Programowanie zorientowane obiektowo

Autor: Bertrand Meyer
Tłumaczenie: Michał Dadan (wstęp, rozdz. 1–6, 9, 10, 15, dod. A, C, D), Jarosław Dobrzański (rozdz. 7, 17–25), Jan Ostrowski (rozdz. 26–32), Jaromir Senczyk (rozdz. 16, 33–36, dod. B), Krzysztof Szafranek (rozdz. 8, 11–14)
ISBN: 83-7361-738-8
Tytuł oryginału: Object-Oriented Software Construction Second Edition
Format: B5, stron: 1464

Poznaj reguły projektowania i programowania obiektowego

- Elementy techniki obiektowej
- Metodyka tworzenia oprogramowania
- Implementacja mechanizmów obiektowych

Programowanie zorientowane obiektowo to technika, która w ciągu ostatnich lat zyskała niezwykłą popularność. Języki programowania obiektowego święcą triumfy, a metodologie projektowania oparte na analizie obiektowej stają się standardami przemysłowymi. Założenia analizy i programowania obiektowego są pozornie proste, jednakże bez ich właściwego zrozumienia nie można zaprojektować prawidłowo aplikacji implementowanej w obiektowym języku programowania. Technologia obiektowa zmieniła cały przemysł programistyczny, więc jej opanowanie jest niezbędnym elementem wiedzy każdego informatyka, który chce wykorzystywać w pracy nowoczesne metody i techniki.

Książka „Programowanie zorientowane obiektowo" to wyczerpujące omówienie wszystkich zagadnień związanych z projektowaniem i programowaniem obiektowym. Opisuje główne elementy techniki obiektowej oraz wiele spośród ich potencjalnych zastosowań. Dzięki książce poznasz również metodykę projektowania oprogramowania, dowiesz się, czym są wzorce projektowe, i nauczysz się, w jaki sposób zaimplementować lub zasymulować techniki obiektowe w różnych językach programowania.

- Podstawowe elementy projektowania obiektowego
- Wielokrotne wykorzystywanie kodu
- Analiza obiektowa
- Abstrakcyjne typy danych
- Klasy i obiekty
- Zarządzanie pamięcią
- Mechanizmy dziedziczenia
- Obsługa wyjątków
- Metodyka projektowania obiektowego
- Programowanie współbieżne
- Obiektowe bazy danych
- Zastosowanie technik obiektowych w różnych językach programowania

Wykorzystaj techniki obiektowe i popraw jakość tworzonego przez siebie oprogramowania

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

SPIS TREŚCI

17.

Kontrola typów 691

18.

Obiekty i stałe globalne 727

Część IV Metodyka obiektowa — prawidłowe stosowanie metody 747

19.

O metodyce 749

20.

Wzorzec projektowy — wielopanelowe systemy interaktywne 763

24.

Prawidłowe stosowanie dziedziczenia 907

25.

Przydatne techniki 973

26.

Poczucie stylu 979

27.

Analiza obiektowa 1011

28.

Proces tworzenia oprogramowania 1035

29.

Nauczanie 1049

Część V Zagadnienia zaawansowane 1067

30.

Współbieżność, rozproszenie, klient-serwer i internet 1069

31.

Trwałość obiektów i bazy danych 1161

35.

36.

A

B

C

D

E

1

JAKOŚĆ OPROGRAMOWANIA

Słowo inżynieria nieodzownie kojarzy się z jakością. Inżynieria oprogramowania to tworzenie wysokich jakościowo programów. Ta książka opisuje techniki, które potencjalnie mogą przyczynić się do znacznego podniesienia jakości aplikacji komputerowych.

Zanim zacznę je omawiać, muszę napisać, w jakim celu zostały wymyślone. Jakość oprogramowania najlepiej pojmować jako sumę wielu czynników. W tym rozdziale opiszę kilka z nich, zidentyfikuję obszary, jakie najpilniej wymagają usprawnień, i wskażę kierunek, w którym wspólnie będziemy szukać rozwiązań w trakcie dalszej lektury.

1.1. CZYNNIKI ZEWNĘTRZNE I WEWNĘTRZNE

Wszyscy chcemy, aby nasze oprogramowanie było szybkie, niezawodne, łatwe w obsłudze, czytelne, modułowe, sensownie skonstruowane itd. Tyle tylko, że te przymiotniki opisują dwa różne rodzaje zalet.

Po jednej stronie mamy zalety, takie jak szybkość czy łatwość obsługi, których obecność lub brak może z łatwością stwierdzić każdy użytkownik. Tego rodzaju właściwości możemy nazwać **zewnętrznymi** czynnikami decydującymi o jakości.

> Pod pojęciem „użytkownicy" należy rozumieć nie tylko osoby, które będą fizycznie korzystały z gotowych programów (na przykład pracownik linii lotniczych obsługujący system rezerwacji lotów), ale także decydentów, którzy podjęli decyzję o ich zakupie bądź zlecili ich opracowanie (analogicznie dyrektor linii lotniczych, którego uprawnienia na to pozwalają). Tak więc taka właściwość jak łatwość, z jaką oprogramowanie daje się dostosowywać do zmian w specyfikacji (w dalszej części tego rozdziału nazywana *rozszerzalnością*), jest zaliczana do kategorii czynników zewnętrznych, mimo iż „użytkownicy końcowi", tacy jak pracownik dokonujący rezerwacji, mogą nie być nią bezpośrednio zainteresowani.

Inne możliwe zalety programu komputerowego, takie jak czytelność czy modułowość, zaliczamy do czynników **wewnętrznych**, ponieważ są one dostrzegalne jedynie dla informatyków mających dostęp do kodu źródłowego programu.

W ostatecznym rozrachunku liczą się tylko czynniki zewnętrzne. Gdy korzystam z przeglądarki internetowej lub mieszkam w pobliżu elektrowni atomowej sterowanej przez program komputerowy, tak naprawdę nie obchodzi mnie to, czy jego kod źródłowy jest modularny i czytelny. Ważne, żeby obrazki w przeglądarce nie wczytywały się godzinami, a błędne dane wejściowe nie mogły doprowadzić do wysadzenia elektrowni. Jednak kluczem do osiągnięcia satysfakcjonujących czynników zewnętrznych są czynniki wewnętrzne. Aby użytkownicy mogli cieszyć się widocznymi oznakami jakości aplikacji, ich projektanci i implementatorzy muszą stosować techniki gwarantujące wysoką jakość wewnętrzną.

W kolejnych rozdziałach przedstawiam zbiór współczesnych technik pozwalających na uzyskanie właśnie wewnętrznej jakości aplikacji. Czytając ten rozdział, nie powinieneś jednak tracić z oczu szerszego obrazu. Techniki wewnętrzne nie są naszym celem same w sobie. Są to jedynie środki, za pomocą których możemy uzyskać wysoką jakość zewnętrzną tworzonych aplikacji. Dlatego też musimy zacząć od omówienia czynników zewnętrznych. Przeznaczyłem na to pozostałą część tego rozdziału.

1.2. PRZEGLĄD CZYNNIKÓW ZEWNĘTRZNYCH

Poniżej opisuję najważniejsze czynniki zewnętrzne decydujące o jakości, które powinny być obecne w każdym programie komputerowym. Doprowadzenie do takiego stanu jest głównym celem programowania zorientowanego obiektowo.

Poprawność

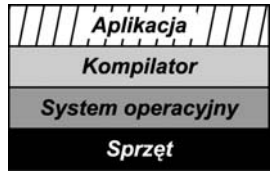
Definicja poprawności

Poprawność to zdolność programów komputerowych do wykonywania dokładnie tych zadań, które im powierzono i które zdefiniowano w ich specyfikacjach.

Poprawność jest zdecydowanie najważniejszym czynnikiem. Jeżeli system nie robi tego, czego się po nim oczekuje, wszystko inne — czy jest szybki, czy ma przyjemny interfejs użytkownika itd. — nie ma większego znaczenia.

Łatwo o niej pisać, ale znacznie trudniej ją osiągnąć. Już pierwszy krok na drodze do poprawności jest trudny. Trzeba bowiem precyzyjnie określić wymagania stawiane systemowi.

Techniki stosowane w celu zapewnienia poprawności zazwyczaj odnoszą się jedynie do pewnego obszaru systemu. Poważne współczesne programy komputerowe, nawet te mniejsze, odnoszą się do zagadnień z tak wielu dziedzin, że niemożliwe byłoby zagwarantowanie ich poprawności po przeanalizowaniu komponentów i właściwości tylko na jednej płaszczyźnie. Dlatego konieczne jest podejście warstwowe, w którym wyższe warstwy polegają na niższych (patrz rysunek 1.1).



