

U mnie działa Język branży IT

WYDANIE II

PAWEŁ BASZURO

Helion 

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Materiały graficzne w książce i na okładce zostały wykorzystane za zgodą Shutterstock.com

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/umnie2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-9003-4

Copyright © Helion S.A. 2022

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Wstęp	7
O autorze	11
Przedmowa	13
Komputery i oprogramowanie	15
Komputer w życiu codziennym	16
Komputer i oprogramowanie w firmie	24
Komputer w wytwarzaniu oprogramowania	28
Działanie oprogramowania	32
Projekty informatyczne	43
Cykl życia i role projektu oprogramowania	46
Zwinne podejście	53
Analiza i wymagania	65
Źródła i podział wymagań	67
Dokumenty wymagań	72
Praca nad nie swoimi wymaganiami	78

Projektowanie	83
Krok po kroku	85
Przykład wymagań kształtujących architekturę	109
Programowanie i programiści	115
Podział programistów	117
Programista programiście nierówny	127
Nieprogramiści tworzący oprogramowanie	128
Testowanie	131
Zgłoszenia błędów	135
Rodzaje testów	138
Środowisko testowe	144
Wdrożenie i utrzymanie	147
Wdrożenie	148
Utrzymanie	150
Przykład skomplikowania utrzymania i wdrożenia w bezpiecznym życiu codziennym	156
Bibliografia	169
Skorowidz	179



Testowanie

Ćmy i motyle to owady bardzo podobne do siebie. Polowanie na motyle kojarzymy z chwytaniem ich w siatkę na wiosennej polanie. Dla kontrastu łapanie ćmy wiążemy z mniej finezyjnym wymachiwaniem klapkiem. Ćmy mają to do siebie, że czasem, wiedzione świetlnym instynktem, podejmują się samobójczych misji, celując w rozgrzane przedmioty. Jednak dzięki temu pewnego dnia 1947 roku programistka (później kontradmirał marynarki wojennej Stanów Zjednoczonych) Grace Hopper odnotowała wadliwe działanie komputera Mark II. Wadliwe działanie było spowodowane przez znaną

w elektromechanicznych podzespołach ómę. Hopper oraz członkowie jej zespołu zaczęli używać terminu „bug” (pluskwa, owad) do opisu niewłaściwego działania komputera. W latach czterdziestych dwudziestego wieku na całym świecie było zaledwie kilka takich urządzeń, a informatyka i programowanie dopiero raczkowały. Podobnie było z językiem opisującym różne zjawiska związane z komputerami, stąd termin „bug” został spopularyzowany do opisu dowolnego, nieoczekiwanego zachowania komputera (niekoniecznie związanego z insektami). Jak wskazują historycy domeny¹, sam termin został po raz pierwszy użyty w 1873 roku przez Thomasa Edisona przy okazji pracy nad urządzeniem do przesyłania telegramów, jednak „bug” na stałe zagościł w żargonie programistycznym do opisu błędów.

Błędy w oprogramowaniu mają różne pochodzenie, począwszy od niedociągnięć programistów, przez nieprzewidziane sytuacje, po błędy wynikające z wymagań. Liczba błędów w oprogramowaniu jest skorelowana z szeroko rozumianą jakością oprogramowania.

¹ *Did You Know? Edison Coined the Term „Bug”, Bugs have plagued technologists for centuries*, Alexander B. Magoun, Paul Israel, IEEE, 23 sierpnia 2013, w języku angielskim dostępny pod adresem <http://theinstitute.ieee.org/tech-history/technology-history/did-you-know-edison-coined-the-term-bug>.

Niedociągnięcia programistów wynikają z braku wiedzy lub doświadczenia konkretnego programisty, jego pośpiechu lub nieuwagi. Takie błędy są zazwyczaj relatywnie łatwe do naprawienia lub wyłapania w trakcie profesjonalnego procesu wytwarzania (faza testów).

Nieprzewidziane sytuacje, takie jak nieoczekiwane działanie sprzętu (np. brak odpowiedzi lub długotrwałe przetwarzanie), nieoczekiwany ciąg zdarzeń użytkownika (np. wybieranie opcji na ekranie w kolejności innej niż zakładana) lub kombinacja obu, są trudniejsze do naprawienia. Zanim programista będzie mógł skutecznie naprawić błąd, musi najpierw zidentyfikować przyczynę i powtórzyć ciąg zdarzeń wiodących do wystąpienia błędu. Dlatego istotne jest dokładne udokumentowanie zgłoszenia błędu lub defektu. Na poziomie technicznym analizę zachowania komputera podczas pracy programu nazywamy debugowaniem (ang. *debugging*, inne polskie tłumaczenie to odpluskwanie). Programiści, pracując nad złożonymi problemami, często prowadzą analizy typu post mortem, czyli analizy po fakcie wystąpienia błąd. Aby skutecznie przeprowadzić tego typu analizę, potrzebne są różne artefakty techniczne, takie jak pliki logów, zrzuty pamięci i pliki symboli.

Pliki logów to ciąg notowań różnych akcji realizowanych przez działający program. Ze względu na poziom skomplikowania współcześnie używanego oprogramowania nie jest możliwe

odnotowanie każdej potencjalnej instrukcji (względy wydajności; czasem także ze względu na dostępną powierzchnię dyskową), stąd domyślne logowanie zdarzeń jest wyłączone. Niektóre aplikacje po wykryciu błędu automatycznie aktywują logowanie, w innych przypadkach użytkownik może zostać poproszony o zmianę konfiguracji programu. Zaawansowany użytkownik może być także poproszony o zrobienie i udostępnienie zrzutu pamięci programu.

