Microsoft

KOD DOSKONAŁY 2

Wydanie drugie

Jak tworzyć oprogramowanie pozbawione błędów

Kultowy podręcznik tworzenia doskonałego oprogramowania!

- Twórz wolny od błędów, najwyższej jakości kod
- Utrzymuj stałą kontrolę nad złożonymi projektami
- Wcześnie wykrywaj i rozwiązuj problemy
- Sprawnie rozwijaj i poprawiaj oprogramowanie

Steve McConnell

Helion

Tytuł oryginału: Code Complete

Tłumaczenie: Paweł Koronkiewicz

ISBN: 978-83-246-5571-7

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Napisali o tej książce

„Doskonały podręcznik poświęcony dobrym praktykom programowania".

— Martin Fowler, *Refaktoryzacja*

„*Kod doskonały* Steve'a McConnella to… najkrótsza droga do prawdziwej wiedzy o programowaniu… Jego książki są przyjemne w lekturze i ani na chwilę nie pozwalają zapomnieć, że autor pisze z pozycji osoby posiadającej wieloletnie doświadczenie w pracy programisty".

— Jon Bentley, *Perełki programowania*

„Jest to po prostu najlepsza książka o budowie oprogramowania, z jaką miałem do czynienia. Jej egzemplarz powinien posiadać każdy programista. Więcej — powinien raz do roku czytać ją całą od deski do deski. Robię tak od dziewięciu lat i wciąż uczę się czegoś nowego!"

— John Robbins, *Debugging Applications for Microsoft .NET and Microsoft Windows*

„Współczesne oprogramowanie *musi* być niezawodne i bezpieczne, a budowa bezpiecznych programów to przede wszystkim uporządkowany proces programowania. Po dziesięciu latach wciąż trudno na tym polu o poważniejszy autorytet niż *Kod doskonały*".

— Michael Howard, dział technologii zabezpieczeń Microsoft Corporation; współautor *Writing Secure Code*

„Całościowy i bardzo szczegółowy przegląd zagadnień związanych z budową wysokiej jakości programów. McConnell omawia szeroką gamę tematów takich jak architektura, konwencje pisania kodu, testowanie, integracja i natura programistycznego rzemiosła".

— Grady Booch, *Object Solutions*

„*Kod doskonały* Steve'a McConnella jest kompletną encyklopedią programowania. Podtytuł *Jak tworzyć oprogramowanie pozbawione błędów* doskonale opisuje zawartość tej niemal 900-stronicowej książki. Jak pisze autor, ma ona zmniejszyć lukę między »wiedzą ekspertów ze środowisk branżowych i akademickich« (na przykład Yourdona i Pressmana) a codzienną praktyką, a także pomóc »tworzyć oprogramowanie wyższej jakości oraz wykonywać swoją pracę szybciej i napotykać mniejszą liczbę problemów«. Egzemplarz książki McConnella powinien posiadać każdy programista. Jej treść i sposób ujęcia tematu pozostają na każdej stronie niezmiennie praktyczne".

— Chris Loosley, *High-Performance Client/Server*

„Przełomowa książka Steve'a McConnella *Kod doskonały* to jedna z najbardziej przystępnych prac szczegółowo omawiających metody budowy oprogramowania…"

— Erik Bethke, *Game Development and Production*

„Kopalnia praktycznych informacji i porad dotyczących projektowania i tworzenia dobrego oprogramowania".

— John Dempster, *The Laboratory Computer: A Practical Guide for Physiologists and Neuroscientists*

„Jeżeli poważnie podchodzisz do rozwijania swoich umiejętności, musisz mieć książkę *Kod doskonały* Steve'a McConnella".

— Jean J. Labrosse, *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C*

„Steve McConnell napisał jedną z najlepszych książek poświęconych budowie oprogramowania, a niezwiązanych z konkretnym środowiskiem… *Kod doskonały*".

— Kenneth Rosen, *Unix: The Complete Reference*

„Mniej więcej raz na pół wieku pojawia się pozycja, która podsumowuje wiele lat doświadczeń i skokowo skraca proces nauki… Trudno oddać w słowach to, jak doskonała jest ta książka. *Kod doskonały* to za mało".

— Jeff Duntemann, *PC Techniques*

„Microsoft Press wydał dzieło, które należy uznać za pełny, całościowy opis tematu budowy oprogramowania. Książka ta powinna znaleźć się na półce każdego programisty".

— Warren Keuffel, *Software Development*

„Tę wspaniałą książkę powinien przeczytać każdy programista".

— T.L. (Frank) Pappas, *Computer*

„Jeżeli chcesz być zawodowym programistą, wydatek na zakup tej książki może być najlepszą inwestycją Twojego życia. Nie trać czasu na czytanie tej recenzji: po prostu idź do sklepu i kup książkę McConnella. Twierdzi on, że stara się zmniejszyć lukę między wiedzą ekspertów ze środowisk branżowych i akademickich a codzienną praktyką… Najciekawsze jest to, że udaje mu się osiągnąć ten cel".

— Richard Mateosian, *IEEE Micro*

„Książka *Kod doskonały* powinna być lekturą obowiązkową dla każdego, kto zajmuje się programowaniem".

— Tommy Usher, *C Users Journal*

„Muszę odstąpić nieco od moich zwyczajów i wyjść przed szereg. Gorąco polecam książkę Steve'a McConnella *Kod doskonały*. Zastąpiła ona u mnie opisy API jako książka, która leży najbliżej klawiatury".

— Jim Kyle, *Windows Tech Journal*

„Ta doskonale napisana, choć dość gruba książka jest najlepszą jak dotąd publikacją poświęconą praktycznym aspektom pisania oprogramowania".

— Tommy Usher, *Embedded Systems Programming*

„Jest to najlepsza książka o inżynierii oprogramowania, jaką znam".

— Edward Kenworth, *.EXE Magazine*

„Jest to książka, która zasługuje na miano klasyki i która powinna być lekturą obowiązkową dla wszystkich programistów oraz osób kierujących ich pracą".

— Peter Wright, *Program Now*

Mojej żonie Ashlie, która nie ma wiele wspólnego z programowaniem,
natomiast wypełnia w niezwykły sposób każdą inną sferę mojego życia.

Spis treści

Wstęp

> *Przepaść między najlepszymi praktykami inżynierii oprogramowania a codzienną pracą w tej dziedzinie jest ogromna — prawdopodobnie nie ma ona sobie równych w żadnej innej dyscyplinie technicznej. Narzędzie pozwalające upowszechniać dobre praktyki byłoby bardzo cenne.*
>
> *— Fred Brooks*

Głównym celem tej książki jest zmniejszenie luki między codzienną praktyką programowania a wiedzą ekspertów ze środowisk branżowych i akademickich. Do jej napisania skłoniło mnie spostrzeżenie, że bardzo wiele metod programowania o ogromnych, nieznanych wcześniej możliwościach przez lata dostępnych jest jedynie czytelnikom artykułów publikowanych w czasopismach akademickich i specjalistycznych.

Choć na polu najbardziej zaawansowanych technik budowy oprogramowania poczyniono w ostatnim czasie znaczne postępy, w codziennej praktyce niewiele się zmieniło. Nie brakuje programów z błędami, spóźnionych, takich, które nie zmieściły się w planowanym budżecie, i takich, które nie spełniają oczekiwań użytkowników. Badacze akademiccy i zatrudniani przez prywatne firmy znają metody pracy eliminujące większość problemów procesu wytwarzania oprogramowania, z których wiele rozpoznano już w latach siedemdziesiątych. Ponieważ jednak wiedza o tych metodach rzadko opuszcza strony specjalistycznych czasopism, są one stosunkowo rzadko wykorzystywane w praktyce. Różne badania wykazały, że postępy w rozwoju dziedziny na płaszczyźnie akademickiej znajdują odzwierciedlenie w codziennej praktyce dopiero po 5 – 15 latach, a nieraz po dłuższym czasie (Raghavan i Chand 1989, Rogers 1995, Parnas 1999). Ta książka jest próbą skrócenia tego okresu i zwrócenia uwagi programistów na najnowsze odkrycia już dziś.

Dla kogo jest ta książka?

Zebrane w tym podręczniku wiedza i doświadczenie pomogą Ci tworzyć oprogramowanie wyższej jakości oraz wykonywać swoją pracę szybciej i napotykać mniejszą liczbę problemów. Dowiesz się, co było przyczyną wielu typowych problemów w przeszłości i jak ich unikać. Opisane metody pracy pomogą Ci utrzymać kontrolę nad dużymi projektami oraz efektywnie rozwijać i modyfikować oprogramowanie w odpowiedzi na zmiany wymagań.

Doświadczeni programiści

Podręcznik ten przysłuży się każdemu doświadczonemu programiście, który poszukuje całościowego i prostego w użyciu przewodnika po procesie wytwarzania oprogramowania. Ponieważ treść książki koncentruje się na programowaniu,

najbliższej programistom części cyklu życia oprogramowania, opis efektywnych technik jego budowy jest zrozumiały zarówno dla samouków, jak i dla programistów z wykształceniem akademickim.

Kierownicy zespołów

Poprzednie wydania tej książki były wielokrotnie wykorzystywane przez liderów zespołów w edukacji mniej doświadczonych programistów. Książka ta może także pomóc uzupełnić luki we własnej wiedzy. Jeżeli jesteś doświadczonym programistą, być może nie zgodzisz się ze wszystkimi zawartymi w niej wnioskami (byłbym zaskoczony, gdyby tak było), ale po zakończeniu lektury i przemyśleniu wszystkich poruszanych zagadnień trudno będzie znaleźć temat z zakresu budowy oprogramowania, któremu nie poświęciłeś wcześniej uwagi.

Programiści bez formalnego wykształcenia

Jeżeli jesteś programistą samoukiem, to jesteś w dobrym towarzystwie. Pracę w tym zawodzie rozpoczyna co roku około 50 tysięcy osób (BLS 2004, Hecker 2004), podczas gdy związanych z oprogramowaniem stopni naukowych przyznaje się corocznie około 35 tysięcy (NCES 2002). Łatwo wywnioskować stąd, jak wielu programistów nie dysponuje formalnym wykształceniem kierunkowym. Programistów samouków można często znaleźć w różnych grupach zawodowych: inżynierów, księgowych, naukowców, nauczycieli i drobnych przedsiębiorców. Programowanie staje się często istotnym elementem ich pracy, nawet jeżeli nie postrzegają oni siebie jako programistów. Niezależnie od zakresu zdobytego wykształcenia książka ta daje możliwość zapoznania się z efektywnymi metodami pracy.

Studenci

Swoistym przeciwieństwem programisty z dużym doświadczeniem, a niewielkim formalnym wykształceniem, jest świeżo upieczony absolwent uczelni. Osoba taka ma często bogatą wiedzę teoretyczną, której nie towarzyszy praktyka w budowaniu programów komercyjnych. Tradycja ustna dobrego programowania jest często przekazywana w powolnym procesie rytualnych tańców plemiennych architektów oprogramowania, liderów zespołów, analityków i starszych programistów. Jeszcze częściej niemal cała wiedza tego rodzaju jest produktem samodzielnych prób i błędów. Ta książka może stać się alternatywą dla mało dynamicznego tradycyjnego przekazu wiedzy. Gromadzi ona dużą liczbę pomocnych wskazówek i efektywnych strategii pracy, wcześniej poznawanych jedynie w drodze polowania na rzadkie sposobności do skorzystania z doświadczenia innych. Książka ta jest pomocną dłonią wyciągniętą w stronę studenta, który przechodzi ze środowiska akademickiego do zawodowego.

Gdzie można znaleźć podobne informacje?

Niniejsza książka jest syntezą wiedzy o metodach programowania pochodzącej z wielu różnych źródeł. Poza jej rozproszeniem istotnym problemem było to, że duża część zgromadzonych informacji o budowie oprogramowania pozostawała przez lata poza dostępną literaturą (Hildebrand 1989, McConnell 1997a). W efektywnych, wysoce skutecznych metodach programowania stosowanych przez doświadczonych programistów nie ma nic tajemniczego, jednak w codziennej gonitwie i walce o skończenie bieżącego projektu rzadko się zdarza, że mają oni czas podzielić się swoimi umiejętnościami. W efekcie tego dostęp do dobrych źródeł pozwalających pogłębić wiedzę jest bardzo utrudniony.

Metody opisywane w tej książce wypełniają lukę między podręcznikami wprowadzającymi do programowania a tymi, które poruszają różne zagadnienia na poziomie specjalistycznym i akademickim. Gdzie szukać możliwości pogłębienia wiedzy o programowaniu po lekturze książek *Wprowadzenie do języka Java*, *Język Java dla zaawansowanych* i *Tajniki języka Java*? Można czytać książki o platformach sprzętowych firm Intel i Motorola, o funkcjach systemów operacyjnych Windows i Linux albo o innym języku programowania — tym bardziej, że nie można efektywnie używać języka czy programu w określonym środowisku bez dobrych źródeł informacji, które zawierają wszystkie potrzebne szczegóły. Jednak ta książka jest jedną z tych, które poświęcone są samemu programowaniu. Wiele z najbardziej pomocnych sposobów pracy to metody, które można stosować niezależnie od środowiska lub języka. O ile w innych książkach są one często pomijane, tutaj znajdują należne im miejsce.

Jak pokazuje rysunek, informacje przestawione w tej książce to kompilacja oparta na bardzo wielu źródłach. Jedyną możliwością zdobycia ich w innych sposób byłoby przekopywanie się przez górę książek i kilkuset magazynów specjalistycznych. Niezbędnym uzupełnieniem byłoby jeszcze pozyskanie znaczącej ilości wiedzy praktycznej. Nawet jeżeli masz to wszystko za sobą, pomocne może być posiadanie publikacji, która gromadzi ten ogromny zasób informacji w jednym miejscu i porządkuje go.

Główne zalety tego podręcznika

Niezależnie od posiadanej przez Ciebie wiedzy i wykształcenia niniejsza książka pomoże Ci pisać lepsze programy w krótszym czasie i z mniejszą liczbą problemów.

Pełne omówienie zagadnień budowy oprogramowania. W książce tej omawiane są ogólne aspekty budowy programów, jak jakość oprogramowania i podejście do procesu programowania, a także najróżniejsze zagadnienia szczegółowe takie jak przebieg budowy klasy, techniki pracy z danymi i strukturami sterującymi, debugowanie, refaktoryzowanie oraz metody i strategie optymalizacji. Aby lepiej poznać wybrane z tych tematów, nie musisz czytać całego podręcznika. Został on zaprojektowany tak, aby wyszukiwanie potrzebnych informacji było jak najłatwiejsze.

Gotowe listy kontrolne. W książce znaleźć można dziesiątki list kontrolnych pomocnych w ocenianiu architektury, projektu, jakości klas i procedur, nazw zmiennych, struktur sterujących, układu, testów oraz wielu innych aspektów programu i procesu programowania.

Najnowsza wiedza. Opisywanych jest tu wiele najnowszych metod pracy, z których znaczna część nie znajduje się jeszcze w powszechnym użyciu. Ponieważ wykorzystywana jest zarówno wiedza praktyczna, jak i wyniki badań naukowych, można oczekiwać, że przedstawione techniki pozostaną użyteczne przez wiele lat.

Szersza perspektywa budowy oprogramowania. Podręcznik ten jest okazją, by wznieść się ponad codzienną gorączkę gaszenia pożarów i określić, które metody pracy się sprawdzają, a które nie. Niewielu aktywnych zawodowo programistów ma czas na czytanie setek książek i artykułów, których kompilacją jest niniejsza pozycja. Zebrane w niej doświadczenia praktyczne i wyniki badań zapewnią podstawy wiedzy i pobudzą do zastanawiania się nad realizowanymi projektami, umożliwiając podjęcie strategicznie ukierunkowanych działań, które uwolnią czytelnika od staczania wciąż tych samych bitew.

Brak szumu informacyjnego. W niektórych książkach na każdy gram rzetelnej wiedzy przypada 10 gramów pochwał, uzasadnień i fascynacji ze strony samego autora. W tej staram się w zrównoważony sposób przedstawiać wady i zalety każdej metody. To Ty znasz wymagania konkretnego projektu lepiej niż ktokolwiek inny. Tutaj znajdziesz obiektywne informacje pozwalające podjąć wyważone decyzje, odpowiednie w określonych okolicznościach.

Pojęcia wspólne wielu językom. Opisywane tu metody pracy można efektywnie stosować niezależnie od języka, czy będzie to C++, C#, Java, Microsoft Visual Basic, czy inny podobny język programowania.

Duża liczba przykładów. Książka zawiera prawie 500 przykładów dobrego i złego kodu. Ich liczba jest tak duża przede wszystkim dlatego, że ja sam najszybciej się uczę, widząc konkretny kod. Wydaje mi się, że tak samo jest w przypadku większości innych programistów.

Przykłady pisane są w wielu różnych językach, ponieważ wyjście poza programowanie w jednym tylko języku jest często przełomem w karierze zawodowego programisty. Świadomość, że zasady programowania wykraczają poza składnię tego czy innego języka, otwiera drzwi do wiedzy, której zastosowanie pozwala systematycznie zwiększać jakość i wydajność pracy.

Aby praca z przykładami w różnych językach sprawiała jak najmniej kłopotu, nie stosowałem żadnych nietypowych mechanizmów poza tymi, które zostały dokładnie wyjaśnione. Nie jest konieczne zrozumienie każdego drobnego szczegółu w przedstawionych fragmentach kodu, by zrozumieć ilustrowaną przez nie ideę. Jeżeli zachowasz koncentrację na omawianym zagadnieniu, szybko stwierdzisz, że jesteś w stanie czytać dowolny program, bez względu na język, w którym został napisany. Bardziej znaczące miejsca w przykładach zostały dodatkowo wyróżnione.

Listy innych źródeł informacji. Niniejsza książka zawiera bardzo wiele informacji o budowie oprogramowania, ale nie wyczerpuje tematu. W większości rozdziałów znajduje się punkt „Więcej informacji", w którym wymieniam książki i artykuły pomocne w zdobywaniu głębszej wiedzy o wybranych zagadnieniach.

cc2e.com/1234

Witryna WWW książki. W witrynie podręcznika, *cc2e.com*, można znaleźć zaktualizowane listy kontrolne, książki, artykuły, łącza WWW i inne materiały. Aby uzyskać dostęp do informacji bezpośrednio związanych z pewnym fragmentem tekstu, wpisz *cc2e.com/*, a następnie czterocyfrowy kod widoczny na marginesie (przykład obok). Odwołania takie można znaleźć niemal na każdej stronie tego podręcznika.

Dlaczego napisałem tę książkę

Potrzeba powstawania dobrych książek o programowaniu, które gromadzą wiedzę o efektywnych metodach pracy, jest dobrze znana w środowisku związanym z inżynierią oprogramowania. Raport Computer Science and Technology Board podkreśla, że najlepszym sposobem poprawy jakości oprogramowania i zwiększenia wydajności pracy przy jego budowie jest kodyfikowanie, ujednolicanie i upowszechnianie dostępnej wiedzy o efektywnych metodach pracy (CSTB 1990, McConnell 1997a). Jako narzędzie służące do zwiększania dostępności tej wiedzy raport wskazuje podręczniki inżynierii oprogramowania.

Temat programowania jest zaniedbywany

Był czas, kiedy nie zauważano różnicy między tworzeniem oprogramowania a pisaniem kodu. Gdy jednak zidentyfikowano w cyklu życia oprogramowania różne czynności składowe, światowej klasy eksperci zaczęli poświęcać czas na analizowanie i opracowywanie metod zarządzania projektami, definiowania wymagań, projektowania i testowania. W gorączce badań nad tymi nowymi dziedzinami programowanie pozostało na uboczu. Nie zyskało ono podobnego zainteresowania co inne obszary wiedzy związanej z procesem wytwarzania oprogramowania.

W dyskusjach na temat programowania zaczęła też przeważać opinia, że traktowanie go jako odrębnej *czynności* w rozbudowanym procesie wymusza traktowanie go także jako odrębnej *fazy* tego procesu. W rzeczywistości czynności i fazy wchodzące w skład procesu tworzenia programów nie muszą pozostawać w ściśle określonej relacji, a o programowaniu można pisać niezależnie od tego, czy praca jest wykonywana w fazach, iteracyjnie czy w inny sposób.

Programowanie jest ważne

Inną przyczyną zaniedbywania programowania przez badaczy i autorów książek jest błędne mniemanie, że — w porównaniu z innymi czynnościami w procesie wytwarzania oprogramowania — jest ono procesem niemalże mechanicznym i nie ma w nim wiele miejsca na usprawnienia metod pracy. Nic nie może być dalsze od prawdy.

Programowanie zajmuje przeciętnie 65 procent czasu w małych projektach i 50 procent w średnich. Odpowiada zarazem za 75 procent błędów powstających w małych projektach i od 50 do 75 procent błędów w projektach średnich oraz dużych. Każda czynność, która powoduje 50 – 75 procent błędów, jest w oczywisty sposób obszarem, w którym można wprowadzić znaczące usprawnienia (więcej podobnych danych statystycznych można znaleźć w rozdziale 27.).

Niektórzy zwracają uwagę na to, że choć błędy popełniane w trakcie programowania stanowią duży procent ogólnej liczby błędów, są one znacznie tańsze w naprawie niż błędy w specyfikacji wymagań i architekturze. Ma z tego wynikać, że są mniej istotne. Teza, że błędy popełnione przy programowaniu są tańsze, jest prawdziwa, ale nie prowadzi w dobrym kierunku, bo koszty wynikające z ich pozostawienia mogą być ogromne. Badania wykazały, że drobne błędy programistyczne mogą być najdroższymi w historii informatyki, a ich koszty sięgają często setek milionów dolarów (Weinberg 1983, SEN 1990). Fakt, że są one tanie w naprawie, nie powoduje, iż należy traktować je jako mniej ważne.

Ironicznym aspektem odwrócenia uwagi od programowania jest to, że stanowi ono jedyną czynność, która zostaje wykonana w każdym projekcie. Wymagania można szybko „przyjąć", zamiast je opracowywać, projektowanie architektury może zastąpić przyjęcie z grubsza określonej idei, a testowanie można skrócić lub pominąć. Jeżeli jednak efektem pracy ma być program, programowanie jest absolutnie niezbędne. Jest to więc obszar, w którym każde usprawnienie musi przynieść widoczne skutki, niezależnie od przyjętego podejścia.

Nie ma podobnych książek

Ponieważ programowanie jest tak ważne, gdy pojawił się w mojej głowie pomysł napisania tej książki, byłem przekonany, że podobny podręcznik — poświęcony efektywnym metodom programowania — już gdzieś istnieje. Potrzeba książki o efektywnym programowaniu wydawała się oczywista. Okazało się jednak, że jest takich książek tylko kilka, a żadna z nich nie podejmuje próby całościowego omówienia tematu. Niektóre miały po 15 lub więcej lat i bazowały

na mało znanych językach jak ALGOL, PL/I, Ratfor czy Smalltalk. Część z nich napisali pracownicy akademiccy, którzy nie mieli doświadczenia w pracy w środowisku komercyjnym. Pisali oni o technikach, które sprawdzały się w projektach realizowanych przez studentów, ale nie mieli rzeczywistej wiedzy o tym, jak przebiega ich stosowanie poza murami uczelni. Inne książki podkreślały najróżniejsze zalety najnowszej, fascynującej autora metodyki, ignorując zarazem szeroki zakres dojrzałych, dopracowanych metod, które potwierdziły swoją użyteczność na przestrzeni wielu lat.

Gdy spotykają się krytycy sztuki, rozmowa dotyczy Formy, Struktury i Znaczenia. Gdy spotykają się artyści, tematem rozmowy jest to, gdzie kupić tani a dobry napitek.
— Pablo Picasso

Krótko mówiąc, nie mogłem znaleźć żadnej książki, która podejmowałaby chociaż próbę opisania zbioru użytecznych metod pracy wykorzystywanych w praktyce zawodowej, pracy akademików i badaniach wykonywanych w dużych przedsiębiorstwach. Opis taki powinien być aktualny. Powinien uwzględniać stosowane współcześnie języki, metody programowania obiektowego i najnowsze metodyki. Wydało mi się oczywiste, że książkę o programowaniu musi napisać osoba, która posiada dużą wiedzę o postępach w badaniach teoretycznych, a zarazem wystarczające doświadczenie, by zwrócić uwagę na metody stosowane w codziennej praktyce. Taka jest idea tej książki — ma być pełnym omówieniem zagadnień programowania napisanym przez programistę i dla programistów.

Od autora

Zachęcam wszystkich do przesyłania pytań związanych z poruszanymi w książce tematami, a także ciekawych spostrzeżeń i informacji o błędach. Mój adres to *stevemcc@construx.com*. Mam też prywatną stronę WWW:

www.stevemcconnell.com.

Bellevue, Washington
Dzień Pamięci 2004

Podziękowania

Nikt nie pisze książki w pełni samodzielnie (przynajmniej mi się to dotąd nie zdarzyło). W jeszcze większym stopniu dotyczy to drugiego wydania.

