Adam Roman, Lucjan Stapp, Michaël Pilaeten

Certyfikowany tester ISTQB®

POZIOM PODSTAWOWY

Wydanie II

Podręcznik do samodzielnej nauki na podstawie sylabusa w wersji 4.0

- Poznaj treść sylabusa
- Opanuj wymagane definicje
- Odpowiedz na ponad 70 pytań testowych
- Wykonaj 14 oryginalnych ćwiczeń
- Zdaj przykładowy egzamin
- Sprawdź, czy Twoje odpowiedzi są poprawne

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

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

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

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/ctisp2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-0439-2

Copyright © Adam Roman, Lucjan Stapp, Michaël Pilaeten 2024

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wprowadzenie **7**

O autorach **11**

CZĘŚĆ I. Certyfikat, sylabus i egzamin poziomu podstawowego **13**

Certyfikat poziomu podstawowego **15**
Okoliczności powstania i historia certyfikatu podstawowego 15
Ścieżki kariery dla testerów 15
Docelowi odbiorcy 17
Cele certyfikatu podstawowego 17
Cele międzynarodowego systemu uzyskiwania kwalifikacji 17
Cele biznesowe 18
Cele nauczania 19
Wymagania stawiane kandydatom 20

Sylabus i egzamin poziomu podstawowego **21**
Odniesienia do norm i standardów 21
Ciągła aktualizacja 21
Nota wydania dla sylabusa w wersji 4.0 22
Zawartość sylabusa 22
Struktura egzaminu 27
Reguły egzaminu 28
Rozkład pytań na egzaminie 29
Wskazówki — przed egzaminem i w jego trakcie 31

CZĘŚĆ II. Omówienie treści sylabusa **33**

ROZDZIAŁ 1. **Podstawy testowania** **35**
1.1. Co to jest testowanie? 37
1.1.1. Cele testów 38
1.1.2. Testowanie a debugowanie 39
1.2. Dlaczego testowanie jest niezbędne? 41
1.2.1. Znaczenie testowania dla powodzenia projektu 44
1.2.2. Testowanie a zapewnienie jakości 45
1.2.3. Pomyłki, defekty, awarie i podstawowe przyczyny 47
1.3. Zasady testowania 51
1.4. Czynności testowe, testalia i role związane z testami 57
1.4.1. Czynności i zadania testowe 57
1.4.2. Proces testowy w kontekście 63
1.4.3. Testalia 64
1.4.4. Śledzenie powiązań między podstawą testów a testaliami 71
1.4.5. Role w procesie testowania 72

Kup ksi k   Pole ksi k

1.5. Niezbędne umiejętności i dobre praktyki w dziedzinie testowania ... 75
1.5.1. Ogólne umiejętności wymagane w związku z testowaniem 75
1.5.2. Podejście „cały zespół” ... 77
1.5.3. Niezależność testowania ... 78
Pytania testowe do rozdziału 1. ... 80

ROZDZIAŁ 2. **Testowanie w cyklu wytwarzania oprogramowania ... 87**
2.1. Testowanie w kontekście modelu cyklu wytwarzania oprogramowania ... 88
2.1.1. Wpływ cyklu wytwarzania oprogramowania na testowanie 89
2.1.2. Model cyklu wytwarzania oprogramowania a dobre praktyki testowania ... 99
2.1.3. Testowanie jako czynnik określający sposób wytwarzania oprogramowania ... 100
2.1.4. Metodyka DevOps a testowanie ... 105
2.1.5. Przesunięcie w lewo ... 108
2.1.6. Retrospektywy i doskonalenie procesów ... 109
2.2. Poziomy testów i typy testów ... 111
2.2.1. Poziomy testów ... 111
2.2.2. Typy testów ... 127
2.2.3. Testowanie potwierdzające i testowanie regresji ... 137
2.3. Testowanie pielęgnacyjne ... 140
Pytania testowe do rozdziału 2. ... 143

ROZDZIAŁ 3. **Testowanie statyczne ... 147**
3.1. Podstawy testowania statycznego ... 148
3.1.1. Produkty pracy badane metodą testowania statycznego ... 149
3.1.2. Korzyści wynikające z testowania statycznego ... 150
3.1.3. Różnica między testowaniem statycznym a dynamicznym 154
3.2. Informacje zwrotne i proces przeglądu ... 157
3.2.1. Korzyści wynikające z wczesnego i częstego otrzymywania informacji zwrotnych od interesariuszy ... 157
3.2.2. Czynności wykonywane w procesie przeglądu ... 159
3.2.3. Role i obowiązki w przeglądach ... 164
3.2.4. Typy przeglądów ... 166
3.2.5. Czynniki sukcesu przeglądów ... 176
3.2.6. (*) Techniki przeglądu ... 178
Pytania testowe do rozdziału 3. ... 182

