę jako twór o naturze *organicznej*. Jest to informacja o tym, że stosuję w zarządzeniu pewien zespół praktyk, obejmujących takie rzeczy jak to, że przed dostarczeniem kolejnej wersji zawsze wymagam poświęcenia pewnej ilości czasu na pielęgnowanie architektury. Jakie jest Twoje podejście do architektury oprogramowania? Jak wpływa ono na stosowane przez Ciebie praktyki zarządzania?

(2) Oto ćwiczenie, które bardzo pomagało mi w pracy z zespołami sprawiającymi pewne problemy. Jest ono pomocne zwłaszcza w identyfikowaniu potencjalnych źródeł nieporozumień i konfliktów. Polega na tym, że dajemy każdemu członkowi zespołu czystą kartkę i prosimy o narysowanie architektury systemu. Programiści muszą tę architekturę *narysować* — nie mogą po prostu wyszukać w komputerze starego diagramu i wydrukować go. Każdy rysunek musi zostać stworzony samodzielnie, bez konsultacji z innymi członkami zespołu. Po zakończeniu ćwiczenia przyklejamy wszystkie rysunki do ściany, tworząc w ten sposób „galerię architektur". Co zwraca Twoją uwagę i uwagę członków zespołu, gdy przeglądają te rysunki? Czy ich treść jest zgodna ze sobą? Jeżeli nie, dlaczego tak jest?

(3) Czy w widocznym miejscu wisi graficzna reprezentacja architektury? Jeżeli nie, dlaczego? Jeżeli tak, kiedy ostatni raz była aktualizowana?

(4) Czy potrafisz wyliczyć możliwości bieżącej architektury?
(5) Przejrzyj definicje architektury oprogramowania dostępne pod adresem *http://www.sei.cmu.edu/*. Które z tych definicji najbardziej Ci odpowiadają? Dlaczego?
(6) Jakie konkretnie artefakty lub perspektywy architektury przygotowałeś? Czemu one służą?