RYSUNEK 1.1. Warstwy w procesie tworzenia oprogramowania

W tak zwanym **warunkowym** podejściu do poprawności interesuje nas jedynie zagwarantowanie poprawności danej warstwy *przy założeniu*, że wszystkie warstwy leżące niżej na pewno są poprawne. Jest to jedyne realistyczne podejście, ponieważ pozwala rozbić problem na kilka części i koncentrować się w danym momencie tylko na jednej z nich. Nie możesz twierdzić, że program napisany w języku wysokiego poziomu X jest poprawny, jeżeli nie masz pewności, że kompilator, którym dysponujesz, prawidłowo implementuje język X. Nie oznacza to wcale, że musisz ślepo ufać kompilatorowi. Po prostu problem zostaje tu rozbity na dwie części. Poprawność kompilatora i poprawność programu z punktu widzenia semantyki języka.

W technice opisywanej w tej książce poprawność zależy od jeszcze większej liczby warstw (patrz rysunek 1.2), ponieważ tworząc programy komputerowe, będziemy polegali na bibliotekach komponentów przeznaczonych do wielokrotnego użycia, które mogą być wykorzystywane do budowy wielu różnych aplikacji.

RYSUNEK 1.2. Warstwy w procesie tworzenia aplikacji, w którym posługujemy się komponentami wielokrotnego użytku

Również w tym przypadku stosuje się podejście warunkowe. Powinniśmy zagwarantować poprawność bibliotek, a następnie — w oparciu o nią — poprawność aplikacji.

Wielu praktykom stykającym się z zagadnieniem poprawności oprogramowania przychodzi na myśl testowanie i lokalizowanie błędów. Jednak do problemu można podejść bardziej ambitnie. W dalszych rozdziałach przyjrzymy się wielu technikom, w szczególności kontroli typów i asercjom, które wymyślono po to, aby ułatwić tworzenie oprogramowania, które jest poprawne od samego początku. Jest to znacznie lepsze podejście, niż zmuszanie programu do poprawności dopiero na pewnym etapie. Jednak nie zwalnia nas ono z testowania i lokalizowania błędów, czyli czynności, które w tym przypadku pozwalają na upewnienie się, że program faktycznie jest poprawny.

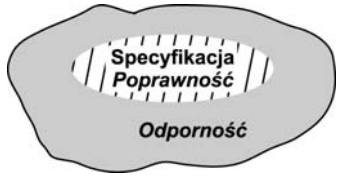
Można iść jeszcze dalej i przyjąć całkiem sformalizowane podejście do tworzenia oprogramowania. Postanowiłem stosować w książce dość potoczny język, dlatego wspomnę jedynie, że wiele technik, które opisuję w dalszych rozdziałach, wywodzi się bezpośrednio ze ścisłych, matematycznych technik formułowania specyfikacji programów i ich weryfikacji, które pozwalają bardzo mocno zbliżyć się do ideału w pełni poprawnego oprogramowania.

Odporność

Definicja odporności

Odporność to zdolność systemów komputerowych do prawidłowego reagowania na niestandardowe sytuacje.

Odporność stanowi uzupełnienie poprawności (patrz rysunek 1.3). Poprawność określa zachowanie systemów w sytuacjach przewidzianych w specyfikacji. Z kolei odporność charakteryzuje działania podejmowane po napotkaniu sytuacji wykraczających poza specyfikację.



RYSUNEK 1.3. Odporność kontra poprawność

Już sama definicja pokazuje, że odporność nie jest tak ścisłym pojęciem jak poprawność. Ponieważ mówimy tu o przypadkach nieprzewidzianych w specyfikacji, nie możemy powiedzieć, że system powinien wówczas po prostu „wykonywać swoje zadania". Gdyby te zadania były znane, każdy taki niestandardowy przypadek stałby się częścią specyfikacji, co przeniosłoby nas z powrotem w obszar poprawności.

Definicja „niestandardowej sytuacji" przyda się jeszcze, gdy będziemy omawiali obsługę wyjątków. Wynika z niej, że to, co nazywamy sytuacją standardową, a co niestandardową, zależy od przyjętej specyfikacji. Sytuacja niestandardowa to po prostu taka, która nie została w niej przewidziana. Jeżeli poszerzymy specyfikację, wówczas sytuacje, które były niestandardowe, staną się standardowe, nawet jeśli dotyczą takich okoliczności jak wprowadzenie przez użytkownika nieprawidłowych danych. Tak więc „standardowe" nie oznacza w tym przypadku „pożądane", a jedynie „przewidziane na etapie projektowania programu". Mimo iż nazywanie wprowadzenia błędnych danych „standardową sytuacją" może z początku wydawać się paradoksalne, każde inne rozwiązanie wymagałoby ustalenia obiektywnych kryteriów, a tym samym byłoby bezużyteczne.

Informacje na temat obsługi wyjątków znajdziesz w rozdziale 12.

Zawsze mogą zdarzyć się sytuacje, których specyfikacja bezpośrednio nie przewiduje. Znaczenie odporności polega na gwarancji, że pojawienie się takich okoliczności nie spowoduje żadnej katastrofy. Program powinien po prostu wyświetlić stosowne komunikaty o błędzie i zakończyć w normalny sposób swoje wykonywanie lub wejść w tak zwany „tryb łagodnego zejścia".

Rozszerzalność

Definicja rozszerzalności

Rozszerzalność to możliwość łatwego dostosowywania programu komputerowego do zmian w specyfikacji.

Oprogramowanie, w przeciwieństwie do sprzętu, z założenia powinno dawać się łatwo modyfikować. Tak właśnie jest, jeśli tylko mamy dostęp do kodu źródłowego. Jego zmiana jest w takiej sytuacji banalnie prosta. Wystarczy posłużyć się dowolnym edytorem tekstów.

Z zagadnieniem rozszerzalności wiąże się problem skali. W małych aplikacjach wprowadzenie zmian zazwyczaj nie stanowi problemu. Jeśli jednak mamy do czynienia z coraz bardziej rozbudowanymi programami, to coraz trudniej je modyfikować. Tego rodzaju aplikacje przypominają czasami ogromne domki z kart, a wyciągnięcie tylko jednego elementu może spowodować zawalenie się całej konstrukcji.

Rozszerzalność jest potrzebna ze względu na to, że u podstaw każdego programu komputerowego leży czynnik ludzki, co czyni go podatnym na błędy, które trzeba później poprawiać. Weźmy pod uwagę na przykład programy biznesowe (takie jak informacyjne systemy zarządzania). Wystarczy, że zmienią się jakieś przepisy prawa lub firma zostanie przejęta przez konkurencję, a już założenia, na których oparty był system, przestaną być aktualne. Jest to dość sztampowy przykład, ale niewiele osób zdaje sobie sprawę z tego, że z podobną sytuacją mamy do czynienia nawet w programach naukowych, w przypadku których możemy zakładać, że za miesiąc lub dwa będą nadal obowiązywały te same prawa fizyki co dzisiaj. Jednak może ulec zmianie sposób, w jaki je pojmujemy, lub procedura modelowania pewnych zjawisk.

W tradycyjnym podejściu do inżynierii oprogramowania nie przykładano wystarczająco dużej wagi do zmian. Zamiast tego opierano się na idealistycznym postrzeganiu cyklu życia oprogramowania, w którym wymagania były zamrażane już na etapie analizy potrzeb. Na dalszych etapach skupiano się już wyłącznie na projektowaniu i tworzeniu rozwiązania. Jest to zrozumiałe. Pierwszym krokiem na drodze rozwoju inżynierii programowania było opracowanie przejrzystych technik formułowania i rozwiązywania stałych, niezmiennych problemów. Dopiero później zaczęto się martwić, co należy robić w sytuacji, gdy problem ulega zmianie w czasie, kiedy wciąż szukamy rozwiązania. Obecnie podstawowe techniki inżynierii oprogramowania są już opracowane i dlatego pora zająć się opisywanym problemem. Zmiany są nieodłącznym elementem procesu tworzenia oprogramowania. Możemy mieć do czynienia ze zmianami wymagań, sposobu ich pojmowania, algorytmów, sposobu prezentacji danych lub technik implementacyjnych. Umożliwienie łatwego ich wprowadzania jest głównym zadaniem techniki obiektowej i tematem przewodnim tej książki.

Choć wiele technik zwiększających rozszerzalność można prezentować na prostych przykładach, ich znaczenie staje się jasne dopiero w przypadku większych projektów. Przy zwiększaniu rozszerzalności kluczową rolę odgrywają dwie zasady:

- *Prostota projektu* — proste architektury zawsze łatwiej zaadaptować do zmian niż złożone.
- *Decentralizacja* — im większa autonomia poszczególnych modułów, tym większe prawdopodobieństwo, że prosta zmiana specyfikacji będzie wymagała zmodyfikowania tylko jednego lub kilku modułów i nie wywoła reakcji łańcuchowej w całym systemie.

Technika programowania zorientowanego obiektowo jest przede wszystkim techniką tworzenia takich architektur systemów, które pomagają projektantom w tworzeniu zdecentralizowanych programów o prostej strukturze (nawet jeśli są to duże projekty). Prostota i decentralizacja to kluczowe zagadnienia, które będą się przewijały w naszej dyskusji do czasu, aż w kolejnych rozdziałach sformułujemy zasady rządzące techniką obiektową.

