



Paweł Baszuro

**U MNIE
DZIAŁA**
JĘZYK
BRANŻY IT

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Małgorzata Kulik

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/umnie>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5613-9

Copyright © Helion 2019

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	73
Praca nad nie swoimi wymaganiami	79

Projektowanie	81
Krok po kroku	83
Przykład wymagań kształtujących architekturę	103
Programowanie i programiści	107
Podział programistów	109
Programista programiście nierówny	118
Nieprogramiści tworzący oprogramowanie	119
Testowanie	121
Zgłoszenia błędów	125
Rodzaje testów	128
Środowisko testowe	133
Wdrożenie i utrzymanie	135
Wdrożenie	136
Utrzymanie	138
Skorowidz	145



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 Grace Hopper odnotowała wadliwe działanie komputera Mark II. Wadliwe działanie było spowodowane przez znaną w elektromechanicznych podzespołach ćmę. 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 domeny¹¹, 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.

Niedociągnięcia programistów wynikają z braku wiedzy lub doświadczenia konkretnego programisty, jego pośpiechu

¹¹ *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>.

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 bugu. 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 (względny 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 umożliwiające 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. Jedynka 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 pracę¹². Donald 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 podzięcie za zgłoszenie. Zazwyczaj im większa firma, im groźniejsze ryzyko

¹² Wpis w Wikipedii w języku angielskim dostępny pod adresem https://en.wikipedia.org/wiki/Knuth_reward_check.

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 podzięcie za zgłoszenie błędu związanego z bezpieczeństwem zwykła rozsyłać czarne karty kredytowe¹³. Zaś firma Microsoft za pomocą programu Windows Insider¹⁴ 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.

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

¹³ Facebook hands out White Hat debit cards to hackers, ELI-NOR 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/>.

¹⁴ Strona Microsoft Windows Insider Program, dostępna pod adresem: <https://insider.windows.com/pl-pl/>.

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, czego oczekiwaliśmy 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 niektórych schorzeniach. Przetestowanie tego rozwiązania zajęło jednak kilka tysięcy lat. Oczekiwaniem względem przedstawionej metody była poprawa stanu zdrowia chorego, co udowadniało 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 oraz jej zakres (najlepiej zanim zlecimy zespołowi technicznemu pracę nad oprogramowaniem). Wyróżnia się testy jednostkowe, komponentowe, integracyjne oraz systemowe. 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 linii kodu do liczby wszystkich linii 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 linii 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 niezniekształconej 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,
 Patrz: wymagania kryteria
 akceptacyjne
administrator, 52, 144
Agile Manifesto, *Patrz:* programowanie
 zwinne manifest
algorytm, 83
Amazon Web Services, 97
analiza biznesowa, 47
API, 95
aplikacja
 architektura, 95, 109, 110
 internetowa
 bogata, *Patrz:* RIA
 progresywna, *Patrz:* PWA
 mobilna, 111
 multimedialna, 113
 projektowanie, 104
 tworzenie, 111, 112, 113, 115
application programming interface,
 Patrz: API
autoryzacja, 114

B

backend, *Patrz:* warstwa dostępu
 do danych
bajt, 125
batch processing, *Patrz:* dane
 przetwarzanie wsadowe
baza danych, 115, 116, 117
BI, *Patrz:* Business Intelligence
biblioteka, 95
 Cocoa, 112
Big Data, 104
blockchain, 105
błąd, 122, 123, 124
 out of memory, 19
 wyszukiwanie, 125
 zgłoszenie, 125, 126, 127
BPM, *Patrz:* system zarządzania
 procesami biznesowymi
BPMN, 84, 94, 95
bramka, 85
BRD, 73, 74
bug, 122
Business Intelligence, 119

Business process management,
Patrz: system zarządzania
 procesami biznesowymi
 Business Process Model and Notation,
Patrz: BPMN
 Business Requirements Document,
Patrz: BRD

