



Testuj oprogramowanie jak



Metody automatyzacji

Poznaj najlepszą
na świecie metodę
testowania!

James Whittaker • Jason Arbon • Jeff Carollo

Tytuł oryginału: How Google Tests Software

Tłumaczenie: Julia Szajkowska

ISBN: 978-83-246-8656-8

Authorized translation from the English language edition, entitled:
HOW GOOGLE TESTS SOFTWARE; ISBN 0321803027; by James A. Whittaker;
and Jason Arbon; and Jeff Carollo; published by Pearson Education, Inc,
publishing as Addison Wesley.
Copyright © 2012 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Polish language edition published by HELION S.A.
Copyright © 2014.

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

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/teopgo>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa Alberto Savoi	11
Przedmowa Patricka Copelanda	15
Wstęp	21
O autorach	27
Rozdział 1. Wprowadzenie do procedur testowania oprogramowania stosowanych przez firmę Google	29
Jakość ≠ testy	35
Podział ról	36
Struktura organizacyjna	39
Od raczkowania, przez chód, do biegu	41
Rodzaje testów	43
Rozdział 2. Inżynier do spraw testowania oprogramowania	47
Z życia inżyniera do spraw testowania oprogramowania	50
Praca nad kodem i testy	50
Kim właściwie jest ITO?	56
Wczesny etap projektowania	57
Struktura w zespole	59
Dokumentacja projektu	60
Interfejsy i protokoły	63
Planowanie automatyzacji	64
Testowalność	65
System pracy inżyniera do spraw testowania oprogramowania — przykład	69
Wykonanie testu	79
Definicje rozmiarów testów	80
Wykorzystanie testów w infrastrukturze dzielonej	83
Korzyści z różnych rodzajów testów	84
Wymagania stawiane czasom wykonywania testów	88
Certyfikowany w testach	95
Wywiad z twórcami programu „Certyfikowany w testach”	97
Rozmowa kwalifikacyjna z inżynierem do spraw testowania oprogramowania	104
Wywiad z programistą narzędzi Tedem Mao	112
Rozmowa z twórcą aplikacji WebDriver Simonem Stewartem	114

Rozdział 3. Inżynier testujący	119
Testowanie z uwzględnieniem potrzeb użytkownika	119
Z życia inżyniera testującego	120
Planowanie testów	124
Ryzyko	144
Życie przypadku testowego	158
Życie błędu	164
Zatrudnianie inżynierów testujących	179
Kierowanie testami w Google	187
Testowanie w trybie utrzymania	193
Badanie jakości za pomocą botów. Doświadczenie	197
BITE. Nowe doświadczenie	211
Analizy testowe — Google Test Analytics	223
Prowadzenie darmowych testów	230
Testerzy zewnętrzni	235
Rozmowa z inżynierem testującym Lindsay Webster	237
Rozmowa z inżynierem testującym pracującym przy serwisie YouTube Apple Chow	244
 Rozdział 4. Kierownik zespołów inżynierskich	 251
Z życia kierownika zespołów inżynierskich	251
Zdobywanie pracowników i pomysłów	254
Wpływ	256
Rozmowa z KZI usługi Gmail Ankitem Mehtą	258
Rozmowa z KZI zespołu Android Hungiem Dangiem	265
Rozmowa z KZI projektu Chrome Joelem Hynoskim	270
Dyrektor testów	276
Rozmowa z dyrektorem testów w projektach Search i Geo Sheltonem Marem	277
Rozmowa z dyrektorem zespołu inżynierii narzędziowej Ashishem Kumarem	281
Rozmowa z dyrektorem testów w Google India Sujayem Sahnim	286
Rozmowa z kierownikiem testów Bradem Greenem	291
Rozmowa z Jamesem Whittakerem	294
 Rozdział 5. Jak poprawić testowanie w Google	 303
Poważne niedociągnięcia systemu pracy w Google	303
Przyszłość inżynierów do spraw testowania oprogramowania	306
Przyszłość inżynierów testujących	308
Przyszłość kierowników i dyrektorów zespołów testujących	309
Przyszłość infrastruktury testującej	309
Wnioski	311

Dodatek A Plan testowania systemu operacyjnego Chrome	313
Przegląd tematów	313
Ocena ryzyka	315
Testy podstawowe w kolejnych wersjach kompilacji	315
Dzienne testy ostatniej dobrej wersji	316
Testy wersji przeznaczonej do wprowadzenia na rynek	316
Testy ręczne kontra automatyczne	317
Dbłość o jakość — programiści kontra testerzy	317
Kanały dystrybucji	317
Udział użytkowników	317
Repozytoria przypadków testowych	318
Panel testowania	318
Wirtualizacja	318
Wyniki	319
Praca w pełnym obciążeniu, testy długotrwałe i stabilizacja	319
Szkielet wykonawczy (Autotest)	319
Partnerzy OEM	319
Laboratorium sprzętowe	320
Automatyzacja w strukturach E2E	320
Testowanie menedżera AppManager	320
Testy w przeglądarce	321
Sprzęt	322
Harmonogram działań	322
Osoby odpowiedzialne za prowadzenie testów i obszary ich działania	324
Powiązane dokumenty	324
 Dodatek B Wycieczki testowe dla Chrome	 325
Wyprawa do sklepu	325
Wyprawa studenta	326
Sugerowane obszary prowadzenia badań	327
Rozmowa międzynarodowa	327
Sugerowane obszary prowadzenia badań	327
Bieg na orientację	328
Sugerowane obszary prowadzenia badań	328
Do białego rana	329
Sugerowane obszary prowadzenia badań	329
Rzemieślnik	329
Narzędzia w Chrome	330
Kiepska okolica	330
Nieciekawe dzielnice przeglądarki Chrome	330

Ustawienia własne	331
Sposoby modyfikowania przeglądarki Chrome	331
Dodatek C Wpisy z bloga dotyczące narzędzi i kodowania	333
BITE — kilka stron na temat błędów i nadmiarowej pracy	333
Wypuście boty QualityBots	336
RPF — szkielet funkcji nagrywania i odtwarzania Record Playback Framework	338
Google Test Analytics — teraz w wersji otwartej	341
Szczegółowość	341
Szybkość	341
Wymowność	342
Wartość praktyczna	342
Skorowidz	347

ROZDZIAŁ 1

Wprowadzenie do procedur testowania oprogramowania stosowanych przez firmę Google

James Whittaker

Jedno pytanie słyszę częściej niż jakiekolwiek inne. Niezależnie od tego, w jakim przebywam kraju czy w jakiej biorę udział konferencji, ono zawsze pojawia się prędzej czy później. Nawet Nooglerzy, jak nazywamy nowych pracowników korporacji Google, zadają je, gdy tylko nabiorą nieco praktyki w pracy. Brzmi ono: „**Jak Google testuje oprogramowanie?**”.

Nie potrafię powiedzieć, ile razy już na nie odpowiadałem ani nawet określić, ile różnych wersji odpowiedzi udzieliłem, wiem natomiast, że sama odpowiedź zmienia się — im dłużej pracuję dla Google, tym więcej dowiaduję się o niuansach technik kontrolowania jakości programów. Od dawna już ścigała mnie wizja napisania takiej książki, więc gdy Alberto, który zazwyczaj odgraża się, że przerobi każdą książkę dotyczącą testowania oprogramowania na pieluchy dla dorosłych (twierdząc, że wtedy istnienie takich podręczników byłoby chociaż trochę uzasadnione), zaproponował, żebym usiadł do pracy, nie miałem już żadnych wątpliwości — ta książka musiała powstać.

A mimo to zwlekałem. Główny problem polegał na tym, że zupełnie nie nadawałem się na autora takiego podręcznika. Przede wszystkim w samym Google pracuje wiele bardziej niż ja obeznanych z tym tematem osób, zatem chciałem, by mogły pierwsze zmierzyć się z tym zadaniem. Druga sprawa, która nie pozwalała mi podjąć się pisania z czystym sumieniem, to fakt, że jako kierownik grupy testującej

przeglądarkę Chrome i system Chrome OS (które to stanowisko przejął obecnie jeden z moich byłych podwładnych) miałem wgląd jedynie w niewielki fragment rozwiązań testowych stosowanych w firmie Google. Czekало mnie jeszcze wiele nauki.

Testowanie oprogramowania w Google leży w gestii większego, kierowanego centralnie działu, tak zwanej grupy wydajności projektowej (ang. *Engineering Productivity Group*). Grupa ta obejmuje programistów, zespoły testujące oraz zespoły odpowiedzialne za pracę nad kolejnymi wersjami oprogramowania, a do obowiązków jej członków należy testowanie programu na wszystkich poziomach — od poprawności funkcjonowania poszczególnych modułów do sprawdzenia testów rozpoznawczych. Zespoły testujące dysponują szerokim wachlarzem narzędzi i infrastrukturą pozwalającą testować działanie wielu znanych z sieci rozwiązań: wyszukiwarki, wyświetlania reklam, dodatkowych aplikacji, serwisu YouTube i innych, które kojarzy się z marką Google. Google zdołało pokonać wiele trudności związanych z tempem pracy i skalą działania, dzięki czemu mimo rozległej infrastruktury ciągle jesteśmy w stanie wprowadzać nowe aplikacje z werwą właściwą nowym firmom. Jak zauważył w przedmowie Patrick Copeland, swój sukces Google zawdzięcza przede wszystkim zespołom testującym.

Testowanie oprogramowania w Google leży w gestii większego, kierowanego centralnie działu, tak zwanej grupy wydajności projektowej.

Gdy w grudniu 2010 system operacyjny Chrome OS trafił wreszcie do odbiorców, a kierowanie zespołem przekazałem szczęśliwie jednemu ze swoich podwładnych, mogłem więc wreszcie poświęcić nieco więcej uwagi pozostałym produktom Google. Tak rozpoczęła się praca nad tą książką. Zanim jednak przystąpiłem do pisania, postanowiłem spróbować swoich sił bardziej kameralnie, na blogu¹, we wpisie „How Google Tests Software”² — reszta to już historia. Sześć miesięcy później książka była gotowa. Teraz wiem, że nie powinienem był zwlekać tak długo z przystąpieniem do pracy. W ciągu tamtych sześciu miesięcy dowiedziałem się więcej na temat procesów testowania w Google niż przez dwa lata pracy, a książka uzupełniła materiały szkoleniowe dla Nooglerów.

To nie jedyna pozycja poświęcona testowaniu oprogramowania przez wielkie firmy. Gdy pracowałem w Microsoftzie, Alan Page, BJ Rollison i Ken Johnston napisali *How We Test Software at Microsoft*³, której większa część bazowała na doświadczeniach własnych autorów. Microsoft wyznaczał wtedy standardy jakości w dziedzinie testowania oprogramowania. Polityka tej firmy sprawiła, że etap testów zyskał najwyższy priorytet w oczach najlepszych inżynierów świata. Testerzy pracujący

¹ <http://googletesting.blogspot.com/2011/01/how-google-tests-software.html>

² „Testuj oprogramowanie jak Google” — przyp. tłum.

³ *Jak testujemy oprogramowanie w Microsoftzie* — przyp. tłum.

wcześniej dla Microsoftu byli wręcz rozchwytywani przez innych pracodawców, a Roger Shrirman, pierwszy przełożony grupy testującej w Microsoftzie, ściągał do Redmont wszystkich, którzy przejawiali zdolności w tej dziedzinie. To były złote czasy testów oprogramowania.

Wtedy firma wydała obszerną dokumentację całego procesu.

Nie trafiłem do Microsoftu na tyle wcześnie, by móc uczestniczyć w przygotowaniu tej publikacji, ale dostałem drugą szansę. Gdy trafiłem do Google, testy nad produktami zaczynały dopiero nabierać rozpędu. Grupa wydajności projektowej liczyła wtedy zaledwie kilkaset osób — dla porównania: w obecnych czasach liczba ta wzrosła do 1200. Problemy, o których Pat wspominał w przedmowie, właśnie odchodziły w niepamięć, a firma wchodziła w etap najbardziej dynamicznego rozwoju w swoich dziejach. Blog poświęcony zagadnieniom testowania oprogramowania w Google odwiedzały setki tysięcy osób miesięcznie, a GTAC⁴ na stałe weszła do kalendarza imprez branżowych. Niedługo po moim przejściu do Google Patrick dostał awans i stał się bezpośrednim przełożonym przynajmniej kilkunastu innych szefów projektów i kierowników działów. Jeśli szukać źródeł nowej fali złotego wieku testowania oprogramowania, to z pewnością można zaliczyć do nich Google.

Oznacza to, że procedury testowania stosowane w korporacji Google również zasługują na obszerną relację. Cały kłopot polega na tym, że nie potrafię takowych zdawać. Jednocześnie Google słynie ze stosowania prostego i bezpośredniego podejścia do prac nad oprogramowaniem, może więc i ja powinienem przyjąć tę politykę.

W książce *Testuj oprogramowanie jak Google* znajdziesz odpowiedzi na pytania dotyczące tego, co to znaczy być testerem w Google czy w jaki sposób podchodzimy do zagadnień skali, złożoności czy powszechnego użycia przygotowywanych u nas programów. Znajdziesz tu informacje, których próżno szukać gdzie indziej, ale jeśli nie zaspokoją one Twojej ciekawości, pamiętaj, że w sieci znajdziesz mnóstwo innych. Wystarczy je „wygooglać”!

Jednak to nie koniec tej historii, a wydaje mi się, że warto przedstawić ją całą. W każdym razie ja wreszcie dojrzałem do jej przedstawienia. Metody testowania oprogramowania w Google mogą z czasem stać się pewnego rodzaju standardem dla innych firm, gdyż coraz większa liczba producentów rezygnuje z zamykania swoich produktów na dyskach komputerów, odchodząc od tego modelu na rzecz wolności, jaką oferuje sieć. Nie spodziewaj się zatem znaleźć w tej książce wielu punktów stycznych z wydawnictwem sygnowanym przez Microsoft. Podobieństwa ograniczają się do liczby autorów — w obu przypadkach za powstanie książki odpowiadały trzy osoby — oraz ogólnej zbieżności tematów — w każdej z tych książek znajdziesz opis praktyk związanych z testowaniem oprogramowania, stosowanych w wielkiej firmie. Poza tymi dwoma aspektami książki te różnią się we wszystkim.

⁴ GTAC, czyli Google Test Automation Conference (<https://developers.google.com/google-test-automation-conference/>), to organizowana co roku konferencja poświęcona zagadnieniom testowania oprogramowania.