Przystosowanie do wielokrotnego użycia

Definicja przystosowania do wielokrotnego użycia

Przystosowanie do wielokrotnego użycia to zdolność elementów składowych oprogramowania do pełnienia roli elementów konstrukcyjnych wielu różnych aplikacji.

Potrzeba przystosowania do wielokrotnego użycia wynika z obserwacji, że w wielu programach komputerowych przewijają się te same wzorce. Dlatego powinna istnieć możliwość wykorzystania tych podobieństw, pozwalająca na uniknięcie ponownego wynajdywania koła. Dostrzegając taki wzorzec, możemy wyodrębnić element programu, który nadaje się do wykorzystania w wielu różnych projektach.

Przystosowanie do wielokrotnego użycia ma wpływ na wszystkie inne aspekty jakości oprogramowania, ponieważ dzięki niemu nie trzeba pisać tak dużo kodu jak dawniej i można skupić się (przy takim samym całkowitym koszcie projektu) na poprawianiu innych czynników, takich jak poprawność czy odporność.

Jest to przykład kolejnego zagadnienia, na które nie położono wystarczająco dużego nacisku w tradycyjnym pojmowaniu cyklu życia aplikacji. Powody historyczne są tu takie same jak w poprzednim przypadku. Przyjęto, że najpierw trzeba znaleźć rozwiązanie jednego problemu, a dopiero potem można stosować je w odniesieniu do innych problemów. Jednak rozrastanie się programów komputerowych i próba uczynienia z informatyki prawdziwego przemysłu doprowadziły do tego, że przystosowanie kodu do wielokrotnego użycia stało się palącą potrzebą.

Przystosowanie do wielokrotnego użycia będzie odgrywało kluczową rolę przy omawianiu zagadnień przedstawionych w następnych rozdziałach. Jeden z nich jest praktycznie w całości poświęcony dogłębnej analizie tego czynnika decydującego o jakości, omówieniu wymiernych korzyści, jakie on daje, i problemów, jakie się z nim wiążą.

O przystosowaniu do wielokrotnego użycia piszę w rozdziale 4.

Zgodność

Definicja zgodności

Zgodność to łatwość, z jaką można ze sobą łączyć poszczególne składniki oprogramowania.

Zgodność jest bardzo istotna, ponieważ składniki programistyczne nie powstają w próżni. Muszą ze sobą współpracować. Jednak bardzo często mają z tym problemy, ponieważ ich twórcy poczynili sprzeczne założenia dotyczące świata zewnętrznego. Jednym z przy-

kładów tego typu sytuacji jest ogromna różnorodność niezgodnych ze sobą formatów plików, obsługiwanych przez różne systemy operacyjne. Każdy program komputerowy może bezpośrednio przyjmować wyniki działania innego programu tylko wtedy, jeżeli formaty plików stosowane przez obie aplikacje są ze sobą zgodne.

Brak zgodności może odprowadzić do prawdziwej katastrofy. Oto skrajny przykład.

San Jose (Kalifornia), Mercury News, 20 czerwca 1992. Artykuł cytowany na grupie dyskusyjnej „comp.risks", 13.67, w czerwcu 1992. Niektóre fragmenty pominąłem.

> DALLAS. W zeszłym tygodniu AMR, spółka będąca właścicielem firmy American Airlines, Inc., przyznała, że połamała zęby na próbie stworzenia doskonałego, uniwersalnego systemu rezerwacji lotów, który miał też umożliwiać rezerwowanie samochodów i pokoi hotelowych.
>
> AMR zaprzestała rozwijania swojego nowego systemu rezerwacji Confirm w parę tygodni po upływie pierwotnego terminu oddania go do użytku, kiedy to miała ruszyć współpraca w zakresie obsługi rezerwacji z partnerami firmy: Budget Rent-A-Car, HiltonHotels Corp. oraz Marriott Corp. Zawieszenie czteroletniego projektu, który pochłonął 125 milionów dolarów, doprowadziło do zmniejszenia zysków netto firmy o 165 milionów dolarów i zrujnowało jej reputację lidera branży. […]
>
> Już w styczniu kierownictwo projektu Confirm zdało sobie sprawę z tego, że wysiłki ponad 200 programistów, analityków systemowych i inżynierów poszły na marne. Najważniejsze elementy tego ogromnego projektu, których dokumentacja zajęła 47 000 stron, były rozwijane niezależnie od siebie i w dodatku przy użyciu różnych technik. Po złożeniu ich w jedną całość okazało się, że mimo wysiłków programistów, nie chcą ze sobą współpracować. Poszczególne „moduły" aplikacji nie dawały sobie rady z „wyciąganiem" potrzebnych im informacji z innych modułów.
>
> Firma AMR Information Services zwolniła wtedy ośmiu głównych programistów, w tym kierownika projektu.[…] Pod koniec czerwca firmy Budget i Hilton ogłosiły, że wycofują się z projektu.

Kluczem do zgodności jest homogeniczność projektu i wypracowanie jednego standardu przesyłania informacji wewnątrz aplikacji. W tym celu należy zastosować:

- standaryzowane formaty plików, tak jak w systemie operacyjnym Unix, w którym każdy plik tekstowy jest po prostu sekwencją znaków,
- standaryzowane struktury danych, tak jak w programach napisanych w języku Lisp, w których wszystkie dane i programy są reprezentowane przez drzewa binarne, nazywane listami,
- standaryzowane interfejsy użytkownika, tak jak w różnych wersjach systemu Windows, OS/2 czy MacOS, w których wszystkie narzędzia komunikują się z użytkownikiem w taki sam sposób, oparty na wykorzystaniu standardowych komponentów, takich jak okna, ikony, menu itd.

Informacje na temat abstrakcyjnych typów danych znajdziesz w rozdziale 6.

Bardziej ogólne rozwiązania uzyskujemy, definiując standaryzowane protokoły dostępu do wszystkich istotnych elementów, na których operuje oprogramowanie. Właśnie ta idea stoi za abstrakcyjnymi typami danych i podejściem obiektowym, a także za tak zwanymi protokołami *warstwy pośredniczącej*, takimi jak CORBA czy OLE-COM (ActiveX) Microsoftu.

Wydajność

Definicja wydajności

Wydajność to zdolność programu do stawiania komputerowi jak najmniejszych wymagań sprzętowych, rozumianych między innymi jako czas procesora, przestrzeń zajmowana w pamięci wewnętrznej i zewnętrznej, szerokość pasma wykorzystywanego przy przesyłaniu danych itd.

Środowisko programistów wykazuje zazwyczaj dwa różne podejścia do problemu wydajności.

- Niektórzy programiści mają na tym punkcie obsesję, w związku z czym poświęcają mnóstwo czasu na optymalizację kodu i wkładają w nią wiele wysiłku.
- Jednak istnieje również inna tendencja, przejawiająca się w mądrościach ludowych, takich jak „najpierw spraw, żeby to działało, a dopiero potem zajmij się przyspieszaniem", czy „za rok komputery i tak będą o połowę szybsze".

Często zdarza się, że ta sama osoba raz jest zwolennikiem pierwszej, a raz drugiej teorii. To swego rodzaju programistyczne rozdwojenie jaźni (Dr Abstrakcja i Mr Mikrosekunda).

Które podeście jest lepsze? Cóż, programiści udowadniali już wielokrotnie, że kwestię optymalizacji kodu potrafią traktować przesadnie. Jak już wspomniałem, wydajność nie na wiele się zda, jeżeli oprogramowanie nie jest poprawne (to prowadzi do sformułowania kolejnej złotej zasady: „nie przejmuj się tym, czy program jest szybki, dopóki nie jest poprawny", która przypomina poprzednią, ale nie do końca znaczy to samo). Mówiąc bardziej ogólnie, musimy wprowadzić pewną równowagę między potrzebami w zakresie wydajności a pozostałymi celami, takimi jak rozszerzalność czy przystosowanie do wielokrotnego użycia. Przesadne optymalizacje mogą uczynić aplikację tak wyspecjalizowaną, że nie będzie się poddawała modyfikacjom, ani nadawała do wielokrotnego wykorzystania. Ponadto stale rosnąca szybkość komputerów pozwala przyjąć bardziej luźne podejście do kwestii wydajności i zwalnia z walki o każdy bajt i każdą mikrosekundę.

Jednak żaden z tych czynników nie umniejsza znaczenia wydajności. Nikt nie lubi czekać na reakcję interaktywnych aplikacji ani dokupywać pamięci tylko po to, żeby w ogóle można było uruchomić dany program. Także luźne podejście do wydajności często jest tylko pozą. Jeżeli końcowa wersja aplikacji jest tak wolna, że trudno z niej normalnie korzystać, to nawet osoby twierdzące, że „szybkość nie ma znaczenia", zaczynają narzekać.