ROZDZIAŁ 4. **Analiza i projektowanie testów ... 187**
4.1. Ogólna charakterystyka technik testowania ... 188
4.1.1. Kategorie technik testowania i ich cechy charakterystyczne 189
4.2. Czarnoskrzynkowe techniki testowania ... 195
4.2.1. Podział na klasy równoważności (KR) ... 195
4.2.2. Analiza wartości brzegowych (AWB) ... 206
4.2.3. Testowanie w oparciu o tablicę decyzyjną ... 212
4.2.4. Testowanie przejść pomiędzy stanami ... 218
4.2.5. (*) Testowanie oparte na przypadkach użycia ... 227
4.3. Białoskrzynkowe techniki testowania ... 231
4.3.1. Testowanie instrukcji i pokrycie instrukcji kodu ... 232
4.3.2. Testowanie gałęzi i pokrycie gałęzi ... 234
4.3.3. Korzyści wynikające z testowania białoskrzynkowego ... 238

Kup ksi k Pole ksi k

4.4. Techniki testowania oparte na doświadczeniu 240
4.4.1. Zgadywanie błędów 240
4.4.2. Testowanie eksploracyjne 243
4.4.3. Testowanie w oparciu o listę kontrolną 246
4.5. Podejścia do testowania oparte na współpracy 250
4.5.1. Wspólne pisanie historyjek użytkownika 250
4.5.2. Kryteria akceptacji 253
4.5.3. Wytwarzanie sterowane testami akceptacyjnymi (ATDD) 255
Pytania testowe do rozdziału 4. 258
Ćwiczenia do rozdziału 4. 268

ROZDZIAŁ 5. **Zarządzanie czynnościami testowymi 273**
5.1. Planowanie testów 274
5.1.1. Cel i treść planu testów 275
5.1.2. Wkład testera w planowanie iteracji i wydań 279
5.1.3. Kryteria wejścia i kryteria wyjścia 280
5.1.4. Techniki szacowania 282
5.1.5. Ustalanie priorytetów przypadków testowych 291
5.1.6. Piramida testów 296
5.1.7. Kwadranty testowe 297
5.2. Zarządzanie ryzykiem 299
5.2.1. Definicja i atrybuty ryzyka 300
5.2.2. Ryzyka projektowe i produktowe 301
5.2.3. Analiza ryzyka produktowego 303
5.2.4. Kontrola ryzyka produktowego 306
5.3. Monitorowanie testów, nadzór nad testami i ukończenie testów 309
5.3.1. Metryki stosowane w testowaniu 309
5.3.2. Cel, treść i odbiorcy raportów z testów 310
5.3.3. Przekazywanie informacji o statusie testowania 313
5.4. Zarządzanie konfiguracją 315
5.5. Zarządzanie defektami 317
Pytania testowe do rozdziału 5. 320
Ćwiczenia do rozdziału 5. 326

ROZDZIAŁ 6. **Narzędzia testowe 329**
6.1. Narzędzia wspomagające testowanie 329
6.2. Korzyści i ryzyka związane z automatyzacją testów 331
Pytania testowe do rozdziału 6. 332

CZĘŚĆ III. Odpowiedzi i rozwiązania 335

ROZDZIAŁ 7. **Odpowiedzi do pytań testowych 337**
Rozdział 1. 337
Rozdział 2. 342
Rozdział 3. 345
Rozdział 4. 348
Rozdział 5. 358
Rozdział 6. 363

ROZDZIAŁ 8. **Odpowiedzi do ćwiczeń 365**
Rozdział 4. 365
Rozdział 5. 376

Kup ksi k Pole ksi k

CZĘŚĆ IV. Oficjalny przykładowy egzamin 379

Egzamin 381

Dodatkowe przykładowe pytania 399

Egzamin — odpowiedzi 411

Dodatkowe przykładowe pytania — odpowiedzi 427

Bibliografia 437

Źródła internetowe 442

Skorowidz 443

Wprowadzenie

Cel podręcznika

Niniejszy podręcznik jest skierowany do osób przygotowujących się do egzaminu ISTQB® Certyfikowany Tester — Poziom Podstawowy w oparciu o nowy sylabus poziomu podstawowego (wersja 4.0). Naszym celem było dostarczenie kandydatom rzetelnej wiedzy na bazie tego dokumentu. Z doświadczenia wiemy bowiem, że w internecie można znaleźć mnóstwo informacji na temat sylabusów i egzaminów ISTQB®, ale duża ich część jest niestety złej jakości. Zdarza się nawet, że materiały znalezione w sieci zawierają poważne błędy. Ponadto, ze względu na istotne zmiany, jakie zaszły w sylabusie w stosunku do poprzedniej wersji (3.1.1), ilość dostępnych dla kandydatów materiałów opartych na nowym sylabusie jest wciąż niewielka.