Metody testowania oprogramowania w Google mogą z czasem stać się pewnego rodzaju standardem dla innych firm, gdyż coraz większa liczba producentów rezygnuje z zamykania swoich produktów na dyskach komputerów, odchodząc od tego modelu na rzecz wolności, jaką oferuje sieć.

Patrick Copeland w przedmowie przygotowanej do tej książki opisał, w jaki sposób narodziła się metodologia stosowana dziś w Google. Oczywiście jej początki, odpowiadające początkom samej firmy, stały się zrębem dla metod używanych dziś, które w naturalny sposób wykształciły się na tej podstawie wraz z rozwijaniem się samego Google. Należy mieć świadomość, że Google to tygiel umysłów ścisłych, miejsce spotkania ludzi, którzy szlify w fachu inżynierskim zdobywali w wielu innych firmach. Kulturowana w nim polityka innowacyjności sprawiła, że techniki, które nie sprawdziły się w innych przedsiębiorstwach, były przez pracowników Google odrzucane lub modyfikowane. Z czasem, gdy szeregi testerów w Google zaczęły się rozrastać, zaczęto wprowadzać nowe pomysły i wdrażać nieznane dotąd procedury. Te z nich, które sprawdziły się w praktyce w Google, wrosły na dobre w model testowy stosowany w firmie, a pozostałe — jako zbędny balast — odrzucono. Testerzy pracujący dla Google z chęcią dadzą szansę każdemu pomysłowi, ale bardzo szybko odrzucają te z rozwiązań, które nie sprawdzają się w specyficznych warunkach firmy.

Google to przedsiębiorstwo, w którym ceni się innowacyjność i szybkość działania i które słynie z tego, że udostępnia swoje oprogramowanie, gdy tylko kod nadaje się do użycia (i jednocześnie kiedy na ewentualne skutki błędów naraża się zaledwie garstkę użytkowników), a także ze zbierania opinii na temat funkcji oprogramowania już wraz z pierwszymi zainteresowanymi (co zwiększa ich liczbę i pozwala lepiej ocenić możliwości programu). Prowadzenie testów w takich warunkach musi przebiegać przede wszystkim sprawnie, zatem techniki wymagające wcześniejszego planowania czy stałego nadzoru zwyczajnie się nie sprawdzają. Zdarza się, że testowanie odbywa się jednocześnie z pracami nad oprogramowaniem, tak że często trudno jest określić, w którym miejscu przebiega granica między tymi dwoma dziedzinami. Z kolei w innych przypadkach testy są całkowicie niezależne od etapu programowania, więc twórcy aplikacji nie mają nawet świadomości, że ich produkt podlega badaniom.

Zdarza się, że testowanie odbywa się jednocześnie z pracami nad oprogramowaniem, tak że często trudno jest określić, w którym miejscu przebiega granica między tymi dwoma dziedzinami. Z kolei w innych przypadkach testy są całkowicie niezależne od etapu programowania, więc twórcy aplikacji nie mają nawet świadomości, że ich produkt podlega badaniom.

Rozwój Google spowodował oczywiście spadek tempa prac nad oprogramowaniem, ale w stopniu ledwie zauważalnym. Jesteśmy w stanie przygotować system operacyjny w czasie niewiele dłuższym niż rok, nowe wersje aplikacji klienckich, takich jak przeglądarki Chrome, pojawiają się raz na kilka tygodni, zaś aplikacje internetowe aktualizujemy codziennie — a wszystko mimo to, że referencje, którymi szczyciliśmy się na początku działania, dawno straciły już na aktualności. Wydaje się, że w przypadku tego środowiska znacznie łatwiej przyjdzie wskazać cechy, jakich brak prowadzonym w nim testom — dogmatyczności, sztywnych ram, wielkich nakładów pracy czy czasu — niż podać te, jakimi takie testy powinny się charakteryzować. Mimo to właśnie je postaram się określić. Jedno mogę stwierdzić z całą pewnością: testy nie mogą stanowić czynnika hamującego wprowadzanie innowacyjnych rozwiązań czy spowalniającego tempo prowadzenia prac nad aplikacją. W każdym razie żadna procedura testowania nie zdoła zrobić tego dwukrotnie.

Doskonałe wyniki otrzymywane dzięki metodom testowania stosowanym w Google z pewnością nie są skutkiem małej liczby oferowanych przez firmę aplikacji. Zakres i stopień złożoności testów prowadzonych w Google są takie same jak w każdej innej dużej firmie. Niezależnie od tego, czy będziemy mówić o systemach operacyjnych przeznaczonych dla urządzeń indywidualnych, aplikacjach internetowych, programach działających na urządzeniach mobilnych, czy aplikacjach wielostanowiskowych, rozwiązaniach dla biznesu lub sieci społecznościowych, z pewnością znajdziemy wśród nich towary z oferty Google. Nasze aplikacje są ogromne i niebywale złożone, korzystają z nich miliony użytkowników, bywają też celem ataków hakerskich. Spore partie kodu są powszechnie dostępne, wiele fragmentów stanowi schedę po stosowanych dawniej rozwiązaniach, a sama firma podlega częstym kontrolom. Oferowane przez nas programy działają w setkach krajów na całym świecie, funkcjonując oczywiście w różnych wersjach językowych, a gdyby tego było mało, zawsze można wspomnieć o podstawowym wymogu, jaki stawiają im użytkownicy — aplikacje Google powinny dać się łatwo używać i „po prostu działać”. Z pewnością nie można powiedzieć, by praca testera w Google polegała na rozwiązywaniu banalnych problemów. Nasi testerzy przypuszczalnie muszą codziennie mierzyć się z każdym wyzwaniem, jakie można sobie wyobrazić.

To, czy Google wywiązuje się dobrze ze stawianych sobie zadań (pewnie nie), to kwestia dyskusyjna, natomiast z pewnością można powiedzieć jedno — stosowane w tej firmie podejście do zagadnienia testowania oprogramowania różni się wyraźnie od procedur, które miałem okazję obserwować w innych przedsiębiorstwach, a sądząc po postępującej nieubłaganej tendencji do przenoszenia oprogramowania z komputerów stacjonarnych do chmury, wydaje się, że praktyki stosowane w Google wkrótce staną się powszechne w tej branży. Mamy nadzieję, że wiedza zawarta w tej książce pozwoli rzucić nieco światła na rozwiązania stosowane w Google i tym samym doprowadzić do konstruktywnej dyskusji w temacie działań, jakie należałoby podjąć, by twórcy oprogramowania mogli oferować odbiorcom solidne, niezawodne produkty, na których ci mogliby polegać. Oczywiście pomysły wdrażane w Google nie są pozbawione wad, ale chcielibyśmy przedstawić je światu i poddać ocenie

międzynarodowego środowiska zajmującego się zagadnieniem prowadzenia testów oprogramowania. Dzięki temu będziemy mogli dalej rozwijać je i poprawiać.

Reguły testowania w Google wydają się klócić z nakazami logiki — w całej firmie pracuje mniej testerów niż w niejednym przedsiębiorstwie nad poszczególnymi projektami. Google Test to nie liczące miliony oddziały piechoty. Jesteśmy niewielkim i doborowym oddziałem specjalnym, którego podstawę działania stanowią starannie przygotowana taktyka i zaawansowane technologicznie uzbrojenie — tylko one dają nam szansę powodzenia w podejmowanych misjach. Brak potężnego zaplecza ludzi zmusza nas do konkretnego określania priorytetów. Larry Page ujął to doskonale: „Niedostatek podnosi przejrzystość”. Nieważne, czy mówimy tu o funkcjach oprogramowania, czy metodach prowadzenia testów — doświadczenie nauczyło nas, że w obydwu przypadkach najlepsze wyniki uzyskuje się, stosując wysoce wydajne, ale nie nazbyt odporne metody działania. Również dzięki niedostatkom nauczyliśmy się cenić zasoby wykorzystywane do prowadzenia testów. Dlatego też doceniamy każdego pracownika i staramy się, by zatrudniani w Google błyskotliwi testerzy angażowali się całymi sobą w prowadzone prace. Gdy ktoś pyta mnie o klucz do sukcesu, odpowiadam zazwyczaj: „Nie zatrudniaj zbyt wielu osób w dziale testów”.

Gdy ktoś pyta mnie o klucz do sukcesu, odpowiadam zazwyczaj: „Nie zatrudniaj zbyt wielu osób w dziale testów”.

Jak zatem firma taka jak Google radzi sobie z tak małym zespołem testującym? Gdybym miał ująć to najprościej, jak potrafię, powiedziałbym, że w Google ciężar dbania o odpowiednią jakość oprogramowania spoczywa przede wszystkim na barkach programistów. Jakość nie jest nigdy problemem „tych tam, testerów”. Każdy pracujący w Google programista jest jednocześnie testerem, więc o jakość dba dosłownie kolektyw (rysunek 1.1). Mówienie o stosunku zatrudnienia programistów do testerów w Google ma mniej więcej taki sam sens jak rozważanie stopnia zanieczyszczenia atmosfery przy powierzchni Słońca. Takie rozważania nie mają zwyczajnie sensu. Jeśli jesteś inżynierem, musisz zajmować się też testowaniem. Jeśli zaś jesteś inżynierem ze słowem „testowanie” w tytule, zdołasz nauczyć czegoś kolegów, którzy są tylko programistami.

To, że tworzymy oprogramowanie klasy światowej, dowodzi stanowczo, że wdrożone w Google rozwiązania zasługują na bliższe poznanie. Być może niektóre z naszych pomysłów sprawdzą się także w innych firmach, jednak z całą pewnością można stwierdzić, że część z nich wymaga ulepszenia. W dalszej części rozdziału znajdziesz skrócony opis metod testowania stosowanych w Google, a w pozostałych postaram się opisać, w jaki sposób usiłujemy łączyć praktyki testowe z tworzeniem kodu.



RYSUNEK 1.1. W Google stawiamy na jakość, a nie na wymyślne funkcje

Jakość \neq testy

„Jakości nie da się wdrożyć za pomocą testów”. To truizm, ale warto o tym pamiętać. Nieważne, czy mówimy o samochodach, czy o aplikacjach komputerowych — jeśli o jakość nie zadamy już w fazie projektu, produkt nigdy nie będzie dobry. Wystarczy posłuchać skarg na koszty tych producentów samochodów, którzy musieli wprowadzać poprawki do ostatecznej wersji produktu. Jeśli to, nad czym pracujesz, nie będzie przygotowane starannie od samego początku, przygotuj się na poważne kłopoty.

Niestety nie jest to ani tak proste, jak mogłoby się wydawać, ani zawsze skuteczne. Choć jakości rzeczywiście nie wdraża się za pomocą testów, praktyka dowodzi, że bez testów nie da się osiągnąć odpowiedniej jakości. Bo jak bez sprawdzania ocenić, czy produkt jest wystarczająco dobry, by przekazać go w ręce klientów?

Podejdźmy do problemu jak do zagadki logicznej. Najprostsze rozwiązanie zakłada rezygnację ze stanowiska, że prace nad produktem i testowanie są od siebie niezależne. Testowanie i opracowywanie powinny przebiegać jednocześnie. Napisz fragment kodu i sprawdź, jak działa. Dopisz kolejny kawałek i znów przetestuj. Testowanie nie gwarantuje jakości. Jakość osiąga się, łącząc pracę nad kodem z ciągłą weryfikacją wyników — należy mieszać je ze sobą tak długo, aż staną się jednolitą masą.

Testowanie nie gwarantuje jakości. Jakość osiąga się, łącząc pracę nad kodem z ciągłą weryfikacją wyników — należy mieszać je ze sobą tak długo, aż staną się jednolitą masą.

To właśnie staramy się osiągnąć w Google — połączyć pracę nad kodem i testowanie go, tak by jedno nie istniało bez drugiego. Napisz coś, a potem to sprawdź. Napisz następną część i znowu sprawdź. W takim modelu najważniejsze jest to, *kto* przeprowadza testy. Ponieważ liczba faktycznych testerów w Google jest znacznie mniejsza niż liczba programistów, to właśnie na tych ostatnich spada obowiązek sprawdzania kodu. A kto lepiej poradzi sobie z oceną jakości zaimplementowanych rozwiązań, jeśli nie osoba odpowiedzialna za powstanie kodu? Kto szybciej znajdzie błąd w programie niż jego twórca? Google może pozwolić sobie na ograniczenie liczby weryfikatorów, ponieważ zatrudnia najwyższej klasy programistów. Jeśli aplikacja zawodzi, winą obarcza się wtedy przede wszystkim programistę, który popełnił błąd, a nie testera, który go nie odnalazł.

Oznacza to, że jakość wynika głównie z zapobiegliwości, a nie z działań mających na celu wykrywanie błędów. Jakość wynika bezpośrednio ze sposobu rozwijania projektu, a nie z metod, za pomocą których jest on testowany. Staraliśmy się stworzyć proces wielostopniowy — na tyle, na ile mogliśmy scalić testowanie z pracą nad kodem — dzięki czemu wszelkie błędy pojawiające się na danym etapie prac nad projektem można bardzo łatwo wycofać. Dzięki temu nie dość, że oszczędzamy wielu przykrych niespodzianek naszym klientom, to jeszcze udało nam się znacznie zmniejszyć liczbę osób niezbędnych do usuwania błędów wywołania. Ogólnie staramy się sprowadzić kwestię testowania do badania, na ile sprawdzają się procedury zapobiegania występowaniu błędów.

Polityka łączenia prac nad kodem z testowaniem jest nierozzerwalnie związana ze sposobem myślenia preferowanym w Google. Na pytanie: „A gdzie wyniki testów?” można natknąć się u nas wszędzie — począwszy od uwag na marginesach raportów, po ściany toalet, gdzie umieszczamy plakaty przypominające naszym programistom o konieczności testowania kodu⁵. Testowanie musi być nieodzownym aspektem tworzenia kodu, gdyż dopiero mariaż procedur weryfikujących i programowania pozwala uzyskać produkt odpowiedniej jakości.

Testowanie musi być nieodzownym aspektem tworzenia kodu, gdyż dopiero mariaż procedur weryfikujących i programowania pozwala uzyskać produkt odpowiedniej jakości.