Chciałbym podziękować wszystkim, którzy recenzowali i przeglądali znaczące części tej książki. Są to: Hákon Ágústsson, Scott Ambler, Will Barns, William D. Bartholomew, Lars Bergstrom, Ian Brockbank, Bruce Butler, Jay Cincotta, Alan Cooper, Bob Corrick, Al Corwin, Jerry Deville, Jon Eaves, Edward Estrada, Steve Gouldstone, Owain Griffiths, Matthew Harris, Michael Howard, Andy Hunt, Kevin Hutchison, Rob Jasper, Stephen Jenkins, Ralph Johnson i jego Software Architecture Group przy University of Illinois, Marek Konopka, Jeff Langr, Andy Lester, Mitica Manu, Steve Mattingly, Gareth McCaughan, Robert McGovern, Scott Meyers, Gareth Morgan, Matt Peloquin, Bryan Pflug, Jeffrey Richter, Steve Rinn, Doug Rosenberg, Brian St. Pierre, Diomidis Spinellis, Matt Stephens, Dave Thomas, Andy Thomas-Cramer, John Vlissides, Pavel Vozenilek, Denny Williford, Jack Woolley i Dee Zsombor.

Setki czytelników przesłały swoje uwagi dotyczące pierwszego wydania. Liczba komentarzy na temat drugiego była jeszcze większa. Dziękuję każdemu, kto poświęcił swój czas, aby podzielić się swoimi odczuciami po lekturze tej książki w różnych formach, w jakich była udostępniana.

Specjalne podziękowania kieruję pod adresem recenzentów z Construx Software, którzy przeprowadzili formalną inspekcję całego rękopisu. Byli to Jason Hills, Bradey Honsinger, Abdul Nizar, Tom Reed i Pamela Perrott. Byłem naprawdę zachwycony ich sumiennością, której nie przeszkodziła świadomość, iż zanim rozpoczęli pracę, książkę przeglądało już bardzo wiele osób. Dziękuję też Bradeyowi, Jasonowi i Pameli za ich wkład w przygotowanie witryny *cc2e.com*.

Praca z Devonem Musgrave'em, redaktorem prowadzącym tej książki, była prawdziwą przyjemnością. Przy wcześniejszych projektach zetknąłem się z wieloma wspaniałymi redaktorami, ale muszę w tym miejscu wspomnieć niezwykłą sumienność Devona i to, jak łatwo się z nim pracowało. Dziękuję! Dziękuję też Lindzie Engleman, która kierowała pracami nad drugim wydaniem. Bez niej ta książka by nie powstała. Dziękuję również innym pracownikom Microsoft Press. Osoby, których nazwiska chciałbym wymienić, to Robin Van Steenburgh, Elden Nelson, Carl Diltz, Joel Panchot, Patricia Masserman, Bill Myers, Sandi Resnick, Barbara Norfleet, James Kramer i Prescott Klassen.

Chciałbym też przypomnieć nazwiska pracowników Microsoft Press, którzy przygotowywali pierwsze wydanie. Byli to: Alice Smith, Arlene Myers, Barbara Runyan, Carol Luke, Connie Little, Dean Holmes, Eric Stroo, Erin O'Connor, Jeannie McGivern, Jeff Carey, Jennifer Harris, Jennifer Vick, Judith Bloch, Katherine Erickson, Kim Eggleston, Lisa Sandburg, Lisa Theobald, Margarite Hargrave, Mike Halvorson, Pat Forgette, Peggy Herman, Ruth Pettis, Sally Brunsman, Shawn Peck, Steve Murray, Wallis Bolz i Zaafar Hasnain.

W pracy nad pierwszym wydaniem uczestniczyli też recenzenci: Al Corwin, Bill Kiestler, Brian Daugherty, Dave Moore, Greg Hitchcock, Hank Meuret, Jack Woolley, Joey Wyrick, Margot Page, Mike Klein, Mike Zevenbergen, Pat Forman, Peter Pathe, Robert L. Glass, Tammy Forman, Tony Pisculli i Wayne Beardsley. Specjalne podziękowania dla Tony'ego Garlanda za jego niezwykle dokładny przegląd. Po upływie 12 lat jeszcze bardziej doceniam jego wyjątkowy wkład wyrażający się w kilku tysiącach niezwykle cennych uwag.

Listy kontrolne

Tabele

Rysunki

Część I
Proces budowy oprogramowania

W tej części:

Rozdział 1.

Budowa oprogramowania

cc2e.com/0178

W tym rozdziale

- 1.1. Czym jest budowa oprogramowania — strona 37
- 1.2. Znaczenie procesu budowy oprogramowania — strona 40
- 1.3. Jak korzystać z tej książki — strona 41

Podobne tematy

- Dla kogo jest ta książka: wstęp
- Główne zalety tego podręcznika: wstęp
- Dlaczego napisałem tę książkę: wstęp

Dobrze wiemy, co znaczy słowo „budować" poza światem programowania. Budowanie to praca wykonywana przez pracowników budowlanych, prowadząca do powstania nowego domu, szkoły czy apartamentowca. W codziennym użyciu słowo „budowa" odnosi się do tworzenia czegoś nowego. Proces budowania może obejmować pewne elementy planowania, projektowania i sprawdzania, ale stanowią go przede wszystkim bezpośrednie działania prowadzące do powstania nowej rzeczy.

1.1. Czym jest budowa oprogramowania

Tworzenie oprogramowania komputerowego to często bardzo złożony proces. W ciągu ostatnich 25 lat w badaniach nad jego przebiegiem wyróżniono długą listę jego składowych. Należą do nich:

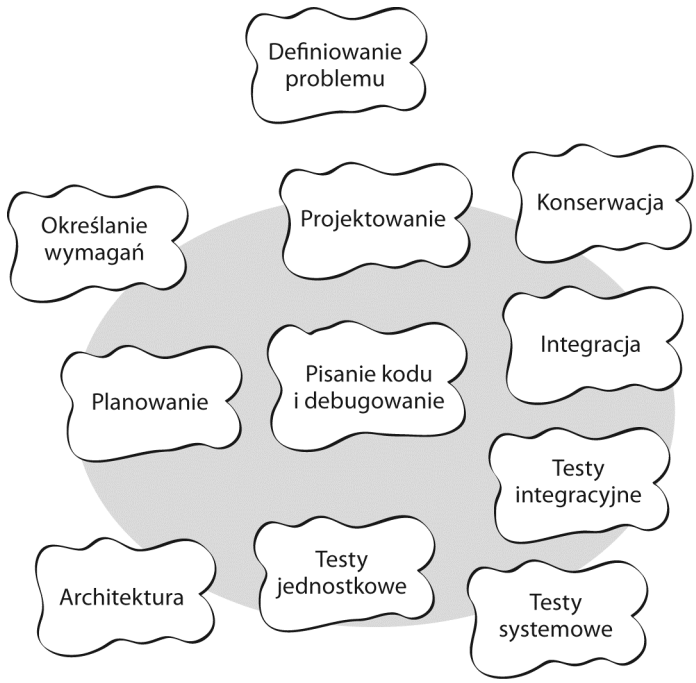
- definiowanie problemu,
- określanie wymagań,
- planowanie,
- projektowanie architektury,
- projektowanie szczegółowe,
- pisanie kodu i debugowanie,
- testy jednostkowe,
- testy integracyjne,
- integrowanie,
- testy systemowe,
- konserwowanie (pielęgnowanie) systemu.

Jeżeli miałeś do czynienia wyłącznie z małymi, niesformalizowanymi projektami, opis tego rodzaju może wydawać się niepotrzebną komplikacją. Z drugiej

strony, jeżeli miałeś styczność z projektami nadmiernie sformalizowanymi, dobrze *wiesz*, jak uciążliwy może być tak usystematyzowany proces. Dobranie optymalnego poziomu formalizacji nie jest łatwe. Jest to jedno z zagadnień poruszanych w tej książce.

Jeżeli dotąd pracowałeś sam lub w nieformalnych grupach, nie wprowadzałeś zapewne podziału na różne fazy prac wykonywanych w procesie tworzenia produktu. Są to czynności określane ogólnie jako „programowanie". W nieformalnych projektach aktywność zaprzątająca przede wszystkim naszą uwagę to ta, którą nazywam w tej książce „budową oprogramowania".

Intuicyjna interpretacja słowa „budowa" pasuje do kontekstu oprogramowania, ale brakuje w niej pewnej perspektywy. Warto ukazać budowę oprogramowania w kontekście innych działań, aby lepiej uwidocznić, co należy do tego procesu, a co nie. Rysunek 1.1 przedstawia jego miejsce pośród innych czynności związanych z programowaniem.



Rysunek 1.1. Czynności należące do budowy oprogramowania znajdują się wewnątrz szarej elipsy. Budowa oprogramowania to przede wszystkim pisanie kodu i debugowanie, ale także wiele elementów projektowania, testów jednostkowych, testów integracyjnych i innego rodzaju działań

Jak widać na rysunku, budowa oprogramowania to przede wszystkim pisanie kodu i debugowanie, ale także projektowanie, planowanie, testy jednostkowe, integracja, testy integracyjne i inne czynności. Aby w pełni omówić wszystkie aspekty procesu wytwarzania oprogramowania, należałoby poświęcić odpowiednią ilość miejsca na opis każdego z wymienionych tu jego elementów. Ponieważ jednak niniejsza książka jest podręcznikiem, w którym omawiane są metody budowy oprogramowania, zagadnienia wykraczające poza ten temat nie będą szerzej prezentowane.

Proces budowy oprogramowania opisuje się potocznie słowem „programowanie" lub, mniej formalnie, „kodowanie". Oba, dość niefortunnie, sugerują mechaniczny proces translacji pewnego projektu na język komputera. Budowanie oprogramowania w żadnej mierze nie jest zajęciem mechanicznym — wymaga kreatywności i umiejętności podejmowania decyzji. W książce tej używam terminów „budowa oprogramowania" i „programowanie" zamiennie.

Rysunek 1.2 przedstawia elementy z rysunku 1.1 w „widoku 3D".

Rysunek 1.2. Ta książka koncentruje się na pisaniu kodu i debugowaniu, projektowaniu, planowaniu, testach jednostkowych, integracji, testach integracyjnych i innych czynnościach w przedstawionych na rysunku proporcjach

Rysunki 1.1 i 1.2 przedstawiają bardzo ogólną perspektywę. Jak to wygląda w szczegółach? Oto przykłady zadań związanych z budowaniem oprogramowania:

- Weryfikowanie, czy przygotowane zostały materiały niezbędne do rozpoczęcia programowania.
- Określanie sposobu testowania kodu.
- Projektowanie oraz pisanie klas i procedur.
- Tworzenie oraz nazywanie zmiennych i stałych.
- Wybieranie struktur sterujących i organizowanie kodu w bloki instrukcji.
- Testowanie jednostkowe, testowanie integracyjne i debugowanie własnego kodu.
- Przeglądanie projektów i kodu niskiego poziomu innych członków zespołu oraz udostępnianie im własnego.
- Dopracowywanie formatowania kodu i komentarzy.
- Integrowanie składników oprogramowania, które były budowane odrębnie.
- Modyfikowanie kodu w celu uzyskania większej wydajności i zmniejszenia wykorzystania zasobów.

Aby uzyskać pełniejszą listę, wystarczy spojrzeć na spis tytułów rozdziałów tej książki.

Budowa oprogramowania obejmuje tak szerokie spektrum działań, że może nasuwać się pytanie: „OK, Johnny, co właściwie *nie jest* procesem należącym do budowy oprogramowania?". Jest ono uzasadnione. Do najważniejszych tego typu czynności należą: zarządzanie, określanie wymagań, opracowywanie architektury oprogramowania, projektowanie interfejsu użytkownika, testowanie systemowe i konserwacja. Każda z nich wpływa na końcowy sukces projektu w takim samym stopniu jak budowa oprogramowania — przynajmniej gdy mowa o projektach, które wymagają udziału więcej niż jednej czy dwóch osób i trwają dłużej niż kilka tygodni. Nie brakuje dobrych książek szczegółowo opisujących te zadania. Wiele z nich wymienionych jest też w różnych rozdziałach w punktach zatytułowanych „Więcej informacji". Listę polecanych lektur można znaleźć w rozdziale 35. „Gdzie znaleźć więcej informacji".

1.2. Znaczenie procesu budowy oprogramowania

Ponieważ czytasz tę książkę, można oczekiwać, że uważasz zagadnienia jakości oprogramowania i efektywności swojej pracy za ważne. Różnego rodzaju oprogramowanie wykorzystuje się w najciekawszych systemach naszych czasów. Internet, efekty specjalne w filmach, medyczne systemy podtrzymywania życia, programy kosmiczne, lotnictwo, złożona obliczeniowo analiza finansowa czy badania naukowe to tylko kilka przykładów. Wszystkie takie rozwiązania, jak i te bardziej konwencjonalne, mogą wiele zyskać na stosowaniu lepszych praktyk programistycznych — fundamentalne zasady działania programów pozostają w dużej mierze niezmienne.

Jeżeli zgadzasz się ze stwierdzeniem, że ważne jest doskonalenie procesu wytwarzania oprogramowania, interesująca może być dla Ciebie, jako czytelnika tej książki, odpowiedź na pytanie: dlaczego wybranym tematem jest samo programowanie?

Proces ten jest ważny z kilku przyczyn.

Patrz też: Szczegółowe omówienie relacji między rozmiarem projektu a czasem poświęcanym na programowanie można znaleźć w punkcie „Wielkość a udział poszczególnych czynności" podrozdziału 27.5.

Programowanie to znaczna część procesu wytwarzania oprogramowania. W zależności od rozmiarów projektu zajmuje ono od 30 do 80 procent czasu pracy. Już sam ten fakt sprawia, że jest to część procesu o kluczowym znaczeniu dla jego powodzenia.

Programowanie to centralny element procesu wytwarzania oprogramowania. Analiza wymagań i opracowywanie architektury następują przed programowaniem, aby mogło ono być efektywne. Testowanie systemowe (w ścisłym znaczeniu niezależnego testowania), służące weryfikacji przebiegu całego procesu, następuje po napisaniu kodu. Samo programowanie znajduje się w centrum procesu tworzenia i rozwijania oprogramowania.

Patrz też: O różnicach między programistami piszemy w części „Różnice między programistami" podrozdziału 28.5.

Koncentracja na programowaniu pozwala znacznie zwiększyć produktywność poszczególnych programistów. Klasyczne studium Sackmana, Eriksona i Granta dowiodło, że między poszczególnymi programistami różnice wydajno-

ści pracy w trakcie budowy oprogramowania mogą być nawet 10- i 20-krotne (1968). Od tego czasu jego wyniki zostały wielokrotnie potwierdzone (Curtis 1981, Mills 1983, Curtis et al. 1986, Card 1987, Valett i McGarry 1989, DeMarco i Lister 2002, Boehm et al. 2000). Ta książka będzie pomocą dla tych, którzy chcą poznać metody pracy stosowane przez najlepszych programistów.

Produkt programowania, kod źródłowy, jest często jedynym dokładnym opisem oprogramowania. W wielu projektach jedyną dokumentacją dostępną programistom jest sam kod. Specyfikacja wymagań i inne dokumenty projektowe mogą ulec dezaktualizacji, podczas gdy kod źródłowy jest aktualny zawsze. Jest to kolejny czynnik, który decyduje o ogromnym znaczeniu jego jakości. Stosowanie spójnych technik pisania i rozbudowy kodu stanowi o różnicy między maszyną Rube Goldberga[1] a poprawnym, szczegółowo opisanym i przez to czytelnym programem. Techniki te muszą być konsekwentnie stosowane w trakcie budowy oprogramowania.

Programowanie to jedyna część procesu, która na pewno zostanie wykonana. Idealny projekt informatyczny przed rozpoczęciem programowania przechodzi przez fazy uważnej analizy wymagań i projektowania architektury. W idealnym projekcie stosuje się całościowe, kontrolowane statystycznie testy systemowe. W rzeczywistych projektach, które nigdy nie przebiegają idealnie, programowanie rozpoczyna się często bez wcześniejszego określania wymagań i projektowania. Testowanie zostaje pominięte, bo produkt zawiera zbyt wiele błędów i brakuje czasu. Jednak, bez względu na pośpiech i złe planowanie, programowanie to część procesu, która zostaje wykonana. Doskonalenie jego metod prowadzi więc do usprawnienia każdego przedsięwzięcia związanego z tworzeniem oprogramowania, niezależnie od jego indywidualnych cech.

1.3. Jak korzystać z tej książki

Niniejszą książkę można czytać od deski do deski lub wybierając poszczególne tematy. Jeżeli decydujesz się na pierwsze podejście, możesz rozpocząć lekturę rozdziału 2. „Metafory procesu programowania". Jeżeli szukasz konkretnych wskazówek dotyczących pisania kodu, możesz zacząć od rozdziału 6. „Klasy z klasą", a dalej kierować się odwołaniami do szerszych omówień co ciekawszych tematów. Jeżeli nie wiesz, co wybrać, rozpocznij od podrozdziału 3.2 „Określanie rodzaju budowanego oprogramowania".

Podsumowanie

- Programowanie znajduje się w centrum procesu tworzenia i rozwijania oprogramowania. Jest to jedyna część tego procesu, która musi wystąpić w każdym projekcie.

[1] Amerykański satyryk, który zasłynął z serii rysunków maszyn wykonujących proste operacje w bardzo złożony, mało zrozumiały sposób — *przyp. tłum.*

- Główne czynności wchodzące w skład programowania to projektowanie, pisanie kodu, debugowanie, integrowanie składników i testy programisty (jednostkowe oraz integracyjne).
- „Budowa oprogramowania" i „programowanie" to terminy używane w tej książce zamiennie na określenie części szerszego procesu tworzenia i rozwijania oprogramowania.
- Stosowane metody programowania znacząco wpływają na jakość produktu końcowego.
- W ostatecznym rozrachunku to znajomość programowania decyduje o tym, jak dobrym programistą jest dana osoba. Temu tematowi poświęcona jest cała dalsza część książki.

Rozdział 2.

Metafory procesu programowania

cc2e.com/0278

W tym rozdziale

Podobny temat

- Heurystyka w projektowaniu: „Projektowanie jest procesem heurystycznym" w podrozdziale 5.1

Spośród wielu dziedzin techniki informatyka operuje jednym z najbarwniejszych języków. Jakie inne zajęcie pozwala wejść do sterylnego pokoju stale utrzymywanego w temperaturze 20 stopni, aby znaleźć się w świecie wirusów, koni trojańskich, robaków, pluskw, bomb, upadków systemu, zapór ogniowych, pokręconych zmieniaczy płci i radiowych myszy?

Takie obrazowe metafory opisują ściśle określone zjawiska. Równie barwne przenośnie określają szersze mechanizmy — możesz wykorzystać je, aby lepiej zrozumieć proces programowania.

Dalsza część książki nie będzie opierać się bezpośrednio na przedstawionym w tym rozdziale omówieniu metafor. Jeżeli wolisz od razu przejść do części praktycznej, możesz je pominąć. Ten rozdział pomoże jednak spojrzeć na programowanie z szerszej perspektywy.

2.1. Znaczenie metafor

Właściwe stosowanie analogii pozwala często wysnuć bardzo celne wnioski. Porównując temat, który słabo znamy, z takim, który jest nam bliższy, możemy uzyskać spojrzenie pozwalające lepiej zrozumieć nowy temat. Takie wykorzystywanie metafor nazywane jest modelowaniem.

W historii nauki można znaleźć niezliczone przypadki odkryć, do których doprowadziło umiejętne wykorzystanie potęgi metafory. Chemik Kekulé miał sen, w którym wąż zjadał własny ogon. Po obudzeniu się zdał sobie sprawę, że struktura molekularna oparta na prostym pierścieniu wyjaśnia właściwości benzenu. Dalsze eksperymenty potwierdziły hipotezę (Barbour 1966).

Teoria kinetyki gazów wywodzi się z modelu kul bilardowych — cząsteczki gazu traktuje się jako obiekty posiadające masę i ulegające zderzeniom sprężystym, podobnie jak kule na stole do bilardu. Model ten posłużył do wyprowadzenia wielu praktycznych twierdzeń.

Teoria falowa światła została opracowana w dużej mierze w oparciu o podobieństwo światła i dźwięku. Mają one amplitudę (jaskrawość, głośność), częstotliwość (kolor, wysokość) i inne wspólne właściwości. Porównywanie ich teorii falowych było tak owocne, że naukowcy włożyli wiele wysiłku w poszukiwania medium pozwalającego na propagację światła w podobny sposób, jak powietrze umożliwia propagację dźwięku. Nadali mu nawet nazwę — „eter" — ale nigdy nie zostało ono znalezione. Analogia, która była pod wieloma względami niezwykle celna, okazała się w tym przypadku zwodnicza.

Ogólnie, siła modeli polega na ich wyrazistości i tym, że można je uchwycić jako pewną pojęciową całość. Sugerują one różne właściwości, zależności i istnienie dodatkowych obszarów, którym trzeba poświęcić stosowną uwagę. Niekiedy model wskazuje na niewłaściwe obszary i wówczas mamy do czynienia z jego nadmiernym rozwinięciem. Tak stało się w przypadku eteru.

Jak można oczekiwać, są metafory lepsze i gorsze. Dobra metafora jest prosta, pasuje do innych, powiązanych z nią metafor i wyjaśnia znaczącą część wyników eksperymentów lub zjawisk.

Rozważmy przykład poruszającego się, podwieszonego na linie ciężkiego kamienia. Przed Galileuszem spadkobiercy szkoły Arystotelesa, którzy patrzyli na kamień w ruchu, widzieli ciężki obiekt przemieszczający się w sposób naturalny z pozycji położonej wyżej do stanu spoczynku w pozycji położonej niżej. Uczeń Arystotelesa widział przede wszystkim utrudniony spadek w dół. Gdy na poruszający się kamień spojrzał Galileusz, zobaczył wahadło. Dla niego w istocie powtarzał on wielokrotnie, w niemal idealny sposób, ten sam ruch.

Sugestywność obu modeli bardzo się różni. Uczeń Arystotelesa, który widział poruszający się kamień jako spadający przedmiot, badał jego masę, wysokość, na którą się wzniósł, i czas wymagany do przejścia do pozycji spoczynkowej. W galileuszowskim modelu wahadła kluczowe były zupełnie inne czynniki. Galileusz badał masę kamienia, promień ruchu wahadła, przemieszczenie kątowe i czas trwania cyklu. Odkrył prawa, których spadkobiercy Arystotelesa odkryć nie mogli, ponieważ ich model nakazywał obserwować inne zjawiska i zadawać inne pytania.

Metafory przyczyniają się do lepszego zrozumienia problemów związanych z programowaniem w podobny sposób, jak dzieje się to w przypadku zagadnień rozważanych przez naukowców. W swoim odczycie towarzyszącym wręczeniu Nagrody Turinga w 1973 roku Charles Bachman mówił o przejściu od geocentrycznego postrzegania świata do spojrzenia heliocentrycznego. Ptolemejski model geocentryczny obowiązywał, praktycznie niezagrożony, przez 1400 lat. W 1543 roku Kopernik ogłosił teorię heliocentryczną, ideę, w której nie Ziemia, ale Słońce znajduje się w środku wszechświata. Zmiana modelu

pojęciowego doprowadziła ostatecznie do odkrycia nowych planet, uznania Księżyca za satelitę Ziemi (a nie planetę) i innego postrzegania miejsca ludzkości we wszechświecie.

Wartość metafor nie powinna być lekceważona. Ważną ich zaletą jest to, że wprowadzają pewne oczekiwane, dla wszystkich zrozumiałe zachowania i mechanizmy. Upraszcza to komunikację i zmniejsza liczbę nieporozumień, a procesy uczenia się przebiegają szybciej. W efekcie metafory są cenną metodą internalizowania i uogólniania pojęć, która pozwala myśleć o problemie na wyższym poziomie oraz unikać błędów w działaniach wykonywanych na poziomach niższych.
— Fernando J. Corbató

Bachman porównał zmianę paradygmatu w astronomii do zmiany w programowaniu komputerów, która nastąpiła na początku lat siedemdziesiątych. Gdy dokonywał tego zestawienia, w 1973 roku, w procesach przetwarzania danych następowało właśnie przejście od postrzegania systemów informatycznych jako skoncentrowanych wokół komputera do nowego spojrzenia, w którym centrum stała się baza danych. Bachman zaznaczył, że spadkobiercy starej szkoły chcieli widzieć dane jako sekwencyjny strumień przepływających przez komputer kart. Zmiana polegała na koncentracji na puli danych, na których komputer w danym momencie operuje.

Dzisiaj trudno sobie wyobrazić, aby ktokolwiek mógł myśleć, że Słońce krąży wokół Ziemi. Trudno też wyobrazić sobie programistę, który uważa, że wszystkie dane powinny być traktowane jako sekwencyjny strumień kart. W obu przypadkach po upadku starej teorii wydaje się nieprawdopodobne, że cieszyła się ona powszechną akceptacją w przeszłości. Co jeszcze bardziej niesamowite, ci, którzy wierzyli w starą teorię, uważali, że nowa jest równie niedorzeczna, jak nam wydaje się być jej poprzedniczka.