Zrzut pamięci (ang. *memory dump*) zawiera stan pracy programu wraz z wczytanymi bibliotekami, stanem używanych urządzeń oraz aktualnie przetwarzanymi danymi użytkownika. Zrobienie zrzutu pamięci działającego programu umożliwia jest przez system operacyjny (powstały plik może mieć duży rozmiar). Za pomocą narzędzi programistycznych można otworzyć plik zrzutu pamięci i przeanalizować ciąg aktualnie przetwarzanych instrukcji oraz zawartości pamięci. Pliki symboli pozwalają programiście połączyć zrzut pamięci z kodem źródłowym programu (wiele instrukcji wykonywanego programu przekłada się na jedną instrukcję w kodzie źródłowym).

Najtrudniejszymi błędami są błędy wynikające z wadliwych, nieprecyzyjnych lub sprzecznych wymagań. Takie błędy wymagają ponownej analizy wymagań, poprawienia ich oraz zaprojektowania, zaimplementowania i przetestowania nowego

rozwiązania. Ze względu na mnogość zadań i czas poświęcony na dostarczenie poprawionej wersji jest to najbardziej kosztowny błąd.

Zgłoszenia błędów

„Okrągłe” liczby w świecie IT, jak 64 czy 1024, mogą wydawać się trudne do zapamiętania dla przeciętnego użytkownika. Są to jednak liczby nieprzypadkowe, to potęgi liczby dwa. Liczba dwa jest związana z dwoma poziomami napięcia — napięcie jest (poziom jeden) lub go nie ma (poziom zero). Ciąg zmian poziomu napięcia tworzy ciąg zer i jedynek. Jedynek na kolejnej pozycji oznacza potęgę dwójki. I tak liczba 1000 w zapisie binarnym oznacza liczbę 8. W pamięci komputera wszystkie dane są reprezentowane właśnie w postaci binarnej (kod binarny). Ze względu na łatwiejsze zarządzanie tymi ciągami liczb bity przyjęło się grupować po osiem i nazywać bajtami. Osiem bitów to 256 możliwości. Profesor uniwersytetu Stanforda Donald Knuth zwykł wystawiać czeki na okrągłą liczbę 2,56 dolara dla każdej osoby (okrągłe 256 centów), która znajdzie błąd lub będzie miała znaczący wkład w jego pracę². Donald

² Wpis w Wikipedii w języku angielskim dostępny pod adresem https://en.wikipedia.org/wiki/Knuth_reward_check.

Knuth być może zapoczątkował program nagradzania za wyszukiwanie błędów w oprogramowaniu (ang. *bug bounty program*). Tego typu program polega na wynagradzaniu każdego, kto znajdzie błąd w oprogramowaniu, w podziękę za zgłoszenie. Zazwyczaj im większa firma, im groźniejsze ryzyko związane z błędem, tym wyższa wartość materialna wynagrodzenia. Firma Facebook, stojąca za największą siecią społecznościową (w momencie pisania tej książki serwis Facebook miał ponad dwa miliardy użytkowników), w podziękę za zgłoszenie błędu związanego z bezpieczeństwem zwykła rozsyłać czarne karty kredytowe³. Zaś firma Microsoft za pomocą programu Windows Insider⁴ zbiera opinie (w tym także zgłoszenia o błędach) na temat kolejnych wersji systemu Windows, zanim nowe wydania staną się publicznie dostępne dla szerokiego grona odbiorców. Tym samym każdy uczestnik programu może wpłynąć na kształt jednego z najpopularniejszych systemów operacyjnych.

³ Facebook hands out White Hat debit cards to hackers, ELINOR MILLS, CNET, 31 grudnia 2011, w języku angielskim dostępny pod adresem <https://www.cnet.com/news/facebook-hands-out-white-hat-debit-cards-to-hackers/>.

⁴ Strona Microsoft Windows Insider Program, dostępna pod adresem: <https://insider.windows.com/pl-pl/>.

Warto zatem wiedzieć, jak napisać dobre zgłoszenie błędu. Bug wiąże się z konkretnym oprogramowaniem, akcją i reakcją systemu. Przygotowując zgłoszenie, należy sprawdzić wersję oprogramowania (lub jego komponentu), które podlega zgłoszeniu. Jeśli oprogramowanie działa w wielu systemach i na wielu platformach, także je warto scharakteryzować (np. jaki system operacyjny, w jakiej wersji, jaka przeglądarka internetowa i w jakiej wersji). W zależności od typu zgłoszenia charakterystyka urządzenia może być istotna (np. nazwa i model telefonu komórkowego). Im więcej tego typu szczegółów umieścimy, tym szybciej programiści będą mogli zbadać problem.