C

cache, *Patrz:* pamięć podręczna
 CAT, 28
 central processing unit, *Patrz:* procesor
 chmura obliczeniowa, 96
 cloud computing, *Patrz:* chmura
 obliczeniowa
 CPU, *Patrz:* procesor
 CSS, 110
 czynniki
 wewnętrzne, *Patrz:* wymagania
 wewnętrzne
 zewnętrzne, *Patrz:* wymagania
 zewnętrzne

D

Dalvik, *Patrz:* środowisko produkcyjne
 Dalvik
 dane
 analiza, 117, 120
 baza, *Patrz:* baza danych
 big data, *Patrz:* Big Data
 eksploracja, 120
 ekstrakcja, 118
 format, 99, 100, 101
 hurtowania, 117
 model, 75, 83, 88
 modelowanie, 120
 przetwarzanie wsadowe, 104

data mining, *Patrz:* dane eksploracja
 Data Requirements, *Patrz:*
 wymagania danych
 data warehouse, *Patrz:* dane
 hurtowania
 debugging, *Patrz:* debugowanie
 debugowanie, 123, 124, 125
 deployment, *Patrz:* oprogramowanie
 wdrożenie
 development environment,
 Patrz: środowisko programistyczne
 diagram, 86
 dług techniczny, 57, 58
 dokumentacja, 56
 dziedziczenie, 89

E

Edison Thomas, 122
 edytor kodu źródłowego, *Patrz:* kod
 źródłowy edytor
 ekspert dziedzinowy, 47
 enterprise resource planning,
 Patrz: system planowania zasobów
 przedsiębiorstwa
 enumeracja, *Patrz:* typ wyliczeniowy
 ERP, *Patrz:* system planowania zasobów
 przedsiębiorstwa
 eXtensive Markup Language,
 Patrz: XML

F

Facebook, 126
 Faynman Richard, 17
 feedback, *Patrz:* sprzężenie zwrotne
 frontend, *Patrz:* warstwa prezentacji
 danych
 FSD, 74

Functional Specification Document,
Patrz: FSD

G

garbage collector, 36
Google Cloud, 97

H

Hopper Grace, 121
hot fix, *Patrz:* kod poprawki
na gorąco
HTML, *Patrz:* język HTML
Hypertext Markup Language,
Patrz: język HTML
Hypertext Transport Protocol,
Patrz: protokół HTTP
Hypertext Transport Protocol Secure,
Patrz: protokół HTTPS

I

IDE, 30
INT environment, *Patrz:* środowisko
integracyjne
Integrated Development Environment,
Patrz: IDE
integration environment,
Patrz: środowisko integracyjne
interesariusz, 47, 52, 57, 59, 63, 73, 82
interfejs, 89
programistyczny, *Patrz:* API
użytkownika, 47, 113
dokumentacja, 76
Internet of Things, *Patrz:* IoT
Internet Rzeczy, *Patrz:* IoT
interoperacyjność, 103

inżynier
jakości, 49
wsparcia, 50, 141, 144
wymagań, 47
IoT, 105

J

JavaScript Object Notation, *Patrz:* JSON
język
assembler, 34
C, 35, 40
C#, 39, 112, 115, 118
C++, 35, 112
CoffeeScript, 41
HTML, 110
Java, 39, 40, 88, 112, 115, 118
JavaScript, 39, 41, 99, 111, 112,
115, 118
Kotlin, 112
Objective-C, 112
PHP, 115
Python, 115, 120
R, 40, 120
Ruby, 115
Scala, 115
skryptowy, 115
Swift, 112
TypeScript, 41
UML, 86, 95
Visual Basic, 112, 115
wyższego poziomu, 35
zapytań baz danych, 116
JSON, 99
JVM, 39, 115