Geocentryczna wizja wszechświata zaślepiła astronomów, którzy próbowali trzymać się jej nawet wtedy, gdy pojawiła się nowa. W podobny sposób postrzeganie komputera jako centralnego punktu we wszechświecie informatyki okaleczyło informatyków, którzy starali się zachować ten punkt widzenia nawet po sformułowaniu teorii, zgodnie z którą w centrum jest baza danych.

Łatwo ulec pokusie trywializowania znaczenia metafor. Przedstawione przykłady można w prosty sposób podsumować stwierdzeniem: „Oczywiście właściwa metafora jest lepsza. Ta druga była błędna!". Jest to naturalna reakcja, ale zarazem uproszczenie. Historia nauki nie jest ciągiem przejść od „błędnej" metafory do „poprawnej". To ciąg przejść od metafor „gorszych" do metafor „lepszych", od tych węższych do bardziej uogólnionych, od sugestywnych w jednym obszarze do sugestywnych w obszarze innym.

W istocie wiele modeli, które zastępuje się modelami lepszymi, zachowuje swoją użyteczność. Inżynierowie wciąż rozwiązują większość problemów, korzystając z dynamiki Newtona, mimo że — z teoretycznego punktu widzenia — jej miejsce zajęła teoria Einsteina.

Programowanie to dziedzina młodsza niż większość dyscyplin naukowych. Nie jest jeszcze dojrzała na tyle, aby obowiązywał w niej pewien ujednolicony zespół metafor. Nie brakuje natomiast analogii uzupełniających się, a także wzajemnie sprzecznych. Jedne są lepsze, inne gorsze. Właściwe rozumienie dostępnego zasobu metafor jest niezbędne do poprawnego zrozumienia procesu tworzenia i rozwoju oprogramowania.

2.2. Jak korzystać z metafor w programowaniu

Metafora dotycząca programowania przypomina bardziej reflektor niż mapę terenu. Nie wyjaśnia ona, gdzie szukać odpowiedzi, ale pokazuje, w jaki sposób ją znaleźć. Jest w większym stopniu narzędziem heurystycznym niż algorytmem.

Algorytm to zbiór ściśle określonych instrukcji opisujących sposób wykonywania pewnego zadania. Jest przewidywalny, deterministyczny i nie ma w nim miejsca na przypadek. Mówi o tym, jak przejść od punktu A do punktu B bez żadnego zbaczania z wytyczonej drogi, bez punktów pośrednich D, E i F, bez postojów i podziwiania widoków.

Heurystyka to metoda, która pomaga w poszukiwaniu odpowiedzi. Wyniki jej zastosowania nie są pewne, ponieważ mówi ona tylko, w jaki sposób szukać, ale nie informuje o tym, co ma zostać znalezione. Nie jest to opis bezpośredniej drogi z punktu A do punktu B. Położenie tych punktów może wręcz nie być znane. W efekcie heurystyka jest algorytmem w przebraniu klauna. Jest mniej przewidywalna, jest z nią więcej zabawy i nie daje żadnej gwarancji.

Oto algorytm dojazdu do domu pewnej osoby: Jedź autostradą nr 167 do Puyallup. Zjedź przy South Hill Market i jedź 4,5 mili pod górę. Skręć w prawo przy latarni obok sklepu spożywczego, a następnie w pierwszą drogę w lewo. Wjedź na podjazd dużego domu po lewej o numerze 714 na ulicy North Cedar.

Patrz też: Stosowanie heurystyki w projektowaniu oprogramowania omawiamy w części „Projektowanie jest procesem heurystycznym" podrozdziału 5.1.

Heurystyczne wskazówki (heurystyki) wyglądają następująco: Znajdź ostatni odebrany od nas list. Jedź do miasta wskazanego w adresie zwrotnym. Gdy się w nim znajdziesz, spytaj dowolną osobę o nasz dom. Wszyscy nas znają i chętnie wskażą drogę. Jeżeli nie będziesz mógł nikogo znaleźć, zadzwoń z budki telefonicznej, a przyjedziemy po ciebie.

Różnica między algorytmem a metodą heurystyczną jest dość subtelna i oba te pojęcia w pewnym zakresie pokrywają się. Na potrzeby tej książki wystarczy stwierdzenie, że głównie jest ona widoczna w poziomie bezpośredniości rozwiązania. Algorytm to bezpośrednie wskazówki. Metoda heurystyczna opisuje sposób ich samodzielnego szukania albo miejsce, w którym można je znaleźć.

Wskazówki dokładnie opisujące sposoby rozwiązywania problemów napotykanych w trakcie programowania byłyby na pewno dużym ułatwieniem, a wyniki ich stosowania charakteryzowałyby się większą przewidywalnością. Jednak inżynieria oprogramowania nie jest jeszcze na tyle zaawansowaną nauką i być może nigdy taką nie będzie. Największym wyzwaniem przy programowaniu jest ujęcie pojęciowe problemu, a wiele błędów ma charakter koncepcyjny. Ponieważ każdy program jest inny, znalezienie ogólnego zbioru wskazówek, które w każdym przypadku prowadziłyby do rozwiązania, jest trudne lub niemożliwe. W efekcie ogólna wiedza o tym, jak radzić sobie z problemami, jest co najmniej tak samo wartościowa jak znajomość konkretnych rozwiązań określonych problemów.

Jak korzystać z metafor w programowaniu? Wykorzystuj je w taki sposób, aby uzyskiwać szersze spojrzenie na rozwiązywane problemy i analizowane procesy. Staraj się, aby były pomocą w myśleniu o tym, co robisz, i ułatwiały

rozważania nad lepszymi sposobami wykonania tych samych zadań. Nie dojdziesz do poziomu, na którym wystarczyłoby Ci spojrzeć na wiersz kodu, abyś mógł stwierdzić, że narusza on jedną z opisywanych w tym rozdziale metafor. Z czasem jednak osoba, która korzysta z analogii, aby uczynić proces tworzenia oprogramowania bardziej przejrzystym, będzie postrzegana jako taka, która lepiej zna się na programowaniu i sprawniej pisze wysokiej jakości kod niż osoby, które ich nie używają.

2.3. Popularne metafory programowania

Programowanie obrosło przytłaczającą liczbą metafor. David Gries twierdzi, że pisanie oprogramowania jest nauką (1981). Donald Knuth pisze, że to sztuka (2001). Watts Humphrey mówi, że jest to proces (1989). P.J. Plauger i Kent Beck uważają, że praca programisty jest jak jazda samochodem, ale obaj dochodzą do niemal przeciwstawnych wniosków (Plauger 1993, Beck 2006). Alistair Cockburn przyrównuje programowanie do gry (2008). Eric Raymond widzi podobieństwa do bazaru (2000). Andy Hunt i Dave Thomas uważają, że jest jak pielęgnowanie ogrodu. Paul Heckel napisał, że jest ono jak filmowanie „Królewny Śnieżki i siedmiu krasnoludków" (1994). Fred Brooks uważa, że to połączenie pracy rolnika, polowań na wilkołaki i topienia dinozaurów w smole (2000). Które metafory są najlepsze?

Pisanie listów — pisanie kodu

Najbanalniejsza metafora programowania wywodzi się z wyrażenia „pisanie kodu". Sugeruje ono, że budowanie programu przypomina pisanie lekkiego listu — siadasz z piórem, atramentem i papierem, aby napisać go od początku do końca. Nie wymaga on formalnego planowania, a lista poruszanych tematów powstaje na bieżąco wraz z nim.

Z metafory pisania kodu wywodzi się wiele ważnych pojęć i idei. Jon Bentley twierdzi, że dobry program można czytać przy kominku ze szklanką brandy, dobrym cygarem i wiernym psem przy boku z równym zainteresowaniem co dobrą powieść. Brian Kernighan i P.J. Plauger zatytułowali swoją książkę o stylu programowania *The Elements of Programming Style* (1978), nawiązując przy tym do tytułu znanej książki o pisaniu *The Elements of Style* (Strunk i White 2000). Programiści często rozmawiają o „czytelności programu".

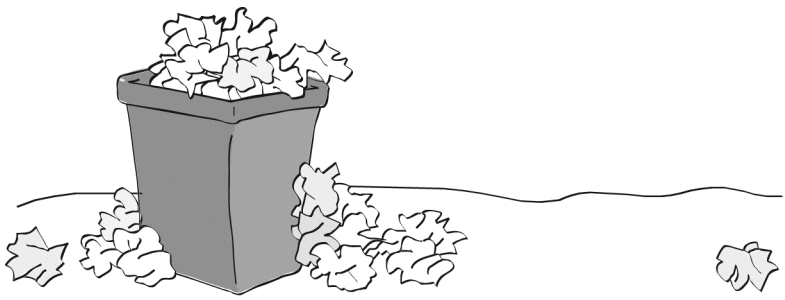
Przy pracy w pojedynkę i przy małych projektach metafora pisania listu sprawdza się dość dobrze, ale w innych sytuacjach wychodzą na jaw jej braki — nie jest to ani pełny, ani wystarczający opis procesu tworzenia i rozwijania oprogramowania. Pisanie to zazwyczaj zajęcie wykonywane indywidualnie, podczas gdy w projektach mamy najczęściej do czynienia z wieloma osobami o różnie określonych zakresach odpowiedzialności. Po zakończeniu pisania listu wkładamy go do koperty i wysyłamy. Nie można go już zmienić, jest skończony. Oprogramowanie dużo łatwiej zmodyfikować i praca nad nim praktycznie nigdy nie zostaje uznana za zakończoną. Po jego wdrożeniu lub udostępnieniu użytkownikom pierwszej jego wersji wykonuje się średnio dwie trzecie, a często nawet 90 procent całości włożonej w jego utworzenie i rozwój pracy

(Pigoski 1997). Gdy mamy do czynienia z pisaniem, oryginalność jest zwykle ceniona. W programowaniu próby znalezienia oryginalnych rozwiązań są zazwyczaj mniej efektywne niż koncentracja na wykorzystywaniu koncepcji, kodu i testów ze starszych projektów. Krótko mówiąc, metafora pisania implikuje proces programowania, który jest zbyt prosty i surowy, niż to pożądane w rzeczywistości.

Przygotuj się na to, że cokolwiek zrobisz, wcześniej czy później pójdzie do kosza.
— Fred Brooks

Jeżeli przygotujesz się na to, że cokolwiek zrobisz, wcześniej czy później pójdzie do kosza, będziesz wyrzucał dwa razy.
— Craig Zerouni

Niestety, metafora pisania listu została rozpropagowana za pośrednictwem jednej z najpopularniejszych książek o programowaniu na świecie, *Mityczny osobomiesiąc* Freda Brooksa (Brooks 2000). Autor pisze w niej: „Przygotuj się na to, że cokolwiek zrobisz, wcześniej czy później pójdzie do kosza". Przywołuje to na myśl obraz piętrzącego się w koszu na śmieci stosu pogniecionych kartek z nieudanymi próbami twórczymi (patrz rysunek 2.1).



Rysunek 2.1. Metafora pisania listu sugeruje, że proces programowania bazuje na kosztownej metodzie prób i błędów, a nie na precyzyjnym planowaniu i projektowaniu

Praca z myślą o tym, że cokolwiek zrobisz, pójdzie do kosza, może sprawdzać się przy pisaniu listu z zapytaniem o zdrowie cioci. Jednak rozumienie metafory „pisania" oprogramowania w ten sposób, że w przyszłości powstanie inna wersja, nie jest dobrą radą, gdy pracujemy nad systemem, którego koszt dorównuje 10-piętrowemu biurowcowi lub statkowi oceanicznemu. Nie jest sztuką osiągnięcie nawet bardzo trudnego celu, gdy dysponujemy nieograniczoną liczbą prób. Wyzwaniem jest dochodzenie do celu w pierwszym podejściu, a przynajmniej na tyle sprawnie, aby wykorzystanie czasu i zasobów pozostało efektywne. Inne metafory lepiej wskazują ścieżki prowadzące do osiągnięcia tego ideału.

Praca na roli — hodowanie systemu

Niektórzy programiści twierdzą, że — w przeciwieństwie do surowej metafory pisania — tworzenie oprogramowania jest jak sianie i zbieranie plonów. Programista projektuje kolejne elementy, zapisuje je w postaci programów, testuje i dodaje jeden po drugim do systemu. Metoda małych kroków minimalizuje skalę potencjalnych problemów.

Czasem zła metafora opisuje dobrą metodę. Próbuj wtedy zachować daną technikę i znaleźć dla niej lepsze porównanie. W tym technika przyrostowa może być pomocna, podczas gdy analogia do uprawy roślin nie sprawdza się.

Więcej informacji: Inną metaforę z uprawą roślin, odnoszącą się do konserwacji oprogramowania, można znaleźć w rozdziale „On the Origins of Designer Intuition" książki *Rethinking Systems Analysis and Design* (Weinberg 1988).

Idea wykonywania małych kroków może mieć coś wspólnego z uprawą zbóż, ale analogia z pracą na roli jest słaba i niesie ze sobą niewiele informacji. Łatwo zastąpić ją jedną z metafor opisywanych dalej. Ta jest trudna do rozwinięcia poza opis prostej idei wprowadzania niewielkich, odizolowanych zmian. Jeżeli podążysz tym tropem, co ilustruje rysunek 2.2, zaczniesz zastanawiać się nad planem nawożenia systemu, odchwaszczaniem szczegółów projektu, zwiększaniem ilości zbieranego kodu przez efektywne zarządzanie gruntami i żniwami. Będziesz rozmawiał o płodozmianie C++ i wstrzymywał obsiewanie części pól, aby odbudować poziom azotu w twardym dysku.

Słabość rolniczej metafory tkwi w tym, że sugeruje ona brak bezpośredniej kontroli nad rozwojem kodu. Siejesz jego ziarna na wiosnę. Jeżeli pogoda dopisze i partia pozwoli, zbierzesz piękny kod jesienią.

Rysunek 2.2. Trudno w pożyteczny sposób rozwinąć rolniczą metaforę rozwoju oprogramowania

Hodowla ostryg — przyrastanie systemu

Gdy ktoś mówi o „uprawianiu oprogramowania", często ma w rzeczywistości na myśli jego stopniowe przyrastanie (czy też narastanie). Metafora rolnicza i analogia do narastania są sobie bliskie, jednak ta druga pozwala uzyskać głębsze spojrzenie. Proces narastania można powiązać z wieloma obrazami, takimi jak obraz pereł — które powstają przez stopniowe dołączanie do jądra niewielkich ilości węglanu wapnia — albo obszaru lądu zwiększającego się w wyniku osadzania rozpuszczonych w wodzie substancji.

Patrz też: O stosowaniu metod przyrostowych w integracji systemów piszemy w podrozdziale 29.2 „Częstość integracji — końcowa czy przyrostowa?".

Nie oznacza to, że musisz nauczyć się wytwarzać kod z rozpuszczonych w wodzie minerałów, ale zwraca uwagę na fakt, że warto nauczyć się rozbudowywania systemów w wielu drobnych krokach. Przyrostowe projektowanie, budowanie i testowanie należą do najefektywniejszych technik inżynierii oprogramowania.

W przyrostowym procesie tworzenia oprogramowania praca rozpoczyna się od przygotowania najprostszej możliwej, ale już działającej wersji systemu. Nie musi ona pobierać właściwych danych wejściowych, nie musi wykonywać użytecznych operacji ani też generować właściwego wyniku — wystarczy, aby był to sam szkielet, który będzie wystarczająco mocny, aby dało się osadzić na nim właściwy system. Jego praca może sprowadzać się do wywoływania pustych klas dla każdej ze zidentyfikowanych funkcji systemu. Jest to początek podobny do ziarenka piasku, wokół którego ostryga tworzy nową perłę.

Po utworzeniu szkieletu krok po kroku, małymi etapami, osadzamy na nim mięśnie, inne tkanki, skórę. Zmieniamy kolejne puste klasy w rzeczywiste implementacje. Przechodzimy od symulacji odczytu danych wejściowych do ich rzeczywistego pobierania i przetwarzania oraz od symulacji generowania danych wyjściowych do wyprowadzania użytecznych informacji. Dodajemy kolejne elementy aż do uzyskania w pełni funkcjonalnego systemu.

Nie brakuje pośrednich dowodów potwierdzających zalety takiego podejścia. Fred Brooks, który w 1975 roku radził, aby zawczasu oswoić się z myślą, że każdy projekt wcześniej czy później pójdzie do kosza, dekadę po wydaniu bestsellera *Mityczny osobomiesiąc* przyznał, że nic nie miało takiego wpływu na jego pracę i jej efektywność jak metody programowania przyrostowego (2000). Tom Gilb zamieszcza podobne stwierdzenie w swojej przełomowej książce *Principles of Software Engineering Management* (1988), w której opisał metodykę Evolutionary Delivery i fundamentalne zasady stojące u podstaw popularnych dzisiaj procesów Agile. Z podejścia przyrostowego czerpią liczne współczesne metodyki programowania (Beck 2006, Cockburn 2008, Highsmith 2002, Reifer 2002, Martin 2003, Larman 2004).

Siła metafory programowania przyrostowego ma swoje źródło w tym, że sugeruje ona więcej, niż może w istocie dać. Trudniej o jej niewłaściwe rozwinięcie niż w przypadku metafory rolniczej. Obraz powoli formującej się wewnątrz ostrygi perły jest celną ilustracją realizowanego procesu.

Programista jako budowniczy — budowa oprogramowania

Obraz „budowania" oprogramowania jest bardziej praktyczny niż wizja jego „pisania" czy „hodowli". Nie jest on sprzeczny z ideą przyrastania, a wprowadza bardziej precyzyjne ukierunkowanie. Budowa implikuje różne fazy planowania, przygotowań i realizacji, które różnią się rodzajem i zakresem w zależności od tego, co jest budowane. Dalsza eksploracja tej metafory prowadzi do wielu użytecznych analogii.

Do budowy dwumetrowej wieży potrzeba pewnej ręki, płaskiej powierzchni i 10 nieuszkodzonych puszek po piwie. Do budowy wieży sto razy większej nie wystarczy po prostu sto razy więcej puszek. Wymagany jest zupełnie inny sposób planowania i realizacji projektu.

Jeżeli budujesz prostą konstrukcję — na przykład budę dla psa — możesz pojechać do supermarketu, kupić drewno i gwoździe, a do wieczora nowy dom dla Azora będzie gotowy. Jeżeli zapomnisz o wejściu (patrz rysunek 2.3) lub popełnisz jakiś inny błąd, nie jest to wielki kłopot. Możesz wprowadzić poprawki, a nawet zacząć od nowa. Wszystko, co możesz stracić, to jedno popołudnie. Takie swobodne podejście sprawdza się także w przypadku niewielkich projektów. Jeżeli napiszesz tysiąc wierszy kodu, kierując się kiepskim projektem, możesz zrobić refaktoryzację lub rozpocząć od nowa, a strata czasu nie jest duża.

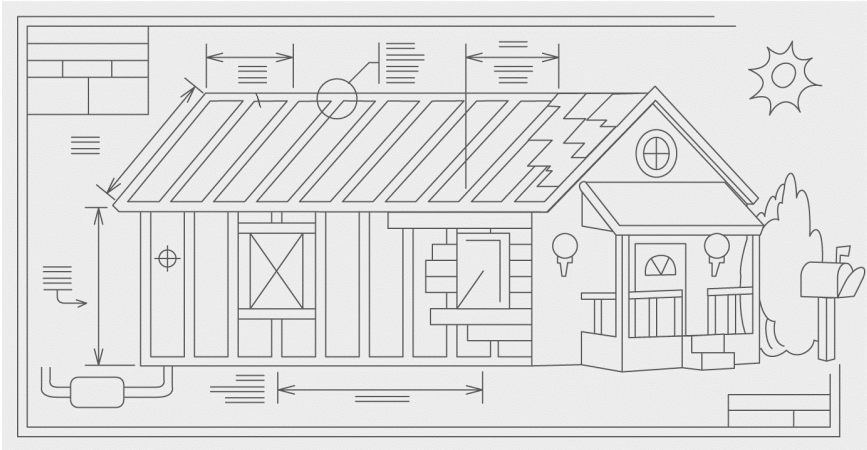
Jeżeli stawiasz dom, proces budowy jest bardziej złożony. Podobnie jest z konsekwencjami zastosowania niewłaściwego projektu. Najpierw decydujesz, jaki rodzaj domu ma zostać zbudowany — odpowiada to w programowaniu fazie

Rysunek 2.3. Koszt błędu popełnionego przy budowie niewielkiej konstrukcji to tylko odrobina czasu i, ewentualnie, zażenowania

definiowania problemu. Następnym krokiem jest opracowanie we współpracy z architektem ogólnego projektu i uzyskanie jego akceptacji, co jest analogią fazy projektowania architektury oprogramowania. Rysujesz plany i schematy, po czym podpisujesz umowę z wykonawcą. Przypomina to fazę projektowania szczegółowego w programowaniu. Przygotowujesz plac budowy, wylewasz fundament, stawiasz ściany i dach, montujesz instalacje. To etap programowania lub budowy oprogramowania. Gdy dom jest gotowy, architekci wnętrz, krajobrazu i dekoratorzy starają się nadać jego konstrukcji możliwie korzystny wygląd. Można to porównać do optymalizacji oprogramowania. W trakcie całego procesu towarzyszą nam najróżniejsi inspektorzy, którzy kontrolują przygotowanie placu budowy, wykonanie fundamentu, ścian, dachu, instalacji i innych elementów. Podobnie kontrolowane są poszczególne elementy oprogramowania i całość przygotowywanego systemu.

Większa złożoność i rozmiar prowadzą do poważniejszych konsekwencji tak przy budowaniu domów, jak i przy budowaniu oprogramowania. W pierwszym przypadku materiały są dość drogie, ale największym kosztem jest robocizna. Rozbieranie ściany i przesuwanie jej o 15 centymetrów jest kosztowne nie ze względu na zużycie gwoździ, ale dlatego, że trzeba zapłacić wykonawcy za dodatkowy czas. Projekt musi być jak najbardziej dopracowany (patrz rysunek 2.4), aby nie trzeba było tracić czasu na korygowanie błędów, których można było uniknąć. Przy budowie oprogramowania materiały są tańsze i wciąż koszt robocizny ma kluczowe znaczenie. Zmiana formatu raportów może być równie kosztowna jak przesunięcie ściany domu, ponieważ w obu sytuacjach głównym komponentem wydatków jest praca.

Czy można znaleźć inne analogie? Przy budowie domu korzystasz w miarę możliwości z gotowych elementów wyposażenia, takich jak pralka, zmywarka, lodówka i zamrażarka, które kupujesz. Jeżeli nie jesteś hobbystą, nie rozważasz konstruowania ich samodzielnie. Kupujesz też gotowe szafki, prefabrykowane i przycięte blaty, okna na wymiar, gotowe drzwi i wyposażenie łazienki. Podobnie postępujesz przy budowie systemu informatycznego. Korzystasz szeroko z możliwości języków wysokiego poziomu i nie bierzesz pod uwagę pisania kodu na poziomie systemu operacyjnego. Możesz używać także gotowych bibliotek klas kontenerowych, funkcji statystycznych, klas interfejsu użytkownika i klas współpracujących z bazami danych. Ogólnie rzecz biorąc, nie ma zazwyczaj sensu pisanie elementów kodu, które można kupić gotowe i dopracowane.



Rysunek 2.4. Bardziej złożone konstrukcje wymagają dokładniejszego planowania

Jeżeli jednak budujesz bardziej wymyślny dom, możesz rozważyć użycie szafek wykonanych na zamówienie. Możesz zadbać o to, aby zmywarka, lodówka i zamrażarka wyglądały jak jedna całość z pozostałym wyposażeniem kuchni. Możesz zamówić okna o oryginalnym kształcie i rozmiarze. Tego rodzaju postępowanie także ma odpowiedniki w świecie programowania. Pracując nad projektem bardziej wyspecjalizowanym czy o szczególnych wymaganiach, możesz posunąć się do napisania własnych funkcji wykonujących obliczenia matematyczne, aby zapewnić odpowiednią dokładność lub szybkość działania. Możesz budować własne klasy kontenerowe, interfejsu użytkownika i współpracy z bazami danych, aby uzyskać w pełni jednolitą i spójną konstrukcję systemu.

Zarówno budowa domów, jak i budowa oprogramowania jest bardzo wrażliwa na błędy w procesie planowania. Jeżeli kod jest pisany w niewłaściwej kolejności, praca programisty jest trudna, debugowanie uciążliwe, a projektowanie testów sprawia kłopot. Projekt trwa dłużej lub nie zostaje ukończony, bo zadania poszczególnych osób są nadmiernie skomplikowane, a całość mało zrozumiała.

Dobre planowanie nie musi oznaczać planowania absolutnie wszystkiego ani być zbyt szczegółowe. Możesz zaplanować rozwiązania konstrukcyjne budynku i odłożyć wybieranie rodzaju podłóg, kolorów ścian czy pokrycia dachu na później. Dobrze przemyślany projekt zwiększa swobodę późniejszej zmiany zdania co do detali. Dodatkowo, im więcej programista zyskuje doświadczenia w tworzeniu określonego rodzaju oprogramowania, tym więcej tych drobiazgów nabiera dla niego charakteru elementów oczywistych. Wyzwaniem jest stworzenie planu dokładnie na tyle szczegółowego, aby pominięcie czegoś nie zemściło się na dalszych etapach pracy.