W ramach zgłoszenia należy precyzyjnie przedstawić ciąg kolejnych kroków, które wiodły do odnalezienia błędu. Taki ciąg można spisać w punktach, np. „Wybranie z menu *Plik* opcji *Otwórz*; w nowym oknie zaznaczenie pliku; wybranie z akcji *Anuluj*”. Powinniśmy również napisać, co otrzymaliśmy w tym momencie, np. „oprogramowanie otwiera zaznaczony plik”. Warto także wyartykułować nasze oczekiwania po takiej akcji, np. „oprogramowanie zamknie okno oraz nie otworzy zaznaczonego pliku”. Dobrą praktyką jest sprawdzenie przed zgłoszeniem, czy błąd miał charakter jednostkowy, czy może powtarza się za każdym razem (także po restarcie urządzenia). Inne „znaleziska” warto odnotować w komentarzach do zgłoszenia.

Do zarządzania stanem zgłoszeń często używa się dedykowanych systemów zarządzania zgłoszeniami (ang. *bug tracking system*), np. JIRA lub Redmine.

Rodzaje testów

Ludzkość przez tysiąclecia traktowała upuszczanie krwi jako ważny element terapii różnych chorób. Nietrudno sobie wyobrazić kapłana starożytnej religii, który źródła gorszego stanu zdrowia upatruje w „złej” krwi, więc należy się jej pozbyć. Tak postawiona hipoteza wymagała natychmiastowej weryfikacji. Uzdrawiony chory utwierdzał kapłana w pozytywnym działaniu terapii (test pozytywny). Co ciekawe, chory bez jakiegokolwiek innej kuracji zapewne nie odczuwał poprawy, co także mogło służyć do potwierdzenia tej tezy (test negatywny). Takie obserwacje zapisały upuszczanie krwi jako istotny element terapii medycznej na kolejne stulecia. W XIX wieku zaczęto weryfikować skuteczność tej metody i dziś jest ona stosowana w nielicznych schorzeniach. Przetestowanie tego rozwiązania zajęło jednak kilka tysięcy lat. Oczekiwaniem względem przedstawionej metody była poprawa stanu zdrowia chorego, co udawadniało prawdziwość tej (pozytywnej) hipotezy. Nieprecyzyjna ocena stanu zdrowia przed i po zastosowaniu metody leczniczej pozwalała na zakończenie jej testów z sukcesem.

Powyższy przykład pozwala na wyprowadzenie bardziej ogólnych wskazówek na temat testowania (w tym także oprogramowania). Po pierwsze w testach należy być precyzyjnym w określaniu stanu wejściowego (w testach „aranżujemy” stan wejściowy, ang. *arrange*), podejmowanych kroków (ang. *act*) i stanu oczekiwanego (ang. *assert*). Po drugie należy zweryfikować nie tylko pozytywne scenariusze, ale również negatywne (np. jak powyższa metoda zadziała na zdrowego człowieka?) i przypadki skrajne (np. jak pacjent będący na granicy progu wyleczenia innymi metodami zachowa się pod wpływem wyżej wymienionej metody?). Weryfikując wyniki testu, należy skategoryzować wyniki — co to znaczy, że test zakończył się sukcesem? Kiedy mówimy o porażce?

Sukcesem (ang. *test success* lub *test pass*) jest każdy wynik testu, który wskazuje, że oprogramowanie zachowuje się zgodnie z oczekiwaniami (zarówno testy pozytywne, jak i negatywne mogą, a wręcz powinny kończyć się sukcesem). Porażką (ang. *test fail*) jest każdy test, w wyniku którego otrzymany stan jest inny od oczekiwanego.

Mówiąc o precyzji, należy jasno określić metodę testowania i jej zakres (najlepiej zanim zlecimy zespołowi technicznemu pracę nad oprogramowaniem). Zazwyczaj testujemy to, na co mamy wpływ. Tworzymy plan testowy, który mówi, co i jak testujemy. Często nie testujemy całego wszechświata, opieramy się

na umowach i zaufaniu. Jeżeli kupujemy sprzęt (np. standardowy komputer PC) lub popularny komponent oprogramowania systemowego (np. system operacyjny Windows lub OSX), to przeważnie ufamy a priori, że został skonstruowany zgodnie z zasadami sztuki i przetestowany, więc jest poprawny. Na przykład jeśli operacja dodawania dwóch liczb w aplikacji kalkulatora udostępnionej przez system operacyjny działa poprawnie, to komponent dodawania działa. W naszej aplikacji nie musimy testować samej operacji dodawania udostępnionej przez bibliotekę systemową, lecz jedynie korzystający z niej nasz kod.

Rzeczywiste przypadki często są bardziej złożone i wymagają podejmowania bardziej skomplikowanych decyzji dotyczących przedmiotu testów. Wówczas podejmujemy decyzję o tym, co, jak i na jakim poziomie testujemy. Zdarza się, że część błędów jest akceptowalna (np. dotyczy małego grona użytkowników albo funkcji dodatkowych) i z pełną świadomością podejmuje się decyzję o wdrożeniu oprogramowania z błędami (np. gdy koszt naprawy jest nieuzasadniony).