Ta kwestia odzwierciedla coś, co — moim zdaniem — najlepiej charakteryzuje inżynierię oprogramowania (i myślę, że to się szybko nie zmieni). Tworzenie oprogramowania jest trudne głównie dlatego, że wymaga brania pod uwagę wielu różnych czynników, z których jedne, takie jak poprawność, są abstrakcyjne i względne, a inne, takie jak wydajność, bardzo konkretne i związane z właściwościami konkretnego sprzętu komputerowego.

Dla niektórych naukowców tworzenie oprogramowania jest dziedziną matematyki. Dla innych jest to gałąź techniki. W rzeczywistości obie grupy mają rację. Programista musi pogodzić abstrakcyjne idee i założenia z niedoskonałymi możliwościami ich implementacji, a teorie matematyczne niezbędne do przeprowadzenia poprawnych

obliczeń z ograniczeniami czasowymi i przestrzennymi, wynikającymi z praw fizyki i z niedoskonałości współczesnego sprzętu. Prawdopodobnie ta potrzeba zadowolenia zarówno wilków, jak i owiec, stanowi największe wyzwanie inżynierii oprogramowania.

Choć stale rosnąca szybkość komputerów robi duże wrażenie, nie można z jej powodu zaniechać wysiłków na rzecz uzyskania wysokiej wydajności i to co najmniej z trzech powodów.

- Ktoś, to kupuje większy i szybszy komputer, chce w ten sposób osiągnąć jakieś konkretne korzyści — mieć możliwość rozwiązywania nowych problemów bądź szybszego rozwiązywania dotychczasowych lub też rozwiązywania starych problemów w takim samym czasie jak dawniej, tyle że rozpatrywanych w większej skali. Natomiast kupowanie nowego komputera po to, aby rozwiązywać stare problemy w takim samym czasie co poprzednio, po prostu nie ma sensu!
- Jednym z najbardziej widocznych skutków zwiększania się szybkości komputerów jest *wzrost* przewagi dobrych algorytmów nad kiepskimi. Załóżmy, że nowy komputer jest dwa razy szybszy od starego. Niech n określa rozmiary problemu, który należy rozwiązać, zaś N będzie maksymalnym n, jakie dany algorytm jest w stanie obsłużyć w danym czasie. Jeżeli algorytm ma złożoność $O(n)$, to znaczy wykonuje się w czasie proporcjonalnym do n, to wówczas nowy komputer pozwoli rozwiązać problem o rozmiarze $2*N$ (dla dużego N). W przypadku algorytmów w postaci $O(n^2)$ nowy komputer pozwoli zwiększyć N jedynie o 41%. Natomiast algorytm typu $O(2^n)$, taki jak pewne algorytmy wyszukiwania, zwiększy N jedynie o jeden. Nie za dużo, jak na wydane pieniądze.
- W niektórych sytuacjach wydajność może wpływać na poprawność. W specyfikacji może być na przykład napisane, że komputer musi zareagować na określone zdarzenie nie później niż po upływie określonego czasu. Na przykład komputer pokładowy w samolocie musi być przygotowany na wykrycie i przetworzenie komunikatu od czujnika przepustnicy w czasie na tyle krótkim, aby było możliwe przeprowadzenie ewentualnej korekty lotu. Takie powiązanie między wydajnością a poprawnością nie jest ograniczone jedynie do programów, które potocznie nazywa się aplikacjami czasu rzeczywistego. Mało kto uświadamia sobie, że z podobną sytuacją mamy do czynienia w programach przewidujących pogodę na następny dzień, które analizują na bieżąco dane napływające przez 24 godziny.

  Inna przykładowa sytuacja, która może nie stanowiła wielkiego problemu, ale strasznie mnie denerwowała. Używałem kiedyś systemu zarządzania oknami, który był na tyle wolny, że zdarzało mu się nie zarejestrować przesunięcia kursora myszy z jednego okna do drugiego. W rezultacie znaki wpisywane przeze mnie na klawiaturze nie trafiały do tego okna, do którego powinny.

  W tym przypadku ograniczenia wydajności powodują naruszenie specyfikacji, a konkretnie wpływają na poprawność, a to nawet w niewinnych, używanych na co dzień aplikacjach może mieć niemiłe konsekwencje. Pomyśl, co mogłoby się stać, gdyby dwa otwarte na ekranie okna, o których wspomniałem, były oknami nowych wiadomości pocztowych, skierowanych do dwóch różnych odbiorców. Wojny wybuchały już z dużo bardziej błahych powodów.

Ponieważ w tej książce skupiam się na założeniach inżynierii obiektowej, a nie na kwestiach implementacyjnych, tylko w kilku podrozdziałach piszę wprost o kosztach w zakresie wydajności aplikacji. Musisz jednak pamiętać, że problem istnieje i że jestem

jego świadomy. Za każdym razem gdy omawiam przykładowe obiektowe rozwiązanie jakiegoś problemu, możesz być pewien, że przemyślałem je tak, aby było nie tylko eleganckie, ale też wydajne. Zawsze gdy wprowadzam jakiś nowy mechanizm obiektowy, taki jak odzyskiwanie pamięci (i inne techniki zarządzania pamięcią w aplikacjach obiektowych), wiązanie dynamiczne, generyczność czy wielokrotne dziedziczenie, pamiętam o tym, że musi on być zaimplementowany w taki sposób, aby był wydajny pod względem czasu i wymaganej przestrzeni. Dlatego tam, gdzie uznam to za przydatne, będę wspominał o wpływie przyjętego podejścia na wydajność.

Wydajność jest tylko jednym z czynników składających się na jakość. Dlatego nie powinniśmy (jak niektórzy nasi koledzy po fachu) dopuszczać do tego, żeby kierowała naszymi działaniami. Niemniej jednak należy o niej pamiętać, zarówno przy tworzeniu konkretnej aplikacji, jak i na etapie projektowania nowego języka programowania. Jeżeli powiesz wydajności: „Do widzenia", wydajność to samo powie Tobie.

Przenośność

Definicja przenośności

Przenośność to łatwość, z jaką dane oprogramowanie daje się przenosić między różnymi środowiskami sprzętowymi i programowymi.

Tak więc przenośność nie odnosi się jedynie do różnic między fizycznymi urządzeniami sprzętowymi, ale do bardziej ogólnej, **sprzętowo-programowej** natury komputera. Tej, z którą mamy styczność, pisząc aplikacje, a więc między innymi do systemu operacyjnego, systemu okien (jeśli jest on stosowany) i innych podstawowych narzędzi. W dalszej części książki tego rodzaju urządzenie sprzętowo-programowe nazywać będziemy „platformą". Przykładem platformy jest na przykład „komputer z procesorem Intel X86, pracujący pod kontrolą Windows NT", w skrócie „Wintel".

Wiele obecnych niezgodności między różnymi platformami wydaje się nie mieć sensownego usprawiedliwienia. Naiwny obserwator mógłby nawet stwierdzić, że z pewnością chodzi o spisek przeciwko ludzkości, a w szczególności przeciw programistom. W każdym razie, niezależnie od przyczyn takiego stanu rzeczy, różnorodność sprawia, że przenośność oprogramowania jest wielkim problemem zarówno dla programistów, jak i użytkowników.

Łatwość obsługi

Definicja łatwości obsługi

Łatwość obsługi to łatwość, z jaką ludzie o różnym doświadczeniu i kwalifikacjach mogą nauczyć się korzystania z aplikacji i stosowania ich do rozwiązywania problemów. Termin ten obejmuje też łatwość instalacji i nadzorowania pracy systemu.

W definicji tej położono nacisk na zróżnicowanie poziomu umiejętności potencjalnych użytkowników. Stanowi to jedno z największych wyzwań dla projektantów aplikacji zajmujących się kwestiami łatwości obsługi. W jaki sposób zapewnić odpowiednią pomoc początkującym i wszystko im wyjaśnić, a przy tym nie zanudzić doświadczonych użytkowników, którzy od razu chcą zabrać się do pracy?

Podobnie jak w przypadku wielu innych czynników opisywanych w tym rozdziale, jednym z kluczy do osiągnięcia łatwości obsługi jest zastosowanie prostej struktury programu. Dobrze zaprojektowany system, zbudowany w oparciu o przejrzystą, przemyślaną strukturę, będzie łatwiejszy do poznania niż program, w którym panuje duży bałagan. Oczywiście nie jest to warunek wystarczający (to, co może być proste i jasne dla projektanta, użytkownikom może wydawać się dziwne i trudne, zwłaszcza jeśli zostanie opisane językiem, jakim posługują się projektanci, a nie użytkownicy), ale zastosowanie się do niego bardzo pomaga.

Jest to jeden z obszarów, w których technika obiektowa jest szczególnie przydatna. Pewne metody, które na pierwszy rzut oka wydają się dotyczyć zagadnień projektowych i implementacyjnych, pozwalają też stworzyć nowego rodzaju interfejsy użytkownika, które ułatwiają obsługę aplikacji. W następnych rozdziałach podam wiele konkretnych przykładów.