Podział ról

Aby pozostać w zgodzie z mottem: „Sam zrobileś, sam rozbierz” (i utrzymać ten stan rzeczy w miarę upływu czasu), poza utrzymywaniem typowego zespołu do zadań programistycznych firma musi określić dla pracowników także zakres innych obowiązków. Chodzi tu przede wszystkim o określenie czynności, które pozwolą

⁵ <http://googletesting.blogspot.com/2007/01/introducing-testing-on-toilet.html>

programistom skutecznie i wydajnie testować programy. W Google zadbaliśmy o to, by niektórzy z naszych inżynierów mieli za zadanie nadzorować pracę innych, dzięki czemu ci drudzy osiągają lepsze wyniki w krótszym czasie. Grupa nadzorująca bardzo często określa się sama mianem *testerów*, ale jej podstawowym zadaniem jest czuwanie nad utrzymaniem wydajności pracy. Rolą testerów jest podnoszenie wydajności w zespołach programistycznych, a unikanie wprowadzania kolejnych poprawek, które muszą pojawiać się w kiepsko opracowanym kodzie, to bardzo istotny czynnik wydajności. Dlatego też w dalszych rozdziałach poświęcimy sporo miejsca na dokładne omówienie stosowanego w Google podziału ról. Na razie zaś przedstawiam krótkie ich zestawienie.

Inżynier oprogramowania (IO) to zwyczajny programista. Do jego obowiązków należy przygotowanie kodu, który ostatecznie trafia do użytkownika. To programiści przygotowują dokumentację, określają struktury danych projektu i jego architekturę, a większość czasu poświęcają na tworzenie kodu i sprawdzanie jego poprawności. Inżynier oprogramowania przygotowuje ogromne ilości kodu testującego, głównie w ramach pracy w systemie projektowania z użyciem testów (ang. *test-driven design*, TDD), tworzy testy jednostkowe i — co jeszcze wyjaśnimy w tym rozdziale — bierze udział w opracowaniu małych, średnich i wielkich procedur testujących. Inżynier oprogramowania odpowiada za jakość wszystkiego, z czym ma styczność, niezależnie, czy to oprogramował, wyrugował potknięcia innych, czy całkowicie zmodyfikował kod. Tak właśnie — jeśli inżynier wprowadza zmiany w istniejących funkcjach i przez to funkcja przestaje przechodzić przez istniejące już testy albo wymaga dodatkowej weryfikacji, to inżynier ma obowiązek przygotować nowe procedury sprawdzające. Inżynier oprogramowania poświęca się niemal całkowicie pisaniu kodu.

Inżynier do spraw testowania oprogramowania (ITO) to także stanowisko programistyczne, przy czym osoby je piastujące mają za zadanie skupiać się przede wszystkim na kwestii testowania i ogólnych założeniach procedur testujących. Do ich obowiązków należy opiniowanie projektów i badanie kodu pod kątem spełniania norm jakościowych oraz oceny ryzyka. Do ich zadań należy refaktoryzacja kodu, tak by dawał się on łatwiej testować, oraz przygotowywanie szkieletu testującego poszczególne moduły i automatyzowanie procesu sprawdzania jakości kodu. Ludzie zatrudnieni na tego rodzaju stanowiskach to także programiści, ale ich zainteresowanie skupia się przede wszystkim na podnoszeniu jakości produktu i prowadzeniu weryfikacji. Kwestie dodawania nowych funkcji czy zwiększania wydajności są dla nich drugorzędne. Choć większość czasu również pochłania im programowanie, to przygotowywany przez nich kod ma przede wszystkim podnosić jakość oferowanego produktu, a nie rozwijać istotne z punktu widzenia użytkownika funkcje.

Choć większość czasu również pochłania im programowanie, to przygotowywany przez nich kod ma przede wszystkim podnosić jakość oferowanego produktu, a nie rozwijać istotne z punktu widzenia użytkownika funkcje.

Inżynier testujący (IT) to stanowisko zbliżone do stanowiska inżyniera do spraw testowania oprogramowania, przy czym osoby je piastujące powinny skupiać się na nieco innym aspekcie prowadzenia testów. Inżynier testujący powinien badać produkt przede wszystkim z punktu widzenia użytkownika, a dopiero potem zastanawiać się, jak na ten sam problem spojrzy programista. Niektórzy z zatrudnionych w Google inżynierów testujących poświęcają mnóstwo czasu na przygotowanie kodu skryptów automatyzujących czy aplikacji realizujących różne scenariusze użytkownika, a czasami wręcz udających działanie użytkownika. Jednocześnie organizują testy prowadzone przez IO czy ITO, interpretują wyniki i przeprowadzają ostateczną weryfikację, szczególnie w końcowych fazach pracy nad projektem, gdy wysiłki wszystkich skupiają się na przygotowaniu produktu do wypuszczenia na rynek. Inżynierowie testujący to eksperci do spraw produktu, doradcy do spraw jakości i analitycy ryzyka. Wielu z nich zajmuje się aktywnie programowaniem, lecz równie liczna grupa praktycznie tego nie robi.

Uwaga

Inżynier testujący sprawdza program przede wszystkim z punktu widzenia użytkownika. Do jego zadań należy przygotowanie wszystkich działań związanych z podniesieniem jakości produktu, interpretowanie wyników testów, przeprowadzanie samych testów i tworzenie kompleksowej automatyzacji testów.

Wracając jednak do rozważań na temat jakości — inżynierowie oprogramowania zajmują się pewnymi aspektami działania aplikacji i kwestią jakości wdrażanych rozwiązań w raczej izolowanym środowisku. Do ich obowiązków należy przygotowanie projektu odpornego na usterki, przywracanie aplikacji do stanu sprzed pojawienia się błędu, projektowanie z użyciem testów, tworzenie testów jednostkowych i aktywna współpraca z inżynierami do spraw testowania oprogramowania podczas pisania kodu testującego funkcje badanej aplikacji.

Z kolei inżynierowie do spraw testowania oprogramowania to programiści przygotowujący funkcje testowe. Oni opracowują szkielet pozwalający wyizolować przygotowywany kod przez symulowanie środowiska, w jakim ma on docelowo działać (tu wykorzystywane są takie narzędzia, jak *stub*, *mock* i *fake* — metody, które opiszę dokładniej nieco dalej), i określanie kolejności, w jakiej kod będzie weryfikowany. Innymi słowy, inżynier do spraw testowania oprogramowania przygotowuje kod, dzięki któremu inżynier oprogramowania będzie mógł sprawdzić poprawność przygotowanych funkcji. Samo prowadzenie testów bardzo często spada już na IO. ITO ma za zadanie sprawdzić, czy poszczególne funkcje w ogóle nadają się do testowania, i dopilnować, by IO aktywnie uczestniczył w przygotowywaniu kodu przypadków testowych.

Z powyższego opisu wynika wyraźnie, że ITO bierze pod uwagę przede wszystkim perspektywę programisty. Jego zadaniem jest zadbać o jak najwyższą jakość

działania poszczególnych funkcji programu i jak najbardziej ułatwić programistom zweryfikowanie przygotowanego przez nich kodu. Prowadzenie testów uwzględniających punkt widzenia użytkownika to zadanie zatrudnianych przez Google inżynierów testujących. Gdy testy na poziomie modułowym i na poziomie funkcjonalności przeprowadzone przez IT oraz ITO dadzą zadowalające wyniki, nadejdzie pora, by sprawdzić, w jaki sposób użytkownik odczuje działanie aplikacji zasilanej odpowiednimi danymi. Testy wykonywane przez IT są poniekąd podwójnym sprawdzeniem sprawności programistów. Wszystkie wykryte przez nich oczywiste błędy będą dowodem, że weryfikacja wykonana na wczesnym poziomie przez programistów została przeprowadzona niestaranie i w nieodpowiedni sposób. Gdy okaże się, że takie błędy są sporadyczne, IT może zająć się podstawowym stawianym przed nim zadaniem, czyli sprawdzeniem, czy oprogramowanie będzie działać poprawnie w typowych warunkach użytkowania, czy będzie spełniać oczekiwania użytkownika, czy jest odpowiednio zabezpieczone, zlokalizowane, czy jest przystępne i tak dalej. Inżynier testujący zajmuje się przede wszystkim przeprowadzaniem testów i koordynowaniem pracy swoich kolegów oraz testerów kontraktowych, testerów „z tłumu”, testerów wewnętrznych (tzw. *dogfooderów*⁶), użytkowników wersji beta i awangardy wśród użytkowników. To właśnie inżynierowie testujący przekazują członkom zainteresowanych grup informacje o problemach mogących wystąpić w podstawowym projekcie, opisują im stopień skomplikowania funkcji aplikacji i metody unikania błędów. Trzeba przyznać, że praca inżyniera testującego nie ma końca.

Struktura organizacyjna

W większości firm, z którymi miałem okazję współpracować, programiści i testerzy tworzą jeden zespół pracujący nad danym projektem — i jedni, i drudzy odpowiadają przed tym samym szefem zespołu. Jeden produkt, jeden zespół i wszyscy stoją po jednej stronie barykady.

Niestety nigdy nie widziałem, żeby ten model sprawdził się w praktyce. Starsi menedżerowie są rekrutowani najczęściej spośród kierowników zespołów lub programistów — bardzo rzadko z grupy testerów. Stąd w gorącym okresie poprzedzającym premierę główny nacisk kładzie się raczej na dopracowanie funkcji aplikacji, a nie na testowanie jakości jej jądra. W tak zorganizowanym zespole testowanie spada na podrzędną pozycję, czego dowody znajdziemy zresztą w nie tak odległej przeszłości, pełnej wadliwych produktów i przedwczesnych wejść na rynek. Czy znajdzie chętnych na Service Pack1?

⁶ Określenie *dogfood* przyjęło się w większości firm informatycznych w Stanach Zjednoczonych. Opisuje ono proces adaptowania oprogramowania przygotowywanego do wprowadzenia na rynek wewnątrz danej firmy. Wyrażenie *eating your own dogfood* (zjadanie karmy swojego psa — *przyp. tłum.*) ma oddawać koncepcję, że zanim sprzeda się komuś własny produkt, należy sprawdzić jego jakość na sobie.

Uwaga

Mimo że zespół pracujący nad danym projektem powinien być zgrany, nadzorujący pracę wywodzą się zazwyczaj z grupy kierowników niższego szczebla lub projektantów, rzadko zaś spośród testerów. W gorącym okresie poprzedzającym premierę aplikacji główny nacisk kładzie się raczej na dopracowanie funkcji aplikacji, a nie na testowanie jakości jej jądra. To struktura organizacyjna sprawia, że testy zawsze będą traktowane z mniejszym namaszczeniem niż programowanie.

Strukturę organizacyjną Google tworzą tak zwane obszary zainteresowania (ang. *Focus Area*), czyli OZ-y. Osobny OZ obsługuje strefę klienta (Chrome, Google Toolbar i im podobne), osobny geolokalizację (Mapy, Google Earth itp.), a jeszcze inne zajmują się reklamami, aplikacjami czy programami na urządzenia mobilne. Wszyscy inżynierowie oprogramowania odpowiadają przed kierownikiem lub wiceprezesem danego OZ.

Inżynierowie do spraw testowania oprogramowania i inżynierowie testujący wyłamują się z tego schematu. Do testów dochodzi w niezależnej jednostce, której kompetencje przenikają przez wszystkie obszary zainteresowania, czyli we wspomnianej już grupie wydajności projektowej. Testerzy są wypożyczani poszczególnym zespołom programistów i nikt nie dziwi się, gdy zgłaszają uwagi dotyczące jakości produktu czy zastrzeżenia dotyczące tych jego funkcji, które nie zostały należycie przetestowane czy też które wykazują zbyt wiele błędów, by można było uznać je za dopuszczalne. Kierujemy się własnymi priorytetami i nie zarzucimy dbania o solidność rozwiązań czy bezpieczeństwo na rzecz niczego, co nie będzie naprawdę istotne. Jeśli zespołowi programującemu zależy na skróceniu testów, musimy wiedzieć o tym wcześniej, ale nawet gdy pojawi się taka prośba, zawsze mamy prawo odmówić.

Dzięki temu Google może ograniczyć liczbę testerów. Kierownik zespołu pracującego nad przygotowaniem aplikacji nie może samodzielnie zdecydować o obniżeniu wymagań jakościowych stawianych produktowi, nie może też zatrudnić większej liczby testerów, by zepchnąć na nich niewdzięczne zadania. Programista zajmujący się przygotowaniem danej funkcji programu musi zmierzyć się też z tymi mniej przyjemnymi i żmudnymi aspektami pracy nad aplikacją i naprawdę nie ma prawa spychać tych zadań na jakiegoś nieszczęsnego testera. O przydziale testerów do poszczególnych projektów decyduje kierownictwo grupy wydajności projektowej. Podstawą do podejmowania decyzji są priorytet przypisany danemu projektowi, złożoność zagadnienia i potrzeby danego zespołu oceniane w stosunku do potrzeb pozostałych grup. Oczywiście i my możemy się mylić — zresztą czasami tak właśnie się dzieje — ale mimo to zastosowane podejście pozwala wykorzystać siły, jakimi dysponujemy, do zaspokojenia prawdziwych, a nie wydumanych potrzeb.

Uwaga

Przydzielaniem testerów do poszczególnych projektów zajmuje się kierownictwo grupy wydajności projektowej. Podstawą do podejmowania decyzji są priorytet przypisany danemu projektowi, złożoność zagadnienia i potrzeby danego zespołu oceniane w stosunku do potrzeb pozostałych grup. Zastosowane podejście pozwala wykorzystać siły, jakimi dysponujemy, do zaspokojenia prawdziwych, a nie wydumanych potrzeb.

Takie „wypożyczanie” testerów sprawia też, że zadania stawiane przed inżynierami do spraw testowania oprogramowania i inżynierami testującymi ciągle się zmieniają, dzięki czemu ITO i IT nie tracą zainteresowania tematem, który przecież zawsze jest nowy i fascynujący. Jednocześnie rozwiązanie to zapewnia w naturalny sposób wzmożony przepływ pomysłów wewnątrz firmy. Tak zwiększamy prawdopodobieństwo ponownego wykorzystania dobrych procedur testowych — tester, który sprawdził jakąś koncepcję w czasie pracy nad projektem z działu geolokacji, użyje jej zapewne także po przeniesieniu go do prac nad przeglądarką Chrome. Nic tak nie przyspiesza rozprzestrzeniania się idei jak przeniesienie jej twórcy w nowe środowisko.