Wyróżnia się testy jednostkowe, komponentowe, integracyjne oraz systemowe. Kolejność, w jakiej je wymieniłem, ma znaczenie — przede wszystkim jeśli chodzi o ich zakres i liczbę, najwięcej bowiem powinno być testów jednostkowych, a zazwyczaj najmniej jest systemowych. Testy jednostkowe

polegają na testowaniu bliżej niezdefiniowanej „jednostki”. Tutaj pojawia się pytanie, czym jest ta mityczna „jednostka”? Jednostka to najmniejszy moduł w oprogramowaniu, który da się przetestować. W praktyce oznacza to zwykle pojedynczą linijkę kodu i odpowiedzenie na pytanie, czy aby na pewno funkcjonuje ona prawidłowo i czy jest to udowodnione (pokryte) testami. Sprawy nieco komplikują się, kiedy testujemy decyzję. W takim wypadku należy przetestować wszystkie możliwe rozgałęzienia danej decyzji (ang. *branch testing*; np. badając warunek, czy liczba jest podzielna przez dwa, powinniśmy przetestować zarówno przypadki, kiedy testowana liczba jest liczbą parzystą, jak i nieparzystą). Często spotykamy się z sytuacją, kiedy prowadzone są metryki testów, w których gromadzone są informacje na temat stosunku przetestowanych linijek kodu do liczby wszystkich linijek kodu, stosunku przetestowanych rozgałęzień do wszystkich rozgałęzień w kodzie, stosunku przetestowanych plików, bytów do wszystkich plików, bytów w kodzie itd. Takie testy realizowane są przez programistów. Jedną z miar oceny jakości oprogramowania jest procent pokrycia kodu testami.

Przechodząc do testów komponentu, zazwyczaj testujemy zespół lub ciąg linijek kodu, które składają się na większy, samodzielny obszar realizujący określoną funkcjonalność. Testy komponentu zazwyczaj nie mają aż tak dużej granulacji jak

testy jednostkowe, pokrywają przepływ danych i sterowania przez komponent, od wejścia do wyjścia danych. W praktyce oznacza to testowanie komponentu w odizolowaniu od pozostałych (np. testowanie komponentu generowania raportu w odizolowaniu od komponentu wprowadzania danych). Istnieją przypadki, kiedy testy komponentu nie mogą być przeprowadzone samodzielnie przez użytkownika i wymagana jest pomoc zespołu programistów. Takie wsparcie może być potrzebne w budowaniu zaślepek dla komponentów wchodzących w interakcję z testowanym komponentem (ang. *stub*) albo symulowaniu zachowania tychże komponentów (ang. *mock*). Innym przypadkiem testów, które także wymagają nakładu prac zespołu technicznego, są testy integracyjne.

Testy integracyjne mają na celu sprawdzenie, czy komponenty poprawnie współpracują ze sobą. Często testuje się komponenty parami. Podczas testów integracyjnych sprawdza się, czy dane z jednego komponentu trafiają do drugiego w poprawnej i nieznieskształconej postaci oraz w dobrej kolejności (np. czy moduł generowania raportu przekaże poprawny raport do modułu wydruku raportu).

Testy systemowe z kolei polegają na testowaniu systemu całościowo. Testy systemowe mogą być przekrojowe i przechodzić przez wszystkie komponenty składające się na system (np. definiujemy dwie pozycje, jakie mają być wprowadzone

do systemu; wprowadzamy je do systemu; wybieramy opcję generowania raportu dla nich; następnie wybieramy opcję wydruku wygenerowanego raportu i weryfikujemy, czy wydruk zawiera wprowadzone wcześniej pozycje).

Kiedy podczas testów realizowane są typowe scenariusze użycia oprogramowania, naśladujące zachowanie realnego użytkownika, to wtedy mówimy o testach akceptacyjnych (ang. *user acceptance test*, UAT).

Oprogramowanie możemy testować manualnie albo ułatwić sobie pracę i testować automatycznie. Testy manualne (innymi słowy ręczne) wymagają interwencji użytkownika (potocznie „przeklikania się” przez system) i zazwyczaj są czasochłonne. Testy manualne pozwalają na wygenerowanie przypadkowych zdarzeń w systemie, które mogą prowadzić do znalezienia nieoczywistych błędów. Takie testowanie nazywamy testowaniem eksploracyjnym. Częste testowanie systemu wymaga dużego nakładu czasu i środków na przetestowanie całości systemu. Automatyzacja znanych i typowych testów przyczyni się do minimalizacji kosztów. W takich przypadkach to komputer przetestuje oprogramowanie (podejmując takie same akcje jak człowiek siedzący przed komputerem) oraz zdecyduje, czy wynik zachowania oprogramowania jest zgodny z oczekiwanym (wprowadzonym wcześniej przez człowieka do testu automatycznego).

Zbiór wszystkich testów, które są powtarzane przy każdej sesji testowania (np. z okazji wydania nowej wersji), nazywa się testami regresyjnymi. Na testy regresyjne mogą składać się zarówno testy manualne, jak i testy automatyczne. Celem tych testów jest zweryfikowanie, czy podstawowe funkcje oprogramowania działają w niezminionej i oczekiwanej formie. Wiele zespołów technicznych realizuje testy regresyjne przy każdej większej zmianie dowolnego z elementów oprogramowania (np. zmian w kodzie źródłowym, zmian w zbiorze używanych bibliotek lub znaczących zmian w konfiguracji komponentu).

Bez względu na to, jakie rodzaje testów zostały przeprowadzone, należy każdorazowo zgromadzić artefakty (dokumenty, raporty, zrzuty ekranu i wideo) potwierdzające fakt przeprowadzenia testów i ich rezultaty. Takie artefakty nazywa się ewidencją testów (ang. *test evidence*).