K

Kanban, 58, 64
karta płatnicza, 21
klasa, 86, 88, 89
 abstrakcyjna, 89
klasyfikacja MoSCoW,
 Patrz: MoSCoW
Knuth Donald, 125
kod
 binarny, 34, 125
 poprawki na gorąco, 137
 repozytorium, 31
 źródłowy, 30, 124
 Apache Tomcat, 38
 DOS, 36
 edytor, 30
 GGPlot2, 39
 JavaScript, 41
 kompilacja, 31, 34
 NodeJS, 37
kodowanie, 48
kompilator, 34
komponent, 94, 98
 test, *Patrz:* test komponentu
komunikator, 25
konsola do gier, 23
kontrakt, 98
koszt posiadania całkowity,
 Patrz: TCO

L

lean, 78
legacy software, *Patrz:*
 oprogramowanie przestarzałe
library, *Patrz:* biblioteka
Little Jason, 78

M

manifest programowania zwinnego,
 Patrz: programowanie zwinne
manifest
maszyna wirtualna, 36, 40, 134
 Java, *Patrz:* JVM
memory dump, *Patrz:* pamięć zrzut
metryka testów, *Patrz:* test metryka
Microsoft, 126
Microsoft Azure, 97
Minimal Viable Product, *Patrz:* MVP
mock-up, 76
MoSCoW, 74
MVP, 67

N

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

O

odpluskwianie, *Patrz:* debugowanie
OLAP, *Patrz:* dane hurtowania
OLTP, 117
Online analytical processing,
 Patrz: dane hurtowania
operacja
 asynchroniczna, 94
 równoległa, 18
 synchroniczna, 94
oprogramowanie, 24
 aktualizacja, 142
 CD, 31, 32, 49
 CI, 31, 32, 49
 cykl życia, 46
 diagnostyka, 140
 jakość, 57, 122

konfiguracja, 141
 monitorowanie, 139, 140
 przestarzałe, 57
 sprzętowe, *Patrz:* oprogramowanie
 wbudowane
 systemowe, 16, 20, *Patrz też:*
 system operacyjny
 tworzenie, 28, 48, 53
 implementacja, 48
 infrastruktura, 29, 48, 109, 111,
 112, 113, 115, 120
 kompilacja, 31
 koordynacja, 52
 model iteracyjny, 54
 na potrzeby własne, 70
 narzędzia, 112, 113, 115
 platforma, 111, 112, 113, 115
 programista, *Patrz:* programista
 projektowanie, 47, 104, 109, 110
 środowisko, *Patrz:* IDE
 testowanie, 31, 49, *Patrz też:* test,
 testowanie
 utrzymanie, 50, 135, 138, 139, 140
 w firmie, 24
 wbudowane, 16
 wdrożenie, 49, 135, 136
 produkcyjne, 136
 testowe, 136
 wersja kandydacka, 50
 wsparcie, 50
 zasady przetwarzania danych
 osobowych, 69
 zastosowania, 25

P

PaaS, 96
 packages, *Patrz:* pakiet

pakiet, 95
 pamięć
 dyskowa, 17, 19
 nieulotna, 19
 operacyjna, 17, 19, 35
 podręczna, 84
 zrzut, 124
 Platform as a Service, *Patrz:* PaaS
 plik
 binarny, 116
 HostRuleSet.java, 38
 konfiguracji, 142
 logów, 123, 140
 margins.R, 39
 node_api.cc, 37
 płaski, 115
 symboli, 124
 SYSINIT.ASM, 36
 tekstowy, 115
 płatność, 21
 POJO, 88, 101
 proces, 84
 biznesowy notacja, *Patrz:* BPMN
 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,
 Patrz: MVP
 właściciel, 68
 programista, 109, 118
 DevOps, 49
 programowanie
 funkcyjne, 98
 obiektywne, 98
 reaktywne, 98
 strukturalne, 98
 zwinne, 73
 dokumentacja, 76
 historijka użytkownika, 77, 78
 manifest, 54, 56, 58
 Progressive Web Apps, *Patrz:* PWA
 protokół, 98
 HTTP, 98, 99
 HTTPS, 98
 REST, *Patrz:* usługa REST
 sieciowy, 98
 SOAP, 102
 prototyp, 76
 przypadek testowy skrajny, 129
 PWA, 110