Analogia do budowy domów pomaga także wyjaśnić, dlaczego różne projekty wymagają innego podejścia do procesu wytwarzania oprogramowania. W budownictwie również stosuje się inne reguły planowania, projektowania i kontroli jakości przy budowie magazynu i reaktora jądrowego czy szpitala. Inaczej buduje się szkołę, inaczej apartamentowiec, a inaczej dom w zabudowie willowej. Podobnie jest w programowaniu. Często spełniająca swoje zadanie

zasada stosowania „lekkiego", dynamicznego podejścia czasem musi zostać złamana na rzecz procesu w pełni sformalizowanego i skrajnie sztywnego, jeżeli wynika to z postawionych celów (na przykład dotyczących bezpieczeństwa).

Gdy rozważamy modyfikowanie projektów programistycznych, nasuwa się kolejna paralela z procesem budowlanym. Przesunięcie ściany o 15 centymetrów ma zupełnie inny wymiar, gdy mamy do czynienia ze ścianą nośną i gdy problem dotyczy ściany działowej. W podobny sposób zmiany struktury programu są bardziej kosztowne niż dodawanie czy usuwanie cech peryferyjnych.

Analogia budowlana daje też wartościowy wgląd w naturę projektów programistycznych o wyjątkowo dużych rozmiarach. Ponieważ kara za błąd w wielkich konstrukcjach jest dotkliwa, dotyczące ich plany i projekty wymagają wielokrotnej weryfikacji, dopracowania każdego szczegółu i niepozostawiającej wątpliwości całościowej wizji. Ponadto stosuje się pewien margines bezpieczeństwa — lepiej zapłacić 10 procent więcej za bardziej wytrzymały materiał, niż rozważać prawdopodobieństwo zawalenia się wieżowca. Dużą wagę przykłada się do charakterystyk czasowych. Gdy budowano Empire State Building, każdej ciężarówce dowożącej materiały przypisano piętnastominutowe „okno". Jeżeli któraś z nich nie dojechała punktualnie, cały projekt ulegał opóźnieniu.

Podobnie jest w świecie informatyki. Projekt wyjątkowo duży wymaga innego poziomu planowania niż typowe zlecenie o dużych rozmiarach. Capers Jones policzył, że system informatyczny o jednym milionie wierszy kodu wymaga średnio 69 różnych *rodzajów* dokumentacji (1998). Specyfikacja wymagań takiego systemu ma około 4 – 5 tysięcy stron, a dokumentacja projektowa szybko rozrasta się do rozmiarów dwu- i trzykrotnie większych. Trudno oczekiwać, aby jedna osoba była w stanie poznać całość projektu tych rozmiarów. Przeniesienie czynności przygotowawczych na inny poziom jest niezbędne.

Jeżeli mamy do czynienia z projektami, których wymiar ekonomiczny jest porównywalny z budową Empire State Building, rozwiązania organizacyjne i techniczne muszą być dopasowywane do wielkości przedsięwzięcia.

Patrz też: Ciekawe uwagi o rozwijaniu metafory budowy oprogramowania można znaleźć w artykule „What Supports the Roof?" (Starr 2003).

Metaforę budowlaną można rozwijać w wielu różnych innych kierunkach. W dużej mierze dowodzi to jej wartości. Wywodzi się z niej wiele popularnych w informatyce pojęć: architektura oprogramowania, szkielet, rusztowanie (ang. *scaffolding*), konstrukcja, klasy fundamentowe (ang. *foundation classes*) czy rozbieranie kodu. Czytelnik mógłby zapewne przytoczyć jeszcze kilka innych.

Przybornik programisty — narzędzia i moduły

Programiści, którzy sprawnie budują wysokiej jakości oprogramowanie, mają za sobą lata poznawania najróżniejszych technik, trików czy wręcz magicznych zaklęć. Te metody pracy nie są regułami. Są to raczej instrumenty analityczne. Dobry rzemieślnik wie, którego narzędzia użyć w danej sytuacji, i potrafi się nim posługiwać. Podobnie jest z programistami. Im większe doświadczenie, tym więcej narzędzi analitycznych w mentalnym przyborniku. Rośnie też wiedza, która dyktuje, kiedy i jak je stosować w taki sposób, aby uzyskać jak najlepszy efekt.

Patrz też: O wybieraniu i łączeniu metod stosowanych w procesie projektowania piszemy w podrozdziale 5.3 „Heurystyki — narzędzia projektanta".

Różni konsultanci polecają często, by w pracy z oprogramowaniem zdecydowanie przyjąć pewne metody postępowania, a odrzucić inne. Nie jest to korzystne, ponieważ bezkrytyczne, stuprocentowe przyjęcie jednej metodyki pracy szybko zawęża spojrzenie i sprawia, że wszystko widzimy przez jej pryzmat. W pewnych sytuacjach może to prowadzić do zignorowania nadarzających się możliwości użycia innych technik, bardziej pasujących do bieżącego problemu. Metafora przybornika przypomina o tym, że różne metody, techniki i wypracowane samodzielnie schematy to nie wszystko — potrzebna jest jeszcze perspektywa pozwalająca na stosowanie tych narzędzi, które najlepiej sprawdzą się w konkretnej sytuacji.

Łączenie metafor

Ponieważ metafory mają naturę heurystyczną, a nie algorytmiczną, nie wykluczają się wzajemnie. Możesz bazować na metaforze przyrastania i budowy jednocześnie. Jeśli odczuwasz taką potrzebę, możesz używać analogii do pisania i łączyć ją z metaforami jazdy samochodem, polowania na wilkołaki czy topienia w smole. Ważne jest to, która metafora lub połączenie metafor najlepiej pobudza proces myślowy i wspomaga komunikację w zespole.

Korzystanie z analogii wymaga zachowania pewnego poziomu czujności. Aby się nimi posługiwać, konieczne jest ich rozwijanie. Jednak metafora rozwinięta zbyt mocno lub w złym kierunku prowadzi na manowce. Podobnie jak przy korzystaniu z każdego narzędzia o dużych możliwościach, okazji do niewłaściwego stosowania metafor nie brakuje. Mimo to właśnie ich ogromny potencjał decyduje o tym, że są cennym elementem intelektualnego arsenału programisty.

Więcej informacji

cc2e.com/0285

Pośród książek o metaforach, modelach i paradygmatach kamieniem milowym było dzieło Thomasa Kuhna.

Kuhn, Thomas S., *Struktura rewolucji naukowych*, Aletheia, 2009. Książka Kuhna, traktująca o tym, w jaki sposób teorie naukowe powstają, ewoluują i stapiają się z innymi w darwinowskim cyklu przemian, wzbudziła w 1962 roku, gdy ukazała się po raz pierwszy, wielkie zainteresowanie w świecie filozofii nauki. Jest czytelna i krótka, a zarazem nie brakuje w niej ciekawych przykładów powstawania i upadku metafor, modeli i paradygmatów w nauce.

Floyd, Robert W., „The Paradigms of Programming", odczyt na ceremonii wręczenia Nagrody Turinga w 1978 roku, *Communications of the ACM*, sierpień 1979, s. 455 – 460. Niezwykle wciągające omówienie zagadnienia modeli w procesie wytwarzania oprogramowania z wieloma odniesieniami do idei zaprezentowanych wcześniej przez Kuhna.

Podsumowanie

- Metafory to heurystyki, a nie algorytmy. Jako takie nie zawsze prowadzą najkrótszą drogą do celu.
- Metafory pomagają zrozumieć proces tworzenia i rozwijania oprogramowania poprzez przyrównanie go do innych działań, o których posiadamy szerszą wiedzę.
- Są metafory lepsze i gorsze.
- Przyrównanie budowy oprogramowania do budowy domu lub innej konstrukcji przypomina o konieczności podjęcia przemyślanych przygotowań oraz zwraca uwagę na różnicę między małymi i dużymi projektami.
- Przyrównanie technik tworzenia i rozwijania oprogramowania do narzędzi w intelektualnym przyborniku zwraca uwagę na bogactwo dostępnych metod pracy i na to, że nie można w każdym projekcie bazować na tych samych. Umiejętność wybierania właściwych narzędzi, odpowiednich dla rozwiązywanego problemu, to klucz do efektywnego programowania.
- Metafory nie wykluczają się wzajemnie. Można korzystać z takiego ich wyboru, jaki najlepiej sprawdza się w indywidualnej praktyce.

Rozdział 3.

Przed programowaniem — przygotowania

cc2e.com/0309

W tym rozdziale

Podobne tematy

- Kluczowe decyzje konstrukcyjne: rozdział 4.
- Wpływ rozmiaru projektu na proces budowy oprogramowania i przygotowania: rozdział 27.
- Relacja między celami jakościowymi a przebiegiem procesu budowy oprogramowania: rozdział 20.
- Kierowanie procesem budowy oprogramowania: rozdział 28.
- Projektowanie: rozdział 5.

Przed rozpoczęciem budowy domu budowniczy przegląda schematy, sprawdza, czy uzyskano wszystkie niezbędne zezwolenia, i bada podstawę fundamentu. Inaczej wygląda to przy wznoszeniu wieżowca, inaczej przy budowie osiedla domków jednorodzinnych, a jeszcze inaczej przy składaniu budy dla psa. Niezależnie od projektu zakres przygotowań jest dostosowany do jego specyfiki, a związane z nimi czynności są wykonywane przed rozpoczęciem właściwej budowy.

W tym rozdziale zajmiemy się zadaniami, które muszą zostać wykonane, aby przygotować proces budowy oprogramowania. Podobnie jak w pracy budowniczego, powodzenie projektu zależy w dużej mierze od poziomu przygotowania do jego realizacji. Jeżeli fundament ma wady albo plany są niekompletne, jedyne, co można zrobić, gdy rozpocznie się już proces budowy, to ograniczać powstałe zagrożenia.

Stara ciesielska zasada „zanim przetniesz, zmierz dwa razy" doskonale pasuje do programistycznej części procesu wytwarzania oprogramowania, która może pochłaniać nawet 65 procent całkowitych kosztów projektu. W najgorszych projektach wykonywana w tym etapie praca powtarzana jest dwa, trzy, a nawet więcej razy. Powtarzanie najbardziej kosztowej fazy przedsięwzięcia jest poważnym problemem niezależnie od jego charakteru.

Choć w tym rozdziale piszemy o podstawowych warunkach powodzenia w budowie oprogramowania, nie omawiamy związanych z nimi zagadnień bezpośrednio. Jeżeli szukasz bardziej konkretnych porad lub temat cyklu życia w inżynierii oprogramowania jest Ci już dobrze znany, możesz przejść od razu do bardziej praktycznego rozdziału 5. „Projektowanie". Jeżeli nie podoba Ci się myśl o długich przygotowaniach do programowania, zapoznaj się z podrozdziałem 3.2 „Określanie rodzaju budowanego oprogramowania", aby zobaczyć, jakie czynności wstępne znajdują zastosowanie w Twojej sytuacji, a następnie przejrzyj dane w podrozdziale 3.1, gdzie opisujemy koszty pominięcia stosownych przygotowań.

3.1. Przygotowania i ich znaczenie

Patrz też: Położenie dużego nacisku na jakość jest też najlepszym sposobem poprawienia efektywności pracy. Piszemy o tym w podrozdziale 20.5 „Ogólna Zasada Jakości Oprogramowania".

Cechą, która łączy programistów tworzących wysokiej jakości oprogramowanie, jest stosowanie wysokiej jakości metod pracy. Ich „dobre praktyki programowania" zapewniają utrzymanie odpowiedniego poziomu na początku, w trakcie trwania, a także na końcowych etapach projektu.

Jeżeli przykładamy wagę do jakości w końcowej fazie pracy nad programem, to przykładamy wagę do testowania systemowego. Większość osób, myśląc o kontroli jakości oprogramowania, ma na uwadze właśnie testowanie, jednak jest ono tylko jedną z części pełnej strategii zapewniania jakości. Przy tym nie jest to część o największym znaczeniu. Testowanie nie umożliwia wykrycia problemów takich jak to, że budowany jest inny produkt niż zamierzono albo że budowa właściwego produktu przebiega w niewłaściwy sposób. Braki tego rodzaju muszą zostać rozpoznane wcześniej, na początku procesu budowy.

Jeżeli przywiązujemy wagę do jakości w trakcie trwania projektu, zajmujemy się w rzeczywistości metodami budowy oprogramowania. O nich głównie traktuje ta książka.

Jeżeli ważny jest dla nas poziom jakości na początku projektu, to planujemy i projektujemy produkt naprawdę wysokiej jakości. Jeżeli rozpoczniemy pracę od zaprojektowania samochodu pontiac aztek, to niezależnie od tego, jak wyszukane testy będziemy wykonywać później, nigdy nie doprowadzą one do konkluzji, że stworzyliśmy rolls-royce'a. Możemy zbudować najlepszego na świecie azteka, ale jeżeli chcemy rolls-royce'a, musimy planować jego budowę od samego początku. W programowaniu planowanie takie następuje w fazach definiowania problemu, określania rozwiązania i projektowania tego rozwiązania.

Ponieważ programowanie znajduje się w centrum procesu wytwarzania oprogramowania, w chwili gdy dochodzimy do tego etapu, istnieją już pewne artefakty wytworzone wcześniej. W dużej mierze to w nich zapisany jest przyszły sukces lub niepowodzenie. Gdy rozpoczyna się programowanie, powinniśmy jednak być w stanie przynajmniej określić, w jakiej znaleźliśmy się sytuacji. Jeżeli widzimy, że jest źle, nie możemy tego ignorować. W dalszej części rozdziału wyjaśnimy bardziej szczegółowo, dlaczego właściwe przygotowania są tak ważne i w jaki sposób stwierdzić, czy są wystarczające, aby rozpocząć budowę.

Czy współcześnie przygotowania wciąż obowiązują?

Wykorzystywana metodyka pracy powinna opierać się na tym, co najnowsze i najlepsze, a nie na ignorancji. Odpowiednio należy też wykorzystać to, co starsze i sprawdzone.
— Harlan Mills

W ostatnim czasie wiele osób skłania się ku tezie, że czynności początkowe, takie jak projektowanie architektury, projektowanie szczegółowe i planowanie projektu, to przeżytek, który nie ma zastosowania we współczesnych projektach. Ogólnie rzecz biorąc, stwierdzenia tego rodzaju nie zostały rzetelnie uzasadnione żadnymi danymi ani badaniami, czy to wykonywanymi w przeszłości, czy współczesnymi (więcej na ten temat w dalszej części rozdziału). Przeciwnicy czynności wstępnych pokazują często przykłady projektów, w których przygotowania pozostawiały wiele do życzenia, i wnioskują z tego, że poświęcony na nie czas nie był czasem wykorzystanym efektywnie. Jednak czynności wstępne można wykonać dobrze, a gromadzone od lat siedemdziesiątych dane przekonująco dowodzą, że projekty, które przebiegają najsprawniej, to te, w których przed rozpoczęciem programowania poczyniono odpowiednie przygotowania.

Głównym celem przygotowań jest zmniejszenie ryzyka: dobry projekt zapewnia możliwie wczesną eliminację głównych zagrożeń, dzięki czemu jego realizacja przez większość czasu przebiega bez zakłóceń. Bez wątpienia najczęściej spotykane zagrożenia to mało precyzyjne określenie wymagań i zły plan projektu. Stąd też koncentracja czynności przygotowawczych na wymaganiach i planowaniu.

Przygotowania do budowy oprogramowania nie są nauką ścisłą i podejście do metod zmniejszania ryzyka musi być dostosowane do konkretnego projektu. Szczegóły mogą różnić się bardzo znacznie. Więcej na ten temat w podrozdziale 3.2.

Przyczyny braków w przygotowaniu projektu

Mogłoby się wydawać, że każdy zawodowy programista zna znaczenie przygotowań i jeszcze przed wzięciem się do pracy sprawdza, czy wszystkie niezbędne kroki wstępne zostały wykonane. Niestety, rzeczywistość jest inna.

Patrz też: Opis programu rozwoju zawodowego, który pomaga w przyswojeniu takich umiejętności, można znaleźć w rozdziale 16. książki *Professional Software Development* (McConnell 2004).

Typową przyczyną braków w procesie przygotowawczym jest to, że odpowiedzialni za jego przeprowadzenie programiści nie dysponują odpowiednim poziomem wiedzy. Umiejętności wymagane do stworzenia planu projektu, przekonującego studium biznesowego, pełnej i dokładnej specyfikacji wymagań oraz wysokiej jakości architektury nie należą do banalnych, a większość programistów nie ma okazji zdobyć rzetelnej wiedzy w tym zakresie. Gdy nie wiedzą oni, jak przeprowadzić przygotowania, mówienie o „zwiększeniu zakresu czynności wstępnych" traci sens. Jeżeli nie można czegoś zrobić poprawnie, jakiekolwiek pogłębianie procesu nic nie wniesie. Pełny opis działań poprzedzających programowanie wykracza poza tematykę tej książki, ale sekcja „Więcej informacji" na końcu tego rozdziału wskazuje podręczniki, które umożliwiają zdobycie potrzebnej wiedzy.

cc2e.com/0316

Niektórzy programiści wiedzą, jak przeprowadzić przygotowania, ale pomijają je, bo nie potrafią oprzeć się pragnieniu jak najszybszego rozpoczęcia programowania. Jeżeli masz takie skłonności, miałbym dwie sugestie. Sugestia 1.:

przeczytaj następny podrozdział. Być może znajdziesz w nim zagadnienia, o których dotąd nie pomyślałeś. Sugestia 2.: zwracaj uwagę na napotykane problemy. Już po napisaniu kilku większych programów łatwo dojść do wniosku, że wcześniejsze planowanie pozwoliłoby uniknąć wielu kłopotliwych sytuacji. Niech Twoim przewodnikiem będzie własne doświadczenie.

Ostatnią przyczyną nieprzykładania się do przygotowań jest to, że kierownictwo rzadko patrzy przychylnie na programistę, który zajmuje się czymś innym niż pisanie kodu. Barry Boehm, Grady Booch, Karl Wiegers i inni od 25 lat starają się zwrócić uwagę na znaczenie właściwego przygotowania wymagań i projektu, można by więc oczekiwać, że każdy menedżer słyszał, iż tworzenie oprogramowania to nie tylko pisanie kodu. Życie jednak toczy się swoim trybem.

Więcej informacji: Wiele ciekawych odmian tego schematu opisuje Gerald Weinberg w swojej znanej książce *The Psychology of Computer Programming* (Weinberg 1998).

Kilka lat temu pracowałem nad projektem zleconym przez ministerstwo obrony, w którym opracowywanie specyfikacji wymagań było bardzo znaczącą częścią pracy. Gdy przybył z wizytą jeden z ważnych generałów, wyjaśniliśmy mu, że pracujemy nad specyfikacją wymagań i głównie rozmawiamy z klientami, określamy ich potrzeby oraz przygotowujemy zarys projektu. Generał jednak upierał się, że chce zobaczyć kod. Powiedzieliśmy mu, że nie ma jeszcze żadnego kodu, na co ten, zdeterminowany, obszedł dookoła sto stanowisk w poszukiwaniu choć jednej osoby pracującej z rzeczywistym kodem. Zniechęcony widokiem tak wielu ludzi, którzy nie siedzieli przy swoich biurkach, lecz pracowali nad wymaganiami i projektem, ten wielki, postawny mężczyzna o donośnym głosie wskazał w końcu inżyniera siedzącego obok mnie i wykrzyknął: „A co on robi? On wyraźnie pisze program!". W rzeczywistości inżynier pracował nad narzędziem do formatowania dokumentów, ale generał pragnął widzieć kod, zobaczył coś, co na niego wyglądało, i chciał myśleć, że inżynier pracuje z kodem, więc ostatecznie przyznaliśmy mu rację.

To zjawisko określa się czasem jako syndrom WISCA lub WIMP: „Why Isn't Sam Coding Anything?" lub „Why Isn't Mary Programming?" („Dlaczego Jasiu/Marysia nie zajmuje się programowaniem?").

Jeżeli menedżer projektu wciela się w rolę generała brygady i rozkazuje natychmiastowe rozpoczęcie programowania, łatwo odpowiedzieć „Tak jest!". (Komu to szkodzi? Starszy człowiek pewnie ma w tym jakiś cel). Nie jest to dobra odpowiedź. Istnieje co najmniej kilka lepszych. Przede wszystkim możesz bezpośrednio odmówić wykonywania swoich zadań w kolejności, która jest dalece nieefektywna. Jeżeli Twoje relacje z szefem i Twoje konto w banku są na tyle zdrowe, abyś mógł sobie na to pozwolić, życzę powodzenia.

Drugą kontrowersyjną alternatywą jest udawanie. Połóż z brzegu biurka listing starego programu i weź się za pracę nad wymaganiami oraz architekturą niezależnie od akceptacji szefa — projekt zostanie zrealizowany szybciej i lepiej. Choć w niektórych takie rozwiązanie budzi wątpliwości natury etycznej, to z punktu widzenia przełożonego ignorowanie go będzie w tym przypadku błogosławieństwem.

Trzecią możliwością jest uświadamianie i edukacja, czyli wyjaśnianie szefowi, na czym polega specyfika pracy nad projektem technicznym. Jest to podejście

o tyle dobre, że zwiększa liczbę oświeconych przełożonych — możesz mieć poczucie, że zrobiłeś coś naprawdę dobrego dla świata. W kolejnym podrozdziale przedstawię szczegółowe uzasadnienie konieczności poświęcenia pewnej ilości czasu na przygotowanie do budowy oprogramowania.

Ostatnim rozwiązaniem jest zmiana pracodawcy. Choć sytuacja ekonomiczna jest raz lepsza, a raz gorsza, dobrych programistów praktycznie zawsze brakuje (BLS 2002), a życie jest zbyt krótkie, aby spędzać je na pracy, której mimo najlepszych chęci nie można wykonywać rzetelnie.

Przekonujące uzasadnienie konieczności wykonania przed programowaniem czynności wstępnych

Załóżmy, że zdobyłeś już szczyt definicji problemu, przeszedłeś milę z wymaganiami, zrzuciłeś odzienie przy fontannie architektury i zanurzyłeś się w czystych wodach pełnej gotowości. Wiesz już, że przed implementacją musisz dokładnie wiedzieć, co system ma robić i jak.

Częścią Twojej pracy, jako osoby zajmującej się zagadnieniami technicznymi, jest edukowanie — wyjaśnianie tym, którzy nie mają wykształcenia inżynierskiego, na czym polega realizowany proces wytwarzania oprogramowania. Ten podrozdział ma za zadanie pomóc w radzeniu sobie z menedżerami i przełożonymi, którzy nie do końca orientują się, czego powinni oczekiwać. Jest to rozbudowane uzasadnienie konieczności poświęcenia pewnej ilości czasu na opracowanie wymagań i architektury — ustalenie krytycznych cech systemu — przed rozpoczęciem programowania, testowania i debugowania. Naucz się na pamięć przedstawionych tu argumentów i usiądź z szefem na długą, szczerą rozmowę o naturze całego procesu.

Oprzyj się na logice

Jedną z podstawowych zasad efektywnego programowania jest przykładanie stosownej wagi do przygotowań. Łatwo zrozumieć, że przed rozpoczęciem pracy nad dużym projektem trzeba zaplanować jego przebieg. Duże projekty wymagają więcej planowania, małe wymagają mniej. Od strony zarządzania procesem planowanie to określanie ilości czasu oraz liczby osób i niezbędnych komputerów. Od strony technicznej jest to uściślanie, co ma zostać zbudowane, czyli zabezpieczanie się przed marnotrawieniem pieniędzy na tworzenie czegoś innego niż zamierzono. Czasem użytkownicy nie mają na początku wyraźnego obrazu tego, czego właściwie potrzebują. Wymusza to poświęcenie większej ilości czasu, niż mogłoby się z pozoru wydawać, na dokładne ustalenie, jaki produkt czy system jest w istocie pożądany. Jest to jednak tańsze niż pisanie nietrafionego programu, wyrzucanie pracy do kosza i rozpoczynanie od nowa.

Ważne jest także, aby już przed rozpoczęciem pracy zastanowić się nad tym, jak zbudować właściwy system. Nie chcesz przecież tracić czasu i pieniędzy na podążanie ślepymi uliczkami, gdy nie jest to konieczne, zwłaszcza że zwiększa to sumę kosztów.

Oprzyj się na analogiach

Budowanie oprogramowania nie różni się tak bardzo od innych procesów, które wymagają zaangażowania ludzi i pieniędzy. Jeżeli budujesz dom, przed wbiciem pierwszego gwoździa (czy wylaniem pierwszej tony betonu) dysponujesz wszystkimi projektami i schematami. Co więcej, wszystkie dokumenty są wielokrotnie przejrzane i roją się od pieczątek oraz podpisów potwierdzających ich poprawność. Plany tego rodzaju są równie ważne w budowie oprogramowania.