Patrz Wilfred J.Hansen, „User Engineering Principles for Interactive Systems", Sprawozdanie z FJCC 39, AFIPS Press, Montvale (NJ), 1971, s. 523 – 532.

Projektanci oprogramowania zajmujący się zagadnieniem łatwości obsługi powinni ostrożnie podchodzić do sloganu najczęściej cytowanego w literaturze fachowej, pochodzącego ze starego artykułu Hansena, który brzmi: „Poznaj użytkownika". Stoi za nim argument, że dobry projektant powinien poświęcić czas na poznanie docelowej społeczności użytkowników programu. Jednak takie podeście nie uwzględnia jednej z charakterystycznych cech programów, które odnoszą rynkowy sukces. Po jakimś czasie są one wykorzystywane przez szerszą grupę użytkowników, niż pierwotnie sądzono. (Dwoma dobrymi przykładami są tu Fortran, który miał być początkowo narzędziem służącym do rozwiązywania problemów, jakie napotkała mała grupa naukowców i inżynierów zajmujących się pisaniem programów na komputer IBM 704, oraz system operacyjny Unix, który pierwotnie miał służyć jedynie do wewnętrznych zastosowań w firmie Bell Laboratories). Każdy system zaprojektowany z myślą o specyficznej grupie użytkowników będzie oparty na założeniach, które po prostu przestaną być adekwatne dla szerszego grona odbiorców.

Dlatego dobrzy projektanci interfejsów użytkownika trzymają się bardziej ostrożnej zasady. Czynią oni tak mało założeń dotyczących przyszłych użytkowników, jak to możliwe. Projektując interaktywny program komputerowy, możesz oczekiwać, że jego użytkownicy będą przedstawicielami rasy ludzkiej i będą potrafili czytać, poruszać myszą, klikać przycisk i pisać (powoli), nic ponadto. Jeżeli aplikacja ma służyć do rozwiązywania problemów z jakiejś konkretnej dziedziny, można też pewnie założyć, że jej użytkownicy będą mieli choć podstawową wiedzę w tym zakresie. Ale nawet takie założenie jest już ryzykowne. Parafrazując radę Hansena, mogę sformułować zasadę projektowania interfejsu użytkownika.

Zasada projektowania interfejsu użytkownika

Nie udawaj, że znasz użytkownika. Nie znasz.

Funkcjonalność

Definicja funkcjonalności

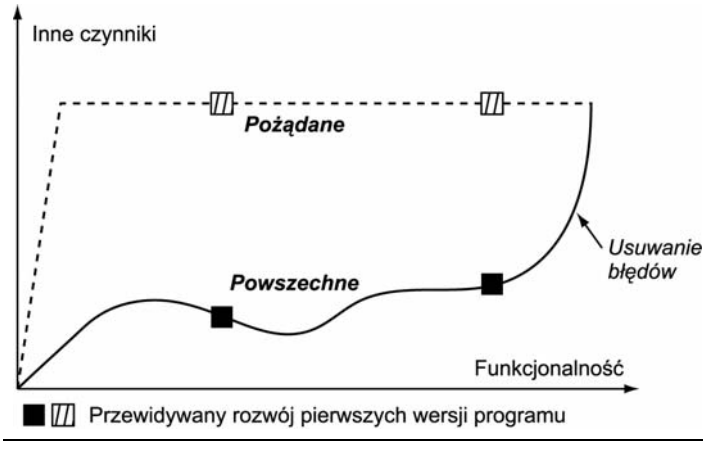
Funkcjonalność to zbiór możliwości oferowanych przez system.

Jednym z największych problemów, przed jakimi stają szefowie projektów, jest określenie, jak duża funkcjonalność jest już wystarczająca. W przemyśle programistycznym stale odczuwa się nacisk na dodawanie do programów kolejnych możliwości, co powszechnie nazywane jest *pogonią za wodotryskami*. Jej konsekwencje są złe już dla wewnętrznych projektów, w których naciski pochodzą ze strony pracowników tej samej firmy, a w przypadku produktów komercyjnych jeszcze gorsze. Niestety większość pism komputerowych przy sporządzaniu recenzji kilku konkurencyjnych produktów zazwyczaj skupia się jedynie na przedstawieniu tabeli, w której porównuje się ich możliwości.

Pogoń za możliwościami to właściwie dwa oddzielne problemy, z których jeden jest trudniejszy do rozwiązania. Prostszy problem polega na tym, że, dodając nowe funkcje, możemy utracić spójność, co odbije się na łatwości obsługi. Użytkownicy często narzekają, że wszystkie te „wodotryski" dodawane do kolejnych wersji znanych programów czynią je jedynie niesamowicie złożonymi. Tego rodzaju uwagi trzeba jednak traktować z przymrużeniem oka, ponieważ nowe funkcje nie biorą się z próżni. Zwykle proszą o nie użytkownicy, tyle że niekoniecznie ci sami. To, co mnie wydaje się świetnym ułatwieniem, dla Ciebie może być tylko nieprzydatnym gadżetem.

Jedynym rozwiązaniem jest tu stała praca nad zachowaniem spójności całego programu i podejmowanie prób wciśnięcia wszystkich funkcji w już wyznaczone ramy. Każdy dobry program opiera się na kilku nowatorskich pomysłach. Jeśli nawet ma on wiele funkcji, powinny one wynikać z podstawowych założeń. W konstrukcji programu musi być widoczny jakiś konkretny plan, w którym każdy element ma swoje miejsce.

Trudniejszy problem polega na tym, iż można tak bardzo zaangażować się w dodawanie nowych funkcji, że zapomni się o innych czynnikach. Jest to powszechnie popełniany błąd, zobrazowany przez Rogera Osmonda na wykresie w postaci dwóch możliwych ścieżek prowadzących do ukończenia projektu (patrz rysunek 1.4).



RYSUNEK 1.4. Krzywe Osmonda, za [Osmond 1995]

W życiu sytuacja wygląda najczęściej tak, jak pokazuje to dolna (czarna) krzywa. W szaleńczym wyścigu za dodawaniem nowych funkcji programiści zapominają o ogólnej jakości programu. Ostatnia faza projektu, w której wszystko powinno być już doprowadzone do porządku, potrafi być długa i stresująca. Jeżeli pod wpływem presji użytkowników lub konkurencji jesteś zmuszony do wypuszczenia programu we wczesnej fazie produkcji, na jednym z etapów oznaczonych na wykresie czarnymi kwadratami, pamiętaj, że może to zaszkodzić Twojej reputacji.

Osmond sugeruje (krzywa linią przerywaną), abyś — korzystając z możliwości, jakie daje programowanie zorientowane obiektowo — utrzymywał ogólną jakość programu (poza funkcjonalnością) przez cały czas na takim samym poziomie. Chodzi więc o to, abyś nigdy nie oszczędzał na niezawodności, rozszerzalności itp. Nie wolno Ci dodawać nowych funkcji, dopóki te, które już masz, będą działać, tak jak trzeba.

Takie podejście jest trudniejsze do stosowania na co dzień ze względu na wspomniane naciski, ale prowadzi do wydajniejszego procesu tworzenia aplikacji, a często także do uzyskania w ostatecznym rozrachunku lepszego produktu. Jeśli nawet końcowe rezultaty są w obu przypadkach takie same, tak jak na przytoczonym wykresie, to postępując zgodnie z podanymi zasadami, powinniśmy uzyskać je szybciej (wykres nie uwzględnia czasu). Gdy trzymamy się sugerowanej ścieżki rozwoju projektu, ewentualne wypuszczenie wczesnej wersji programu (na jednym z etapów oznaczonych ukośnie kreskowanymi kwadratami) jest znacznie łatwiejsze. Trzeba jedynie odpowiedzieć sobie na pytanie, czy zakres funkcjonalności, który został już zaimplementowany w programie, jest wystarczająco szeroki, aby przyciągnąć do niego klientów i nie zrazić niektórych z nich. Natomiast nie musimy już zastanawiać się „czy to jest wystarczająco dobre".

Każdy czytelnik, który miał już okazję kierować jakimś projektem programistycznym, przyzna, że dużo łatwiej dawać takie rady, niż je stosować. Niemniej jednak w przypadku każdego projektu powinno się dążyć do ideału reprezentowanego przez wyższą krzywą Osmonda. Doskonale współgra on z *modelem klastrowym* wprowadzonym w jednym z kolejnych rozdziałów i będącym jednym ze schematów prowadzenia projektu programistycznego.

Adekwatność czasowa

Definicja adekwatności czasowej

Adekwatność czasowa to gotowość programu komputerowego do wypuszczenia na rynek wtedy, gdy użytkownicy go potrzebują, lub jeszcze zanim pojawi się taka potrzeba.