Przyjmujemy zasadniczo, że po osiemnastu miesiącach tester może (ale nie musi) zażądać przeniesienia do nowego projektu bez żadnych konsekwencji. Z jednej strony oznacza to oczywiście, że projekt traci eksperta, lecz jednocześnie pamiętajmy, że w konsekwencji stosowanej polityki testerzy Google znają się ogólnie na wszystkim i mają bogate doświadczenie w pracy nad różnymi rodzajami aplikacji. Można powiedzieć, że Google to firma zatrudniająca testerów, którzy rozumieją potrzeby klienta, znają możliwości sieci i przeglądarek, wiedzą, czego wymaga praca z technologiami mobilnymi, i potrafią pisać programy w wielu językach i na różne platformy. A ponieważ nasze aplikacje i usługi są teraz powiązane ze sobą ściślej niż kiedykolwiek w przeszłości, tester zmieniający zespół tak czy inaczej dysponuje odpowiednim doświadczeniem.

Od raczkowania, przez chód, do biegu

Google osiąga tak zdumiewające wyniki mimo zatrudniania mniejszej niż w innych firmach liczby testerów, ponieważ w odróżnieniu od innych dużych firm rzadko usiłujemy wprowadzać do naszych produktów mnóstwo innowacji naraz. W zasadzie działamy zupełnie inaczej — najpierw przygotowujemy starannie jądro aplikacji i w chwili, gdy staje się ono zdadne do użycia, staramy się udostępnić je jak najszerszej grupie odbiorców. Następnie zapoznajemy się z ich uwagami i próbujemy wprowadzić je w życie, a gdy poprawki są gotowe, znów udostępniamy je klientom. Tak postąpiliśmy w przypadku aplikacji Gmail, która przez cztery lata funkcjonowała formalnie jako wersja beta. Etykieta pojawiająca się przy nazwie usługi ostrzegała

użytkowników, że produkt jest ciągle w fazie ulepszeń. Usunęliśmy ją dopiero, gdy udało się nam osiągnąć zamierzony wcześniej cel — serwery obsługujące rzeczywiste konta użytkowników przez 99,99% czasu. Podobnie postąpiliśmy z Androidem, udostępniając go wyłącznie na telefonie G1, które to połączenie zyskało sobie uznanie użytkowników i recenzentów. Nexus, telefon nowej linii i następcą G1, funkcjonował już z nową wersją systemu — o bardziej rozbudowanych funkcjach. Tu należy podkreślić, że gdy klient płaci za wczesną wersję towaru, musi ona być funkcjonalna na tyle, by nie miał on poczucia, że został oszukany. To, że jest to wczesna wersja, nie musi oznaczać, że towar nie nadaje się do użycia.

Uwaga

Google często wprowadza na rynek „najmniej rozbudowaną działającą wersję” programu i dopiero z czasem rozwija ją o kolejne możliwości, zbierając jednocześnie wrażenia użytkowników — tak tych wewnątrz firmy, jak i pierwszych klientów. Przed wykonaniem kolejnego niewielkiego kroku naprzód zawsze starannie rozważamy jego wpływ na jakość produktu. Każda aplikacja Google przechodzi przez kilka etapów: wersję „kanarkową” (ang. *canary*), deweloperską, testową, beta i RC (ang. *release channel*), zanim trafi w ostatecznej postaci do najszerzego grona odbiorców.

Rozwiązanie to nie niesie ze sobą tak wielkiego ryzyka, jak mogłoby się wydawać na pierwszy rzut oka. W rzeczywistości zanim produkt osiągnie fazę beta, musi przejść kilka innych etapów, na których sprawdzamy, czy jest wart przyznania mu etykiety „beta”. Przeglądarka Chrome — nad tym projektem spędziłem pierwsze dwa lata pracy dla Google — była oferowana różnym grupom odbiorców w różnych kanałach dystrybucji, zanim zebraliśmy odpowiednio wiele opinii, by uznać, że jest produktem godnym zaufania. Na poszczególnych etapach rozwoju aplikacja jest dostępna w kilku różnych kanałach dystrybucji:

- **kanarkowym** — kanał ten wykorzystujemy do rozpowszechniania wczesnych, zmieniających się niemal codziennie wersji oprogramowania, które prawdopodobnie nie nadają się jeszcze do zaprezentowania szerszemu gronu odbiorców. Te „wypusty” odgrywają taką samą rolę jak kanarek w kopalni — jeśli aktualna wersja nie przetrwa fazy testów, stanowi to sygnał, że prace nad nią przebiegały zbyt chaotycznie i konieczne trzeba zastanowić się nad ich kształtem. Wersje oferowane w kanale kanarkowym są przeznaczone wyłącznie dla bardzo odpornych psychicznie użytkowników, którzy są zainteresowani przede wszystkim prowadzeniem testów. Z pewnością nie należy proponować ich ludziom potrzebującym działającego w pełni programu. Ogólnie z kanału tego korzystają wyłącznie inżynierowie (programiści i testerzy) oraz kierownicy nadzorujący tworzenie danej aplikacji;

Uwaga

Zespół odpowiedzialny za rozwijanie systemu Android poszedł o krok dalej. Niemal wszyscy jego członkowie korzystają z wypuszczanych codziennie nowych wersji systemu. Chodzi o to, że jeśli sami nie będą mogli zadzwonić do domu, nie będą tak ochotczo przechodzić do porządku dziennego nad błędami w kodzie.

- **deweloperskim** — z kanału tego korzystają na co dzień programiści. Za jego pośrednictwem udostępnia się zazwyczaj wersje będące wynikiem prac z całego tygodnia, które sprawdziły się już w niektórych obszarach i przeszły niektóre z testów (do tego tematu wrócimy w dalszych rozdziałach). Wszyscy pracujący nad danym produktem mają *obowiązek* pobierać aplikacje udostępniane tą drogą, korzystać z nich w pracy i przeprowadzać testy. Jeśli wersja deweloperska aplikacji nie sprawdzi się w codziennych zdaniach, wraca do kanału kanarkowego. Tego rodzaju wydarzenia nie dają powodów do radości i zazwyczaj prowadzą do wzmoczonych badań mających pozwolić raz jeszcze ocenić całość projektu;
- **testowym** — tym kanałem rozpowszechnia się zazwyczaj wersję, którą można by nazwać mianem zwycięzcy miesiąca, czyli taką, która przejdzie pozytywnie większość prowadzonych na tym etapie testów i cieszy się największym uznaniem w oczach jej autorów. Z kanału testowego korzystają wewnętrzni dogfooderzy, a prezentowana im wersja aplikacji jest zazwyczaj silnym kandydatem na pozycję wersji beta. Na pewnym etapie prac wersja testowa staje się na tyle stabilna, że można używać jej bezpiecznie wewnątrz firmy, a z czasem udostępniać także wybranym współpracownikom zewnętrznym i partnerom, którzy mogliby skorzystać na wcześniejszym poznaniu nowego towaru;
- **beta** lub **RC** — do tego etapu docierają stabilne wersje testowe, które zostały sprawdzone już w wewnętrznym użyciu firmowym i zdołały pokonać wszystkie poprzeczki testowe stawiane przez członków zespołu deweloperskiego. To one jako pierwsze trafiają do szerszego grona odbiorców.

Stabilny rozwój — od raczkowania, przez chód, aż do biegu — pozwala nam przeprowadzać testy już w najwcześniejszych wersjach aplikacji i zbierać uwagi nie tylko z prowadzonych codziennie w każdym z kanałów automatycznych testów programowych, lecz także od ludzi, którzy mieli styczność z produktem.

Rodzaje testów

Google nie stosuje typowego podziału testów na testy kodowania, integracji i systemowe. Zamiast takiego rozróżnienia wprowadzamy własną nomenklaturę — testy *małe*, *średnie* i *duże* (nie mylmy tych pojęć ze stosowanymi przez lotną społeczność przybliżonymi rozmiarami ubrań). Stosowane nazwy mają oddawać zasięg prowadzonych badań. Za pomocą małych testów sprawdzamy niewielkie

wycinki kodu i tak dalej. Każda ze wspomnianych wcześniej grup inżynierów może zajmować się prowadzeniem dowolnego rodzaju testów; testy te mogą mieć charakter tak automatyczny, jak i ręczny.

Zamiast przeprowadzać osobno testy kodowania, integracji i systemowe, Google woli stosować *małe, średnie* bądź *duże* testy — w zależności od tego, jaki zakres aplikacji jest sprawdzany za ich pomocą.

Małe testy obejmują przede wszystkim (choć nie tylko) weryfikację automatyczną, której celem jest sprawdzenie kodu zawartego wewnątrz jednej funkcji czy w danym module. W czasie ich trwania bada się przede wszystkim sposób działania kodu, szuka błędów zapisu danych, określa warunki ich występowania i wyszukuje błędy logiczne typu *off-by-one*, czyli pomyłki polegające na przesunięciu o jeden wartości dyskretnych, bardzo często o charakterze brzegowym. Małe testy nie pochłaniają wiele czasu — zazwyczaj przeprowadza się je w kilka sekund. Odpowiednie funkcje przygotowują IO, rzadziej ITO, a IT prawie nigdy nie mają z nimi do czynienia. Małe testy prowadzi się w środowiskach próbnych za pomocą tak zwanych mocków i fake'ów. (Mock i fake to odmiany obiektów typu stub, czyli obiekty zastępujące rzeczywiste funkcje. Przechowuje się w nich funkcje, które być może w ogóle nie zostaną wykorzystane w ostatecznej wersji aplikacji, są zbyt niedopracowane, by można było ich użyć, czy zbyt skomplikowane, by określić warunki, w jakich mogą wystąpić w nich błędy. W dalszych rozdziałach wrócę jeszcze do tego tematu). Inżynierowie testujący nie piszą wprawdzie zbyt często małych testów, za to zdarza się im uruchamiać je, by określić przyczyny występowania konkretnego błędu. Małe testy mają za zadanie uzyskać odpowiedź na pytanie: „Czy ten fragment kodu działa tak, jak powinien?”.

Średnie testy przeprowadza się najczęściej automatycznie, by sprawdzić za ich pomocą przynajmniej dwie współdziałające funkcje aplikacji. Na tym etapie sprawdza się przede wszystkim relacje funkcji wywołujących się wzajemnie albo oddziałujących w jakiś sposób na siebie. Funkcje tego rodzaju nazywamy *najbliższymi sąsiadami*. Inżynierowie z grupy ITO nadzorują przygotowanie tych testów jeszcze we wczesnej fazie pracy nad aplikacją, równoległe do przygotowywania poszczególnych funkcji. Za przygotowanie ich od strony kodu, sprawdzenie poprawności i przeprowadzenie testów odpowiadają zazwyczaj IO. W razie gdyby średni test nie dał się uruchomić lub dawał błędne wyniki, programista może sam wprowadzić w nim poprawki. W późniejszej fazie prac nad aplikacją średnie testy wykonują inżynierowie testujący — czasami ręcznie (jeśli automatyzacja procesu byłaby zbyt trudna lub zbyt kosztowna), czasami zaś automatycznie. Średnie testy pozwalają szukać odpowiedzi na pytanie: „Czy ten zestaw sąsiadujących ze sobą funkcji działa w sposób, w jaki powinien?”.

Duże testy obejmują sprawdzenie współdziałania przynajmniej trzech (a zazwyczaj większej liczby) funkcji aplikacji i zawierają scenariusze faktycznego użytkowania. Przeprowadzenie ich zajmuje zazwyczaj co najmniej kilka godzin. Choć przy okazji

sprawdza się ogólny stopień zintegrowania poszczególnych funkcji, to najważniejszym ich aspektem jest badanie zwracanych w czasie ich trwania wyników, na podstawie których ocenia się, czy program spełni oczekiwania użytkownika. W pracach nad przygotowaniem dużych testów biorą udział wszystkie grupy inżynierów, a sam test może przebiegać w dowolny sposób — od pełnego zautomatyzowania procesów do prowadzenia ręcznie testów badawczych. Przeprowadzając duże testy, staramy się odpowiedzieć na pytanie: „Czy aplikacja działa w sposób, jakiego oczekiwałby użytkownik, i czy wyniki jej pracy są zadowalające?”. Do tej grupy testów zalicza się między innymi przekrojowe scenariusze badające wyniki pracy pełnych wersji oprogramowania i usług.

Uwaga

Małe testy mają za zadanie zweryfikować poprawność działania niewielkich fragmentów kodu uruchamianego w sztucznie przygotowanym środowisku. **Średnie testy** pozwalają sprawdzić działanie wielu współdziałających ze sobą modułów w warunkach sztucznie przygotowanych, ale także w faktycznym środowisku działania aplikacji. **Duże testy** pozwalają badać dowolną liczbę modułów uruchamianych w dedykowanym im środowisku i z wykorzystaniem autentycznych zasobów.

Sama nomenklatura nie jest istotna. Nie musisz nazywać testów małymi, średnimi i dużymi, jeśli nie masz ochoty — grunt, żeby zastosowane nazwy nie wprowadzały nikogo w błąd⁷. Najważniejsze, że testerzy w Google mają język, w którym mogą się porozumiewać i uzgadniać zakres prowadzonych prac. A jeśli któremuś bardziej rzutkiem zamarzyłoby się przeprowadzenie testów jeszcze wyższego poziomu, w związku z czym wprowadziłby nazwę *olbrzymie testy*, każdy inny pracownik firmy domyśliłby się od razu, że chodzi o sprawdzenie działania całego systemu, każdej jego funkcji, i że należy na to zarezerwować odpowiednio dużo czasu. Żadne dodatkowe wyjaśnienia nie są już konieczne⁸.

Wytyczne dotyczące elementów poddawanych testom i dynamiki procesu weryfikacji zmieniają się z projektu na projekt. Google preferuje politykę wprowadzania na rynek częstych aktualizacji, co pozwala szybciej prezentować kolejne wersje

⁷ Nazewnictwo stosowane w Google zrodziło się z autentycznej potrzeby. Okazało się, że należy ustandaryzować terminologię stosowaną przez testerów, którzy pracowali przecież wcześniej w innych firmach, gdzie posługiwano się zupełnie innymi nazwami — czasami były to testy dymne, czasami testy BVT, czasem integracyjne. Terminom tym przypisywano różne, niejednokrotnie przeciwstawne znaczenia, uznano zatem, że Google potrzebuje własnych określeń.