Nie rozpoczynasz dekorowania choinki przed umieszczeniem jej na stojaku. Nie rozpalasz w piecu przed otwarciem komina. Nie wyjeżdżasz w podróż z pustym bakiem. Nie ubierasz się przed kąpielą i nie wkładasz butów przed założeniem skarpetek. Przy pracy z oprogramowaniem również istnieje pewna właściwa kolejność działań.

Programiści znajdują się na końcu łańcucha pokarmowego, jakim jest proces wytwarzania oprogramowania. Pokarmem architektów są wymagania, pokarmem projektantów jest architektura, a pokarmem koderów jest projekt.

Można porównać ten łańcuch pokarmowy z tym, który występuje w świecie istot żywych. W czystym środowisku naturalnym mewy odżywiają się świeżymi łososiami. Łososie są pożywne, bo jadły świeże śledzie, a te z kolei jadły inne żyjątka wodne, równie zdrowe jak one. Łańcuch pokarmowy pozostaje zdrowy tak długo, jak długo zdrowe są wszystkie jego elementy. Podobnie jest w programowaniu — jeżeli na każdym etapie wszystko jest zdrowe i naturalne, uzyskujemy zdrowy kod pisany przez zadowolonych programistów.

Gdy środowisko ulega zanieczyszczeniu, żyjątka wodne pływają w odpadach z elektrowni jądrowej, mięso śledzia zawiera związki PCB, a łosoś po zjedzeniu go przepływa przez wyciek ropy. Biedne mewy znajdują się niestety na końcu łańcucha pokarmowego, więc ich problemy nie kończą się na ropie, przez którą przepłynął łosoś. Zjadają też związki PCB i pozostałości z odpadów nuklearnych. W programowaniu zanieczyszczona specyfikacja wymagań zniekształca architekturę, a ta z kolei sprowadza proces budowy oprogramowania na manowce. Programiści stają się nieznośni i niedożywieni, a radioaktywne, zanieczyszczone oprogramowanie roi się od defektów.

Jeżeli planujesz projekt o wysokim stopniu iteracyjności, musisz określić krytyczne wymagania i elementy architektury dla każdej z budowanych części jeszcze przed rozpoczęciem programowania. Budowniczy, który stawia osiedle domków jednorodzinnych, nie musi znać wszystkich szczegółów każdego jednego domu, zanim rozpocznie prace nad pierwszym. Musi jednak poznać plac budowy, określić przebieg rur kanalizacyjnych, rozkład przewodów elektrycznych i wiele innych elementów. Jeżeli zaniedba przygotowania, ukończenie projektu może zostać opóźnione przez konieczność przeprowadzenia kanalizacji pod zbudowanym już domem.

Oprzyj się na danych

Przeprowadzone na przestrzeni ostatniego ćwierćwiecza badania przekonująco dowiodły, że opłaca się podejmować właściwe, trafne działania już przy pierwszym podejściu. Zmiany, których można było uniknąć, zawsze drogo kosztują.

Badacze z firm Hewlett-Packard, IBM, Hughes Aircraft, TRW i innych stwierdzili, że wykrycie błędu przed rozpoczęciem budowy oprogramowania pozwala wprowadzić zmianę od 10 do 100 razy tańszym kosztem niż w przypadkach, gdy następuje to w ostatniej części procesu, w fazie testów systemowych lub po wdrożeniu wersji (Fagan 1976; Humphrey, Snyder i Willis 1991; Leffingwell 1997; Willis et al. 1998; Grady 1999; Shull et al. 2002; Boehm i Turner 2004).

Ogólną zasadą jest dążenie do wykrycia błędu w jak najkrótszym czasie po jego wprowadzeniu. Im dłużej defekt pozostaje w łańcuchu pokarmowym oprogramowania, tym więcej szkód powoduje w jego dalszych ogniwach. Ponieważ rozpoczynamy pracę od określenia wymagań, braki w ich specyfikacji mają ogromną inklinację do utrzymywania się w systemie, jak również generowania kosztów. Defekty powstałe na początku mają też zazwyczaj znacznie szersze skutki niż te, które pojawiają się w późniejszych etapach pracy. To kolejna przyczyna tego, że błędy popełnione na początku należą do najbardziej kosztownych.

Tabela 3.1 przedstawia relacje między kosztami usuwania defektów w zależności od momentu ich wprowadzenia i momentu ich wykrycia.

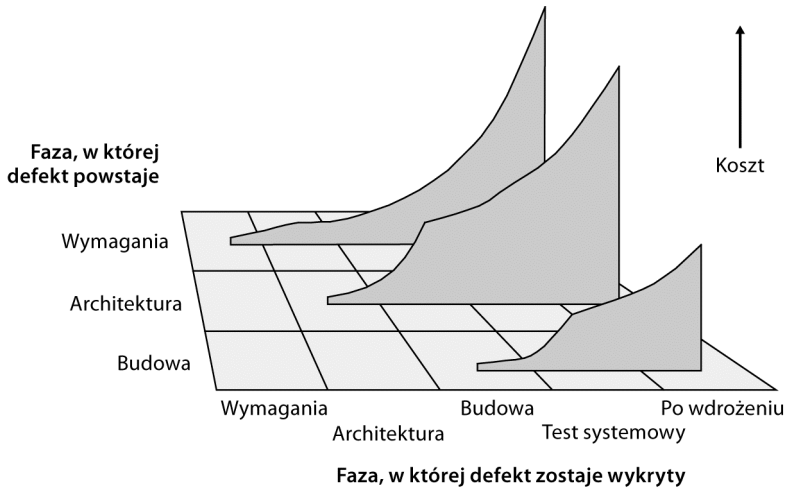
Tabela 3.1. Przeciętny koszt usuwania defektów w zależności od chwili ich wprowadzenia i wykrycia

Wprowadzenie defektu	**Wykrycie defektu**				
	Wymagania	Architektura	Budowa	Test systemowy	Po wdrożeniu
Wymagania	1	3	5 – 10	10	10 – 100
Architektura	—	1	10	15	25 – 100
Budowa	—	—	1	10	10 – 25

Źródło: Na podstawie „Design and Code Inspections to Reduce Errors in Program Development" (Fagan 1976), *Software Defect Removal* (Dunn 1984), „Software Process Improvement at Hughes Aircraft" (Humphrey, Snyder i Willis 1991), „Calculating the Return on Investment from More Effective Requirements Management" (Leffingwell 1997), „Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process" (Willis et al. 1998), „An Economic Release Decision Model: Insights into Software Project Management" (Grady 1999), „What We Have Learned About Fighting Defects" (Shull et al. 2002) i *Balancing Agility and Discipline: A Guide for the Perplexed* (Boehm i Turner 2004).

Dane z tabeli 3.1 pokazują, że na przykład defekt architektury, którego naprawienie kosztuje tysiąc dolarów w czasie prac nad architekturą, może prowadzić do kosztu rzędu 15 tysięcy dolarów, jeżeli zostanie wykryty w testach systemowych. Rysunek 3.1 przedstawia te same dane w postaci graficznej

W typowym projekcie większość działań ukierunkowanych na usuwanie defektów wciąż ma miejsce w obszarze odpowiadającym prawej stronie rysunku 3.1, co oznacza, że debugowanie i związane z nim zmiany zajmują w przeciętnym cyklu rozwoju oprogramowania około 50 procent czasu (Mills 1983; Boehm



Rysunek 3.1. Koszt usunięcia defektu rośnie gwałtownie wraz z czasem, który upływa między jego wprowadzeniem a wykryciem. Zależność taka obowiązuje zarówno w projektach realizowanych sekwencyjnie (100 procent wymagań i kompletny projekt na początku), jak i w tych iteracyjnych (5 procent wymagań i projektu na początku)

1987a; Cooper i Mullen 1993; Fishman 1996; Haley 1996; Wheeler, Brykczynski i Meeson 1996; Jones 1998; Shull et al. 2002; Wiegers 2002). Dziesiątki firm przekonały się, że sama koncentracja na usuwaniu defektów we wczesnych etapach projektu może obniżyć koszty i skrócić czas realizacji o połowę lub więcej (McConnell 2004). Jest to bardzo rzetelny argument za tym, aby wyszukiwać i usuwać wszelkie problemy tak wcześnie, jak to tylko możliwe.

Test gotowości przełożonego

Gdy wydaje Ci się, że szef rozumie, jak ważne jest wykonanie odpowiednich czynności przygotowawczych przed rozpoczęciem budowy oprogramowania, skorzystaj z poniższego testu, aby upewnić się, że faktycznie tak jest.

Które z poniższych stwierdzeń to samospełniające się przepowiednie?

- Lepiej zacznijmy kodowanie od razu, bo przed nami dużo debugowania.
- Nie planujemy poświęcać wiele czasu na testy, bo nie spodziewamy się wielu defektów.
- Zbadaliśmy wymagania i projekt tak dokładnie, że trudno mi wyobrazić sobie jakiekolwiek większe problemy w tracie pisania kodu i debugowania.

Wszystkie powyższe stwierdzenia są samospełniającymi się przepowiedniami. Dąż do tego, aby pewnego dnia wypowiedzieć ostatnie z nich.

Jeżeli wciąż nie jesteś przekonany co do tego, że czynności przygotowawcze są konieczne w realizowanym projekcie, przejdź do lektury następnego podrozdziału.

3.2. Określanie rodzaju budowanego oprogramowania

Capers Jones, kierownik naukowy w Software Productivity Research, podsumował 20 lat badań nad oprogramowaniem stwierdzeniem, że on i jego współpracownicy widzieli 40 różnych metod gromadzenia wymagań, 50 sposobów pracy nad projektem oprogramowania i 30 systemów testujących zastosowanych w projektach bazujących na 700 różnych językach programowania (Jones 2003).

Różne rodzaje zadań wymagają odmiennych proporcji czynności przygotowawczych i programowania. Każdy projekt jest inny, ale można wyróżnić kilka podstawowych stylów pracy. Tabela 3.2 przedstawia trzy najpopularniejsze rodzaje projektów i wymienia praktyki, których stosowanie jest zazwyczaj najbardziej dopasowane do ich potrzeb.

Tabela 3.2. Trzy najpopularniejsze rodzaje projektów i najlepiej sprawdzające się w nich praktyki programistyczne

	Rodzaj oprogramowania		
	Systemy biznesowe	**Systemy pracy ciągłej**	**Osadzone systemy wysokiego bezpieczeństwa**
Typowe aplikacje	Witryny internetowe Witryny intranetowe Zarządzanie magazynem Gry Systemy informacyjne Systemy księgowe	Oprogramowanie osadzone Gry Witryny internetowe Aplikacje desktopowe Narzędzia systemowe Usługi sieciowe	Oprogramowanie lotnicze Oprogramowanie osadzone Urządzenia medyczne Systemy operacyjne Aplikacje desktopowe
Modele cyklu życia	Procesy Agile (Extreme Programming, Scrum, RAD itp.) Prototypowanie ewolucyjne	Produkcja etapami Ewolucyjny model programowania Spiralny model programowania	Produkcja etapami Spiralny model programowania Ewolucyjny model programowania
Planowanie i zarządzanie	Przyrostowe planowanie projektu Testowanie i kontrola jakości warunkowane potrzebami Nieformalna kontrola zmian	Planowanie „z góry” Pełne planowanie testów Kontrola jakości warunkowana potrzebami Formalna kontrola zmian	Rozbudowane planowanie „z góry” Rozbudowane planowanie testów Rozbudowane planowanie kontroli jakości Rygorystyczna kontrola zmian
Wymagania	Nieformalna specyfikacja wymagań	Półformalna specyfikacja wymagań Przeglądy wymagań warunkowane potrzebami	Formalna specyfikacja wymagań Formalne przeglądy wymagań

Tabela 3.2. Trzy najpopularniejsze rodzaje projektów i najlepiej sprawdzające się w nich praktyki programistyczne — *ciąg dalszy*

	Rodzaj oprogramowania		
	Systemy biznesowe	**Systemy pracy ciągłej**	**Osadzone systemy wysokiego bezpieczeństwa**
Projektowanie	Projektowanie i pisanie kodu jako połączony proces	Projektowanie architektury Nieformalne projektowanie szczegółowe Przeglądy projektu warunkowane potrzebami	Projektowanie architektury Formalne przeglądy architektury Formalne projektowanie szczegółowe Formalne przeglądy projektów szczegółowych
Budowa	Programowanie w parach lub indywidualnie Nieformalna procedura zdawania kodu lub brak takiej procedury	Programowanie w parach lub indywidualnie Nieformalna procedura zdawania kodu Przeglądy kodu warunkowane potrzebami	Programowanie w parach lub indywidualnie Formalna procedura zdawania kodu Formalne przeglądy kodu
Testowanie i kontrola jakości	Programiści testują własny kod Programowanie metodą „najpierw test" Niewielka liczba lub brak testów wykonywanych przez odrębną grupę	Programiści testują własny kod Programowanie metodą „najpierw test" Odrębna grupa testująca	Programiści testują własny kod Programowanie metodą „najpierw test" Odrębna grupa testująca Odrębna grupa kontroli jakości
Wdrożenie	Nieformalna procedura wdrożenia	Formalna procedura wdrożenia	Formalna procedura wdrożenia

W rzeczywistych projektach napotkasz niezliczone odmiany trzech przedstawionych powyżej schematów. Mimo to widoczne w tabeli uogólnienia mają istotną wartość. Projekty systemów biznesowych zyskują zazwyczaj na podejściu iteracyjnym, w którym planowanie, opracowywanie wymagań i praca nad architekturą przeplatają się z budową programu, testowaniem systemowym i kontrolą jakości. Systemy, których wady powodują zagrożenie życia, wymagają najczęściej podejścia sekwencyjnego — stabilność wymagań jest tu jednym z warunków zapewnienia najwyższego poziomu niezawodności.

Wpływ podejścia iteracyjnego na przygotowania

Niektórzy autorzy przyjmują tezę, że projekty, w których stosuje się metody iteracyjne, nie wymagają koncentracji na przygotowaniach. Nie jest to podejście uzasadnione. Metody iteracyjne sprzyjają redukcji ilości błędów i braków w fazach przygotowawczych, ale nie eliminują ich. Spójrz na przedstawione w tabeli 3.3 przykłady projektów, w których brak koncentracji na przygotowaniach. Jeden jest realizowany sekwencyjnie i defekty są wykrywane wyłącznie przy testowaniu. Drugi jest realizowany iteracyjnie i błędy są wykrywane

w trakcie całego procesu. W pierwszym ich usuwanie następuje prawie wyłącznie na końcu projektu, co zwiększa koszty (patrz tabela 3.1). W podejściu iteracyjnym poprawki są wprowadzane przez cały czas trwania prac, czego efektem jest ogólne zmniejszenie wydatków. Dane zamieszczone w tej i następnej tabeli mają wyłącznie charakter ilustracyjny, ale proporcje kosztów przy zastosowaniu tych dwóch ogólnych metod pracy znajdują szerokie potwierdzenie w przytoczonych wcześniej w tym rozdziale badaniach.

Tabela 3.3. Skutki pominięcia przygotowań w projekcie sekwencyjnym i iteracyjnym

	Podejście 1.: Sekwencyjne bez przygotowań		**Podejście 2.: Iteracyjne bez przygotowań**	
Stopień zaawansowania projektu	Koszt pracy	Koszt przebudowy	Koszt pracy	Koszt przebudowy
20%	$ 100 000	$ 0	$ 100 000	$ 75 000
40%	$ 100 000	$ 0	$ 100 000	$ 75 000
60%	$ 100 000	$ 0	$ 100 000	$ 75 000
80%	$ 100 000	$ 0	$ 100 000	$ 75 000
100%	$ 100 000	$ 0	$ 100 000	$ 75 000
Przebudowa na końcu projektu	$ 0	$ 500 000	$ 0	$ 0
Suma	$ 500 000	$ 500 000	$ 500 000	$ 375 000
Suma końcowa		$ 1 000 000		$ 875 000

Projekt iteracyjny, w którym przygotowania zostają skrócone lub wyeliminowane, różni się od projektu sekwencyjnego, w którym zrobiono to samo, na dwa sposoby. Po pierwsze, średni koszt usunięcia defektów jest niższy, ponieważ są one wykrywane w krótszym czasie od ich powstania. Defekty te zostają jednak wykryte stosunkowo późno w obrębie iteracji, a pozbycie się ich wymaga, aby część oprogramowania została przeprojektowana lub napisana od nowa. Niezbędne jest też powtórzenie testów. W efekcie koszt usuwania błędów jest wyższy, niż to konieczne.

Po drugie, przy podejściu iteracyjnym koszty są generowane na bieżąco, stopniowo, w trakcie trwania projektu, a nie przesunięte na jego koniec. W ostatecznym rozrachunku koszt całkowity będzie podobny, choć nie będzie wydawał się aż tak wysoki, bo wszystkie wydatki zostaną rozłożone na niewielkie części i poniesione w trakcie realizacji projektu. W projekcie sekwencyjnym mamy do czynienia z jednym dużym wydatkiem na końcu.

Jak ilustruje to tabela 3.4, koncentracja na przygotowaniach może pomóc zredukować koszty niezależnie od tego, czy stosowane jest podejście iteracyjne, czy sekwencyjne. Metodyki iteracyjne są zwykle, z wielu przyczyn, podejściem lepszym, jednak ich stosowanie w połączeniu z ignorowaniem czynności przygotowawczych może doprowadzić do powstania znacznie większych kosztów niż przy wykorzystaniu metodyk sekwencyjnych, w których czynnościom tym poświęcono należną uwagę.

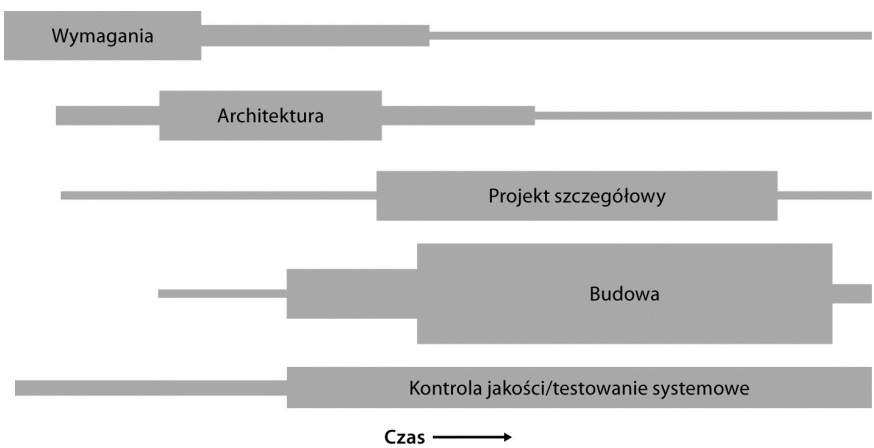
Tabela 3.4. Efekt koncentracji na przygotowaniach w projektach realizowanych sekwencyjnie i iteracyjnie

	Podejście 3.: Sekwencyjne z przygotowaniami		**Podejście 4.: Iteracyjne z przygotowaniami**	
Stopień zaawansowania projektu	Koszt pracy	Koszt przebudowy	Koszt pracy	Koszt przebudowy
20%	$ 100 000	$ 20 000	$ 100 000	$ 10 000
40%	$ 100 000	$ 20 000	$ 100 000	$ 10 000
60%	$ 100 000	$ 20 000	$ 100 000	$ 10 000
80%	$ 100 000	$ 20 000	$ 100 000	$ 10 000
100%	$ 100 000	$ 20 000	$ 100 000	$ 10 000
Przebudowa na końcu projektu	$ 0	$ 0	$ 0	$ 0
Suma	$ 500 000	$ 100 000	$ 500 000	$ 50 000
Suma końcowa		$ 600 000		$ 550 000

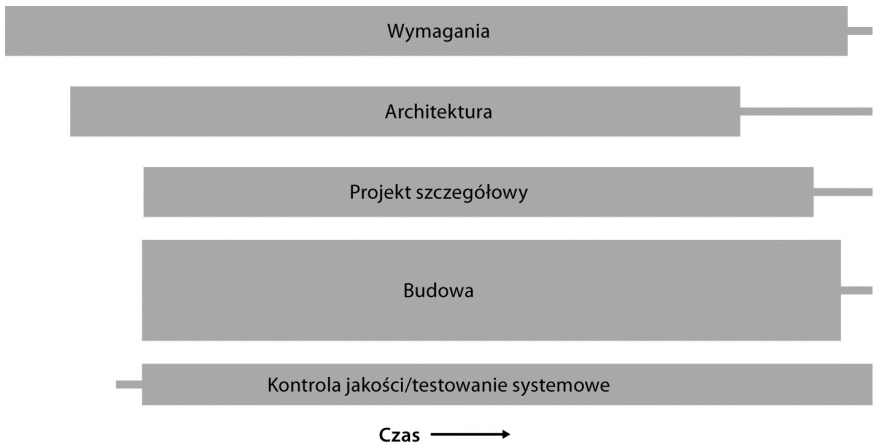
Jak sugeruje tabela 3.4, większość projektów trudno określić jako całkowicie sekwencyjne lub całkowicie iteracyjne. Określanie wymagań i projektu w 100 procentach na samym początku nie jest praktyczne, ale większość przedsięwzięć korzysta na zidentyfikowaniu przynajmniej najbardziej krytycznych elementów należących do zbioru wymagań i do architektury.

Patrz też: Bardziej szczegółowe omówienie zagadnienia adaptacji metodyki do rozmiarów projektu można znaleźć w rozdziale 27. „Jak rozmiar programu wpływa na jego budowę".

Jedną z dobrych praktyk jest planowanie określenia około 80 procent wymagań na początku, alokacja czasu na zdefiniowanie dalszych i stosowanie systematycznej kontroli zmian — ukierunkowanej na akceptację wyłącznie najbardziej wartościowych punktów — w trakcie realizacji projektu. Inną możliwością jest określenie na starcie tylko 20 procent najistotniejszych wymagań i przyjęcie zasady, że pozostałe elementy oprogramowania będą budowane metodą małych kroków, a dalsze wymagania i elementy projektu będą wypracowywane w trakcie prac. Rysunki 3.2 i 3.3 ilustrują oba te podejścia.



Rysunek 3.2. Wykonywane w procesie tworzenia oprogramowania czynności zazwyczaj w pewnym stopniu się pokrywają, nawet jeżeli przyjęto zasadę pracy sekwencyjnej



Rysunek 3.3. W innych przedsięwzięciach wszystkie rodzaje czynności powracają przez cały czas trwania projektu. Jedną z ważnych zasad programowania jest to, aby w pełni zdawać sobie sprawę z poziomu zaawansowania czynności przygotowawczych i odpowiednio dostosowywać swoje podejście

Wybór między podejściem iteracyjnym a sekwencyjnym

Poziom, do jakiego należy doprowadzić czynności przygotowawcze, różni się w zależności od typu projektu (patrz tabela 3.2), jego sformalizowania, otoczenia technicznego, umiejętności personelu i celów biznesowych. Do zastosowania podejścia bardziej sekwencyjnego (najpierw przygotowania, potem budowa) mogą skłaniać wymienione poniżej przesłanki.

- Wymagania są w miarę stabilne.
- Projekt jest dość oczywisty i zrozumiały.
- Zespół dobrze zna obszar zastosowań oprogramowania.
- Projekt wiąże się z niewielkim ryzykiem.
- Ważna jest przewidywalność w długim terminie.
- Szacowany koszt zmiany wymagań, projektu lub kodu jest wysoki.

A oto lista przesłanek dla zastosowania podejścia bardziej iteracyjnego (czynności przygotowawcze wykonywane w trakcie innych prac):

- Wymagania nie są do końca jasne lub można oczekiwać, że nie będą z pewnych przyczyn stabilne.
- Projekt jest złożony lub nowatorski.
- Zespół nie zna obszaru zastosowań oprogramowania.
- Projekt wiąże się z dużym ryzykiem.
- Przewidywalność w długim terminie nie jest istotna.
- Szacowany koszt zmiany wymagań, projektu lub kodu jest niski.

Należy przyznać, że metodyki iteracyjne znajdują zastosowanie dużo częściej niż sekwencyjne. Korzystając z nich, możesz dość swobodnie adaptować zakres

czynności wstępnych do konkretnego projektu, wybierać poziom ich formalizacji i dążyć do tego, aby były bardziej lub mniej kompletne. Bardziej szczegółowe omówienie różnych podejść do projektów dużych i małych (albo może bardziej i mniej sformalizowanych) można znaleźć w rozdziale 27.

Zagadnienie przygotowań do budowy oprogramowania sprowadza się w dużej mierze do tego, że początkiem pracy jest określenie, które z czynności wstępnych muszą zostać wykonane w konkretnym projekcie. W niektórych przedsięwzięciach przygotowaniom poświęca się zbyt mało czasu, co naraża proces budowy oprogramowania na nadmierną liczbę destabilizujących zmian i często skutkuje brakiem regularnych postępów. Innym razem czynności wstępnych jest zbyt wiele, a zespół sztywno trzyma się wymagań i planów, nawet gdy zdobyta w późniejszym okresie wiedza wskazuje na występujące w nich błędy. To także może prowadzić do zatrzymania postępu prac.