W przemyśle programistycznym jednym z najbardziej stresujących czynników jest adekwatność czasowa. Doskonały produkt, który pojawi się na rynku za późno, może okazać się całkowitą klapą. Oczywiście tak samo jest też w innych gałęziach przemysłu, ale niewiele z nich podlega tak szybkim zmianom.

„NT 4.0 BeatsClock", Computer-World, nr 30, 22 czerwca 1996.

Adekwatność czasowa jest czymś rzadko spotykanym, nawet w przypadku dużych projektów programistycznych. Gdy Microsoft ogłosił, że najnowsza wersja jego sztandarowego systemu operacyjnego, tworzonego przez wiele lat, zostanie wypuszczona na rynek o miesiąc wcześniej, niż zakładano, uznano to za tak sensacyjną wiadomość, że zamieszczono ją na okładce magazynu *ComputerWorld* (wraz z komentarzem na temat długich opóźnień, które zdarzały się tej firmie w przeszłości).

Inne czynniki

Poza już omówionymi czynnikami decydującymi o jakości, istnieją też inne, mające wpływ na użytkowników, decydentów zamawiających systemy komputerowe i programistów. Oto niektóre z nich.

- **Weryfikowalność**, czyli łatwość przygotowywania procedur pozwalających określić, czy produkt znajduje się na akceptowalnym poziomie. W szczególności chodzi tu o dane testowe i procedury pozwalające na wykrywanie niepowodzeń i odnajdywanie tych błędów popełnionych na etapie walidacji danych i w fazie operacyjnej, które są przyczyną niepowodzeń.
- **Integralność**, czyli zdolność oprogramowania do chronienia swoich składników (programów, danych) przed nieautoryzowanym dostępem i modyfikacjami.
- **Naprawialność**, czyli udostępnianie możliwości naprawiania defektów.
- **Ekonomiczność**, będąca uzupełnieniem adekwatności czasowej, czyli zdolność systemu do zmieszczenia się w założonym budżecie.

Dokumentacja

Analizując listę czynników decydujących o jakości, możesz dojść do wniosku, że jednym z wymagań powinna być dobra dokumentacja. Jednak nie jest to odrębny składnik. Potrzeba takiej dokumentacji wynika z pozostałych czynników, które już omówiłem. Możemy wyróżnić trzy rodzaje dokumentacji.

- Dokumentacja *zewnętrzna*, dzięki której użytkownicy mogą zrozumieć możliwości systemu i wygodnie z nich korzystać. Jest to konsekwencja wymagania zdefiniowanego jako łatwość obsługi.
- Dokumentacja *wewnętrzna*, która pozwala zrozumieć strukturę i implementację systemu programistom. Jest to konsekwencja wymagania zdefiniowanego jako rozszerzalność.
- Dokumentacja *interfejsów poszczególnych modułów*, dzięki której programiści mogą zrozumieć działanie funkcji dostarczanych przez dany moduł bez potrzeby zagłębiania się w jego implementację. Jest to konsekwencja wymagania zdefiniowanego jako przystosowanie do wielokrotnego użycia. Wynika ona również z rozszerzalności, ponieważ dokumentacja interfejsów modułów pozwala określić, czy dana zmiana będzie miała wpływ na konkretny moduł.

Dlatego, zamiast traktować dokumentację jako osobny produkt, lepiej pisać maksymalnie samodokumentujący się kod. Zasada ta odnosi się do wszystkich trzech rodzajów dokumentacji.

- Uwzględniając w programie podsystem pomocy ekranowej, a także trzymając się jasnych i spójnych konwencji przy tworzeniu interfejsu użytkownika, ułatwiamy zadanie autorom podręczników użytkownika i innych materiałów dydaktycznych.

- Zaimplementowanie programu w dobrym języku programowania zlikwiduje w dużej mierze konieczność tworzenia wewnętrznej dokumentacji, jeśli tylko język ten będzie sprzyjał organizacji i przejrzystości kodu. Są to jedne z najważniejszych czynników wpływających na wygląd notacji obiektowej, którą będę wprowadzał w tej książce.
- Notacja ta będzie obsługiwała ukrywanie informacji i inne techniki (takie jak asercje) pozwalające oddzielić interfejsy modułów od ich implementacji. Pozwoli to zastosować narzędzia, które automatycznie wygenerują dokumentację interfejsów modułów na podstawie ich kodów źródłowych. Tym zagadnieniem również zajmiemy się dokładnie w dalszych rozdziałach.

Wszystkie te techniki ograniczają znaczenie tradycyjnej dokumentacji, ale oczywiście nie możemy od nich oczekiwać, aby całkowicie ją zastąpiły.

Kompromisy

W tym przeglądzie zewnętrznych czynników składających się na jakość oprogramowania pojawiły się już wymagania, które kłócą się z innymi.

Czy jest możliwe osiągnięcie *integralności* bez wprowadzania żadnych zabezpieczeń? Z kolei wprowadzenie ich musi się negatywnie odbić na *łatwości obsługi*. *Ekonomiczność* często wydaje się stać w sprzeczności z *funkcjonalnością*. Uzyskanie optymalnej *wydajności* wymagałoby perfekcyjnego wykorzystania możliwości oferowanych przez dane środowisko sprzętowe i programowe, co kłóci się z *przenośnością*. Z jednej strony, mówi się nam, żebyśmy trzymali się ściśle specyfikacji, a z drugiej, popycha się nas w kierunku tworzenia ogólnych rozwiązań, które nadawałyby się do *wielokrotnego użycia*. Naciski na adekwatność czasową mogą skłaniać ku środowiskom szybkiego tworzenia aplikacji (RAD), przy których mamy mocno ograniczoną *rozszerzalność*.

Choć w wielu przypadkach daje się znaleźć rozwiązanie pozwalające pogodzić ze sobą sprzeczne wymagania, czasami trzeba poświęcić pewne czynniki na rzecz drugich. Niestety zbyt często programiści robią to niejawnie, nie zastanowiwszy się wcześniej nad wszystkimi dostępnymi możliwościami i ich konsekwencjami. W rezultacie najczęściej skupiają się na wydajności. Natomiast prawdziwy inżynier oprogramowania powinien zadać sobie trud jawnego sformułowania kryteriów, którymi będzie posługiwał się przy wyborze, i podjąć w pełni świadome decyzje.

Choć między czynnikami decydującymi o jakości trzeba często dokonywać wyborów, nie wolno nigdy zapomnieć, że najważniejszym z nich jest poprawność. Nic nie usprawiedliwia zaniedbania poprawności na rzecz innych czynników, na przykład takich jak wydajność. Jeżeli oprogramowanie nie spełnia swojej roli, to reszta nie ma znaczenia.

Kluczowe zagadnienia

Wszystkie opisywane dotychczas czynniki są ważne. Ale w obecnym stanie przemysłu programistycznego cztery z nich wydają się mieć szczególne znaczenie. Są to:

- *Poprawność* i *odporność*. W dalszym ciągu tworzenie oprogramowania pozbawionego defektów jest zbyt trudne, podobnie jak usuwanie błędów, gdy już uda się je zlokalizować. Wszystkie techniki służące zwiększaniu poprawności i odporności opierają się na tych samych pomysłach — zastosowaniu bardziej usystematyzowanego podejścia do tworzenia oprogramowania, stosowaniu bardziej formalnych specyfikacji, użyciu wbudowanych mechanizmów kontrolnych już na etapie tworzenia oprogramowania (a nie bazowanie jedynie na testowaniu i usuwaniu błędów już po zakończeniu projektu), wykorzystanie lepszych mechanizmów językowych, takich jak statyczna kontrola typów, asercje, automatyczne zarządzanie pamięcią czy zdyscyplinowana obsługa wyjątków, które pozwalają programistom spełniać wymagania dotyczące poprawności i odporności, a narzędziom programistycznym wykrywać wszelkie niespójności, zanim doprowadzą one do powstania defektów. Ze względu na tak bliskie powiązanie poprawności z odpornością, wygodniej stosować bardziej ogólny termin — obejmujący swym znaczeniem oba te czynniki — **niezawodność**.
- *Rozszerzalność* i *przystosowanie do wielokrotnego użycia*. Oprogramowanie powinno dawać się łatwiej modyfikować. Tworzone przez nas komponenty programistyczne powinny stwarzać możliwość zastosowania w większej liczbie sytuacji. Powinno ich też być więcej. Również w przypadku tych dwóch czynników źródłem usprawnień są te same pomysły. Ogólnie można by rzec, że pomocne są wszystkie techniki, które przyczyniają się do tworzenia bardziej zdecentralizowanych architektur, w których komponenty mają budowę zamkniętą i komunikują się ze sobą jedynie za pomocą ograniczonych i ściśle zdefiniowanych kanałów. Rozszerzalność i przystosowanie do wielokrotnego użycia będziemy określali wspólnym terminem **modułowości**.