Podręcznik rozszerza i uszczegóławia wiele kwestii, które w samym sylabusie opisane są zdawkowo lub ogólnie. Zgodnie z wytycznymi ISTQB® dla szkoleń opartych na sylabusie dla każdego celu nauczania na poziomie K3 należy przeprowadzić ćwiczenie, a dla każdego celu na poziomie K2 — podać praktyczny przykład. Czyniąc zadość tym wymogom, przygotowaliśmy ćwiczenia oraz przykłady dla wszystkich celów nauczania na tych poziomach. Ponadto dla każdego celu nauczania przedstawiamy jedno lub kilka pytań testowych, podobnych do tych, na jakie kandydat będzie odpowiadał na egzaminie. Dzięki temu podręcznik stanowi znakomitą pomoc w nauce, przygotowaniu się do egzaminu oraz weryfikacji nabytej wiedzy.

Struktura podręcznika

Podręcznik składa się z czterech części.

Część I — certyfikat, sylabus i egzamin poziomu podstawowego

Część I podręcznika zawiera oficjalne informacje dotyczące treści i układu sylabusa oraz egzaminu ISTQB® Certyfikowany Tester — Poziom Podstawowy. Omawiana jest w niej także struktura certyfikacyjna ISTQB®. W tej części wyjaśniamy również podstawowe pojęcia techniczne, na których oparta jest budowa sylabusa oraz egzaminu. Wyjaśniamy, czym są cele nauczania, poziomy K oraz jakie są zasady budowy oraz przeprowadzania rzeczywistego egzaminu. Warto zapoznać się z tymi kwestiami, ponieważ ich zrozumienie pomoże kandydatowi o wiele lepiej przygotować się do egzaminu.

Kup ksi k Pole ksi k

Część II — omówienie treści sylabusa

Część II to zasadnicza część podręcznika. Omawiamy tu szczegółowo wszystkie treści i cele nauczania sylabusa poziomu podstawowego. Część ta składa się z sześciu rozdziałów, odpowiadających sześciu rozdziałom sylabusa. Każdy cel nauczania na poziomie K2 ilustrowany jest praktycznym przykładem, a każdy cel na poziomie K3 — zadaniem do samodzielnego wykonania.

Na początku każdego rozdziału podano definicje **słów kluczowych** obowiązujących w danym rozdziale. Każde słowo kluczowe, w miejscu jego pierwszego, istotnego użycia w tekście, zaznaczone jest wytłuszczoną czcionką oraz ikonką książki.

Na końcu każdego rozdziału Czytelnik znajdzie przykładowe pytania testowe pokrywające wszystkie cele nauczania zawarte w danym rozdziale sylabusa. Podręcznik zawiera **70 oryginalnych pytań testowych**, pokrywających wszystkie cele nauczania, a także **14 ćwiczeń odpowiadających celom nauczania na poziomie K3**. Te pytania i ćwiczenia nie występują w oficjalnych materiałach ISTQB®, ale są skonstruowane z zachowaniem zasad i reguł obowiązujących przy ich tworzeniu w przypadku rzeczywistych egzaminów. Są więc dodatkowym materiałem dla Czytelnika, pozwalając mu zweryfikować swoją wiedzę po przeczytaniu każdego rozdziału i lepiej zrozumieć przedstawiony materiał.

Tekst w ramce oznacza materiał nadobowiązkowy. Odnosi się on do treści sylabusa, ale wykracza poza treści ujęte w samym sylabusie i nie podlega egzaminowaniu. Jest to materiał „dla ciekawskich".

Rozdziały oznaczone gwiazdką (*) są nadobowiązkowe. Obejmują materiał, który obowiązywał na egzaminie według starej wersji sylabusa. Zdecydowaliśmy się na pozostawienie tych rozdziałów w książce ze względu na ich ważność i praktyczne zastosowanie. Czytelnik korzystający z podręcznika wyłącznie w celu nauki do egzaminu może podczas lektury te rozdziały pominąć.