Jeśli już dobrze przestudiowałeś tabelę 3.2 i określiłeś, które czynności wstępne mają zastosowanie w Twoim projekcie, możesz przejść do lektury dalszej części rozdziału, w której piszemy o tym, jak ocenić, czy poszczególne aspekty przygotowań zostały uwzględnione we właściwy sposób.

3.3. Definicja problemu

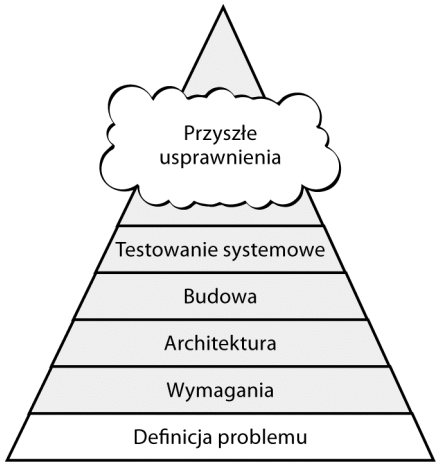
Jeżeli „szablonem" są rzeczywiste ograniczenia i warunki, sztuka polega na znalezieniu tego szablonu. Nie myśl nieszablonowo, znajdź szablon.
— Andy Hunt i Dave Thomas

Pierwszą czynnością wstępną, o którą należy zadbać przed rozpoczęciem programowania, jest jasne określenie problemu, który budowany system ma rozwiązywać. Często określa się to pojęciami „wizja produktu", „misja" czy „definicja produktu". Tutaj stosujemy termin „definicja problemu". Ponieważ jest to książka poświęcona budowie oprogramowania, nie dowiesz się z tego rozdziału, jak napisać taką definicję. Dowiesz się natomiast, jak stwierdzić, czy ona faktycznie istnieje i czy jest dobrą podstawą do rozpoczęcia pracy.

Definicja problemu określa jego naturę bez żadnych odniesień do potencjalnych rozwiązań. Jest to proste rozpoznanie, zajmujące nie więcej niż jedną lub dwie strony, które powinno mieć formę opisu pewnego zagadnienia. Stwierdzenie „Nie nadążamy z zamówieniami z Gigatronu" mówi o pewnym problemie i jest jego dobrą definicją. Stwierdzenie „Musimy zoptymalizować nasz zautomatyzowany system wprowadzania danych, aby nadążyć z zamówieniami z Gigatronu" nie definiuje dobrze problemu — nie brzmi ono jak jego opis, lecz jak jego rozwiązanie.

Jak ilustruje to rysunek 3.4, zdefiniowanie problemu następuje przed rozpoczęciem pracy nad uszczegółowieniem wymagań. Te wiążą się z głębszym rozpoznaniem zagadnienia.

Definicja problemu powinna być formułowana w języku użytkownika i przedstawiać jego perspektywę. Nie jest zazwyczaj pożądane używanie w niej jakichkolwiek terminów informatycznych czy, ogólnie, technicznych. Najlepszym rozwiązaniem nie zawsze jest program komputerowy. Przypuśćmy, że potrzebujesz raportu, który przedstawiałby rachunek zysków w skali roku. Dysponujesz już komputerowymi raportami z kwartalnym rachunkiem zysków.



Rysunek 3.4. Definicja problemu stoi u podstawy wszystkich innych elementów procesu

Patrząc na świat oczami programisty, dojdziesz szybko do wniosku, że uzupełnienie systemu, który już generuje raporty kwartalne, o raport roczny będzie stosunkowo proste. Zapłacisz więc programiście za napisanie i zdebugowanie czasochłonnego programu obliczającego zyski osiągnięte w ciągu roku. Jeżeli nie zamykasz się w świecie programowania, poniesiesz koszt wynagrodzenia sekretarki za minutę pracy, którą zajmie obliczenie wyników rocznych przy użyciu kieszonkowego kalkulatora.

Wyjątkiem od przedstawionej powyżej reguły jest sytuacja, gdy problem dotyczy samego komputera: kompilowanie trwa zbyt długo lub narzędzia programistyczne zawierają błędy. Wówczas opis problemu w kategoriach informatycznych jest właściwy.

Jak pokazuje rysunek 3.5, bez dobrej definicji możesz skierować swoje wysiłki w stronę niewłaściwego problemu.

Rysunek 3.5. Zanim strzelisz, upewnij się, w co celujesz

Karą za brak poprawnej definicji problemu jest ryzyko straty dużej ilości czasu poświęconego na rozpatrywanie złego zagadnienia. Dodatkowo jest to czas, w trakcie którego rozwiązywanie właściwego problemu nie zostaje nawet rozpoczęte.

3.4. Określenie wymagań

Wymagania opisują szczegółowo, co system informatyczny ma robić. Są zarazem pierwszym krokiem w stronę rozwiązania problemu. Związane nimi określenia to: „opracowywanie wymagań", „specyfikacja wymagań", „określanie wymagań", „analiza wymagań", „definicja wymagań", „specyfikacja funkcjonalna" lub „specyfikacja".

Czemu służy oficjalna specyfikacja wymagań?

Jasno określony zbiór wymagań jest ważny z kilku powodów.

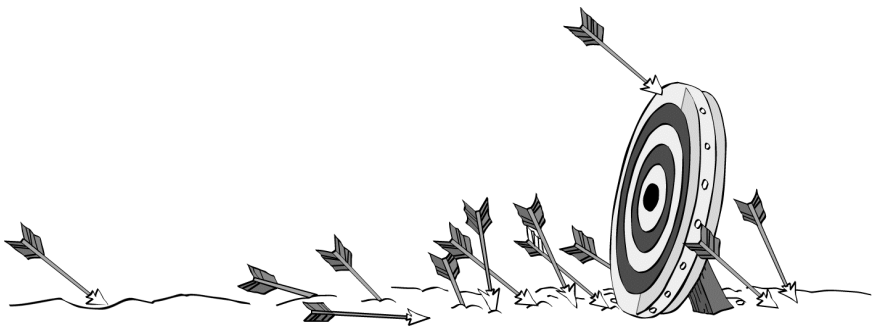
Zapisane wymagania pomagają zachować zasadę, że to użytkownik, a nie programista określa zespół funkcji systemu. Może on taką specyfikację przeglądać i zgadzać się z nią lub nie. Gdy wymagania nie zostają jawnie zdefiniowane, określanie funkcji systemu zostaje scedowane na programistę, który podejmuje konieczne decyzje w trakcie pisania kodu. Specyfikacja pozwala uniknąć zgadywania i domyślania się.

Jawnie określone wymagania chronią przed nieporozumieniami i starciami personalnymi. Zakres działania systemu zostaje wyznaczony przed rozpoczęciem programowania, więc kiedy między programistami pojawia się różnica zdań dotycząca tego, co program powinien robić, przegląd zapisanych wymagań pozwala w prosty sposób uzyskać porozumienie.

Dobrze przygotowane wymagania pomagają zminimalizować liczbę zmian wprowadzanych po rozpoczęciu programowania. Jeżeli w jego trakcie napotykasz błąd natury programistycznej, zmieniasz kilka wierszy kodu i praca toczy się dalej. Jeżeli jest to błąd związany ze specyfikacją wymagań, musisz zmienić projekt tak, aby odpowiadał zmianie w tej specyfikacji. Może to wymagać odrzucenia części wcześniejszego projektu, a ponieważ ważne jest, aby wykorzystywać to, co zostało już napisane, przygotowanie nowej wersji zajmuje więcej czasu niż opracowanie właściwej konstrukcji na samym początku. Dodatkowo konieczne jest napisanie nowego kodu i testów w miejsce tych, które w wyniku zmiany stają się niepotrzebne lub niepoprawne. Nawet kod, na który zmiana nie miała żadnego wpływu, musi zostać ponownie przetestowany dla uzyskania pewności, że modyfikacje nie spowodowały wprowadzenia nowych błędów.

Jak pokazuje tabela 3.1, dane zgromadzone w licznych organizacjach sygnalizują, że w dużych projektach naprawa błędu w specyfikacji wymagań wykrytego w fazie projektowania architektury jest mniej więcej trzykrotnie bardziej kosztowna niż naprawa podobnego błędu ujawnionego w fazie określania wymagań. Jeżeli wykrycie błędu następuje na etapie programowania, koszt wzrasta od 5 do 10 razy, jeżeli ma miejsce w trakcie testów systemowych — 10 razy, a po wdrożeniu — od 10 do 100 razy. W mniejszych projektach o niższych kosztach administracyjnych mnożnik po wdrożeniu lub publikacji wersji nie sięga 100, ale wciąż jest to zakres od 5 do 10 (Boehm i Turner 2004). W każdym przypadku jest to wydatek, którego można i należy unikać.

Właściwa specyfikacja wymagań to klucz do sukcesu projektu. Można wręcz posunąć się do stwierdzenia, że jest ona ważniejsza niż stosowanie efektywnych technik programowania (patrz rysunek 3.6). Na temat określania wymagań napisano już wiele dobrych książek, kolejnych kilka stron poświęcimy więc nie opisowi tego procesu, ale temu, jak określić, czy został on przeprowadzony poprawnie, i jak najlepiej wykorzystać dostępną specyfikację.



Rysunek 3.6. Bez dobrej specyfikacji wymagań możesz dysponować właściwą ogólną definicją problemu, ale ryzykujesz przeoczenie jego bardziej szczegółowych aspektów

Mit stabilnych wymagań

Wymagania są jak woda. Łatwiej na nich budować, gdy zostaną zamrożone.
— anonim

Stabilne wymagania to święty Graal programowania. Dzięki nim projekt podąża ścieżką od określenia architektury, poprzez projektowanie i programowanie, po testowanie w sposób uporządkowany, przewidywalny i spokojny. To istne niebo programisty! Wydatki są przewidywalne i nie trzeba się martwić o funkcje, których zaimplementowanie kosztuje 100 razy więcej, niż to konieczne, tylko dlatego, że przyszły użytkownikowi do głowy dopiero w chwili, kiedy zakończono debugowanie.

Można przyjąć założenie, że po zaakceptowaniu specyfikacji wymagań przez klienta dalsze zmiany nie będą potrzebne, jednak w typowych projektach klient nie jest w stanie wiarygodnie opisać, czego potrzebuje, przed rozpoczęciem pracy z kodem. Problemem nie jest bynajmniej jego ociężałość umysłowa. Programista pracujący nad projektem wraz z upływem czasu zdobywa coraz wyższy poziom jego zrozumienia. Podobnie jest z klientem — proces tworzenia oprogramowania pomaga mu lepiej poznać własne potrzeby. To właśnie jest głównym źródłem zmian w specyfikacji (Curtis, Krasner i Iscoe 1988; Jones 1998; Wiegers 2003). Plan przewidujący sztywne trzymanie się wymagań to w istocie plan, w którym zakładamy zablokowanie komunikacji z odbiorcą.

Jaką ilość zmian należy uznać za typową? Badania prowadzone w IBM i innych firmach doprowadziły do ustalenia, że w przeciętnym projekcie zmianie ulega około 25 procent wymagań (Boehm 1981, Jones 1994, Jones 2000). Odpowiada to 70 – 85 procentom działań związanych z przebudową systemu w typowym przedsięwzięciu (Leffingwell 1997, Wiegers 2003).

Być może uważasz, że pontiac aztek to najlepszy samochód w historii świata, należysz do Stowarzyszenia Płaskiej Ziemi i co cztery lata pielgrzymujesz do miejsca lądowania kosmitów w Roswell w USA. Jeżeli tak, możesz równie

dobrze wierzyć, że wymagania pozostaną niezmienione w trakcie realizowanych projektów. Jeżeli jednak przestałeś wierzyć w Świętego Mikołaja, a przynajmniej nie przyznajesz się do takich przekonań publicznie, to zapewne chętnie zwrócisz uwagę na kilka wskazówek, których stosowanie pozwoli ograniczyć wpływ zmian w specyfikacji.

Zmiany wymagań w trakcie budowy oprogramowania

Oto kilka rzeczy, które możesz zrobić, aby uzyskać możliwie najwięcej korzyści ze zmian wymagań w trakcie budowy oprogramowania:

Używaj zamieszczonych na końcu tego podrozdziału list kontrolnych do oceny jakości specyfikacji. Jeżeli wymagania nie są wystarczająco dobrze określone, przerwij pracę, cofnij się o krok i doprowadź je do porządku. Oczywiście przerywanie pisania kodu na tym etapie może być odczuwane jako wprowadzanie niepotrzebnych opóźnień, czy jednak, gdy jedziesz z Chicago do Los Angeles i widzisz tablice sygnalizujące, że zbliżasz się do Nowego Jorku, spojrzenie na mapę jest stratą czasu? Nie. Jeżeli podążasz w złym kierunku, postój w celu skorygowania planu podróży jest jedynym rozsądnym działaniem.

Zadbaj o to, aby wszyscy mieli świadomość kosztów zmian w specyfikacji wymagań. Gdy pojawia się idea nowej funkcji, klienci łatwo wpadają w entuzjazm. Wywołuje to u nich rozrzedzenie krwi, która w większych ilościach napływa do rdzenia przedłużonego, czego skutkiem są zawroty głowy i zapominanie o wcześniejszych długich spotkaniach, na których omawiano wymagania, o ceremonii podpisywania specyfikacji, a czasem w ogóle o istnieniu tego ważnego dokumentu. Najprostszym sposobem radzenia sobie z takim odurzeniem jest pełne zrozumienia stwierdzenie: „O rany, świetny pomysł. Ponieważ nie uwzględniliśmy go w specyfikacji wymagań, opracuję niezbędną korektę harmonogramu realizacji projektu i oszacuję koszty, aby mógł pan podjąć decyzję, czy wprowadzać to od razu, czy w późniejszym terminie". Słowa „korekta harmonogramu" i „koszty" trzeźwią lepiej niż kawa i zimny prysznic. „Koniecznie" szybko zmienia się wtedy w „dobrze by było".

Jeżeli w Twojej organizacji nie przypisuje się wielkiego znaczenia specyfikacji wymagań, zwróć uwagę, że zmiany uwzględniane przy jej opracowywaniu są znacznie tańsze niż wprowadzane później. Pomocne będzie zamieszczone w tym rozdziale „Przekonujące uzasadnienie konieczności wykonania przed programowaniem czynności wstępnych".

Patrz też: Więcej informacji na temat pracy ze zmianami w projekcie i kodzie można znaleźć w podrozdziale 28.2 „Zarządzanie konfiguracją".

Wprowadź procedurę kontroli zmian. Jeżeli entuzjazm klienta utrzymuje się, rozważ ustanowienie formalnej rady nadzorującej zmiany, która będzie odpowiadać za przeglądanie i przyjmowanie proponowanych modyfikacji. Nie ma nic złego w tym, że klienci zmieniają zdanie i dochodzą do wniosku, że potrzebują czegoś więcej. Problemem może być sytuacja, gdy robią to na tyle często, że trudno za nimi nadążyć. Standardowa procedura kontroli zmian pozwala zadowolić obie strony. Programista jest szczęśliwy, bo modyfikacje nie pojawiają się często i nagle, ale w ściśle określonym czasie, klient natomiast jest zadowolony, bo wie, że wykonawca jest gotowy na przyjęcie jego sugestii i wniosków.

Patrz też: Więcej informacji na temat iteracyjnych metodyk programowania można znaleźć w punkcie „Iteruj" w podrozdziale 5.4 i w podrozdziale 29.3 „Przyrostowe strategie integracji".

Korzystaj z metod organizacji pracy, które ułatwiają wprowadzanie zmian. Niektóre metodyki wytwarzania oprogramowania zwiększają możliwości reagowania na zmiany w wymaganiach. Prototypowanie ewolucyjne ułatwia opracowywanie wymagań przed rozpoczęciem programowania. Ewolucyjny model programowania to podejście, w którym system jest dostarczany etapami. Można zbudować małą część, uzyskać od użytkowników informacje zwrotne, skorygować projekt, wprowadzić kilka zmian i przejść do kolejnej części. Kluczem są krótkie cykle, które umożliwiają szybkie reagowanie na wnioski użytkowników.

Więcej informacji: Opis metodyk wytwarzania oprogramowania, które umożliwiają elastyczną pracę z wymaganiami, można znaleźć w książce *Rapid Development* (McConnell 1996).

Przerwij projekt. Jeżeli specyfikacja wymagań jest wyjątkowo niechlujnie opracowana lub mało stabilna, a żadna z powyższych wskazówek nie okazuje się pomocna, przerwij pracę. Nawet jeżeli nie możesz tego naprawdę zrobić, spróbuj wyobrazić sobie, jak mogłoby to wyglądać. Pomyśl o tym, o ile gorsza musiałaby być sytuacja, aby faktycznie tak się stało. Jeżeli taka sytuacja jest wyobrażalna, zastanów się, jak bardzo różni się ona od tej, w której jesteś.

Patrz też: O różnicach między projektami formalnymi i nieformalnymi (często wynikającymi z różnic w rozmiarze przedsięwzięć) można przeczytać w rozdziale 27. „Jak rozmiar programu wpływa na jego budowę".

Pamiętaj o celu biznesowym projektu. Wiele kwestii związanych z wymaganiami przestaje mieć znaczenie, gdy przypomnimy sobie cel biznesowy projektu. Wymagania, które wydawały się świetnymi pomysłami, gdy rozważano je jako nowe cechy i funkcje, tracą często swój urok, gdy zastanowić się nad ich „wartością marginalną". Programiści, którzy nie zapominają u uwzględnianiu szeroko rozumianych konsekwencji ekonomicznych swoich decyzji, są na wagę złota.

cc2e.com/0323

Lista kontrolna: Wymagania

Lista kontrolna wymagań to szereg pytań, które warto zadać sobie w trakcie przeglądu specyfikacji. Nie piszę w tej książce o tym, jak ją dobrze przygotować, i informacji na ten temat nie zawiera również niniejsza lista. Ma ona ułatwić na etapie programowania ocenę solidności przygotowanego podłoża — określenie, jakich wstrząsów w skali Richtera można się spodziewać.

Nie wszystkie pytania listy kontrolnej będą miały zastosowanie w każdym projekcie. Jeżeli projekt jest nieformalny, część z nich jest zupełnie nieadekwatna. Inne pytania są natomiast ważne, ale nie musisz udzielać na nie formalnej odpowiedzi. Jeżeli jednak pracujesz nad projektem, który jest duży i sformalizowany, nie polecałbym opuszczania któregokolwiek z punktów.

Wymagania funkcjonalne

- ❑ Czy opisano wszystkie wejścia systemu — źródła danych, dokładność, zakres wartości i częstotliwość?
- ❑ Czy opisano wszystkie wyjścia systemu — miejsca docelowe danych, dokładność, zakres wartości, częstotliwość i format?
- ❑ Czy opisano wszystkie formaty wyjściowe stron WWW, raportów itp.?

- ❑ Czy opisano wszystkie zewnętrzne interfejsy sprzętowe i programowe?
- ❑ Czy opisano wszystkie zewnętrzne interfejsy komunikacyjne, w tym sposób nawiązywania komunikacji, metody kontroli błędów i protokoły komunikacyjne?
- ❑ Czy opisano wszystkie funkcje, których oczekuje użytkownik?
- ❑ Czy opisano dane wykorzystywane przez każdą funkcję i jej dane wynikowe?

Wymagania jakościowe

- ❑ Czy zapisano dla wszystkich operacji wymagany przez użytkownika czas reakcji?
- ❑ Czy określono parametry czasowe takie jak czas przetwarzania, szybkość transmisji danych i przepustowość systemu?
- ❑ Czy określono poziom bezpieczeństwa?
- ❑ Czy określono poziom niezawodności, w tym konsekwencje awarii oprogramowania, kluczowe informacje, które nie mogą zostać utracone, oraz strategię wykrywania i usuwania błędów?
- ❑ Czy określono minimalną ilość pamięci komputera i miejsca na dysku?
- ❑ Czy określono poziom elastyczności systemu — zdolność adaptacji do zmian w określonych funkcjach, w środowisku operacyjnym i w interfejsach łączących go z innym oprogramowaniem?
- ❑ Czy specyfikacja zawiera definicję udanego projektu? A nieudanego?

Jakość wymagań

- ❑ Czy wymagania zapisano w języku użytkowników? Czy użytkownicy też tak uważają?
- ❑ Czy wymagania nie są wzajemnie sprzeczne? Czy występuje możliwość powstania konfliktów?
- ❑ Czy określono akceptowalne kompromisy dla konkurujących ze sobą atrybutów, na przykład wydajności i poprawności?
- ❑ Czy w specyfikacji unika się określania cech projektu?
- ❑ Czy poziom szczegółowości wymagań jest w miarę jednolity? Czy są wymagania, które trzeba uszczegółowić, albo takie, które są określone zbyt dokładnie?
- ❑ Czy wymagania są na tyle zrozumiałe, że mogłyby zostać przekazane innemu zespołowi? Czy programiści też tak uważają?
- ❑ Czy poszczególne elementy mają odniesienie do problemu i jego rozwiązania? Czy można powiązać każdy z nich z jego źródłem w środowisku problemu?

- ❑ Czy poszczególne wymagania są testowalne? Czy byłoby możliwe określenie tego, czy zostały spełnione, przez niezależny zespół?
- ❑ Czy określono możliwe zmiany w wymaganiach i prawdopodobieństwo ich wystąpienia?

Kompletność wymagań

- ❑ Czy dla informacji niedostępnych przed rozpoczęciem budowy oprogramowania określono obszary niekompletności wymagań?
- ❑ Czy wymagania są kompletne w tym znaczeniu, że spełnienie każdego z nich będzie jednoznaczne z akceptacją produktu?
- ❑ Czy wymagania wydają Ci się rozsądne? Czy wyeliminowałeś te z nich, których implementacja jest niemożliwa, a których włączenie do specyfikacji było tylko ustępstwem wymuszonym przez przełożonego lub klienta?

3.5. Architektura

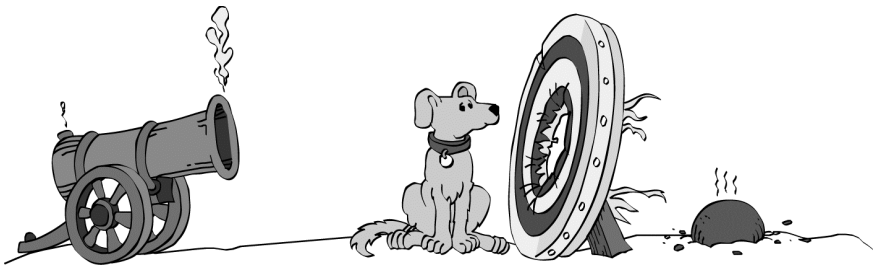
Patrz też: Więcej informacji o projektowaniu na różnych poziomach można znaleźć w rozdziałach od 5. do 9.

Architektura oprogramowania to wysokopoziomowa część jego projektu, szkielet, na którym opierają się poszczególne jego elementy (Buschman et al. 1996; Fowler 2005; Bass Clements, Kazman 2003; Clements et al. 2003). Architekturę określa się również terminami „architektura systemowa", „projekt wysokiego poziomu" i „projekt najwyższego poziomu". Opisuje się ją najczęściej w pojedynczym dokumencie określanym nazwą „specyfikacja architektury". Niektórzy dokonują rozróżnienia między architekturą, która odnosi się do ograniczeń konstrukcyjnych obowiązujących w całym systemie, a projektem wysokiego poziomu, odnoszącym się do ograniczeń konstrukcyjnych w podsystemie lub grupie klas, ale niekoniecznie w całości systemu.

Ponieważ niniejsza książka traktuje o budowie oprogramowania, nie piszę w tym podrozdziale o tworzeniu architektury, koncentruję się natomiast na metodach określania jej jakości. Ponieważ jednak architektura jest o jeden krok bliżej programowania niż omówione wcześniej wymagania, związane z nią zagadnienia przedstawię nieco bardziej szczegółowo.

Czemu służy przygotowywanie architektury przed rozpoczęciem budowy oprogramowania? Jej jakość decyduje o spójności koncepcji systemu. Od niej zależy z kolei to, jaki ten system jest. Przemyślana architektura zapewnia strukturę niezbędną do utrzymania jedności koncepcyjnej od najwyższych poziomów systemu po najbardziej szczegółowe. Wskazuje ona drogę programiście — na poziomie szczegółowości odpowiadającym jego umiejętnościom i wykonywanemu zadaniu. Architektura dzieli też pracę, dzięki czemu wielu programistów lub wiele zespołów może zajmować się nią niezależnie od siebie.

Dobra architektura ułatwia budowę oprogramowania. Zła sprawia, że zbudowanie rozwiązania jest zadaniem ledwie wykonalnym. Rysunek 3.7 ilustruje jeszcze jeden problem związany ze złą architekturą.