Środowisko testowe

Woda wrze w temperaturze stu, a zamarza poniżej zera stopni Celsjusza. Na podstawie tych faktów można byłoby przetestować, czy termometr pokazuje właściwą temperaturę. Jednak kiedy z tym samym termometrem udałoby nam się wspiąć na wysoką górę (np. Mount Everest), wtedy wskazałby znacząco

inną temperaturę dla tych samych zjawisk. Zmiana środowiska w tym wypadku powoduje inny punkt odniesienia (względem przemian termodynamicznych) i musi to być uwzględnione w testach. W idealnym przypadku test powinien odbywać się w takim samym środowisku (a w opisanym przypadku pod takim samym ciśnieniem), aby uzyskać takie same wyniki. Nie inaczej jest z testami oprogramowania. Środowisko, w którym odbywają się testy, ma równie istotne znaczenie.

Każdy z wyżej wymienionych rodzajów testów realizowany jest (w najbardziej optymistycznym scenariuszu) w osobnych środowiskach testowych. Testy jednostkowe zazwyczaj wykonywane są w środowisku programistycznym (ang. *development environment*, w skrócie *dev environment*), testy komponentów w środowisku zapewnienia jakości (ang. *quality assurance environment*, *QA environment*), testy integracyjne w środowisku integracyjnym (ang. *integration environment*, *INT environment*), zaś testy całego systemu z perspektywy użytkownika odbywają się w środowisku akceptacyjnym (ang. *User Acceptance Test environment*, *UAT environment*). Środowisko nietestowe i z realnymi danymi nazywamy produkcją (ang. *Production environment*, *PROD environment*). Podczas wytwarzania oprogramowania czasem używa się dodatkowych środowisk albo całkowicie innej terminologii nazewniczej. Istnieją także przypadki ograniczenia liczby środowisk testowych (np. ze względu

na koszty zarządzania lub zakupu licencji). W ostatnich latach istnieje trend automatyzacji procesu tworzenia, zarządzania i konfigurowania różnych środowisk, począwszy od automatyzacji opartej na prostych skryptach systemowych, po rozwiązania oparte na wirtualnych maszynach (np. Docker i zarządzanie konfiguracją za pomocą narzędzia o nazwie Kubernetes).

Skorowidz

A

abstrakcja, 33
Acceptance Criteria, 77
administrator, 52, 156
adres
 internetowy, 157
 IP, 158
Agile Manifesto,
 Patrz: programowanie
 zwinne manifest
algorytm, 88
 iteracyjny, 88
 rekurencyjny, 88
Amazon Web Services, 102
analiza
 biznesowa, 47
 systemowa, 47
 techniczna, 47
Apache
 Cordova, 121
API, 100
aplikacja
 architektura, 101, 117,
 118
 internetowa
 bogata, *Patrz:* RIA
 progresywna,
 Patrz: PWA
 mobilna, 119
 multimedialna, 121
 projektowanie, 110
 tworzenie
 narzędzia, 120, 121,
 123

 platforma, 119, 120,
 121, 123
application programming
 interface, *Patrz:* API
autoryzacja, 123

B

backend, *Patrz:* warstwa
 dostępu do danych
bajt, 135
batch, *Patrz:* przetwarzanie
 wsadowe
batch processing, *Patrz:*
 dane przetwarzanie
 wsadowe
baza danych, 124, 125, 126
BI, *Patrz:* Business
 Intelligence
biblioteka, 101
 Cocoa, 121
 Log4J, 167
 OpenSSL, 166
big data, 109
blackhat, 164
blockchain, 111
błąd, 132, 133, 134, 163
 out of memory, 19
 wyszukiwanie, 136
 zgłoszenie, 136, 137,
 138
BPM, *Patrz:* system
 zarządzania procesami
 biznesowymi
BPMN, 90, 100, 101

bramka, 90
BRD, 73
bug, 132
Business Intelligence, 128
Business process
 management,
 Patrz: system
 zarządzania procesami
 biznesowymi
Business Process Model
 and Notation,
 Patrz: BPMN
Business Requirements
 Document, 73

C

cache, *Patrz:* pamięć
 podręczna
CAT, 28
central processing unit,
 Patrz: procesor
chmura obliczeniowa, 102
cloud computing, 102
CPU, *Patrz:* procesor
Creative Commons, 113
CSS, 118, 160
CVE, 165
 CVE-2002-0721, 165
 CVE-2002-1145, 165
 CVE-2014-0160, 166
 CVE-2021-44228, 167
czynniki wewnętrzne, 69
czynniki zewnętrzne, 69

D

Dalvik, 120
 dane
 analiza, 126
 narzędzia, 128
 atrybuty, 85
 baza, *Patrz:* baza danych
 big data, 109
 eksploracja, 128
 ekstrakcja, 126
 format, 105, 106
 hurtowania, 126
 model, 75, 85, 89, 94
 modelowanie, 128
 przetwarzanie
 wsadowe, 110
 rekordy, 85
 data feed, 110
 data mining, 128
 Data Requirements,
 Patrz: wymagania danych
 data warehouse, *Patrz:*
 dane hurtowania
 debugging, *Patrz:*
 debugowanie
 debugowanie, 133, 134,
 136
 deployment, *Patrz:*
 oprogramowanie
 wdrożenie
 development environment,
 145
 diagram, 92
 dług techniczny, 57, 58
 dokumentacja, 56
 Domain Name System,
 Patrz: protokół DNS
 domena, 159
 dziedziczenie, 95