⁸ I rzeczywiście idea olbrzymich testów trafiła do oficjalnej dokumentacji. W infrastrukturze Google funkcjonują na co dzień pojęcia testów małych, średnich i tak dalej — w ten sposób definiuje się kolejkę wykonywania zadań w czasie trwania testów automatycznych. Więcej szczegółów znajdziesz w rozdziale poświęconym zadaniom stawianym przed inżynierami do spraw testowania.

użytkownikom i szybciej zbierać ich odzew. W ten sposób przyspieszamy prace nad kolejnymi zmianami. W Google staramy się rozwijać tylko te aplikacje i usługi, które przypadły użytkownikom do gustu, i wprowadzać nowe funkcje tak szybko, jak jest to możliwe, by odbiorca nie musiał długo na nie czekać. Jednocześnie jesteśmy w stanie ograniczyć liczbę zbędnych rozwiązań na bardzo wczesnym etapie. Oczywiście tego rodzaju model działania wymaga wczesnego wydania aplikacji w ręce użytkowników i zewnętrznych programistów, ale właśnie dzięki niemu mamy pewność, że oferujemy ludziom dokładnie to, czego potrzebują.

Wreszcie trzeba stwierdzić wyraźnie, że w połączeniu testów automatycznych z prowadzonymi ręcznie te pierwsze stoją na zdecydowanie uprzywilejowanej pozycji. Jeśli zagadnienie da się zautomatyzować, a sam problem nie wymaga odwoływania się do ludzkiej intuicji i zdolności umysłowych, należy bezwzględnie stosować rozwiązania maszynowe. Testy ręczne warto stosować tylko tam, gdzie osąd człowieka jest absolutnie niezbędny, na przykład podczas podejmowania decyzji dotyczących estetyki interfejsu użytkownika czy określania, które z danych mogą naruszać prawo do prywatności.

We wszystkich trzech opisanych wcześniej metodach testowania w połączeniu testów automatycznych i ręcznych te pierwsze sprawdzają się znacznie lepiej. Jeśli zatem zagadnienie da się zautomatyzować, a sam problem nie wymaga odwoływania się do ludzkiej intuicji i zdolności umysłowych, należy bezwzględnie stosować rozwiązania maszynowe.

To jedna strona medalu. Jednocześnie Google przeprowadza mnóstwo testów ręcznych, zarówno skryptowych, jak i eksploracyjnych, ale nawet one przebiegają pod czujnym okiem zautomatyzowanych procedur. Techniki nagrywania pozwalają tworzyć testy automatyczne na podstawie zapisów z przeprowadzanej wcześniej ręcznej weryfikacji — każde przesunięcie kursora i każde kliknięcie jest rejestrowane, by w kolejnych wersjach aplikacji móc wykorzystać te nagrania i dzięki temu zminimalizować regresję prac nad programem. Takie rozwiązanie pozwala testerom pracującym własnoręcznie nad badaniem aplikacji zająć się nowymi problemami. Automatyzujemy też procedurę przesyłania raportów i organizowania wykonywanych ręcznie zadań testowych⁹. Jeżeli przykładowo funkcja nie przejdzie automatycznych testów poprawnie, system wskaże ostatnie z wprowadzonych w kodzie zmian, typując tym samym najbardziej prawdopodobnego sprawcę nieszczęścia, wyśle wiadomość do jej autora i sam wprowadzi do dziennika odpowiednią informację o wystąpieniu błędu. Nieustannie podejmowane wysiłki, mające na celu wprowadzić automatyzację również w ostatnim odcinku ludzkiego umysłu, stanowią swojego rodzaju specyfikację narzędzi testowych nowej generacji, jakie powstają w Google.

⁹ Więcej na temat mechanizmów rejestrujących i automatyzacji testów ręcznych znajdziesz w rozdziałach poświęconych pracy inżynierów testujących.

Skorowidz

#include, 99
3G, 235

A

Accepted, 173
ActionScript
 ActionScript2, 246
 ActionScript3, 246
addurl, 72
AddUrl, 71
addurl.pb.cc, 72
addurl.pb.h, 72
addurl.proto, 71
addurl_frontend, 74
addurl_frontend.cc, 73
addurl_frontend_test, 78, 79
addurl_frontend_test.cc, 74
AddUrlFrontend, 70, 72, 73
 destruktor, 73
 konfiguracja testu, 76
 konstruktor, 73
AddUrlFrontendTest, 75
AddUrlReply, 71
AddUrlRequest, 71
 pole uri, 71
AddUrlService, 70
 definicja protokołu, 70
 deklaracja konstruktora kopiowania
 i operatora, 72
 fałszywa usługa, 75
 wstrzykiwanie zależności, 72, 73
aktualny stan aplikacji, 125
alokacja, 254
 decydujące argumenty, 255
 dopasowanie predyspozycji, 255
 poprzednie alokacje, 255
 potrzeby projektu, 255
 pragnienia pracownika, 255

analiza
 dwóch izolowanych zmian w kodzie, 92
 ryzyka, 139
WSM, 127, 128, 133
 analiza ryzyka, 139
 budżet i czas, 139
 czasowniki aplikacji, 133
 dla aplikacji Google+, 139
 lista właściwości, 129
 macierz, 227
 możliwości, 133
 możliwości aplikacji, 136
 para właściwość - składowa, 136, 153
 przykłady możliwości, 135
 przymiotniki aplikacji, 128, 129
 reguły, 127
 rzeczowniki systemu, 132
 składowe, 132
 składowe aplikacji, 136
 stopniowanie możliwości aplikacji, 157
 tabela możliwości, 137
 właściwości, 128
 właściwości aplikacji, 129, 130, 136, 227
 właściwości systemu, 129
 wskazanie składowych, 132
zależności, 92
Android, 265
 filary, 267
 podnoszenie wartości, 266
 zakładanie zespołu, 265
API
 automatyzujące, 118
aplikacja
 Ads, 247
 BITE
 funkcja RPF, 220
 polecenie Record and Play, 219
 przeglądanie informacji o błędach, 216
 wpływ na kształt usługi Maps, 216
 zapisywanie i odtwarzanie danych, 217

- aplikacja
 - etapy rozwoju, 42
 - Google Test Analytics, 224
 - kanały dystrybucji, 42
 - kompatybilność, 272
 - rozwój każdej z funkcji, 306
 - RPF, 218
 - Selenium, 217
 - skuteczne znajdowanie błędów, 305
 - stabilny rozwój, 43
 - tryb utrzymania jakości, 196
 - w stanie utrzymania
 - działania przygotowawcze, 197
 - wysyłająca adresy URL do Google, 70
 - AppManager, 320
 - arkusz kalkulacyjny
 - wady, 223
 - As3Unit, 248
 - Assigned, 173
 - Assigned to, 170
 - atak
 - typu cross-site, 221
 - atrybuty
 - ID, 219
 - Attachments, 171
 - automatyczne testy, 46
 - przedwysyłkowe, 67
 - automatyczny system testujący, 90
 - automatyzacja
 - działań
 - rad, 284
 - interfejsu użytkownika, 273
 - P0, 315
 - testów, 79, 83, 87, 280
 - Android, 267
 - aplikacji przeglądarek, 117
 - sytuacje, 268
 - w strukturach E2E, 320
 - Autotest, 274, 319
- B**
- badanie
 - jakości za pomocą botów
 - ChromeBot, 206
 - doświadczenie, 197
 - indeks, 198
 - pełzanie, 198
 - pochodzenie botów, 206
 - ranking, 199
 - wyniki, 199, 202
 - przypadków testowych dla systemu Chrome OS
 - ręczne, 316
 - stabilności działania, 316
 - Banana Proxy, 248
 - baza błędów, 113
 - beta, 43
 - Bialik Tracy, 97
 - biblioteki
 - addurl, 72
 - addurl_frontend, 74
 - C++ AddUrlFrontend, 74
 - ctype, 194
 - dla platform systemowych, 53
 - funkcje nowej usługi, 55
 - PyAuto, 273
 - testowanie, 55
 - współdzielone
 - zmiany, 93
 - bieg na orientację, 328
 - BITE, 211, 333
 - GTCM, 222
 - i RPF
 - BITE Web Test Framework, 222
 - Flux Capacitor, 222
 - kukielka, 221
 - początki, 220
 - WTF, 222
 - interfejs do wprowadzania danych dotyczących
 - błędu, 335
 - konsola nagrywania i odtwarzania działań
 - użytkownika, 334
 - menu dodatku, 334
 - przewodzenie testów ręcznych i eksploracyjnych, 222
 - RPF, 340
 - skrypty, 223
 - system zarządzania przypadkami testowymi, 222
 - warstwy, 223
 - bieżące, 223
 - wprowadzenie informacji o błędzie, 334
 - zalety, 334
 - Blocking, 171
 - błędy, 164
 - bardzo rzadkie, 145

- baza błędów Buganizer, 165, 216, 217
 - autoryzacja logowania, 165
 - błędy o priorytecie P0, 166
 - błędy o priorytecie P1, 166
 - błędy o priorytecie P2, 166
 - błędy o priorytecie P3, 166
 - błędy o priorytecie P4, 166
 - cechy, 175
 - elastyczna hierarchia n-poziomowa, 165
 - generowanie podsumowań, 165
 - informacje sumaryczne, 166
 - odpowiedzialność za wykrycie błędów, 165
 - omówienie, 170
 - podnoszenie komfortu pracy, 165
 - pole formularza zgłoszenia błędu, 170
 - procedury postępowania, 175
 - segregacja błędów, 175
 - średnia życia błędu, 166
 - tworzenie list pilnych zadań, 165
 - wyszukiwanie bloków tekstu, 165
 - zestaw danych, 165
 - zestaw ustawień domyślnych, 165
 - baza błędów Bugs DB, 165
 - baza błędów Mozilla Bugzilla, 169
 - błąd 56859, 201
 - błąd 77261, 201
 - częste, 146
 - integracji, 67
 - katalogowanie, 164
 - logiczne
 - typu off-by-one, 44
 - minimalne, 147
 - na stronach internetowych, 333
 - narzędzia śledzenia błędów Issue Tracker, 169
 - niewielkie, 147
 - okazjonalne, 146
 - rzadkie, 146
 - wykrywanie, 164
 - zgłaszanie, 164
 - znaczne, 147
 - życie błędu, 164, 177
 - sposoby rejestracji błędów, 178
 - bot
 - projekt Bots, 209
 - zakład i rozwinięcie na potrzeby sieci, 209
 - Bot Chromebot, 321
 - Browser Integrated Test Environment, 211
 - Browser Integrated Testing Environment, 333
 - BrowserUX, 321
 - buforowa składnia protokołu, 63
 - bufory protokołów, 63
 - Bug, 174
 - Buganizer, 113
 - BugsDB, 113
 - build, 54
 - build system, 72
 - build target, 54
 - buzz_client_tests, 94
- ## C
- C++ AddUrlFrontend, 74
 - canary, 42
 - Capability Maturity Model, 95
 - CC, 171
 - cel kompilacji, 54
 - biblioteka, 54, 55
 - binariów, 55
 - integrowanie, 55
 - plik testu, 54, 55
 - celność działania wyszukiwarki dla zestawu
 - wyników, 205
 - centra komunikacyjne, 286
 - Certyfikowany w testach, 19, 95
 - korzyści, 97
 - kryteria programu, 102
 - nagradzający system punktowy, 101
 - największe trudności, 103
 - opis programu, 98
 - pomysł, 97
 - poziomy, 96, 98
 - projekty spadkowe, 103
 - waga projektu, 100
 - wprowadzenie w Google, 100
 - wskazówki podczas wprowadzania, 104
 - wymagania, 96
 - wywiad z twórcami programu, 97
 - change list, 66
 - Changed, 171
 - Changelists, 171
 - chodzenie za słońcem, 291
 - ChromeBot, 206
 - Chrome, 270
 - aplety Java, 330
 - aplikacje Flash, 330

Chrome

- automatyzacja interfejsu użytkownika, 273
- badanie poprawności produktu od strony interfejsu, 273
- bieg na orientację, 328
- BITE, 333
- błędy regresji, 273
- czas, 329
- do białego rana, 329
- dodatki, 329
- filmy na stronach, 330
- języki, 327
- karty i okna, 329
- kiepska okolica, 330
- kompatybilność aplikacji, 272
- konsola JavaScript, 330
- menedżer
 - zadań, 330
 - zakładek, 328
- motywy, 331
- narzędzia dla programistów, 328, 330
- O3D, 330
- odtwierzanie plików Flash, 134
- okno incognito, 328
- opcje sieciowe, 327
- pobrane pliki, 328
- podręczny pasek nawigacji, 328
- poziomy dostęp, 327
- przenoszenie danych z dysku przez chmurę, 327
- rozdzielenie profili, 331
- rozmowa międzynarodowa, 327
- rozszerzenia, 330, 331
- RPF, 340
- rzemieślnik, 329
- sposoby modyfikowania przeglądarki, 331
- sprawdzenie przenośności, 327
- strona motywów, 328
- system operacyjny, 327
- testowanie, 273
- ustawienia, 328, 331
 - własne, 331
- Web, 338
- wiele instancji, 330
- wycieczki testowe, 325
- wyprawa
 - do sklepu, 325
 - studenta, 326
- wyświetlanie źródła, 330

zbadanie

- funkcji Kopiuj i wklej, 327
- możliwości, 327
- okna przeglądarki, 328

zmiany sieci, 272

Chrome OS, 58, 273

- automatyzacja w strukturach E2E, 320

Autotest, 319

- dbałość o jakość, 317

- główna przeglądarka, 320

- harmonogram działań, 322

- kanał dystrybucji, 317

- wprowadzenie, 320

- laboratorium sprzętowe, 320

- osoby odpowiedzialne za prowadzenie testów i obszary ich działania, 324

- pakiet testów ręcznych, 321

- panel testowania, 318

- partnerzy OEM, 319

- plan testowania, 313

- Chrome OS jako pierwotna platforma przeglądarki, 314

- dostarczanie danych, 314

- macierz automatyzacji testów sprzętowych, 313

- możliwość testowania i efekt mnożnika, 314

- narzędzia i testy o otwartym kodzie, 314

- ocena ryzyka, 313, 315

- umożliwianie szybkich zmian, 314

- powiązane dokumenty, 324

- praca w pełnym obciążeniu, 319

- repozytoria przypadków testowych, 318

- sprzęt, 322

- stabilizacja, 319

- szkielet wykonawczy, 319

- testowanie menedżera AppManager, 320

testy

- dla opcji zasilania, 322

- dla problemów sprzętowych, 322

- długotrwałe, 319

- dzienne ostatniej dobrej wersji, 316

- kompatybilności stron, 320

- podstawowe w kolejnych wersjach

- kompilacji, 315

- ręczne eksploracyjne, 321

- ręczne kontra automatyczne, 317

- w przeglądarce, 321

- wersji przeznaczonej do wprowadzenia na rynek, 316

- udział użytkowników, 317
- wirtualizacja, 318
- wycieczki, 321
- wyniki działania, 319
- Chrome Web Store, 219
- Chromebook, 221
- chromium.org, 216
- ciągłość systemu integracji, 90
- CiW, 98
- CKO, 192
- Cloud Code Coverage, 287
- committers, 65
- common_collections_util, 93
- Component, 171
- crawling, 88
- Create an AddUrlFrontend, 76
- Created, 172
- cross-talk, 110
- cuke, 159
- Customer Issue, 174
- czas wykonywania testów, 88
- czasowniki aplikacji, 133