Część III — odpowiedzi i rozwiązania

W części III podajemy rozwiązania do wszystkich przykładowych pytań i ćwiczeń zamieszczonych w części II podręcznika. Rozwiązania te nie ograniczają się wyłącznie do podania poprawnych odpowiedzi, ale zawierają również ich uzasadnienia. Pomogą one Czytelnikowi lepiej zrozumieć sposób tworzenia prawdziwych pytań egzaminacyjnych i lepiej przygotować się do ich rozwiązywania podczas prawdziwego egzaminu.

Część IV — oficjalne przykładowe egzaminy i pytania

Ostatnia, IV część podręcznika zawiera oficjalny przykładowy egzamin ISTQB® dla certyfikatu Poziom Podstawowy, dodatkowe pytania pokrywające niepokryte w egzaminie cele nauczania oraz informację o poprawnych odpowiedziach i uzasadnienia tych odpowiedzi.

Kup ksi k Pole ksi k

Podręcznik zbudowany jest więc tak, aby wszystkie niezbędne informacje:

- struktura i zasady przeprowadzania egzaminu,
- treści omawiane w sylabusie wraz z ich wyczerpującym omówieniem i przykładami,
- definicje pojęć, których znajomość obowiązuje na egzaminie,
- przykładowe, oryginalne pytania testowe i ćwiczenia, wraz z poprawnymi odpowiedziami i ich uzasadnieniem,
- oficjalny, przykładowy egzamin ISTQB® wraz z poprawnymi odpowiedziami i ich uzasadnieniem

były dostępne dla kandydata w jednym miejscu. Mamy nadzieję, że prezentowany w niniejszej publikacji materiał pomoże wszystkim osobom zainteresowanym uzyskaniem certyfikatu ISTQB® Certyfikowany Tester — Poziom Podstawowy.

Kup ksi k Pole ksi k

ROZDZIAŁ 1.

Podstawy testowania

Słowa kluczowe

analiza testów — czynność polegająca na identyfikowaniu warunków testowych w wyniku analizy podstawy testów.

awaria — zdarzenie, w którym moduł lub system nie wykonuje wymaganej funkcji w określonym zakresie.

cel testów — przyczyna lub powód testowania.

dane testowe — dane niezbędne do wykonania testów.

debugowanie — proces wyszukiwania, analizowania i usuwania przyczyn awarii w module lub systemie.

defekt — niedoskonałość lub wada produktu pracy, polegająca na niespełnieniu wymagań.

implementacja testów — czynność polegająca na przygotowaniu testaliów potrzebnych do wykonania testów, oparta na analizie i projektowaniu testów.

jakość — stopień, w jakim moduł lub system spełnia określone lub domniemane potrzeby klienta lub użytkownika.

monitorowanie testów — czynność polegająca na sprawdzaniu statusu aktywności testowych, identyfikowaniu odchylenia od planu lub oczekiwanego statusu oraz raportowaniu statusu do interesariuszy.

nadzór nad testami — czynność, podczas której rozwija i stosuje się działania naprawcze, aby utrzymać w toku testy projektu, gdy odbiegają one od tego, co zostało zaplanowane.

planowanie testów — czynność tworzenia planów testów lub wprowadzania do nich zmian.

podstawa testów — zasób wiedzy używany jako podstawa do analizy i projektowania testów.

podstawowa przyczyna — przyczyna defektu, która — gdy zostanie wyeliminowana — wystąpienie tego typu defektu redukuje lub usuwa.

Kup ksi k Pole ksi k

pokrycie — stopień, w jakim określone elementy pokrycia zostały określone lub sprawdzone przez zestaw testowy, wyrażony w procentach. *Synonim*: pokrycie testowe.

pomyłka — działanie człowieka powodujące powstanie nieprawidłowego rezultatu. *Synonim*: błąd.

procedura testowa — sekwencja przypadków testowych w kolejności wykonywania oraz wszelkie powiązane działania, które mogą być wymagane do ustanowienia warunków wstępnych i wszelkich czynności podsumowujących po wykonaniu.

projektowanie testów — czynność wyprowadzania i specyfikowania przypadków testowych z warunków testowych.

przedmiot testów — moduł lub system podlegający testowaniu.

przypadek testowy — zbiór warunków wstępnych, danych wejściowych, akcji (w stosownych przypadkach), oczekiwanych rezultatów i warunków końcowych opracowany w oparciu o warunki testowe.

testalia — produkty prac stworzone w ramach procesu testowego, używane do planowania, projektowania, wykonywania, oceny i raportowania testów.

testowanie — proces składający się ze wszystkich czynności cyklu wytwarzania, zarówno statycznych, jak i dynamicznych, skoncentrowany na planowaniu, przygotowaniu i ocenie oprogramowania oraz powiązanych produktów w celu określenia, czy spełniają one wyspecyfikowane wymagania, na wykazaniu, że są one dopasowane do swoich celów, oraz na wykrywania usterek.