E

Edison Thomas, 132
 edytor kodu
 źródłowego, 30
 ekspert dziedzinowy, 47
 Electron, 121
 enterprise resource
 planning, 27
 enumeracja, 98
 ERP, 27
 exploit, 164
 eXtensive Markup
 Language, *Patrz:* XML

F

Facebook, 136
 Faynman Richard, 17
 feedback, 67
 frontend, *Patrz:* warstwa
 prezentacji danych
 FSD, 74
 Functional Specification
 Document, *Patrz:* FSD

G

garbage collector, 36
 Google TensorFlow, 129
 Google Cloud, 102

H

haker, 164
 Hardware Description
 Language, 34
 Heartbleed, 166
 Hopper Grace, 131
 hot fix, 149
 HTML, 118, 160
 Hypertext Markup
 Language, *Patrz:* HTML

HyperText Transport
 Protocol, *Patrz:* protokół
 HTTP
 HyperText Transport
 Protocol Secure, *Patrz:*
 protokół HTTPS

I

i18n, 86
 IDE, 30
 INT environment, 145
 Integrated Development
 Environment, 30
 integration environment,
 145
 interesariusz, 47, 52, 57,
 58, 62, 73, 84
 interfejs, 94
 programistyczny, 100
 użytkownika, 47, 121
 dokumentacja, 75
 Internationalization, 86
 Internet of Things, 111
 Internet Rzeczy, 111
 interoperability, 109
 interoperacyjność, 109
 inżynier
 jakości, 49
 wsparcia, 50, 153, 156
 wymagań, 47
 IoT, 111

J

JavaScript Object Notation,
Patrz: JSON
 język
 assembler, 34
 C, 35, 40
 C#, 39, 120, 123, 127
 C++, 35, 120
 CoffeeScript, 40
 HTML, 118, 160

Java, 39, 40, 92, 120,
123, 127
JavaScript, 39, 40, 105,
119, 120, 123, 127, 160
Kotlin, 120
Objective-C, 120, 121
PERL, 123
PHP, 123
Python, 123, 128
R, 40, 128
Ruby, 123
Scala, 123
skryptowy, 123
Swift, 120
TypeScript, 40
UML, 91, 92, 101
VHDL, 34
Visual Basic, 120, 123
wyższego poziomu, 35
zapytań baz danych, 125
JSON, 105
JVM, 39, 123

K

Kanban, 58, 63, 64
Karta płatnicza, 21
Keras, 129
klasa, 92, 94
abstrakcyjna, 95
klasyfikacja MoSCoW, 74
Knuth Donald, 135
kod
binarny, 34, 135
błędu 200, 158
błędu 404, 158
błędu 500, 158
poprawki na gorąco, 149
repozytorium, 31
spaghetti, 89
źródłowy, 30, 134
Apache Tomcat, 38
DOS, 36
edytor, 30

GGPlot2, 39
JavaScript, 41
kompilacja, 30
NodeJS, 37
kodowanie, 48
kompilator, 34
komponent, 104
aktywny, 98
kontrakt, 100
pasywny, 98
techniczny, 100
test, 141
komputer
zarządzanie, 16
zasoby, 16
komunikator, 25
konsola do gier, 23
kontrakt, 104
kryptologia, 166
kwerenda, 160

L

l10n, *Patrz:* lokalizacja
LDAP, 167
lean, 78
legacy software, 57
library, *Patrz:* biblioteka
Little Jason, 78
localhost, 158
Localization, 86
Log4Shell, 167
lokalizacja, 86
loopback, 158

M

manifest programowania
zwinnego, *Patrz:*
programowanie zwinne
manifest
maszyna wirtualna, 36, 40,
146
Java, *Patrz:* JVM

memory dump, 134
metryka testów, 141
Microsoft, 136
Microsoft Azure, 102
Microsoft SQL Server, 165
Minimal Viable Product, 67
MITRE, 165
mock-up, 76
model OSI/ISO, 159
MoSCoW, 74
MVP, 67

N

NFC, 21, 22
notacja procesów
biznesowych, *Patrz:*
BPMN

O

odpluskwanie,
Patrz: debugowanie
OLAP, *Patrz:* dane
hurtowania
OLTP, 125
Online analytical
processing, 126
Open Source, 112, 166
OpenSSF, 166
operacja
asynchroniczna, 100
równoległa, 18
synchroniczna, 100
oprogramowanie, 24
aktualizacja, 154, 162
CD, 31, 32, 49
CI, 31, 32, 49
cykl życia, 46
diagnostyka, 152
jakość, 57, 132
konfiguracja, 153
monitorowanie, 151,
152, 167

oprogramowanie
 przestarzałe, 57
 sprzętowe, 16
 systemowe, 16, 20,
Patrz też: system
 operacyjny
 tworzenie, 28, 48, 53
 implementacja, 48
 infrastruktura, 28,
 48, 117, 119, 120,
 121, 123, 124, 128
 kompilacja, 30
 koordynacja, 52
 model iteracyjny, 54
 na potrzeby własne, 70
 narzędzia, 120, 121,
 123
 platforma, 119, 120,
 121, 123
 programista, 117,
 127
 projektowanie, 47,
 110, 117, 118
 środowisko, 30
 testowanie, 30, 31,
 49, *Patrz też:* test,
 testowanie
 utrzymanie, 50, 147,
 150, 151, 152
 w firmie, 24
 wbudowane, 16
 wdrożenie, 49, 147, 148
 produkcyjne, 148
 testowe, 148
 wersja kandydacka, 50
 wsparcie, 50
 zasady przetwarzania
 danych osobowych,
 69
 zastosowania, 25
 otwarte oprogramowanie,
 112
 outsourcing, 80