D

- Dang Hung, 265
- deadlock, 110
- defect-driven development, 116
- definicje rozmiarów testów, 80
- Depends On, 171
- deweloperski kanał dystrybucji, 43
- do białego rana, 329
- dogfooder, 39, 154
- dokumentacja
 - projektu, 60
 - cele dla członków zespołu, 63
 - interfejsy i protokoły, 62, 63
 - inżynier do spraw testowania
 - oprogramowania, 61
 - kompletność, 62
 - moment kończenia wstępnych prac, 60
 - poprawność, 62
 - projekt, 62
 - recenzowanie, 61, 62
 - spójność, 62
 - testowanie, 62
 - środowiska testowego, 89
 - wersji, 79

- DOM, 207
 - model obiektowy, 218
- dopasowanie rozmyte, 338
- droid-food, 267
- drzewo zasobów, 105
- Duplicate, 173
- duże testy, 44, 45, 83, 85
 - mocne i słabe strony, 85
 - pokrycie kodu, 87
 - wykorzystanie zasobów, 85
 - zakres działania, 83
- dwadzieścia procent czasu, 51, 57, 255
- dwudziestoprocentowe prototypy, 58
- dynamika
 - działań, 259
 - zespołu, 259
- dyrektor testów, 276
 - przyszłość, 309
 - rozmowa
 - z dyrektorem testów w Google, 286
 - z dyrektorem w projektach Search i Geo, 277
 - z dyrektorem zespołu inżynierii narzędziowej, 281
 - wymagania, 277
 - zadania, 276
- dziesięciominutowy plan testowy, 151

E

- Eclipse, 218
- edytor ACE, 335
- efekt mnożnika, 314
- elementy
 - o dokładnej zgodności, 219
- eliminacja zbędnych testów
 - wytyczne, 196
- Encyklopedia testów, 89
- Engineering Productivity Group, 30

F

- fake, 44, 47, 55
 - definiowanie, 77
 - małe testy, 87
 - testy integracyjne, 63
- FakeAddUrlService, 76, 77
- fałszywe
 - funkcje obliczeniowe, 74
 - zgłoszenie, 206

fanty, 276
 Feature Request, 174
 Feedback, 293, 318
 cel, 293
 Fix later, 173
 Fixed, 173
 fixit, 99
 flaga, 88, 89
 FlexUnit, 248
 Flux Capacitor, 222
 Focus Area, 40
 Found In, 172
 funkcje
 budujące ranking, 199
 fałszywe, 74
 HandleAddUrlFrontendRequest, 74
 Instant Pages, 211
 najbliższy sąsiad, 44
 obsługi języka Java, 220
 obsługujące zdarzenia w aplikacjach
 internetowych, 74
 pauzuj i popraw, 219
 pomocnicze, 74

G

glina kompilacyjny, 69
 globalna inżynieria testowa, 290
 Gmail, 258
 błędy regersywne, 260
 czas działania aplikacji, 260
 gmail_server_test, 92
 Google
 analiza ryzyka, 139
 Checkout, 219
 CKO, 192
 infrastruktura testująca, 309
 kierowanie testami, 187
 rola kierownika zespołu testowego, 188
 zarządzanie zespołami testerów, 188
 kultura testowania, 287
 ośrodki obsługujące obszary geograficzne, 286
 rodzaje stanowisk kierowniczych i
 dyrektorskich, 189
 dyrektor testów, 190
 główny kierownik techniczny, 189
 główny technik, 189

 kierownik zespołów inżynierskich, 189
 starszy dyrektor testów, 190
 wewnętrzne procesy rekrutacyjne, 191
 inicjatywy strategiczne, 191
 negocjacje, 191
 recenzje i ocena działania, 191
 technika, 191
 zdolności komunikacyjne, 191
 zunifikowany system katalogowania błędów, 176
 Google App Engine, 229
 Google Desktop, 193
 Google Diagnostic Utility, 288
 Google Docs, 126
 Google Feedback, 176, 213, 318
 algorytm klastrowy, 176
 Google Groups, 164
 Google India, 286
 Google Search, 198, 277
 granice ogólnego zasięgu, 278
 szczelność rozwiązania, 279
 testowanie, 278
 testy zmian konfiguracji, 281
 współczynnik zgodności trafień
 dynamiczny, 198
 statyczny, 198
 Google Sites, 130
 Google Talk Labs Edition, 194
 Google Test, 34
 Google Test Analytics, 127, 131
 analizy testowe, 223
 przekształcanie listy możliwości w przebieg
 testów, 227
 w wersji otwartej, 341
 ważenie ryzyka w projektach łączonych, 227
 zadania, 229
 Google Test Case Manager, 159
 Google Testing Blog, 333
 Google+, 139
 możliwości, 140
 komentarze, 142
 kręgi, 141
 ludzie, 141
 powiadomienia, 141
 profil, 140
 strumień, 141
 usługa Hangout, 141
 wpisy, 142
 zdjęcia, 142

- składowe, 140
 - komentarze, 140
 - kęgi, 140
 - ludzie, 140
 - powiadomienia, 140
 - profil, 140
 - strumień, 140
 - wpisy, 140
 - zainteresowania, 140
 - zdjęcia, 140
- właściwości, 139
 - ekspreswna, 140
 - istotna, 140
 - łatwa w obsłudze, 140
 - prywatna, 140
 - rozwijalna, 140
 - społeczna, 140
- Google-food, 267
- Green Brad, 291
- grupa wydajności projektowej, 30
 - przydział testerów do projektów, 40
 - przyszłość, 303
- GTA
 - częstotliwość występowania błędów, 145
 - błędy bardzo rzadkie, 145
 - błędy częste, 146
 - błędy okazjonalne, 146
 - błędy rzadkie, 146
- GTCM, 159
 - Google App Engine, 164
 - interfejs programistyczny, 164
 - JSON, 164
 - liczby, 161
 - mechanizmy wewnętrzne, 164
 - menedżer przypadków testowych, 162
 - stopień pokrycia testami, 162

H

- Hajdarabad, 286
- HandleAddUrlFrontendRequest, 73, 74
- Harvester, 88
- historia użytkownika, 138, 139, 153, 155
 - przygotowanie, 155
- hooks, 247
- http_reply, 73
- http_request, 73

- HUD, 212, 318
- HYD, 287
- Hynoski Joel, 270

I

- infrastruktura
 - Amazon EC2, 205
 - badająca wartości parametrów
 - charakteryzujących działanie aplikacji, 290
 - dzielona
 - system testowania, 83
 - wykorzystanie testów, 83
 - kompilująca, 282
 - obliczeniowa
 - system integracji ciągłej, 91
 - SkyTap, 205
 - testowa, 47, 83
 - testująca, 282
 - kierunek rozwoju, 310
 - Matrix, 218
 - przyszłość, 309
 - zewnętrzna, 47
- inicjatywa własna pracowników, 57
- innowacje testowe i eksperymenty, 232
 - GTAC, 232
 - koncepty, 233
 - przebieg eksperymentów, 235
- Instant Pages, 211
- integracja ciągła, 284
- interfejsy, 63
 - DOM, 194
 - TestScribe, 222
- Internal Cleanup, 174
- inżynier do spraw testowania oprogramowania,
 - 37, 38, 47, 49, 50
 - definicje rozmiarów testów, 80
 - dokumentacja projektu, 60
 - recenzja, 61
 - dołączanie do zespołu, 60
 - dwudziestoprocentowy projekt, 58
 - interfejs i protokoły, 63
 - kod buforowy, 63
 - kompilacja celów testów, 55
 - korzyści z różnych rodzajów testów, 84
 - odpowiedzialność za jakość produktu, 65
 - planowanie automatyzacji, 64
 - taktyka, 64

inżynier do spraw testowania oprogramowania
 praca, 50
 nad cudzym kodem, 110
 nad kodem, 50
 predyspozycje do wykonywania zadań ITO, 181
 przyszłość, 306
 rola, 61
 rozmowa kwalifikacyjna, 104
 dopasowanie do kultury firmy, 111
 kandydat doskonały, 111
 ocena kandydatów, 106, 109, 110
 ocena sposobu szukania rozwiązania, 105
 osoba przeprowadzająca rozmowę, 112
 przykład implementacji, 108
 znaczenie pytań, 108
 struktura w zespole, 59
 system pracy
 przykład, 70
 wykonanie testu, 79
 szersza perspektywa, 59, 61
 średnie testy, 82
 testowalność aplikacji, 65
 testy, 50
 funkcjonalne, 110
 integracyjne, 64
 strukturalne, 110
 w strukturze organizacyjnej, 40
 wczesny etap projektowania, 57
 współpraca z programistami, 56
 wykorzystanie testów w infrastrukturze
 dzielonej, 83
 wymagania, 56
 wywiad z programistą narzędzi, 112
 zadania, 56
 inżynier niezależny, 309
 inżynier oprogramowania, 37, 38, 49, 56, 120
 odpowiedzialność za jakość produktu, 65
 przykład aplikacji, 70
 przyszłość, 307
 w strukturze organizacyjnej, 40
 inżynier testujący, 38, 39, 49, 65, 119
 awans, 252
 dobór, 121
 określanie poziomu ryzyka
 wskazówki, 153
 początkowy etap pracy, 121
 predyspozycje do wykonywania zadań IT, 182

przyszłość, 308
 rozmowa z kandydatem na stanowisko ITO, 179
 w strukturze organizacyjnej, 40
 wymagania, 122, 123
 zaangażowanie w projekt, 120
 zadania, 120
 zakres obowiązków, 120, 122
 zatrudnianie, 179
 scenariusze rozmów z kandydatami, 179
 szukanie rozwiązania pośredniego, 180
 zmniejszenie wymagań
 programistycznych, 179
 inżynieria testowania, 279
 IO, 37, 49
 Issue Tracker, 169
 IT, 22, 38, 49, 119
 ITO, 22, 37, 49, 119
 a IT
 predyspozycje do wykonywania zadań IT, 182
 predyspozycje do wykonywania zadań
 ITO, 181
 różnice, 181

J

jakość
 a testy, 35
 aplikacji, 304
 inżynierowie oprogramowania, 38
 substytut, 303
 systemu Chrome OS, 314
 udział użytkowników, 317
 w fazie projektu, 35
 zapobiegliwość, 36
 jednorodna platforma programistyczna, 53
 jednorodność, 54
 język
 specyfikacji wersji budowanej, 54

K

kanal
 droidów, 267
 dystrybucji, 42
 beta, 43
 Chrome OS, 317
 deweloperski, 43
 kanarkowy, 42

- RC, 43
 - testowy, 43
- kanarkowy kanał dystrybucji, 42
- kiepska okolica, 330
- kierownik produktu, 306
- kierownik projektu, 60
 - dokumentacja projektu, 60
- kierownik testów
 - przyszłość, 309
- kierownik zespołów inżynierskich, 251
 - aktywne uczestnictwo w działaniach, 257
 - alokacja, 254
 - decydujące argumenty, 255
 - awans, 252
 - kompletowanie zespołów, 261
 - nowe projekty, 255
 - odpowiednia dynamika działań, 259
 - odpowiedzialność, 257
 - optymalizowanie działań, 253
 - organizacja pracy zespołu, 267
 - poznanie swoich ludzi, 253
 - poznanie swojego produktu, 252
 - pułapki, 263
 - rola, 251
 - rozmowa
 - z Jamesem Whittakerem, 294
 - z kierownikiem testów, 291
 - z KZI projektu Chrome, 270
 - z KZI usługi Gmail, 258
 - radę dla kierowników, 259
 - z KZI zespołu Android, 265
 - swobodny przepływ pracowników, 254
 - techniczna strona pracy, 261
 - wpływ, 256
 - współpraca międzyzespolowa, 257
 - zadania, 252, 257
 - zakładanie zespołu, 265
 - zasady testowania, 262
 - zdobywanie pracowników i pomysłów, 254
- klasy
 - AddUrlFrontend, 72, 73
- kod
 - buforowy, 63
 - testowy, 47
 - testów, 79
- kodowanie
 - wpisy z bloga, 333

- kolejki wysyłania, 67, 68, 69, 290
- kolejkowanie kodu, 67
- kolejność testowania, 89
- kompatybilność
 - aplikacji, 272
 - witryn, 316
- kompilacje
 - ciągłe, 67, 69, 290
 - przykłady zależności wewnątrz kompilacji, 92
- kompilator, 54
 - bufora protokołu, 72
- konflikty, 89
- KP, 60, 306
- kukiełka, 221
- kultura pracy, 58
- Kumar Ashish, 281
- KZI, 251

L

- laboratorium sprzętowe, 320
- Last modified, 172
- library build target, 54
- lista
 - celów i wyników, 98
 - możliwości, 138
 - akcja, 138
 - grupa możliwości, 138
 - sposób formułowania, 138
 - zmian, 66, 78
 - długość, 66
 - punkty kontrolne, 66
 - złoty, 68
- LZ, 66