Rysunek 3.7. Brak dobrej architektury może sprawić, że mimo właściwego zdefiniowania problemu nie osiągniesz dobrego rozwiązania. Ukończenie projektu może być wręcz niemożliwe

Wprowadzanie zmian w architekturze po rozpoczęciu programowania jest kosztowne. Czas potrzebny do usunięcia wykrytego w niej błędu należy do tego samego rzędu wielkości co czas usuwania nieścisłości w specyfikacji wymagań, jest więc znacząco dłuższy niż w przypadku naprawy błędów programistycznych (Basili i Perricone 1984, Willis 1998). Podobnie jak zmiany w wymaganiach, mało znaczące na pierwszy rzut oka modyfikacje architektury mogą mieć daleko idące konsekwencje. Niezależnie od tego, czy zmiany te wynikają z konieczności usunięcia błędów, czy z potrzeby wprowadzenia udoskonaleń, im wcześniej zostaną dokonane, tym lepiej.

Typowe składniki architektury

Patrz też: Zagadnieniami związanymi z niskopoziomowym projektem oprogramowania zajmiemy się w rozdziałach od 5. do 9.

W dobrych architekturach systemowych występuje wiele typowych składników. Gdy budujesz cały system samodzielnie, praca nad architekturą pokrywa się z pracą nad projektem szczegółowym. W takim przypadku należy przynajmniej dobrze przemyśleć poszczególne jej składniki. Jeżeli pracujesz nad systemem, którego architekturę stworzył ktoś inny, powinieneś być w stanie określić jej podstawowe komponenty bez ich wyszukiwania, badania czy dogłębnego analizowania. Niezależnie od sytuacji kluczowe elementy dobrej architektury pozostają zbliżone.

Organizacja programu

Jeżeli nie jesteś w stanie wyjaśnić czegoś sześciolatkowi, to tak naprawdę sam tego nie rozumiesz.
— Albert Einstein

Architektura systemowa wymaga przede wszystkim wprowadzenia, w którym system zostaje opisany w kategoriach najbardziej ogólnych. Bez niego uzyskanie spójnego obrazu na bazie tysiąca szczegółów czy choćby kilkunastu klas jest znacznie utrudnione. Gdyby system był małą, 12-elementową układanką typu puzzle, półtoraroczne dziecko układałoby go pomiędzy wypluwaniem kolejnych łyżeczek zupy na śliniak. Puzzle z 12 podsystemów trudniej jest ułożyć, a dopóki nie jesteś w stanie tego zrobić, nie masz możliwości pełnego zrozumienia, jakie jest znaczenie budowanej klasy w całości.

W architekturze powinieneś znaleźć dowody na to, że alternatywy dla wybranej organizacji zostały należycie rozważone, oraz przyczyny dokonanego wyboru. Nie jest przyjemnością praca nad klasą programu, gdy wszystko wskazuje na to, że jej rola w systemie nie została jasno określona. Poprzez opis alternatywnych organizacji architektura wyjaśnia powody wybrania danego rozwiązania i pokazuje, że każda klasa jest elementem dokładnie przemyślanym.

Jedno z badań nad procesem projektowania wykazało, że uzasadnienie wybranej konstrukcji jest co najmniej równie ważne w procesie rozwijania systemu jak sam projekt (Rombach 1990).

Patrz też: O blokach różnych rozmiarów w projektowaniu można przeczytać w podrozdziale 5.2 „Podstawowe pojęcia projektowania".

Architektura powinna definiować główne bloki konstrukcyjne programu. W zależności od jego rodzaju mogą być nimi pojedyncze klasy lub podsystemy złożone z dużej liczby klas. Blok konstrukcyjny to klasa albo zbiór klas lub procedur, które współpracują w realizacji funkcji wysokiego poziomu, takich jak interakcje z użytkownikiem, wyświetlanie stron WWW, interpretowanie poleceń, hermetyzacja reguł biznesowych czy dostęp do danych. Każdej pozycji na liście wymagań powinien odpowiadać co najmniej jeden taki blok. Jeżeli za realizację funkcji odpowiadają dwa lub więcej bloków, niezbędne jest zapewnienie ich współpracy i unikanie konfliktów.

Patrz też: Minimalizowanie ilości wiedzy, którą jeden blok musi posiadać o drugim, to podstawowa zasada ukrywania informacji. Więcej na ten temat w punkcie „Ukrywaj tajemnice (ukrywanie informacji)" w podrozdziale 5.3.

To, za co odpowiada dany blok konstrukcyjny, powinno być precyzyjnie zdefiniowane. Każdy blok powinien mieć jeden obszar odpowiedzialności i jak najmniejszą wiedzę o obszarach odpowiedzialności innych bloków. Takie ograniczenie pozwala zamykać informacje o projekcie w obrębie pojedynczych bloków.

Precyzyjnie powinna zostać zdefiniowana także komunikacja między blokami. Architektura powinna opisywać, które z innych bloków konstrukcyjnych dany blok może wykorzystywać bezpośrednio, do których ma dostęp pośredni i z których nie może korzystać w ogóle.

Podstawowe klasy programu

Patrz też: Projektowanie klas jest omawiane w rozdziale 6. „Klasy z klasą".

Zadaniem architektury jest określać podstawowe klasy projektu. Powinna ona identyfikować ich zakres odpowiedzialności oraz zasady interakcji z innymi klasami, a także zawierać opisy hierarchii klas, przejść stanowych i czasu trwania obiektów. Jeżeli system jest rozbudowany, architektura powinna opisywać też organizację klas w podsystemy.

Architektura wymienia inne rozważone projekty klas i podaje powody zastosowania wybranej organizacji. Nie musi wymieniać każdej klasy systemu. Pomocna jest zasada „80/20", polegająca na przygotowaniu specyfikacji 20 procent klas, które odpowiadają za 80 procent funkcji systemu (Jacobson, Booch i Rumbaugh 1999; Kruchten 2000).

Organizacja danych

Patrz też: Zagadnienia związane z pracą ze zmiennymi są omawiane w rozdziałach od 10. do 13.

Architektura opisuje organizację podstawowych plików i tabel. Również w tym przypadku powinna wskazywać rozważone rozwiązania alternatywne i uzasadniać podjęte decyzje. Jeżeli aplikacja korzysta z listy identyfikatorów klientów i architekci zdecydowali, że będzie to lista o dostępie sekwencyjnym, specyfikacja architektury powinna wyjaśniać, dlaczego taki rodzaj listy jest lepszy niż lista o dostępie swobodnym, stos lub tablica asocjacyjna. Programiście daje to istotny wgląd w sposób myślenia architektów. Jest on bezcenny przy wprowadzaniu zmian i rozwijaniu systemu. Gdy go brak, można poczuć się jak przy oglądaniu zagranicznego filmu bez napisów i lektora.

W normalnych sytuacjach bezpośredni dostęp do danych powinien mieć tylko jeden podsystem lub jedna klasa. Wyjątkiem są klasy lub procedury dostępowe, które umożliwiają korzystanie z danych w kontrolowany i abstrakcyjny sposób. Zagadnienie to zostało omówione szerzej w punkcie „Ukrywaj tajemnice (ukrywanie informacji)" podrozdziału 5.3.

Architektura ma także za zadanie określać wysokopoziomową organizację i zawartość wykorzystywanych baz danych. Powinna też wyjaśniać, dlaczego pojedyncza baza jest lepsza niż wiele baz (lub odwrotnie), dlaczego baza danych jest właściwsza niż pliki tekstowe, oraz identyfikować możliwe interakcje z innymi programami korzystającymi z tych samych danych, objaśniać, jakie widoki są dostępne, itd.

Reguły biznesowe

Jeżeli działanie architektury opiera się na określonych regułach biznesowych, powinny one zostać zidentyfikowane i opisane pod kątem wpływu na projekt. Załóżmy, że w systemie ma być przestrzegana reguła biznesowa głosząca, że dane klienta nie mogą pozostawać nieaktualne dłużej niż 30 sekund. W takim przypadku należy opisać wpływ tej zasady na zastosowane metody aktualizowania i synchronizowania danych klientów.

Projekt interfejsu użytkownika

Interfejs użytkownika zostaje często wyspecyfikowany już na etapie opracowywania wymagań. Jeżeli tak nie jest, powinien zostać opisany wraz z architekturą, która powinna określać podstawowe elementy formatu stron WWW, graficznego interfejsu użytkownika, interfejsu wiersza poleceń itp. Dopracowana w tym zakresie architektura decyduje o różnicy między programem, który jest powszechnie lubiany, a tym, którego nikt nie używa.

Architektura powinna mieć konstrukcję modułową, aby wymiana interfejsu użytkownika była możliwa bez zakłócania funkcjonowania reguł biznesowych i mechanizmów wyprowadzających dane. Zamiana klas interfejsu interakcyjnego na klasy wiersza poleceń powinna być stosunkowo prostą operacją. Możliwość ta często się przydaje, zwłaszcza że interfejsy wiersza poleceń są wygodne przy testowaniu jednostkowym oraz testowaniu podsystemów.

cc2e.com/0393

Projektowanie interfejsu użytkownika to temat na osobną książkę i nie będziemy go tutaj omawiać.

Zarządzanie zasobami

Architektura powinna opisywać plan zarządzania zasobami, które podlegają istotnym ograniczeniom, takimi jak połączenia z bazą danych, wątki i uchwyty. Zarządzanie pamięcią to kolejny ważny element jej opisu, zwłaszcza w przypadku projektów dotyczących aplikacji osadzonych i sterowników. Architektura powinna szacować typowy poziom wykorzystania zasobów i ich wykorzystanie w sytuacjach ekstremalnych. W najprostszym przypadku jej zadaniem jest pokazać, że wymagania dotyczące zasobów mieszczą się z dużym zapasem w możliwościach środowiska implementacji. W sytuacjach bardziej złożonych

aplikacja może być zmuszona do aktywnego zarządzania zasobami. Gdy tak jest, menedżerowi zasobów należy poświęcić równie dużo uwagi co innym częściom systemu.

cc2e.com/0330

Zabezpieczenia

Więcej informacji: Doskonałe omówienie tematu zabezpieczeń można znaleźć w książce *Writing Secure Code, 2nd Ed.* (Howard i LeBlanc 2003), a także w numerze czasopisma „IEEE Software" ze stycznia 2002 roku.

Architektura powinna określać wybrane podejście do zabezpieczeń na poziomie projektu i na poziomie kodu. Jeżeli model zagrożeń nie został wcześniej przygotowany, powinien powstać w trakcie pracy nad architekturą. Należy opracować wskazówki dotyczące pisania kodu, które uwzględniałyby wpływ przyjmowanych rozwiązań na bezpieczeństwo. Obejmuje to między innymi obsługę buforów, reguły obsługi danych, które mogą stanowić zagrożenie (dane wprowadzane przez użytkowników, cookies, dane konfiguracyjne i inne dane z interfejsów zewnętrznych), szyfrowanie, poziom szczegółowości komunikatów błędów, ochronę ładowanych do pamięci danych poufnych i inne podobne kwestie.

Wydajność

Więcej informacji: Dobre omówienie tematu projektowania systemów pod kątem wydajności można znaleźć w książce Connie Smith *Performance Engineering of Software Systems* (1990).

Jeżeli mogą wystąpić problemy z wydajnością systemu, w wymaganiach powinny zostać określone minimalne parametry. Lista celów w zakresie wydajności może obejmować poziom wykorzystania zasobów — w takim przypadku należy także wskazać priorytet każdego z nich z uwzględnieniem szybkości pracy, poziomu wykorzystania pamięci i kosztów.

Architektura powinna określać wartości szacunkowe i wyjaśniać, dlaczego architekci uważają je za osiągalne. Jeżeli w pewnych obszarach pojawia się ryzyko, że cele mogą być poza zasięgiem, należy o tym napisać. W przypadku gdy pewne obszary do osiągnięcia celów wymagają użycia specyficznych algorytmów lub typów danych, również należy o tym wspomnieć. Architektura może także przypisywać każdej klasie lub obiektowi określoną ilość czasu i zasobów.

Skalowalność

Skalowalność to szeroko rozumiane możliwości rozrastania się systemu odpowiednio do przyszłych potrzeb. Architektura powinna opisywać, jak rozwiązywane są problemy wzrostu liczby użytkowników, serwerów, węzłów sieciowych, rekordów w bazie danych, rozmiarów tych rekordów, wolumenu transakcji i inne tego rodzaju. Jeżeli takie rozrastanie się systemu nie będzie następowało i problem skalowalności go nie dotyczy, powinno to być jasno zadeklarowane.

Współdziałanie

Jeżeli system ma wymieniać dane lub zasoby z innym oprogramowaniem lub wyposażeniem, architektura powinna opisywać, jak zadania te będą realizowane.

Internacjonalizacja i lokalizacja

Internacjonalizacja (żargonowo skracana do „I18n" od ang. *internationalization*, „I"+18 znaków+„n") to dostosowywanie programu do obsługi wielu *locale* (różnych ustawień regionalnych i językowych). Lokalizacja („L10n", od ang. *localization*) to tłumaczenie programu tak, aby w pełni pracował w określonym języku.

Zagadnienia internacjonalizacji są istotne w architekturze systemów interakcyjnych. Większość systemów tego rodzaju wyświetla dziesiątki lub setki komunikatów, wezwań do wprowadzenia danych, informacji pomocniczych, instrukcji, powiadomień o błędach itp. Zasoby, których wymagają takie ciągi, powinny zostać oszacowane. Jeżeli program ma być wykorzystywany komercyjnie, architektura powinna pokazywać, że zostały wzięte pod uwagę typowe problemy z ciągami znakowymi i zestawami znaków, w tym stosowany zestaw (ASCII, DBCS, EBCDIC, MBCS, Unicode, ISO 8859), rodzaj ciągów znakowych (ciągi języka C, ciągi języka Visual Basic) oraz możliwość modyfikowania ciągów bez zmiany kodu i tłumaczenia ich na języki obce przy zachowaniu minimalnego wpływu na kod i interfejs użytkownika. Architektura może decydować o użyciu ciągów wpisywanych bezpośrednio w kodzie, ciągów przechowywanych w klasie i wywoływanych za pośrednictwem interfejsu lub ciągów przechowywanych w pliku zasobów. Powinna też wyjaśniać przyczyny takiego, a nie innego wyboru.

Wejście-wyjście

Wejście-wyjście (we-wy) to kolejny obszar, któremu trzeba poświęcić uwagę przy projektowaniu architektury. Powinna ona określać, czy stosowany jest schemat odczytu „z wyprzedzeniem", „z opóźnieniem", czy „na czas", a także opisywać, na jakim poziomie wykrywane są błędy wejścia-wyjścia: pola, rekordu, strumienia czy pliku.

Przetwarzanie błędów

Przetwarzanie błędów okazuje się jednym z najtrudniejszych problemów współczesnej informatyki i nie można sobie pozwolić na jego lekkie potraktowanie. W pewnych badaniach stwierdzono, że przeciętnie 90 procent kodu programu zapewnia obsługę sytuacji nietypowych, przetwarzanie błędów lub korygowanie stanu systemu (porządkowanie). Wynika z tego, że tylko 10 procent to kod scenariuszy nominalnych (Shaw w Bentley 1982). Przy tak wielkim udziale kodu obsługi błędów konieczność opisania sposobu podejścia do problemu w architekturze jest naturalna.

Zagadnienie obsługi błędów sprowadza się często do poziomu konwencji programowania (o ile w ogóle temat jest poruszany), ponieważ jednak jest to problem dotyczący całego systemu, może być efektywniej rozwiązywany na poziomie architektury. Oto lista najważniejszych zagadnień do rozważenia:

- Czy przetwarzanie błędów ma charakter korekcyjny, czy jedynie detekcyjny? Przetwarzanie korekcyjne oznacza, że program podejmuje próby naprawienia defektów. Przy przetwarzaniu detekcyjnym może on kon-

tynuować pracę, tak jakby błąd nie wystąpił, lub przerwać działanie. W obu przypadkach użytkownik musi zostać powiadomiony o wystąpieniu i charakterze błędu.

- Czy wykrywanie błędów ma charakter aktywny czy pasywny? System może antycypować ich wystąpienie — na przykład sprawdzać poprawność wprowadzanych danych — lub pasywnie reagować na nie, gdy jest to nie do uniknięcia — na przykład gdy określone połączenie danych wejściowych prowadzi do przekroczenia zakresu liczby. Program może korygować stan po awarii lub po prostu przywracać stan zerowy. Podjęta w tych kwestiach decyzja wpływa na budowę interfejsu użytkownika.
- Jak program propaguje błędy? Po wykryciu sytuacji awaryjnej może on natychmiast odrzucić powodujące ją dane, potraktować zakłócenie jako błąd i wejść w stan jego przetwarzania albo też poczekać na zakończenie operacji i powiadomić użytkownika o zaistniałym zdarzeniu.
- Jakie są konwencje obsługi błędów? Jeżeli architektura nie określa jednolitej, spójnej strategii, interfejs użytkownika łatwo może się stać chaotycznym kolażem kilku interfejsów stosowanych w różnych częściach programu. Aby uniknąć takich wątpliwej jakości kompozycji, architektura powinna wyznaczać konwencje komunikatów błędów.
- Jak będzie wyglądała obsługa wyjątków? Architektura powinna opisywać, kiedy kod może je zgłaszać, gdzie będą one przechwytywane, jak będą rejestrowane, dokumentowane itp.
- Na jakim poziomie wewnątrz programu znajduje się obsługa błędów? Błędy można obsługiwać w miejscu wykrycia, przekazywać je do specjalnej klasy obsługi błędów lub przekazywać w łańcuchu wywołań.

Patrz też: Usystematyzowana metoda obsługi niewłaściwych parametrów to kolejny aspekt strategii przetwarzania błędów, który trzeba wziąć pod uwagę przy projektowaniu architektury. Przykłady można znaleźć w rozdziale 8. „Programowanie defensywne".

- Jaki jest poziom odpowiedzialności każdej klasy za sprawdzanie poprawności danych wejściowych? Czy każda z klas odpowiada za ich weryfikowanie, czy też jest w systemie grupa klas, która się tym zajmuje? Czy klasy na pewnych poziomach mogą pracować w oparciu o założenie, że otrzymywane przez nie dane są „czyste"?
- Czy zamierzasz używać standardowego w stosowanym środowisku mechanizmu obsługi wyjątków, czy zbudować własny? To, że środowisko dysponuje pewnym systemem obsługi błędów, nie oznacza jeszcze, że będzie on optymalną częścią rozwiązania problemu określonego w wymaganiach.

Zabezpieczenia przed awariami

Więcej informacji: Dobre wprowadzenie do tematu zabezpieczeń przed awariami można znaleźć w magazynie „IEEE Software" z lipca 2001 roku. Jest w nim także wskazanych wiele tytułów książek i artykułów poruszających pokrewne zagadnienia.

Architektura powinna także wskazywać planowane zabezpieczenia przed awariami, czyli zbiór mechanizmów, które zwiększają niezawodność systemu poprzez wykrywanie błędów i usuwanie ich lub też ograniczanie zasięgu ich skutków.

Oto przykładowe możliwości zabezpieczenia przed awariami obliczeń wartości pierwiastka kwadratowego liczby:

- W przypadku wykrycia awarii system może wycofać się i ponowić próbę. Jeżeli odpowiedź będzie błędna, nastąpi powrót do punktu, w którym można liczyć na pełną poprawność, i kontynuacja pracy.
- System może dysponować pomocniczym kodem wykorzystywanym w przypadku wykrycia błędu w kodzie głównym. Jeżeli pierwsza odpowiedź będzie zła, przełączy się na korzystanie z alternatywnej procedury pierwiastkowania.
- System może wykorzystywać algorytm głosowania. Pierwiastek kwadratowy mogą obliczać trzy klasy stosujące różne metody, po czym może następować porównanie wyników. W zależności od potrzeb systemu może być stosowana wartość średnia, mediana lub wartość modalna (dominanta).
- System może zamieniać wartość błędną na wartością zastępczą, która spowoduje stosunkowo niewielkie szkody w jego działaniu.

Inne rozwiązanie zabezpieczeń przed awariami to przejście systemu w stan, w którym część funkcji przestaje działać lub działają one tylko w ograniczonym zakresie. System może automatycznie wyłączać się lub restartować. Przedstawiony powyżej przykład jest dużym, ale koniecznym uproszczeniem. Zabezpieczenia przed awariami to fascynujący, szeroki i trudny temat — niestety, wykracza on poza ramy tej książki.

Wykonalność

Projektanci mogą wątpić w możliwość osiągnięcia przez system celów wydajnościowych, dostosowania architektury do ograniczeń związanych z zasobami albo w dostępność niezbędnych mechanizmów w różnych środowiskach implementacji. Architektura powinna pokazywać, że system jest technicznie wykonalny. Jeżeli problem z wykonalnością w pewnym obszarze może uniemożliwić zbudowanie w pełni funkcjonalnego systemu, powinno się informować o sposobie analizy tego uwarunkowania. Można to robić poprzez prototypy ilustrujące problem, badania lub innymi odpowiednimi środkami. Zagadnienia związane z tego rodzaju zagrożeniami powinny zostać rozważone przed rozpoczęciem budowy oprogramowania.

Nadmiar konstrukcyjny

Niezawodność to zdolność systemu do kontynuowania pracy po wykryciu błędu. Architektura często wprowadza wyższy poziom niezawodności niż określony w wymaganiach. Jedną z przyczyn jest to, że system złożony z wielu części o minimalnym poziomie niezawodności może nie spełniać wymagań w tym zakresie jako całość. W oprogramowaniu o wytrzymałości łańcucha nie decyduje wytrzymałość najsłabszego ogniwa, ale raczej wszystkie jego słabości przemnożone przez siebie. Architektura powinna wyraźnie wskazywać miejsca, w których programiści mają zadbać o pewien nadmiar konstrukcyjny, i te, w których wystarczy najprostsze rozwiązanie.

Określenie podejścia do nadmiaru konstrukcyjnego jest o tyle istotne, że wielu programistów wprowadza go w swoich klasach automatycznie, z czystej zawodowej pychy. Jasne zdefiniowanie oczekiwań w architekturze może pozwolić uniknąć sytuacji, w których część klas jest wysoce niezawodna, podczas gdy inne są pod tym względem ledwie zadowalające.

Kupować czy budować

Patrz też: Listę rodzajów dostępnych komercyjnie bibliotek i komponentów oprogramowania można znaleźć w punkcie „Biblioteki kodu" w podrozdziale 30.3.

Skrajnie radykalne podejście do budowy oprogramowania to całkowite odejście od budowania na rzecz jego kupienia lub skorzystania z dostępnego bezpłatnie narzędzia open source. Kupić można elementy graficznego interfejsu użytkownika, menedżery baz danych, narzędzia do pracy z obrazem, różnego rodzaju grafiką i wykresami, komponenty do komunikacji internetowej, elementy systemów zabezpieczeń i kryptograficzne, narzędzia do pracy z arkuszami kalkulacyjnymi i tekstem — lista nie ma praktycznie końca. Jedną z wielkich zalet programowania we współczesnych środowiskach GUI jest ogromna liczba mechanizmów, którymi programista od razu dysponuje, takich jak klasy graficzne, menedżery okien dialogowych, procedury obsługi klawiatury i myszy, kod współdziałający automatycznie z dowolną drukarką i monitorem itp.

Jeżeli architektura nie korzysta z gotowych składników, jej specyfikacja powinna wyjaśniać, na czym ma polegać przewaga budowanych komponentów nad dostępnymi bibliotekami.

Decyzje o wykorzystaniu istniejącej bazy

Jeżeli istnieją plany wykorzystania istniejącego już oprogramowania, przypadków użycia, formatów danych lub innych materiałów, architektura powinna wyjaśniać, w jaki sposób zostaną one doprowadzone do zgodności z innymi jej celami — o ile jest to potrzebne i planowane.

Strategia zmian

Patrz też: Więcej o systematycznym zarządzaniu zmianami w podrozdziale 28.2 „Zarządzanie konfiguracją".

Ponieważ budowanie oprogramowania to proces, w którym zarówno programiści, jak i użytkownicy zdobywają nową wiedzę, należy liczyć się z tym, że w trakcie programowania produkt będzie ulegał zmianom. Mogą one wynikać z modyfikacji typów danych i formatów plików czy też decyzji o dodaniu nowych funkcji lub zmianie istniejących. Planowana dalsza rozbudowa może doprowadzić do odkrycia potrzeby wprowadzenia dodatkowych mechanizmów. Może też zapaść decyzja o implementacji części systemu, które zostały odrzucone w jego pierwszej wersji. Jednym z wyzwań stojących przed architektem oprogramowania jest zapewnienie architekturze poziomu elastyczności, który umożliwi łatwe wprowadzenie przyszłych zmian.