P

PaaS, 102
 packages, *Patrz:* pakiet
 pakiet, 100
 pamięć
 dyskowa, 17, 19
 nieulotna, 19
 operacyjna, 17, 19, 35
 podręczna, 89
 zrzut, 134
 Platform as a Service, 102
 plik
 binarny, 124
 HostRuleSet.java, 38
 konfiguracji, 154
 logów, 133, 151
 margins.R, 39
 node_api.cc, 37
 płaski, 124
 symboli, 134
 SYSINIT.ASM, 36
 tekstowy, 124
 pluskwa milenijna, 87
 płatność
 stykowa, 21
 zbliżeniowa, 21
 POJO, 92, 107
 port sieciowy, 159
 problem roku
 2000, 87
 2038, 88
 proces, 90
 procesor, 17, 35
 częstotliwość, 18
 na karcie kredytowej, 21
 rdzeń, 18
 PROD environment, *Patrz:*
 środowisko produkcyjne
 PROD release, *Patrz:*
 oprogramowanie
 wdrożenie produkcyjne
 PROD-PARALLEL
 environment, *Patrz:*
 środowisko produkcyjne
 równoległe
 Product Owner,
Patrz: Scrum
 właściciel produktu
 Production environment,
Patrz: środowisko
 produkcyjne
 produkt
 wartościowy
 minimalny, 67
 właściciel, 68
 programista, 117, 127
 DevOps, 49
 programowanie
 funkcyjne, 104
 obiektowe, 104
 reaktywne, 104
 strukturalne, 104
 zwinne, 73
 dokumentacja, 76
 historyjka
 użytkownika, 76,
 77, 78
 manifest, 54, 55, 58
 Progressive Web Apps,
Patrz: PWA
 protokół, 104
 DNS, 159
 HTTP, 104, 105,
 157, 159
 HTTPS, 104, 166
 REST, 105, 157
 sieciowy, 104
 SOAP, 108
 prototyp, 76
 przetwarzanie
 interaktywne, 110
 wsadowe, 110
 przypadek testowy
 skrajny, 139
 PWA, 118

Q

QA environment, 145
 quality assurance
 environment, 145

R

refactoring, 57
 refaktoryzacja, 57
 regex, 123
 rejestr, 34
 release, *Patrz:*
 oprogramowanie
 wdrożenie
 release candidate,
Patrz: oprogramowanie
 wersja kandydacka
 Representational State
 Transfer, *Patrz:* usługa
 REST
 Return on Investment, 69
 RFC 3987, 161
 RFID, 21, 22
 RIA, 118
 Rich Internet
 Applications, 118
 ROI, 69
 Roles and Entitlements, 75

S

Saarinen Eliel, 83
 SaaS, 102
 Scrum, 58, 59, 62
 estymacja, 60
 mistrz ceremonii, 59, 62
 sprint, 59, 61, 62
 zakres, 61
 właściciel produktu,
 58, 60
 Scrum Master, 59, 62
 SDLC, 46
 serverless computing, 103

Service Level Agreement,
 102
 Simple Object Access
 Protocol, 108
 Single Page Application,
 118, 161
 site map, 76
 SLA, 102
 smartfon, 16, 17, 20
 SME, 47
 SMS, 16
 Software as a Service, 102
 software development
 life cycle, 46
 SPA, 118, 161
 specjalista
 QA, 49
 sprzężenie zwrotne, 67
 stakeholder, *Patrz:*
 interesariusz
 storyboard, 76
 Subject Matter Expert, 49
 system
 kontroli jakości
 danych, 28
 obiegu dokumentów, 26
 operacyjny, 16, 18, 19, 20
 Apple iOS, 20, 119,
 121
 DOS, 36
 Google Android, 20,
 119, 121
 Linux, 121
 Tizen, 20
 Ubuntu, 20
 Unix, 20
 Windows, 20, 120, 156
 planowania zasobów
 przedsiębiorstwa, 27
 sterowania domem, 23
 zarządzania
 dostępem do
 informacji, 26

procesami
 biznesowymi, 27
 ryzykiem, 26

Ś

środowisko
 integracyjne, 145
 produkcyjne, 145, 155
 Dalvik, 120
 równoległe, 148, 155
 programistyczne, 145
 zapewnienia jakości, 145

T

tablet, 119
 TCO, 69, 168
 technical dept, 57, 58
 telewizor, 23
 test
 akceptacyjny, 143, 155
 automatyczny, 143
 beta, 50
 decyzji, 141
 eksploracyjny, 143
 ewidencja, 144
 integracyjny, 140, 142
 jednostkowy, 140
 komponentowy, 140
 komponentu, 141
 manualny, 143
 metryka, 141
 negatywny, 138, 139
 porażka, 139
 pozytywny, 138, 139
 regresyjny, 144
 rozgałęzienia, 141
 sukces, 139
 systemowy, 140, 142
 tester, 49
 testowanie, 31
 Total Cost of Ownership,
Patrz: TCO

typ danych, 85
 data, 87
 liczbowy, 86
 wyliczeniowy, 98
 zmiennoprzecinkowy,
 86