Q

QA environment, *Patrz:* środowisko
 zapewnienia jakości
 quality assurance environment,
 Patrz: środowisko zapewnienia
 jakości

R

refactoring, *Patrz:* refaktoryzacja
 refaktoryzacja, 58
 rejestr, 34

release, *Patrz:* oprogramowanie
 wdrożenie
 release candidate, *Patrz:*
 oprogramowanie wersja kandydacka
 Representational State Transfer,
 Patrz: usługa REST
 Return on Investment, *Patrz:* ROI
 RFID, 21, 22
 RIA, 110
 Rich Internet Applications,
 Patrz: RIA
 ROI, 69
 Roles and Entitlements, *Patrz:*
 wymagania role i uprawnienia

S

Saarinen Eliel, 81
 SaaS, 96
 Scrum, 58, 59, 60, 62
 estymacja, 60
 mistrz ceremonii, 59, 62
 sprint, 60, 62, 63
 zakres, 61
 właściciel produktu, 59, 61
 Scrum Master, *Patrz:* Scrum mistrz
 ceremonii
 SDLC, *Patrz:* oprogramowanie cykl
 życia
 Service Level Agreement, *Patrz:* SLA
 Simple Object Access Protocol,
 Patrz: protokół SOAP
 site map, 76
 SLA, 96
 smartfon, 16, 17, 20
 SME, *Patrz:* ekspert dziedzinowy
 SMS, 16
 Software as a Service, *Patrz:* SaaS

software development life cycle,
Patrz: oprogramowanie cykl życia
 specjalista QA, 49
 sprzężenie zwrotne, 67
 stakeholder, *Patrz:* interesariusz
 storyboard, 76
 Subject Matter Expert, *Patrz:* ekspert
 dziedzinowy
 system
 kontroli jakości danych, 28
 obiegu dokumentów, 26
 operacyjny, 16, 18, 19, 20
 Apple iOS, 20, 111, 112
 DOS, 36
 Google Android, 20, 111, 113
 Linux, 113
 Microsoft Windows, 20
 Tizen, 20
 Ubuntu, 20
 Unix, 20
 Windows, 112, 144
 planowania zasobów
 przedsiębiorstwa, 27
 sterowania domem, 23
 zarządzania
 dostępem do informacji, 26
 procesami biznesowymi, 27
 ryzykiem, 27

Ś

środowisko
 integracyjne, 134
 produkcyjne, 134, 143
 Dalvik, 112
 równoległe, 136, 143
 programistyczne, 134
 zapewnienia jakości, 134

T

tablet, 111
 TCO, 69
 technical dept, *Patrz:* dział techniczny
 telewizor, 23
 test
 akceptacyjny, 132, 143
 automatyczny, 132
 beta, 50
 decyzyjny, 130
 eksploracyjny, 132
 ewidencja, 133
 integracyjny, 129, 131
 jednostkowy, 129
 komponentowy, 129
 komponentu, 130
 manualny, 132
 metryka, 130
 negatywny, 128, 129
 porażka, 129
 pozytywny, 128, 129
 regresyjny, 132
 rozgałęzienia, 130
 sukces, 129
 systemowy, 129, 131
 tester, 49
 testowanie, 31
 Total Cost of Ownership,
 Patrz: TCO
 typ wyliczeniowy, 92

U

UAT, *Patrz:* test akceptacyjny
 UAT environment, *Patrz:* środowisko
 testów akceptacyjnych
 Unified Modelling Language,
 Patrz: język UML

urządzenie przenośne, 20, 21
user acceptance test, *Patrz:* test akceptacyjny
User Acceptance Test environment, *Patrz:* środowisko testów akceptacyjnych
user experience, 109
User story, *Patrz:* wymagania historyjka użytkownika
usługa
 REST, 99
 sieciowa, 99, 102
uwierzytelnienie, 114
użytkownik, 109
 rola, 75
 uprawnienia, 75