Ł

- łatwa automatyzacja, 23
- łączenie testów, 271

M

- macierz automatyzacji testów sprzętowych
 - Chrome OS, 313
- małe testy, 44, 45, 80, 85
 - izolacja kodu, 81
 - konieczne symulacje, 81
 - mocne i słabe strony, 86

małe testy
 pokrycie kodu, 87
 wykorzystanie zasobów, 85
 zakres działania, 81

Mao Ted, 112

mapowanie, 107

MapReduce, 107

Mar Shelton, 277

Matrix, 113, 218

mechanizm
 20%, 255
 nasłuchujący, 247

Mehta Ankit, 258

memory leak, 110

metoda
 AddUrl, 71
 automatyzowania testów, 65
 dopasowania rozmytego, 338, 339
 rozwijania przez błędy, 116
 WSM, 341
 szczegółowość, 341
 szybkość, 341
 wartość praktyczna, 342
 wymowność, 342

mierniki ilościowe, 292

minimalizowanie ryzyka, 153

Mission Control, 307

młodszy specjalista, 307

mock, 44, 47
 małe testy, 87
 testy integracyjne, 63

model
 chodzenia za słońcem, 291
 dojrzałego potencjału, 95
 DOM
 atrybuty rodziców i dzieci, 219

Mondrian, 66

Mozilla Bugzilla, 169

możliwość testowania, 314

N

nadmiarowa praca, 333

nagroda koleżeńska, 53

najbliższy sąsiad, 44

narzędzia
 a oprogramowanie, 305
 Autotest, 274

budowania i wykonywania testów, 79

diagnostyczne, 289

Google Diagnostic Utility, 288

HUD, 318

kontrolne, 69

lokalizujące, 283

o otwartym kodzie
 Chrome OS, 314

oceniania stopnia pokrycia projektu testami, 288

pomiarowanie, widoczność i raportowanie, 283

programistyczne, 282

QualityBots, 336

RPF, 338

Selenium, 115

tworzenie, 285

w Chrome, 330

wpisy z bloga, 333

wykorzystujące mechanizm sprzężenia
 zwrotnego, 290

źródłowe, 282

Żniwiarz, 88

Native Driver, 310

New, 173

Norwitz Neal, 97

Not feasible, 172

Not repeatable, 173

Notes, 172

O

O duże, 109

obliczenia w chmurze, 95

Obsolete, 173

obszary zainteresowania, 40

ocena ryzyka, 144, 315
 Chrome OS, 313, 315
 cele, 315
 częstotliwość występowania błędów, 145
 błędy bardzo rzadkie, 145
 błędy częste, 146
 błędy okazjonalne, 146
 błędy rzadkie, 146

czynniki, 144

konsultacje, 148
 dyrektorzy i wiceprezesi, 149
 handlowcy, 149
 kierownicy projektów, 149
 programiści, 148
 zalety, 149

Test Analytics, 344
 uwagi, 153
 minimalizowanie ryzyka, 153
 wpływ błędów na komfort pracy, 147
 błąd niewielki, 147
 błąd znaczny, 147
 maksymalny, 147
 minimalny, 147
 zasięg błędów, 145
 ocenianie
 jakości całego internetu, 205
 kodu, 65
 ODW, 316
 odznaki certyfikacji w testach, 95
 off-by-one, 44
 ograniczenia rodzajów testów, 85
 ograniczenie ryzyka
 stopień ograniczenia ryzyka, 150
 ogromne testy, 83, 85
 zakres działania, 83
 okno incognito, 328
 opracowywanie kodu
 inżynier do spraw testowania
 oprogramowania, 50
 organizowanie wykonywanych ręcznie zadań
 testowych, 46
 osoba kierująca pracami nad projektem, 60
 osobliwość, 206
 otwarta baza kodu, 52
 otwartość kodu, 314
 OZ, 40

P

Page View, 137
 pakiet
 automatyzujący, 64
 regresyjny, 55
 testujący, 64
 panel testowania, 318
 PAT, 69
 peer bonus, 53
 plan testowania, 124
 dziesięciominutowy, 151
 historia wprowadzania, 126
 przypadki testowe, 128
 system weryfikacji kodu, 125
 systemu operacyjnego Chrome, 313

Test Analytics, 341
 testy, 128
 wymagania, 126
 planowanie automatyzacji, 64
 sposób przekazywania informacji o jakości
 kolejnych kompilacji, 65
 szkielet automatyzujący, 64
 pliki
 .jar, 221
 addurl.pb.cc, 72
 addurl.pb.h, 72
 addurl.proto, 71
 addurl_frontend.cc, 73
 addurl_frontend_test, 78
 addurl_frontend_test.cc, 74
 AddUrlFrontend, 72
 binarny, 55
 common_collections_util, 93
 testu, 55
 youtube_client, 94
 początek prac, 59
 podleganie testom, 136
 podział ról, 36
 inżynier
 do spraw testowania oprogramowania, 37
 oprogramowania, 37
 testujący, 38
 pokrycie kodu, 137, 288
 pole formularza zgłoszenia błędu, 170
 Blokowanie, 171
 Do wiadomości, 171
 Listy zmian, 171
 Mierzone w wersję, 174
 Odbiorca, 170
 Ostatnie zmiany, 172
 Postanowienia, 172
 Priorytet, 172
 Rodzaj, 174
 Składowa, 171
 Status, 173
 Streszczenie, 174
 Szkodliwość, 173
 Utworzono, 172
 Uwagi, 172
 Weryfikator, 174
 Zależny, 171
 Załączniki, 171

- pole formularza zgłoszenia błędu
 - Zgłoszone przez, 172
 - Zmiany, 171
 - Znaleziono w, 172
 - Zweryfikowana wersja, 174
- pomiarowanie, widoczność i raportowanie, 283
- poniejsze projekty, 51
- poprawianie testowania, 303
 - odpowiedzialność za prowadzenie testów, 307
 - przyszłość
 - infrastruktury testującej, 309
 - inżynierów do spraw testowania oprogramowania, 306
 - inżynierów testujących, 308
 - kierowników i dyrektorów zespołów testujących, 309
 - system pracy, 303
 - wnioski, 311
- poziom jakości, 199
- poznanie
 - swoich ludzi, 253
 - swojego produktu, 252, 307
- praca nad kodem, 50, 60
 - biblioteki, 52
 - dla platform systemowych, 53
 - budowanie nowych wersji kompilacji, 54
 - jedno repozytorium, 50
 - język specyfikacji wersji budowanej, 54
 - kompilator, 54
 - nowe rozwiązania, 52
 - ostrzejsze testy, 53
 - praktyki, 52
 - rozbieżności wynikające ze środowisk pracy, 53
 - sprawdzanie poprawności, 53
 - uzgodnienie interfejsów, 55
 - wspólny kod, 52
 - cechy, 52
 - zależności, 52
- Priority, 172
- procedury testowania, 29
 - Certyfikowany w testach, 95
 - innowacje produktów, 41
 - jakość ≠ testy, 35
 - kultura pracy, 58
 - łączenie pracy nad kodem i testowanie, 36
 - ograniczenie liczby pracowników, 253
 - plan automatyzowania procesów testowych, 64
 - płynne i częste zmienianie zespołów, 254
 - podział ról, 36
 - pracownicy a liczba usług, 55
 - proces przygotowywania kodu, 65
 - prostota i jednorodność, 54
 - rodzaje testów, 43
 - struktura organizacyjna, 39
 - swobodne zmienianie zespołów, 276
 - takie same środowisko pracy i zestaw narzędzi, 69
 - typowe przypadki, 71
 - używanie wspólnych fragmentów kodu, 52
 - wprowadzanie częstych aktualizacji, 45
 - zunifikowany system budowania nowych wersji kompilacji, 54
- proces
 - pisania aplikacji
 - narzędzia, 51
 - repozytorium, 50
 - uzgodnienie interfejsów, 55
 - złożone usługi, 55
 - przygotowywania kodu, 65, 66
 - testowania
 - narzędzia nad oprogramowanie, 305
 - niedogodności, 304
 - rozdzielenie ról programistów i testerów, 304
- Process, 174
- profil ryzyka, 122
- program
 - automatycznego testowania, 69
 - wyzwań testowych, 95
- programista, 37
 - kontakt z użytkownikiem, 308
 - pracujący nad funkcjami aplikacji, 49
 - testów, 49
 - użytkowy, 48, 49
 - zaangażowanie w kwestie testowania, 95
- programowanie
 - w sytuacji idealnej, 47
 - kod testowy, 48
- projekt, 62
 - automatyzacji stron korzystających z JavaScript, 221
 - BITE, 211
 - dokładny opis błędu, 216
 - dołączanie adresu URL do raportu o błędzie, 215

- formularz, 214
- informacje o systemie operacyjnym
 - i przeglądarce, 215
- kod HTML niepoprawnego fragmentu strony, 214
- przycisk Bug It, 214
- rozwiązywanie problemów, 212
- segregowanie błędów, 214
- zapis działań podjętych przez testera, 214
- zgłaszanie błędów, 213
- zrzut ekranu, 214
- BITE Web Test Framework, 222
- ryzyko, 121
- UX, 232
- zależny
 - zmiany, 94
- zwrot z inwestycji, 121
- projektanci testów, 309
- projektowanie
 - w sytuacji idealnej
 - potrzeby użytkownika, 48
 - z użyciem testów, 37
- prostota, 54
- proto_library, 72
- protokoły, 63
 - JSON, 164
 - SOAP, 164
- prrowadzenie darmowych testów, 230
 - schemat, 230
 - ocena błędów, 231
 - planowanie za pomocą GTA, 230
 - pokrycie testami, 230
 - segregowanie błędów i ich usuwanie, 231
 - testy eksploracyjne, 231
 - wdrożenie nowej wersji i powrót do kroku 1, 231
 - zgłaszanie błędów, 231
 - używanie botów, 232
 - w sieci, 231
 - warunki, 230
- przegląd wydajności projektowej, 285
- przeglądarka
 - Chrome
 - Chrome Labs, 147
 - mechanizm automatycznej aktualizacji, 147
 - przycisk Odśwież, 147
 - skielet automatyzujący SiteCompat, 209

- Firefox, 208
- WebKit, 206
- przekładanie możliwości na testy wskazówki, 139
- przesyłanie raportów, 46
- przewodnik po stylu programowania w C++, 66
- przygotowanie testów, 79
- przypadek
 - testowy, 124, 128, 139, 158
 - arkusz kalkulacyjny, 158
 - dokumentacja, 159
 - Google Test Case Manager, 159
 - powstawanie, 158
 - przygotowanie, 158
 - regresyjny
 - automatyzacja, 217
 - Test Scribe, 159
 - zarządzanie, 158
 - użytkowania, 138, 153
- przyszłość testów, 303
- przyznawanie awansów, 257
- PyAuto, 273
- Python App Engine, 233

Q

- QualityBots, 336
 - Amazon EC2, 336
 - frontend narzędzia, 336
 - interfejs, 336
 - panel kontrolny, 337
 - przykładowe wyniki uruchomienia, 337

R

- raport
 - z testu pokrycia, 87
 - w chmurze, 88
 - zbiorczy, 251
- RC, 43
- recenzja kodu, 65
- recenzowanie dokumentacji, 62
- Record Playback Framework, 338
- redukcja, 107
- reguły
 - kompilowania
 - proto_library, 72
 - przedwysyłkowe, 66
 - przygotowywania kompilacji, 92

- release channel, 42
- Remote Procedure Call, 71
- Reported by, 172
- repozytorium, 51
 - Issue Tracker, 169
 - kodów źródłowych, 51
 - ostrzejsze testy, 53
- Resolution, 172
- roczne oceny pracowników, 257
- rodzaje testów, 43, 84
 - cele i ograniczenia czasu wykonywania, 85
 - ograniczenia, 85
 - wykorzystanie zasobów, 85
- rozmiar testu, 80, 84
 - korzyści z różnych rozmiarów testów, 84
- rozmowa
 - kwalifikacyjna
 - na stanowisko inżyniera testującego, 183
 - z inżynierem do spraw testowania oprogramowania, 104
 - z dyrektorem testów w Google India Sujayem Sahnim, 286
 - z dyrektorem testów w projektach Search i Geo Sheltonem Marem, 277
 - z dyrektorem zespołu inżynierii narzędziowej Ashishem Kumarem, 281
 - z inżynierem testującym Lindsay Webster, 237
 - baza danych błędów, 239
 - moment zakończenia testów, 242
 - ocena stanu projektu, 238
 - określenie najważniejszych składowych programu, 239
 - sprawdzanie składowych aplikacji, 239
 - testowanie Google Sites, 240
 - testy automatyczne, 238
 - testy jednostkowe, 238
 - usuwanie usterek, 242
 - zestaw testów, 239
 - zrozumienie ogólnej koncepcji, 238
 - z inżynierem testującym pracującym przy serwisie YouTube Apple Chow, 244
 - algorytmy losujące zawartość serwisu YouTube, 249
 - błąd arkusza stylów CSS, 248
 - główny tester w Google, 246
 - koszty utrzymania i naprawiania testów, 250
 - największy błąd, 248

- ocena procesu weryfikacji kandydatów na stanowiska IT i ITO, 244
- praktyki testowe stosowane w Google, 245
- Selenium, 247
- testowanie aplikacji Flash serwisu YouTube, 248
- testy eksploracyjne, 246
- z Jamesem Whittakerem, 294
- z kierownikiem testów Bradem Greenem, 291
- z KZI projektu Chrome Joelem Hynoskim, 270
- z KZI usługi Gmail Ankitem Mehtą, 258
- z KZI zespołu Android Hungiem Dangiem, 265
- z twórcą aplikacji WebDriver, 114
- rozmowa międzynarodowa, 327
- rozszerzenie
 - bazy znanych błędów
 - Google Feedback, 318
- rozwiązanie luźnego wykonania, 219
- RPC, 71
- RPF, 218, 338
 - działanie na stronach internetowych, 340
 - ocena jakości, 338
 - wyniki testowania, 339
- Rufer Russ, 97
- rywalizacja, 293
- ryzyko, 144
 - ocena, 144
 - szacowanie na podstawie częstotliwości występowania błędów i ich zasięgu, 145
 - zmniejszanie, 149
- rzemieślnik, 329