Technika obiektowa, z którą zapoznasz się dokładnie w kolejnych rozdziałach, pozwala wyraźnie ulepszyć te cztery czynniki decydujące o jakości, co czyni ją tak atrakcyjną. Przyczynia się ona również w znacznym stopniu do spełnienia pozostałych wymagań, szczególnie takich jak:

- *Zgodność*. Technika obiektowa sprzyja stosowaniu jednolitego stylu projektowania i tworzeniu standaryzowanych modułów i interfejsów, co z kolei pomaga tworzyć systemy, które mają ze sobą współpracować.
- *Przenośność*. Kładąc nacisk na abstrakcyjność i ukrywanie informacji, technika obiektowa zachęca projektantów do rozróżniania specyfikacji i implementacji, co ułatwia przenoszenie aplikacji na różne platformy. Techniki, takie jak polimorfizm i wiązanie dynamiczne, pozwalają nawet tworzyć systemy, które automatycznie dostosowują się do zastanych komponentów platformy sprzętowo-programowej, na przykład do różnych systemów zarządzania oknami czy bazami danych.
- *Łatwość obsługi*. Wpływ, jaki narzędzia obiektowe wywarły na współczesne interaktywne programy komputerowe, a w szczególności na ich interfejsy użytkownika, jest powszechnie znany. Czasami przypisuje się temu nawet większe znaczenie, niż by wypadało (twórcy reklam nie są jedynymi osobami, które każdy program, w jakim są ikony i okna, obsługiwany za pomocą myszy, nazywają obiektowym).
- *Wydajność*. Jak już wspomniałem, mimo iż z zakupem szybkiego komputera i ze stosowaniem technik obiektowych wiąże się pewien koszt, możliwość wykorzystywania w procesie tworzenia oprogramowania profesjonalnych komponentów wielokrotnego użytku pozwala znacznie zyskać na wydajności.

- *Adekwatność czasowa*, *ekonomiczność* i *funkcjonalność*. Techniki obiektowe pozwalają tym, którzy opanują je do perfekcji, tworzyć programy szybciej i przy mniejszych kosztach. Ułatwiają one dodawanie nowych funkcji, a czasem wręcz sugerują, jakie możliwości przydałoby się jeszcze wprowadzić.

Mimo oczywistego postępu, jaki niosą ze sobą techniki obiektowe, powinniśmy pamiętać, że nie stanowią one panaceum i że w dalszym ciągu musimy rozwiązywać wiele klasycznych problemów inżynierii programowania. To, że coś nam w tym pomaga, nie oznacza jeszcze, że rozwiąże to problem za nas.

1.3. KONSERWACJA OPROGRAMOWANIA

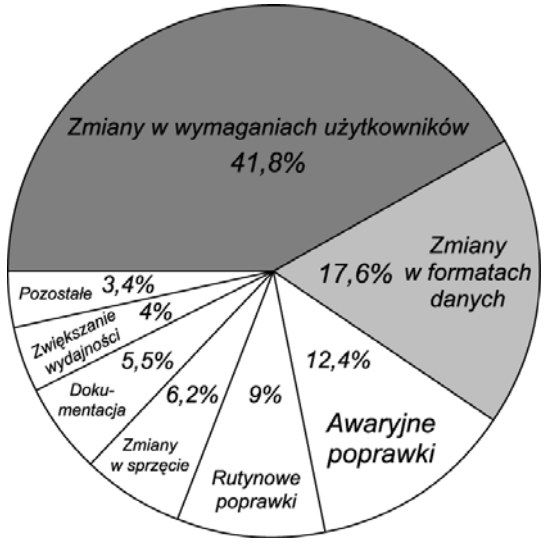
Być może zauważyłeś, że nie wymieniłem dotąd często wspominanego czynnika decydującego o jakości, jakim jest łatwość konserwacji. Aby zrozumieć, dlaczego tak postąpiłem, musisz przyjrzeć się bliżej temu, czym właściwie jest konserwacja oprogramowania.

Terminem konserwacja określamy wszystko, co robi się już po przekazaniu programu odbiorcom. Wszystkie dyskusje na temat metodyki tworzenia oprogramowania z reguły skupiają się jedynie na programowaniu. Podobne podejście obserwuje się na kursach programowania dla początkujących. Tymczasem powszechnie wiadomo, że 70% kosztów tworzenia oprogramowania pochłania właśnie jego konserwacja. Żadne omówienie zagadnienia jakości oprogramowania nie jest wystarczające, jeżeli pomija się w nim ten aspekt.

Zatem co rozumiemy przez „konserwację" w odniesieniu do oprogramowania? Programy komputerowe nie zużywają się od ciągłego korzystania, a tym samym nie trzeba ich „konserwować", tak jak samochodów czy telewizorów. Można by więc rzec, że jest to dość nietrafione określenie. Tym bardziej, że odnosi się ono zarówno do bardziej, jak i do mniej chwalebnych zajęć. Do pierwszej grupy zaliczyć można wszelkie modyfikacje. Wiadomo, że specyfikacje systemów komputerowych zmieniają się po to, aby odzwierciedlać zmiany zachodzące w świecie zewnętrznym. Pociąga to za sobą konieczność modyfikowania samych programów. Natomiast do drugiej grupy zaliczamy czynności służące do lokalizowania i usuwania błędów, których na tym etapie projektu w ogóle nie powinno już być.

Wykres przedstawiony na rysunku 1.5, pochodzący z pamiętnego opracowania Lientza i Swansona, pokazuje już bardziej konkretnie, jakie zagadnienia mieszczą się pod pojęciem konserwacji. Autorzy przepytali 487 zespołów programistycznych rozwijających aplikacje różnych typów. Mimo iż badanie odbyło się już jakiś czas temu, wyniki najnowszych ankiet potwierdzają ich odkrycia. Wykres pokazuje, jaka część kosztów konserwacji oprogramowania przypada na poszczególne zagadnienia, wyodrębnione przez autorów.

Ponad dwie piąte kosztów przeznaczane jest na rozbudowywanie i modyfikowanie aplikacji na życzenie użytkowników. Są to czynności, których praktycznie nie da się uniknąć. Natomiast bez odpowiedzi pozostaje pytanie, ilu spośród tych wysiłków można by uniknąć, gdyby oprogramowanie było od początku projektowane z myślą o rozszerzalności. Mamy pełne podstawy sądzić, że technika obiektowa pozwoli poprawić tę sytuację.



RYSUNEK 1.5.
Koszty konserwacji oprogramowania.
Źródło: [Lientz 1980]

Szczególnie interesująca jest druga pozycja — zmiany w formatach danych. Gdy zmienia się fizyczna struktura plików i innych elementów przechowujących dane, trzeba zaadaptować do tych zmian istniejące aplikacje. Gdy na przykład poczta w Stanach Zjednoczonych przydzieliła parę lat temu dużym firmom kody pocztowe w postaci „5+4" (złożone z dziewięciu znaków zamiast pięciu), trzeba było przepisać od nowa wiele programów, które „wiedziały", że kody pocztowe składają się dokładnie z pięciu znaków. Prasa oszacowała, że kosztowało to kilkaset milionów dolarów.

Inny przykład znajdziesz w podrozdziale: „Jak długi jest środkowy inicjał?", na stronie 167.

> Zapewne wielu czytelników otrzymało swego czasu piękne zaproszenia na cykl konferencji (nie jedną czy dwie, ale cały cykl, prowadzony w wielu miastach) poświęconych „problemowi roku 2000". Mówiono na nich, w jaki sposób unowocześnić wiele programów służących do przechowywania bardzo istotnych danych, których autorzy nawet przez chwilę nie pomyśleli, że po upływie XX wieku również trzeba będzie jakoś wprowadzać daty. W porównaniu z tym problem kodów pocztowych wydaje się błahostką. Jorge Luis Borges stwierdził, że skoro dziś mało kto zastanawia się, co stanie się 1 stycznia 3000 roku, to prawdopodobnie możemy uznać, że był to jedyny cykl konferencji w historii ludzkości poświęcony tak niepozornemu tematowi — *jednej dziesiętnej cyfrze*.

Nie chodzi o to, że jakaś część programu zna fizyczną strukturę danych. To jest akurat nieuniknione, ponieważ w pewnym momencie program musi mieć dostęp do danych po to, aby mógł przeprowadzić na nich jakieś operacje. Jednak w tradycyjnych technikach projektowania ta wiedza jest dostępna w zbyt wielu częściach systemu, przez co po zmianie fizycznej struktury danych (a to zawsze kiedyś następuje) trzeba modyfikować wiele fragmentów programu, a wcale nie musi tak być. Innymi słowy, jeżeli kody pocztowe zostaną wydłużone z pięciu do dziewięciu znaków, a daty zaczną wymagać stosowania dodatkowej cyfry, będzie można mieć uzasadnione podejrzenia, że należy zaadaptować do tych zmian programy operujące na tych wielkościach. Natomiast nie do zaakceptowania jest sytuacja, w której informacje na temat dokładnej długości danych są umieszczone w bardzo wielu miejscach w programie, tak że zmiana tej długości pociąga za sobą konieczność modyfikowania niewspółmiernie wielu fragmentów kodu.