V

virtual machine, *Patrz:* maszyna wirtualna

W

warstwa
 dostępu do danych, 109, 114
 prezentacji danych, 109, 111
 przetwarzania danych, 115
wireframe, 76
workflow, 74
współdzielenie, 18
wymagania, 79
 biznesowe, 70, 73, 74, 76, 77, 83, 94, 95, 97
 danych, 75, 83, 101
 dokumentacja, 71, 75

komentarze, 72
wersja, 72
wymagań biznesowych, *Patrz:* BRD
wymagań funkcjonalnych, *Patrz:* FSD
funkcjonalne, 74, 95
gromadzenie, 70
historyjka użytkownika, 77, 78
jakość, 79
kryteria akceptacyjne, 77
niefunkcjonalne, 74, 82, 95
pola i atrybuty, 75, 86
powiązanie, 72
role i uprawnienia, 75
rozwiązania, 70, 83
techniczne, 70, 77
walidacja, 80
weryfikacja, 80
wewnętrzne, 69
zewnętrzne, 68, 69
źródła, 70

X

XML, 100, 101
XML Schema, *Patrz:* XSD
XSD, 100

Z

zabezpieczenie antykradzieżowe, 22
założenia, 95
zamek elektryczny, 22
zegar częstotliwość taktowania, 18

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 

NIE ROZUMIESZ JĘZYKA BRANŻY IT? SIĘGNIJ PO TĘ KSIĄŻKĘ!

- OPANUJ SŁOWNIK INFORMATYKÓW
- POZNAJ CYKL ŻYCIA OPROGRAMOWANIA
- NAUCZ SIĘ KOMUNIKACJI Z BRANŻĄ IT

Kto choć raz miał do czynienia z informatykami, wie, że rozmowy z nimi często przypominają kontakty z przybyszami z innej galaktyki. Posługują się specjalistycznym żargonem, są mocno skupieni na aspektach technicznych, bywa, że się niecierpliwią, gdy otoczenie nie nadąża za ich tokiem rozumowania. Jeśli rozmówca nie jest zorientowany w branży IT, zwykle trudno mu znaleźć wspólny język z przedstawicielami środowiska i może mieć nie lada kłopot, gdy będzie musiał coś z nimi załatwić.

TA KSIĄŻKA BĘDZIE PRAWDZIWYM OBJAWIENIEM DLA WSZYSTKICH, KTÓRZY WSPÓŁPRACUJĄ Z OSOBAMI ZATRUDNIONYMI W IT. Jej celem jest zaprezentowanie podstawowych terminów używanych w tej dziedzinie, przedstawienie procesu wytwarzania i utrzymania oprogramowania oraz wsparcie w zrozumieniu zagadnień i problemów, które dla informatyków stanowią codzienność. Lektura pomoże tym, którzy chcą poznać specyfikę branży lub są na różne sposoby zaangażowani w projekty IT.

- TERMINOLOGIA UŻYWANA W ŚWIECIE INFORMATYCZNYM
- ETAPY WYTWARZANIA I WDRAŻANIA OPROGRAMOWANIA
- DEFINIOWANIE WYMAGAŃ I ZROZUMIENIE PROCESU
- KOMUNIKACJA Z PRZEDSTAWICIELAMI ŚRODOWISKA
- PRAKTYCZNE PRZYKŁADY Z ŻYCIA CODZIENNEGO
- OPANUJ INFORMATYCZNY ŻARGON W MGNIENIU OKA!

Helion 	<i>Sprawdź nasze szkolenia!</i> ↓ SZKOLENIA  AKADEMIA IT & BUSINESS	KOD KORZYŚCI <i>Sięgnij po więcej!</i> ▶ 
 helion.pl  HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 WWW.SZKOLENIA.HELION.PL	ISBN 978-83-283-5613-9  9 788328 356139
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 34,90 zł