S

- Sahni Sujay, 286
- scenariusze przekrojowe, 122
- Search, 277
- Selenium, 115, 208, 217
- serwery
 - VPN, 205
- sesja przeglądarki, 113
- Severity, 173
- shardowanie, 107
- Sharing, 137
- silnik WebKit, 169, 201
- SiteCompat, 209
- skrypty
 - automatyzujące WebDrive, 198
 - ciągłej kompilacji, 68

specyfikacja wersji testu, 79
 sprawdzanie
 działania pojedynczych modułów kodu, 80
 działania wszystkich funkcji jako całości, 83
 jakości działania, 289
 stopnia zintegrowania, 81
 wzajemnego oddziaływania ograniczonej liczby modułów aplikacji, 81
 stan zielony, 67
 Status, 173
 status odczytanych, 66
 Stewart Simon, 114
 stopień
 oczytania, 66
 ograniczenia ryzyka, 150
 pokrycia kodu, 288
 zintegrowania, 81
 street-food, 267
 Striebeck Mark, 97
 struktura
 organizacyjna, 39
 nadzorujący pracę, 40
 obszary zainteresowania, 40
 wypożyczanie testerów, 41
 w zespole, 59
 kierownik projektu, 60
 zespołu, 251
 Summary, 174
 swobodne zmienianie zespołów, 276
 swobodny przepływ pracowników, 254
 symulacyjne implementacje interfejsów, 55
 system
 BITE UX, 222
 budowania, 72
 nowych wersji kompilacji, 54
 Chrome, 220
 Chrome OS, 219
 funkcje połączeń sieciowych, 236
 integracji ciągłej, 90, 95
 analiza zależności, 91
 przerwanie testu, 91
 zwykły, 91
 kompilacji ciągłej, 68
 ochrona, 68
 kompilacji jednoczesnych, 92
 kontroli wersji
 lista złotych zmian, 68

Linux, 220
 pracy
 w Google, 303
 testowania, 83
 dowolna kolejność wykonywania, 89
 uruchamiany z testowaniem, 47
 szersza perspektywa, 61
 szkielet
 automatyzujący, 64
 automatyzujący SiteCompat, 209
 Record and Play, 219
 Record and Playback, 218
 Selenium, 220
 testowy, 65
 WebDrive, 220
 sztuczny system, 64

Ś

ścieżka XPath, 199
 parametry, 218, 219
 ścieżka zależności, 93
 średnie testy, 44, 45, 81, 85
 infrastruktura, 82
 mocne i słabe strony, 86
 pokrycie kodu, 87
 symulowanie zewnętrznych usług, 82
 wykorzystanie zasobów, 85
 zakres działania, 82

T

tabela możliwości, 137
 para podgląd strony - dzielenie, 138
 podgląd strony, 137
 wartości liczbowe, 137
 tablica testów jednostkowych, 67, 69
 tajne projekty, 310
 Targeted To, 174
 TDD, 37
 techniki nagrywania, 46
 Test Analytics, 131, 341, 342
 kreator, 342
 ocena ryzyka, 153
 odnośniki do baz danych błędów
 i przypadków testowych, 344
 określanie właściwości projektu, 342
 tabela oceny ryzyka, 343

Test Analytics

Test Results, 344

wyświetlanie

możliwości projektu, 343

ocen ryzyka w tabeli zawierającej spis

właściwości i składowych projektu, 343

test harness, 47

Test Scribe, 159

SOAP, 164

TEST_F, 76

test-driven design, 37

testerzy, 37

dobrowolni, 308

kontekst wystąpienia błędu, 212

praca podczas przygotowywania aplikacji, 56

przydział do projektów, 40

w Google, 297, 304

w strukturze organizacyjnej, 40

wczesny etap projektowania, 58

wewnętrzni, 39, 154, 305

wpływ na kształt produktu, 256

wypożyczanie, 41

zewnątrzni, 235

testowalność, 65

testowanie oprogramowania

a jakość oprogramowania, 35

a prace nad produktem, 35

cel, 303

cele testowanego systemu, 280

czynnik hamujący, 33

dbanie o jakość oprogramowania, 34

inicjatywa własna pracowników, 57

jedno repozytorium, 51

jednorodna platforma programistyczna, 53

mały zespół testujący, 34

ograniczenie liczby pracowników, 253

podwójne sprawdzenie sprawności

programistów, 39

pokrycie kodu, 87

relacje funkcji, 44

rozwiązania dostępne w chmurze, 310

scenariusze

faktycznego użytkownika, 44

przekrojowe, 122

sposoby, 32

stabilny rozwój aplikacji, 43

testowanie wieloosobowe, 291

w Google, 30, 303

w tempie i na skalę Google, 90

wskazówki do przygotowania narzędzia, 114

wyniki Google, 33

z poziomu sieci, 310

z punktu widzenia użytkownika, 38

z uwzględnieniem potrzeb użytkownika, 119

zapobieganie występowaniu błędów, 36

zautomatyzowane procedury, 46

zespół, 50

zmiany w Google, 292

zwiększenie wykorzystania dobrych procedur, 41

Testowanie w toalecie, 99

testowanie w trybie utrzymania, 193

przykład Google Desktop, 193

testowy kanał dystrybucji, 43

testy

automatyczne, 46, 65, 264, 267

cele i ograniczenia czasu wykonywania, 85

cuke, 159

czułe na wprowadzone zmiany, 93

dla listy zmian, 66

dokumentacja, 269

dostarczanie danych, 314

duże, 44, 83, 122

mocne i słabe strony, 85

ograniczenie, 85

pokrycie kodu, 87

wykorzystanie zasobów, 85

zakres działania, 83

dymne, 194

dzienne

ostatniej dobrej wersji, 316

eksploracyjne, 137, 153, 156, 198, 204, 268

funkcjonalne, 110

globalne, 83

infrastruktura zewnętrzna, 47

integracyjne, 81

na wczesnym etapie prac, 64

jako funkcja programu, 55, 56

jednostkowe, 80

tablica, 67

język testów, 310

konflikty, 89

korzyści z różnych rodzajów testów, 84

likwidacja przywiązania do konkretnych

zasobów, 89

logika testów, 217
 małe, 44, 80, 122
 izolacja kodu, 81
 konieczne symulacje, 81
 mocne i słabe strony, 86
 ograniczenie czasu, 85
 pokrycie kodu, 87
 w środowiskach próbnych, 44
 wykorzystanie zasobów, 85
 zakres działania, 81
 niedeterministyczne, 86
 niezależność, 88
 o otwartym kodzie
 Chrome OS, 314
 obciążenia, 264
 odpowiedzialność za prowadzenie, 307
 ogromne, 83
 ograniczenie czasu, 85
 zakres działania, 83
 optymalizacja liczby testów po zmianie, 94
 pierwszy przebieg testów, 208
 podstawowe
 w kolejnych wersjach kompilacji, 315
 pokrycia, 87
 prowadzenie darmowych testów, 230
 przekrojowe scenariusze, 45
 przygotowanie, 79
 przyszłych funkcji, 47
 regresywne, 153, 198, 204
 ręczne, 46, 153, 204, 280
 kontra automatyczne, 317
 rozważa, 268
 zautomatyzowane procedury, 46
 rozmiary, 80
 sens stosowania, 79
 specyfikacja wersji, 79
 strukturalne, 110
 systemowe, 83
 średnie, 44, 81, 122
 infrastruktura, 82
 mocne i słabe strony, 86
 ograniczenie czasu, 85
 pokrycie kodu, 87
 symulowanie zewnętrznych usług, 82
 wykorzystanie zasobów, 85
 zakres działania, 82
 testowanie w trybie utrzymania, 193

 w fazie eksperymentalnej, 58
 w przeglądarce, 321
 wersji przeznaczonej do wprowadzenia
 na rynek, 316
 wewnętrzne, 305
 wielokrotne, 139
 wieloosobowe, 156, 204
 korzyści, 156
 trudności, 157
 wykorzystanie
 w infrastrukturze dzielonej, 83
 zasobów, 85
 wymagania stawiane czasom wykonywania, 88
 ToTT, 99
 tworzenie
 kodu, 66
 oprogramowania, 304
 proces tworzenia, 304
 Type, 174

U

udział użytkowników
 Chrome OS, 317
 udzielający się, 65
 uruchomienie
 pod pełnym obciążeniem, 316
 usługa
 zdalnego wywołania procedury, 71
 UX, 313
 użytkownik
 z grupy dogfooder, 154

V

Verified, 174
 Verified In, 174
 Verifier, 174
 Verifier assigned, 173
 Vevo, 246
 VPN, 205

W

wartość testowania, 305
 warunek
 kolejności testowania, 89
 niezależności testów, 88

- Wave, 116
- wczesna wersja towaru, 42
- wczesny etap projektowania, 57
 - dwudziestoprocentowy prototyp, 58
 - eksperymentowanie, 58
 - testerzy, 58
 - zachowawczość, 58
- WebDrive, 198
- WebDriver, 114, 208
- WebKit, 169
- wersja
 - beta, 42, 43
 - deweloperska, 43
 - kanarkowa, 42, 210
 - kompilacji, 54
 - testowa, 43
- weryfikacja
 - automatyczna, 44
 - błędów o priorytecie P0, 316
 - rozwijanej aplikacji, 128
 - scenariuszy, 316
- Wi-Fi, 235
- Will not fix, 173
- wirtualizacja, 318
- Works as intended, 173
- wpisy z bloga, 333
- wpływ, 256
 - roczne oceny pracowników, 257
- wprowadzanie częstych aktualizacji, 45
- wrapper, 107
- WSM, 341
- wspólny kod, 52
 - cechy, 52
 - funkcje a różnice między platformami, 53
 - nowe rozwiązania, 52
 - rozbieżności wynikające ze środowisk pracy, 53
 - sprawdzanie poprawności, 53
 - zależności, 52
- współdzielona biblioteka
 - zmiany, 93
- współpraca międzypespółowa, 257
- wstrzykiwanie
 - błędów, 63
 - skryptu JavaScript, 207
- wszechstronna automatyzacja, 64
- WTF, 222
- wycieczki testowe dla Chrome, 325
 - bieg na orientację, 328
 - sugerowane obszary prowadzenia badań, 328
 - zastosowanie, 328
- do białego rana, 329
 - sugerowane obszary prowadzenia badań, 329
- kiepska okolica, 330
 - nieciekawe dzielnice przeglądarki, 330
- rozmowa międzynarodowa, 327
 - sugerowane obszary prowadzenia badań, 327
 - zastosowanie, 327
- rzemieślnik, 329
 - narzędzia w Chrome, 330
 - zastosowanie, 329
- ustawienia własne, 331
 - sposoby modyfikowania przeglądarki, 331
- wyprawa do sklepu, 325
 - zastosowanie, 326
- wyprawa studenta, 326
 - sugerowane obszary prowadzenia badań, 327
 - zastosowanie, 326
- wykonywanie testu, 79
 - infrastruktura, 79
 - losowość, 89
 - system Google, 84
 - środowisko, 88
- wykrywanie opóźnień w działaniu wersji
 - produkcyjnej, 290
- wymagania testowe projektów, 88
- wyprawa
 - do sklepu, 325
 - studenta, 326
- wywiad
 - z programistą narzędzi, 112
 - z twórcami programu Certyfikowany w testach, 97

Y

- YouTube, 246
 - Ads, 247
 - Watch, 246
- youtube_client, 94

Z

zachowawczość, 58
zależności
 ścieżka, 93
 wewnątrz kompilacji, 92
zamrażanie kodu, 67
zdalne programowanie w parach, 284
zdobywanie pracowników i pomysłów, 254
zespół
 GEO, 216
 inżynierii narzędziowej, 282
 grupy narzędzi, 282
 ocena projektu, 283
 projekty eksperymentalne, 283
testujący, 30, 34
 Chrome OS, 317
 fanty, 276
 kod, 51
 podział ról, 36
 poza zespołem programistów, 305
 współpraca, 278
 zadania, 314
wydajności projektowej, 286

zestaw możliwości
 przekształcanie w historię użytkownika, 138
zewnętrzna maszyna wirtualna, 208
złożone usługi, 55
zmiany
 odmiana regresywna, 203
 w projekcie zależnym, 94
 we współdzielonej bibliotece, 93
zmienianie zespołów, 254
zmniejszanie ryzyka, 149
 procedury, 150
 kod wartowniczy, 150
 narzędzia, 150
 określenie bezpiecznej ścieżki
 postępowania, 150
 przypadki testów regresywnych, 150
 testy, 150
znacznik
 IFRAME, 203
 o zbliżonej zgodności, 219

Ż

Żniwiarz, 88
życie błędu, 164

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

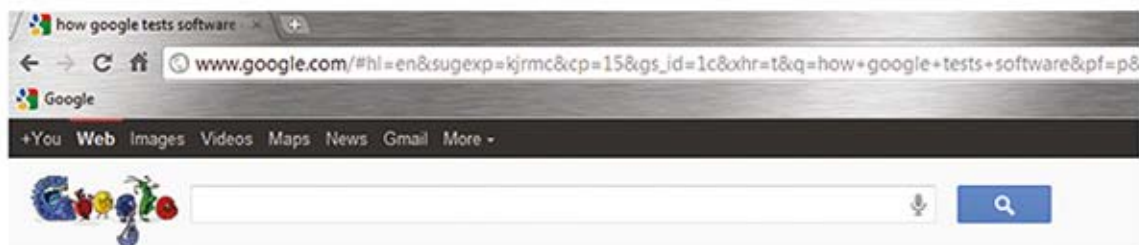
Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA

 **Helion SA**



Sprawdź, jak testują najlepsi!

Oprogramowanie firmy Google to miliony linii kodu źródłowego, dziesiątki wersji językowych, różne systemy operacyjne, przeglądarki i preferencje użytkownika. Jak przy takich wymogach dostarczyć klientom produkt najwyższej jakości? Tu mogą pomóc tylko testy automatyczne. Dzięki nim codziennie bez trudu można uruchomić miliony testów! Google opanowała tę sztukę do mistrzostwa. Warto uczyć się od najlepszych!

Dzięki tej książce dowiesz się, jak zorganizować proces testowania, tak aby był elastyczny, skuteczny i spełniał Twoje oczekiwania. Poznasz rolę inżyniera do spraw testowania oprogramowania, kierownika zespołów inżynierskich oraz inżyniera testującego. Zobaczysz, na jakie problemy natykają się oni każdego dnia oraz jak sobie z nimi radzą. Ponadto nauczysz się oceniać ryzyko, dokumentować proces testowania i raportować błędy. Książka ta jest obowiązkową lekturą dla wszystkich osób, które doskonalą swoje umiejętności programistyczne i chcą polepszyć jakość dostarczanego oprogramowania.

Dzięki tej książce:

- poznasz najlepsze metody zapewniania jakości oprogramowania
- nauczysz się planować i przeprowadzać testy
- poprawnie ocenisz ryzyko
- udoskonalisz proces wytwarzania oprogramowania!

helion.pl
księgarnia
internetowa

Nr katalogowy: 18734



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu



Helion

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nawosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>



ISBN 978-83-246-8656-8



cena: 67,00 zł