Rozwiązaniem tego problemu są abstrakcyjne typy danych, dzięki którym programy mogą odwoływać się do danych za pośrednictwem swych zewnętrznych właściwości, a nie fizycznych implementacji.

Szczegółowe omówienie abstrakcyjnych typów danych znajdziesz w rozdziale 6.

Kolejną ciekawą rzeczą dającą się zaobserwować na wykresie jest niski udział redagowania dokumentacji w całkowitych kosztach konserwacji (5,5%). Pamiętaj, że mówimy tu jedynie o kosztach czynności konserwacyjnych. Ponieważ na ten temat nie mamy bardziej precyzyjnych danych, możemy jedynie spekulować, iż wynika to z tego, że albo dokumentacja tworzona jest na etapie rozwijania kodu, albo nie pisze się jej wcale. W dalszej części książki poznasz styl projektowania, dzięki któremu większa część dokumentacji trafia bezpośrednio do kodu źródłowego. Powiem też o specjalnych narzędziach, które pozwalają ją stamtąd wydobyć.

Kolejne pozycje na liście Lientza i Swansona również są bardzo interesujące, choć nie są, tak jak pozostałe, bezpośrednio związane z zagadnieniami poruszanymi w tej książce. Awaryjne poprawki błędów (opracowywane w pośpiechu po tym, jak któryś z użytkowników zgłosi, że program nie zwraca oczekiwanych wyników lub zachowuje się w nieprawidłowy sposób) kosztują więcej niż rutynowe, zaplanowane poprawki. Wynika to nie tylko z tego, że wymagają one pracy pod dużą presją, ale również z tego, że konieczność ich przygotowania zmienia ustalony harmonogram wypuszczania nowych wersji programu, a w dodatku może prowadzić do wprowadzenia nowych błędów. Ostatnie dwie czynności mają niewielki udział w kosztach.

- Pierwszą z nich jest zwiększanie wydajności. Wykres zdaje się sugerować, że gdy system już działa, szef projektu i programiści nie chcą go ruszać, żeby go nie zepsuć, nawet jeśli potencjalnie mogliby zwiększyć jego wydajność. Gdy przypomnimy sobie zasadę: „najpierw spraw, żeby to działało, a dopiero potem zajmij się przyspieszaniem", dojdziemy do wniosku, że widocznie większość projektów zatrzymuje się na tej pierwszej czynności.
- Podobnie niski udział w kosztach ma również „przenoszenie do nowych środowisk". Można to interpretować w ten sposób (podobnie jak poprzednio nie mamy bardziej szczegółowych danych), że pod względem przenośności programy dzielą się zasadniczo na dwie kategorie, a między nimi właściwie nic nie ma, czyli na programy, które od początku były projektowane z myślą o przenośności, gdzie przeprowadzenie takiej operacji nie jest zbyt kosztowne, oraz programy tak mocno przywiązane do swoich pierwotnych platform, że przeniesienie ich na inne byłoby tak trudne, iż programiści nawet nie próbują tego robić.

1.4. KLUCZOWE POJĘCIA WPROWADZONE W TYM ROZDZIALE

- Zadaniem inżynierii oprogramowania jest opracowywanie sposobów tworzenia wysokich jakościowo programów komputerowych.
- Jakość oprogramowania nie jest pojedynczą wielkością. Należy ją raczej rozpatrywać jako sumę wielu różnych czynników, które w każdym programie występują w nieco innych proporcjach.

- Należy odróżniać zewnętrzne czynniki składające się na jakość, czyli te dostrzegane przez użytkowników i klientów, od czynników wewnętrznych, widocznych jedynie dla projektantów i implementatorów.
- Liczą się jedynie czynniki zewnętrzne, ale do ich uzyskania niezbędne są odpowiednie czynniki wewnętrzne.
- Przedstawiłem listę najważniejszych czynników zewnętrznych decydujących o jakości aplikacji. Te z nich, które najwięcej zyskałyby na wprowadzeniu nowych technik i do których bezpośrednio odnosi się metoda obiektowa, to czynniki związane z bezpieczeństwem (poprawność i odporność), określane wspólnie mianem niezawodności, oraz czynniki wymagające stosowania zdecentralizowanych architektur oprogramowania (przystosowanie do wielokrotnego użycia i rozszerzalność), określane mianem modułowości.
- Konserwacja oprogramowania, na którą przypada większa część kosztów projektu, jest trudna ze względu na fakt, że niełatwo implementuje się zmiany w istniejących programach, oraz z powodu nadmiernego uzależnienia programów od fizycznej struktury danych, na których one operują.

1.5. BIBLIOGRAFIA

Próbę zdefiniowania jakości oprogramowania podjęło już wielu autorów. Spośród pierwszych opracowań na ten temat dwa pozostają wciąż zaskakująco aktualne: artykuł [Hoare 1972] oraz [Boehm 1978] — wyniki pierwszych usystematyzowanych badań na temat tworzenia oprogramowania, prowadzonych przez ludzi z TRW.

Rozróżnienie między czynnikami zewnętrznymi a wewnętrznymi zostało wprowadzone w materiałach przekazanych w 1977 roku lotnictwu Stanów Zjednoczonych przez firmę General Electric [McCall 1977]. To, co my nazywamy czynnikami zewnętrznymi, McCall określa po prostu mianem „czynników", zaś na czynniki wewnętrzne mówi „kryteria". Wiele (choć nie wszystkie) spośród opisanych przeze mnie czynników stanowi bezpośrednie odpowiedniki czynników McCalla. Jeden z jego czynników pominąłem (łatwość konserwacji), ponieważ, jak już wyjaśniłem, pokrywa się on z rozszerzalnością i weryfikowalnością. W swych rozważaniach McCall nie ogranicza się jedynie do czynników zewnętrznych, lecz opisuje też czynniki wewnętrzne („kryteria"), metryki oraz ilościowe techniki gwarantujące uzyskanie satysfakcjonujących czynników wewnętrznych. Jednak wraz z pojawieniem się techniki obiektowej wiele z opisywanych przez niego czynników wewnętrznych i metryk przestało mieć znaczenie ze względu na swe ścisłe powiązanie ze stosowanymi wcześniej praktykami programistycznymi. Przystosowanie tej części pracy McCalla do technik opisywanych w mojej książce jest całkiem interesującym wyzwaniem. Patrz bibliografia i ćwiczenia w rozdziale 3.

Omówienie zależności względnego wzrostu wydajności systemu (po wymianie komputera na szybszy) od rodzaju stosowanego algorytmu wywodzi się z [Aho 1974].

Klasyczną pozycją na temat łatwości obsługi jest [Shneiderman 1987], opierający się na pracy [Shneiderman 1980], poświęconej szerszemu zakresowi zagadnień z dziedziny programowania. Odnośniki do wielu materiałów na ten temat można też znaleźć na stronie internetowej laboratorium Shneidermana, pod adresem *http://www.cs.umd.edu/projects/hcil/*.

Krzywe Osmonda pochodzą z wykładu wygłoszonego przez Rogera Osmonda na konferencji TOOLS USA [Osmond 1995]. Zwróć uwagę, że wykres pokazany w tym rozdziale nie uwzględnia czasu, dzięki czemu możesz lepiej przyjrzeć się zależnościom między funkcjonalnością a innymi czynnikami, wskazywanymi przez dwie alternatywne krzywe. Pamiętaj jednak, że na tym wykresie nie widać, że postępowanie zgodnie z czarną krzywą może doprowadzić do opóźnienia projektu. Oryginalne krzywe Osmonda miały za zadanie ukazywać raczej czas rozwijania aplikacji, a nie zwiększanie jej funkcjonalności.

Wykres prezentujący koszty konserwacji programów pochodzi z wyników badań przeprowadzonych przez Lientza i Swansona, którzy wysłali kwestionariusze do 487 firm [Lientz 1980]. Patrz również [Boehm 1979]. Mimo iż niektóre spośród zbieranych przez nich danych dziś mogą wydawać się nieadekwatne (badanie dotyczyło wsadowych aplikacji na komputery MIS, które składały się średnio z 23 000 wierszy kodu, co jak na dzisiejsze warunki nie jest zbyt dużą liczbą), uzyskane przez nich wyniki nadal wyglądają sensownie. Stowarzyszenie Software Management Association przeprowadza co roku ankietę na temat konserwacji. Wyniki jednej z nich znajdziesz w [Dekleva 1992].

Wyrażenia *programowanie w dużej skali* i *programowanie w małej skali* pochodzą z [DeRemer 1976].

Ogólne omówienie zagadnień z dziedziny inżynierii oprogramowania znajdziesz w książce, którą napisali Ghezzi, Jazayeri i Mandrioli [Ghezzi 1991]. Dwóch spośród tych autorów napisało też książkę o językach programowania, [Ghezzi 1997], stanowiącą uzupełnienie niektórych zagadnień poruszanych w tej książce.