U

UAT, 143
 UAT environment, 145
 umiędzynarodowienie, 86
 Unified Modelling
 Language, *Patrz:* język
 UML
 urządzenie
 przenośne, 20, 21
 wejścia, 20
 wyjścia, 20
 user acceptance test, 143
 User Acceptance Test
 environment, 145
 user experience, 117
 User story, *Patrz:*
 wymagania historyjka
 użytkownika
 usługa
 REST, 105, 157
 sieciowa, 104, 108
 uwierzytelnienie, 122
 użytkownik, 117
 rola, 75
 uprawnienia, 75

V

virtual machine, *Patrz:*
 maszyna wirtualna

W

warstwa
 dostępu do danych, 117,
 122
 prezentacji danych, 117,
 119
 przetwarzania danych,
 124
 wireframe, 76
 workflow, 74
 współdzielenie, 18
 wymagania, 78
 biznesowe, 70, 73, 76,
 77, 85, 100, 101, 103
 danych, 75, 85, 106
 dokumentacja, 71, 74,
 75
 komentarze, 72
 wersja, 71
 wymagań
 biznesowych, 73
 wymagań
 funkcjonalnych, 74
 funkcjonalne, 74, 101
 gromadzenie
 metoda mieszana, 70
 metoda wstępująca,
 70
 metoda zstępująca,
 70
 historyjka
 użytkownika, 76, 77,
 78
 jakość, 79
 kryteria akceptacyjne,
 77

niefunkcjonalne, 74, 84,
 101
 pola i atrybuty, 75, 92
 powiązanie, 72
 role i uprawnienia, 75
 rozwiązania, 70, 85
 techniczne, 70, 77
 walidacja, 80
 weryfikacja, 80
 wewnętrzne, 69
 zewnętrzne, 68, 69
 źródła, 70
 wyrażenie regularne, 123

X

XML, 106, 107
 XML Schema, *Patrz:* XSD
 XSD, 106

Z

zabezpieczenie
 antykradzieżowe, 21
 założenia, 101
 zamek elektryczny, 22
 zegar częstotliwość
 taktowania, 18
 zero-day, 164

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Przeczytaj i zrozum — język branży IT dla każdego

- Informatyczny żargon dla przeciętnego odbiorcy
- Praktyczna wiedza o wytwarzaniu oprogramowania
- Skuteczna komunikacja z przedstawicielami branży IT

Jeśli nie mieszkasz w jaskini na końcu świata, komputery prawdopodobnie opanowały już niemal każdy obszar Twojego życia. Otaczają Cię dosłownie ze wszystkich stron i pomagają w wielu codziennych czynnościach. Z pewnością używasz ich do komunikacji, rozrywki, pracy i nauki, robisz za ich pomocą zakupy i planujesz wakacje. Dzięki komputerom Twoja codzienność jest prostsza i przyjemniejsza... do czasu, gdy musisz coś załatwić z kimś, kto odpowiada za ich programowanie. W tym momencie wszystko się komplikuje, a Ty przestajesz cokolwiek rozumieć.

Jeśli w takich chwilach zadajesz sobie pytanie, o co temu człowiekowi chodzi, a takie terminy jak release, agile, repozytorium, ticket, legacy, implementacja, merge, request, storyboard, bug, backend, branch, log czy mock wywołują u Ciebie dreszcz przerażenia — spieszmy z pomocą! Dzięki tej książce nie tylko poznasz terminologię informatyczną, lecz również zdobędziesz wiedzę o procesie projektowania, tworzenia i utrzymywania oprogramowania komputerowego, a także dowiesz się, jak się skutecznie komunikować z zaangażowanymi w to osobami. Przy użyciu prostego języka i na praktycznych przykładach autor wprowadzi Cię w świat IT i sprawi, że przestaniesz się pocić na widok informatyka. Nauka każdego języka wymaga słownika — oto Twój słownik!

Jeśli chcesz pojąć zagadnienia z dziedziny IT i porozumieć się ze specjalistami z branży, a przy okazji zrelaksować przy lekturze — to książka dla Ciebie! Polecam!

dr Biana Siwińska — Perspektywy

Książkę polecam wszystkim, którzy myślą o przebranżowieniu się do IT lub współpracują ze specjalistami IT. Autor w przystępny sposób wyjaśnia, czym zajmują się programiści, testerzy, analitycy i scrum masterzy, jak działają komputery i kto za co odpowiada w skomplikowanym procesie tworzenia oprogramowania. Wielkim plusem jest wyjaśnienie kilkuset pojęć branżowych, które warto poznać, niezależnie od planów zawodowych.

Joanna Gotfryd — Mampracuj.pl

Bardzo pomocna książka dla wszystkich nowych w branży IT! Terminy wyjaśnione przystępnie, procesy opisane przez analogię do życia codziennego, wiedza w łatwej do przyjęcia formie. Polecam!

Cezary Woszczyk — Sharpeo

	Sprawdź nasze szkolenia!	KOD KORZYŚCI Sięgnij po więcej! ▶ 
 helion.pl	 AKADEMIA IT & BUSINESS	ISBN 978-83-283-9003-4  9 788328 390034
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 49,90 zł