Wady projektowe są często bardzo subtelne i ewoluują wraz z zapominaniem o wczesnych założeniach w trakcie opracowywania nowych funkcji i zastosowań.
— Fernando J. Corbató

Architektura powinna jasno opisywać podejście do przyszłych zmian w systemie, a tym samym pokazywać, że możliwości dalszej rozbudowy zostały rozważone i że najbardziej prawdopodobne kierunki rozwoju są zarazem najprostsze w implementacji. Jeżeli można oczekiwać zmian w formatach danych wejściowych i wyjściowych, zasadach interakcji z użytkownikiem lub wymaganiach w zakresie przetwarzania, architektura będzie miała za zadanie pokazywać, że

zmiany takie zostały wzięte pod uwagę i że skutki żadnej z modyfikacji nie będą wykraczać poza pewną niewielką liczbę klas. Strategia uwzględniania zmian może sprowadzać się do umieszczania w plikach danych numerów wersji, rezerwowania pól do wykorzystania w przyszłości czy projektowania plików w taki sposób, aby można było dodawać nowe tabele. Jeżeli używany jest generator kodu, architektura powinna wykazać, że oczekiwane zmiany mieszczą się w granicach jego możliwości.

Patrz też: Szersze omówienie zagadnienia opóźniania zaangażowania zostało przedstawione w punkcie „Uważnie wybieraj czas wiązania" w podrozdziale 5.3.

Architektura ma również wskazywać strategie opóźniania zaangażowania. Przykładowo, może ona nakazywać stosowanie w testach „if" zewnętrznych tabel — użycie zewnętrznych plików pozwala wtedy wprowadzać zmiany bez rekompilacji.

Ogólny poziom jakości architektury

Patrz też: Więcej informacji o interakcjach między atrybutami jakości można znaleźć w podrozdziale 20.1 „Składowe jakości".

Dobrą specyfikację architektury wyróżnia omówienie klas systemu i informacji ukrytych w każdej z nich oraz uzasadnienie wyboru i odrzucenia wszystkich możliwych alternatyw.

Architektura powinna być dopracowaną całością koncepcyjną, wolną w miarę możliwości od dodatków *ad hoc*. Główną tezą najpopularniejszej w historii książki o inżynierii oprogramowania, *Mityczny osobomiesiąc*, jest to, że kluczowym problemem dużych systemów jest zachowanie ich koncepcyjnej spójności (Brooks 2000). Dobra architektura powinna być dopasowana do zagadnienia. Patrząc na nią, powinieneś być zachwycony naturalnością i prostotą rozwiązania. Niedopuszczalne jest, aby problem i architektura sprawiały wrażenie wiązanych ze sobą na siłę.

Być może widziałeś, jak architektura zmienia się na kolejnych etapach swojego rozwoju. Każda taka zmiana powinna wpasowywać się w ogólną koncepcję. Architektura nie może wyglądać jak amerykańska ustawa budżetowa z wszystkimi jej kosztownymi, pełnymi rozmachu, choć nie zawsze celowości, zastrzeżeniami dla okręgu każdego jednego posła.

Główne cele architektury powinny być jasno określone. Projekt systemu, dla którego jednym z podstawowych celów jest modyfikowalność, będzie inny niż systemu, w którym najważniejsze jest ciągłe działanie, nawet jeżeli oba mają tę samą funkcję.

Architektura powinna podawać uzasadnienie każdej znaczącej decyzji. Warto zachować dystans do wyjaśnień o charakterze „bo tak się robi". Przypomina to historię, w której Beth gotowała szynkę według rodzinnego przepisu jej męża Abdula, który opowiedział, jak matka uczyła go, że należy przyprawić szynkę, obciąć oba końce, włożyć ją do garnka, przykryć i gotować. Gdy Beth spytała: „A po co obcinać końce?", Abdul mógł tylko odpowiedzieć: „Nie wiem. Zawsze tak robiłem. Mogę ją spytać". Wyjaśnienie jego matki było podobne: „Nie wiem, ale zawsze tak robiłam. Mogę spytać babci". Odpowiedź babci wiele wyjaśniła: „Nie wiem, czemu chcecie obcinać końce. Robiłam tak, bo akurat nie miałam większego garnka".

Dobra architektura jest w dużej mierze niezależna od języka i platformy sprzętowej, należy jednak przyznać, że środowiska budowy oprogramowania nie można ignorować. Z drugiej strony, dążąc do utrzymania jak największej niezależności, oddalamy się od pokusy popadnięcia w nadmierną szczegółowość opisu lub wykonania zadań, których miejsce jest w fazie programowania. Wyjątkiem są programy, których działanie dotyczy określonego komputera lub języka.

Twórcy architektury muszą dobrze wyważyć szczegółowość specyfikacji. Żadna część nie powinna być traktowana z większą uwagą, niż na to zasługuje — należy unikać nadmiaru konstrukcyjnego. Projektant nie może skupiać się na jednym fragmencie kosztem innych. Architektura powinna zapewniać spełnienie wszystkich wymagań i nie wprowadzać dodatkowych, niewymaganych elementów.

Zadaniem architektury jest wskazywać obszary ryzyka. Powinna ona wyjaśniać, dlaczego nimi są i jakie kroki podjęto w celu ograniczenia problemu.

W architekturze powinny być uwzględnione różne perspektywy. Plany domu obejmują plan ogólny, plany pięter, diagramy elektryczne i wiele innych schematów o bardzo różnym charakterze. W opisach architektury oprogramowania również stosuje się zróżnicowane spojrzenia na system. Pomaga to ujawnić błędy i niespójności oraz ułatwia programistom uzyskanie pełnego zrozumienia konstrukcji (Kruchten 1995).

Żaden element architektury nie powinien budzić niepokoju. Nie ma miejsca dla takich składowych, których istnienie uzasadnia jedynie żądanie przełożonego. Nie powinno być takich, których zrozumienie sprawia trudność. Jeżeli masz stworzyć implementację systemu, musisz dokładnie wiedzieć, o co w nim chodzi i jak działa.

cc2e.com/0337

Lista kontrolna: Architektura

Oto lista zagadnień, które powinny zostać poruszone w dobrym opisie architektury. Nie jest to w żadnym razie jej pełny plan, ale pragmatyczny sposób oceny wartości odżywczej materiału przekazywanego programiście w łańcuchu pokarmowym, jakim jest cykl tworzenia oprogramowania. Najlepiej wykorzystać tę listę jako podstawę do stworzenia własnej. Podobnie jak w przypadku listy kontrolnej wymagań, jeżeli zajmujesz się projektem nieformalnym, znajdziesz tu rzeczy, które nie mają w Twojej skali odpowiedników, a jeżeli pracujesz jako członek dużego projektu, istotny jest praktycznie każdy punkt.

Zagadnienia architektury

- ❑ Czy organizacja programu jest jasna? Czy opis zawiera dobry przegląd ogólny i uzasadnienie wybranego podejścia?
- ❑ Czy podstawowe bloki konstrukcyjne są jasno określone, łącznie z zakresami odpowiedzialności i interfejsami dla innych bloków?

- ❑ Czy wszystkim funkcjom wymienionym w wymaganiach poświęcono wystarczającą ilość uwagi? Czy za ich obsługę nie odpowiada zbyt wiele lub zbyt mało bloków konstrukcyjnych?
- ❑ Czy opisano większość krytycznych klas i uzasadniono ich istnienie?
- ❑ Czy istnieje projekt danych i jego uzasadnienie?
- ❑ Czy określono organizację i zawartość bazy danych?
- ❑ Czy zidentyfikowano wszystkie kluczowe reguły biznesowe i czy został opisany ich wpływ na system?
- ❑ Czy opisano strategię projektowania interfejsu użytkownika?
- ❑ Czy interfejs użytkownika został wydzielony w taki sposób, że wprowadzane w nim zmiany nie będą wpływały na inne części programu?
- ❑ Czy opisano strategię obsługi danych wejściowych i wyjściowych oraz uzasadniono jej wybór?
- ❑ Czy przedstawiono szacunki wykorzystania zasobów i strategię zarządzania nimi? Czy uzasadnienie obejmuje problemy wszystkich zasobów rzadkich, takich jak wątki, połączenia z bazą danych, uchwyty, przepustowość sieci itp.?
- ❑ Czy opisano wymagania architektury w zakresie zabezpieczeń?
- ❑ Czy architektura określa parametry wydajnościowe i wykorzystanie zasobów przez poszczególne klasy, podsystemy i obszary funkcjonalne?
- ❑ Czy w architekturze opisano, jak osiągane są cele w zakresie skalowalności?
- ❑ Czy architektura porusza zagadnienia współdziałania z innymi systemami?
- ❑ Czy opisano zasady internacjonalizacji i lokalizacji?
- ❑ Czy przedstawiono jednolitą strategię obsługi błędów?
- ❑ Czy zdefiniowano podejście do zagadnienia zabezpieczeń przed awariami (jeżeli są potrzebne)?
- ❑ Czy zweryfikowano wykonalność wszystkich części systemu?
- ❑ Czy określono zasady nadmiaru konstrukcyjnego?
- ❑ Czy podjęto niezbędne decyzje typu „kupować czy budować”?
- ❑ Czy opisano, w jaki sposób starszy kod zostanie dostosowany do innych celów architektury?
- ❑ Czy architektura jest gotowa na przyszłe zmiany?

Ogólna jakość architektury

- ❑ Czy architektura uwzględnia wszystkie wymagania?
- ❑ Czy nie ma części nadmiernie uproszczonych lub rozbudowanych? Czy oczekiwania w tym zakresie zostały jasno określone?
- ❑ Czy koncepcja architektury jest spójna i czy spójność tę zachowują poszczególne elementy?
- ❑ Czy projekt najwyższego poziomu jest niezależny od platformy sprzętowej i języka implementacji?
- ❑ Czy uzasadniono podstawowe decyzje konstrukcyjne?
- ❑ Czy Ty, programista, odbierasz architekturę jako poprawną, kompletną i gotową do implementacji?

3.6. Ilość czasu poświęcanego na przygotowania

Patrz też: Ilość czasu poświęcanego na przygotowania będzie zależeć od typu projektu. O dostosowywaniu procesu przygotowań do różnych projektów pisaliśmy w podrozdziale 3.2 „Określanie rodzaju budowanego oprogramowania".

Ilość czasu, jaki trzeba poświęcić na zdefiniowanie problemu, opracowanie wymagań i przygotowanie architektury, różni się w zależności od potrzeb projektu. Ogólnie przy dobrze prowadzonych projektach poświęca się około 10 – 20 procent energii i około 20 – 30 procent czasu na wymagania, architekturę i wstępne planowanie (McConnell 1998, Kruchten 2000). Liczby te nie obejmują projektowania szczegółowego, które jest już częścią procesu budowy oprogramowania.

Jeżeli projekt jest duży i sformalizowany, a wymagania nie są stabilne, rozwiązanie dotyczących ich problemów zidentyfikowanych w początkowej fazie programowania będzie często wymagało współpracy z analitykiem, który zajmuje się tymi wymaganiami. Uwzględnij czas na takie konsultacje oraz czas potrzebny analitykowi na zapoznanie się z nowymi informacjami i podjęcie niezbędnych decyzji. Jest to czas dzielący Cię od uzyskania dokumentu, na którym będzie opierała się Twoja praca.

Jeżeli wymagania nie są stabilne, ale projekt jest mały i niesformalizowany, rozwiązań tego typu problemów najczęściej szuka się samodzielnie. Uwzględnij wtedy czas na zdefiniowanie wymagań na tyle dobrze, aby ich zmienność nie miała dużego wpływu na budowę oprogramowania.

Patrz też: Metody radzenia sobie ze zmieniającymi się wymaganiami opisane zostały w punkcie „Zmiany wymagań w trakcie budowy oprogramowania" w podrozdziale 3.4.

Przy niestabilnych wymaganiach — niezależnie od poziomu formalizacji projektu — należy traktować pracę z nimi jak drugi, dodatkowy projekt. Oszacuj czas potrzebny na wykonanie reszty zadań dopiero po skończeniu pracy z wymaganiami. Jest to uzasadnione, bo nikt nie może oczekiwać rozsądnego nakreślenia ram czasowych, dopóki nie jest jasno określony przedmiot przedsięwzięcia. Przypominałoby to sytuację, w której klient dowiaduje się o koszty remontu i na pytanie uściślające „Jakie prace są do wykonania?" odpowiada: „Nie wiem, ale ile to będzie kosztowało?". Rozsądna osoba w takiej sytuacji grzecznie, acz konsekwentnie wycofuje się ze współpracy.

Gdy budujemy, brak możliwości podania wyceny przed określeniem przedmiotu umowy jest dość oczywisty. Również klient nie chce, aby wykonawca pojawił się z wszystkimi narzędziami i materiałami i zaczął naliczać koszty, zanim architekt skończy przygotowywanie planów. Jednak programowanie wydaje się na ogół mniej zrozumiałe niż budownictwo, należy więc liczyć się z tym, że nie każdy klient od razu pojmie, dlaczego życzysz sobie planować opracowywanie wymagań jako osobny projekt. Możesz być zmuszony do podania wyczerpującego uzasadnienia.

Przy alokowaniu czasu na pracę nad architekturą można zastosować podejście podobne jak przy opracowywaniu wymagań. Jeżeli pracujesz z oprogramowaniem, którego charakter jest dla Ciebie nowością, daj sobie więcej czasu na trudności związane z nowymi obszarami. Zadbaj o to, aby czas potrzebny do stworzenia dobrej architektury nie był tym odbieranym innym zajęciom. Jeżeli to konieczne, również przygotowanie architektury zaplanuj jako osobny projekt.

Więcej informacji

cc2e.com/0344 Poniżej przedstawiona została lista publikacji, w których można znaleźć więcej informacji o procesie przygotowywania specyfikacji wymagań i pracy z nimi.

cc2e.com/0351

Wymagania

Oto kilka podręczników, które bardziej szczegółowo opisują pracę z wymaganiami:

Wiegers, Karl. *Software Requirements, 2nd Ed.* Redmond, WA, USA, Microsoft Press 2003. Praktyczna, skoncentrowana na codziennych działaniach książka opisująca czynności związane z opracowywaniem wymagań: ich wybieranie, analizowanie, opisywanie, weryfikowanie oraz zarządzanie nimi.

Robertson, Suzanne, i James Robertson. *Mastering the Requirements Process*. Reading, MA, USA, Addison-Wesley, 1999. Dobra alternatywa dla książki Wiegersa dla bardziej zaawansowanych.

cc2e.com/0358 Gilb, Tom. *Competitive Engineering*. Reading, MA, Addison-Wesley 2004. Książka, która opisuje opracowany przez jej autora język specyfikacji wymagań o nazwie Planguage. Zawiera ona także opis specyficznego podejścia Gilba do opracowywania wymagań, projektowania i oceny projektów oraz metodę ewolucyjnego zarządzania projektem. Jest dostępna do pobrania w witrynie autora pod adresem *www.gilb.com*.

IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. Los Alamitos, CA, USA, IEEE Computer Society Press. Podręcznik pisania specyfikacji wymagań oprogramowania opracowany przez IEEE i ANSI. Wymienia elementy, które powinien zawierać dokument specyfikacji, i pokazuje kilka alternatywnych schematów takich dokumentów.

cc2e.com/0365

Abran, Alain, et al. *Swebok: Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA, USA, IEEE Computer Society Press 2001. Zawiera szczegółowe omówienie podstawowych zagadnień związanych z wymaganiami. Książka dostępna do pobrania pod adresem *www.swebok.org*.

Inne dobre alternatywy to:

Lausen, Soren. *Software Requirements: Styles and Techniques*. Boston, MA, USA, Addison-Wesley 2002.

Kovitz, Benjamin L. *Practical Software Requirements: A Manual of Content and Style*. Manning Publications Company 1998.

Cockburn, Alistair. *Writing Effective Use Cases*. Boston, MA, USA, Addison-Wesley 2000.

Architektura oprogramowania

cc2e.com/0372

W ciągu ostatnich kilku lat opublikowano bardzo wiele książek na temat architektury oprogramowania. Niektóre z najlepszych to:

Bass, Len, Paul Clements i Rick Kazman. *Architektura oprogramowania w praktyce.* Wydawnictwo Naukowo-Techniczne 2006.

Buschman, Frank, et al. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. New York, USA, John Wiley & Sons 1996.

Clements, Paul, pod red. *Documenting Software Architectures: Views and Beyond*. Boston, MA, USA, Addison-Wesley 2003.

Clements, Paul, Rick Kazman i Mark Klein. *Architektura oprogramowania. Metody oceny oraz analiza przypadków*. Helion 2003.

Fowler, Martin. *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe*. Helion 2005.

Jacobson, Ivar, Grady Booch i James Rumbaugh. *The Unified Software Development Process*. Reading, MA, USA, Addison-Wesley 1999.

IEEE Std 1471-2000. Recommended Practice for Architectural Description of Software-Intensive Systems. Los Alamitos, CA, USA, IEEE Computer Society Press. Podręcznik pisania specyfikacji architektury oprogramowania opracowany przez IEEE i ANSI.

cc2e.com/0379

Metodyki wytwarzania oprogramowania

Napisano bardzo wiele książek prezentujących różne podejścia do prowadzenia projektu informatycznego. Jedne są bardziej sekwencyjne, inne — bardziej iteracyjne.

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA, USA, Microsoft Press 1998. Książka opisująca jeden szczególny sposób prowadzenia

projektu. W podejściu tym nacisk położony jest na sekwencyjny proces planowania, opracowywania wymagań i projektowania architektury, po którym następuje metodyczne wykonanie. Zapewnia ono długoterminową przewidywalność kosztów i harmonogramów, wysoką jakość i umiarkowany poziom elastyczności.

Kruchten, Philippe. *The Rational Unified Process: An Introduction, 2nd Ed.* Reading, MA, USA, Addison-Wesley 2000. Książka, która prezentuje podejście „skoncentrowane na architekturze i sterowane przypadkami użycia". Podobnie jak w *Software Project Survival Guide* skupia się ono na przygotowaniach wykonywanych na początku procesu i zapewnia długoterminową przewidywalność kosztów i harmonogramów przy umiarkowanej elastyczności. Rational Unified Process to metodyka nieco bardziej wyrafinowana niż opisywane w *Software Project Survival Guide* i *Extreme Programming Explained: Embrace Change*.

Jacobson, Ivar, Grady Booch i James Rumbaugh. *The Unified Software Development Process.* Reading, MA, USA, Addison-Wesley 1999. Dokładniejsze omówienie tematów przedstawionych w *The Rational Unified Process: An Introduction, 2nd Ed.*

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA, USA, Addison-Wesley 2000. Beck opisuje podejście wysoce iteracyjne, które skoncentrowane jest na cyklicznej pracy z wymaganiami i projektami w połączeniu z budową oprogramowania. Extreme Programming praktycznie nie zapewnia przewidywalności w długim terminie, ale pozwala uzyskać wysoki poziom elastyczności.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, Wlk. Brytania, Addison-Wesley 1988. Podejście Gilba skoncentrowane jest na krytycznych problemach planowania, wymagań i architektury na początku projektu, aby potem w systemie ciągłym adaptować plany projektu wraz z postępami w jego realizacji. Umożliwia to łączenie długoterminowej przewidywalności, wysokiej jakości i dużej elastyczności. Jest to metodyka wymagająca większej systematyczności niż opisywane w *Software Project Survival Guide* i *Extreme Programming Explained: Embrace Change*.

McConnell, Steve. *Rapid Development*. Redmond, WA, USA, Microsoft Press 1996. Książka, która prezentuje podejście do planowania projektów bazujące na koncepcji przybornika. Osoba o pewnym doświadczeniu w zakresie planowania może wykorzystać opisane w niej narzędzia do stworzenia planu projektu precyzyjnie dopasowanego do specyficznych potrzeb konkretnego przedsięwzięcia.

Boehm, Barry, i Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA, USA, Addison-Wesley 2003. Próba zestawienia oraz porównania metodyk dynamicznych i bazujących na wczesnym planowaniu. Rozdział 3. zawiera cztery szczególnie ciekawe podrozdziały: „A Typical Day using PSP/TSP", „A Typical Day using Extreme Programming", „A Crisis Day using PSP/TSP" i „A Crisis Day using Extreme Programming". W rozdziale 5. omawiane jest zagadnienie ryzyka w równoważeniu dynamiczności i jego znaczenie dla wyboru między metodami dynamicznymi i opartymi na

wczesnym planowaniu. Rozdział 6., „Conclusions", jest bardzo wyważony i nakreśla bardzo pomocną perspektywę. Dodatek E to prawdziwa kopalnia empirycznych danych opisujących procesy Agile.

Larman, Craig. *Agile and Iterative Development: A Manager's Guide*. Boston, MA, USA, Addison-Wesley 2004. Dobrze przygotowane wprowadzenie do elastycznych, ewolucyjnych metod wytwarzania oprogramowania. Obejmuje opis metod Scrum, Extreme Programming, Unified Process i Evo.

cc2e.com/0386

Lista kontrolna: Przygotowania

- ❑ Czy określiłeś, z jakim rodzajem projektu masz do czynienia, i dostosowałeś odpowiednio stosowaną metodykę?
- ❑ Czy wymagania są wystarczająco dobrze określone i dość stabilne, aby można było rozpocząć programowanie? (Patrz lista kontrolna wymagań).
- ❑ Czy architektura jest wystarczająco dobrze zdefiniowana, aby można było rozpocząć programowanie? (Patrz lista kontrolna architektury).
- ❑ Czy wzięto pod uwagę inne czynniki ryzyka specyficzne dla danego projektu i nie jest on narażony na więcej zagrożeń, niż to konieczne?

Podsumowanie

- Nadrzędny cel przygotowań do budowy oprogramowania to redukcja ryzyka. Zadbaj o to, aby podejmowane działania wstępne ograniczały zagrożenia, a nie zwiększały je.
- Jeżeli chcesz stworzyć oprogramowanie wysokiej jakości, dbanie o nią musi być częścią procesu od początku do końca. Troska o jakość na początku ma większy wpływ na produkt końcowy niż starania w późniejszych etapach pracy.
- Częścią pracy programisty jest edukowanie przełożonych i współpracowników w temacie procesu wytwarzania oprogramowania. Należy do tego przypominanie o znaczeniu przeprowadzenia stosownych czynności przygotowawczych przed rozpoczęciem programowania.
- Rodzaj projektu ma bardzo duży wpływ na przygotowania do niego — jednym projektom służy duży poziom iteracyjności, podczas gdy inne należy realizować bardziej sekwencyjnie.
- Jeżeli problem nie zostanie dobrze zdefiniowany, proces budowy oprogramowania może koncentrować się na rozwiązywaniu złego problemu.
- Źle przygotowane wymagania grożą przeoczeniem istotnych aspektów problemu. Zmiany wymagań wprowadzane po rozpoczęciu programowania kosztują od 20 do 100 razy więcej niż te, które są wprowadzane wcześniej. Warto zadbać o dopracowanie specyfikacji.

- Źle przygotowana architektura to ryzyko, że w procesie budowy będziesz rozwiązywał właściwy problem w niewłaściwy sposób. Koszt wprowadzania w niej zmian rośnie wraz ilością gotowego kodu.
- Należy wiedzieć, jakie podejście zostało wybrane w trakcie przygotowań do projektu, i odpowiednio dostosować sposób pracy przy budowie oprogramowania.

Rozdział 4.

Kluczowe decyzje konstrukcyjne

cc2e.com/0489

W tym rozdziale

- 4.1. Wybór języka programowania — strona 95
- 4.2. Konwencje programowania — strona 100
- 4.3. Twoje położenie na fali technologii — strona 101
- 4.4. Wybór podstawowych praktyk programowania — strona 103

Podobne tematy

- Przygotowania: rozdział 3.
- Określanie rodzaju budowanego oprogramowania: podrozdział 3.2
- Wpływ rozmiarów programu na jego budowę: rozdział 27.
- Zarządzanie budową oprogramowania: rozdział 28.
- Projektowanie oprogramowania: rozdziały od 5. do 9.

Gdy masz już pewność, że fundamenty, na których będziesz budował, zostały odpowiednio przygotowane, możesz przejść do decyzji bezpośrednio związanych z przyszłym kodem. W rozdziale 3., „Przed programowaniem — przygotowania", zajmowaliśmy się tym, co w świecie oprogramowania odpowiada schematom i zezwoleniom na budowę. Programista ma często niewielką kontrolę nad tą częścią procesu, więc opis dotyczył przede wszystkim oceny materiałów dostępnych w chwili rozpoczęcia pracy. W tym rozdziale przechodzimy do tej części przygotowań, za którą, bezpośrednio lub pośrednio, odpowiadają poszczególni programiści i kierownicy zespołów. Można ją porównywać do dobierania narzędzi i ładowania jadącego na plac budowy samochodu.

Jeżeli uważasz, że naczytałeś się już dosyć o różnego rodzaju przygotowaniach, możesz przejść od razu do rozdziału 5. „Projektowanie".

4.1. Wybór języka programowania

> *Dzięki uwolnieniu umysłu od zbędnej pracy dobra notacja pozwala skoncentrować się na bardziej zaawansowanych problemach, czego skutkiem jest zwiększenie możliwości całej rasy. Przed wprowadzeniem notacji arabskiej mnożenie było trudne, a dzielenie nawet liczb całkowitych było zadaniem dla najbardziej lotnych matematyków. Dla greckiego matematyka największym zaskoczeniem we współczesnym świecie byłoby to, że... znaczna część mieszkańców zachodniej Europy potrafi wykonywać operacje dzielenia bardzo dużych liczb.*