ukończenie testów — czynność obejmująca udostępnianie testaliów dla późniejszego użycia, pozostawianie środowisk testowych w zadowalającym stanie i komunikowanie wyników testowania odpowiednim interesariuszom.

walidacja — sprawdzanie poprawności i dostarczenie obiektywnego dowodu, że produkt procesu wytwarzania oprogramowania spełnia potrzeby i wymagania użytkownika.

warunek testowy — testowalna własność modułu lub systemu zidentyfikowana jako podstawa do testowania. *Synonimy*: wymaganie testowe, sytuacja testowa.

weryfikacja — sprawdzenie poprawności i dostarczenie obiektywnego dowodu, że produkt procesu wytwarzania oprogramowania spełnia zdefiniowane wymagania.

wykonywanie testu — czynność polegająca na przeprowadzeniu testu modułu lub systemu, by otrzymać rzeczywiste wyniki.

wynik testu — konsekwencja/wynik wykonania testu.

zapewnienie jakości — działania skoncentrowane na zapewnieniu, że wymagania jakościowe będą spełnione.

Kup ksi k Pole ksi k

1.1. Co to jest testowanie?

FL-1.1.1 (K1)	Kandydat wskazuje typowe cele testów.
FL-1.1.2 (K2)	Kandydat odróżnia testowanie od debugowania.

W obecnych czasach nie ma chyba dziedziny życia, w której nie używałoby się w mniejszym lub większym stopniu oprogramowania. Systemy informatyczne odgrywają coraz większą rolę w naszym życiu, począwszy od rozwiązań dla biznesu (sektor bankowy, ubezpieczenia), a kończąc na urządzeniach dla konsumenta (samochody), rozrywce (gry komputerowe) czy komunikacji. Używanie oprogramowania zawierającego defekty może:

- spowodować utratę pieniędzy lub czasu;
- spowodować utratę zaufania klientów;
- utrudnić zdobycie nowych klientów;
- wyeliminować z rynku;
- w sytuacjach skrajnych — spowodować zagrożenie zdrowia lub życia.

Testowanie oprogramowania umożliwia ocenę jakości oprogramowania i przyczynia się do zmniejszenia ryzyka jego awarii w działaniu. Dlatego dobre testowanie jest niezbędne dla powodzenia projektu. Testowanie oprogramowania to zestaw czynności przeprowadzanych w celu ułatwienia wykrywania defektów i oceny właściwości artefaktów oprogramowania. Te testowane artefakty są znane jako **przedmiot testów** (ang. *test object*).

Wiele osób, w tym pracujących w branży IT, za testowanie uważa mylnie tylko wykonywanie testów, to jest uruchamianie oprogramowania w celu odszukania defektów. Jednak wykonywanie testów stanowi tylko część testowania. Istnieją jeszcze inne czynności związane z testowaniem. Występują one zarówno przed (punkty 1. – 5. poniżej) wykonaniem testów, jak i po nim (punkt 7. poniżej). Są to:

1. Planowanie testów.
2. Monitorowanie testów i nadzór nad testami.
3. Analiza testów.
4. Projektowanie testów.
5. Implementacja testów.
6. Wykonywanie testów.
7. Ukończenie testów.

Czynności testowe są zorganizowane i przeprowadzane w różny sposób w różnych cyklach wytwarzania oprogramowania (ang. *Software Development LifeCycle*, SDLC) (patrz rozdział 2.). Co więcej, często testowanie jest postrzegane jako czynność skupiona wyłącznie na **weryfikacji** (ang. *verification*) wymagań, historyjek użytkownika lub innych form specyfikacji (tzn. sprawdzeniu, czy system spełnia wyspecyfikowane wymagania).

Kup książkę Poleć książkę

Ale w ramach testowania przeprowadza się również **walidację** (ang. *validation*) — czyli sprawdzenie, czy system spełnia wymagania użytkowników oraz inne potrzeby interesariuszy w swoim środowisku operacyjnym.

Testowanie może wymagać uruchomienia testowanego modułu lub systemu — mamy wtedy do czynienia z tzw. testowaniem dynamicznym. Można również wykonywać testy bez uruchamiania testowanego obiektu — takie testowanie nazywa się testowaniem statycznym. Testowanie obejmuje więc również przeglądy produktów pracy takich jak:

- wymagania,
- historyjki użytkownika,
- kod źródłowy.

Testowanie statyczne bardziej szczegółowo opisane jest w rozdziale 3. Testowanie dynamiczne wykorzystuje różne rodzaje technik testowych (np. czarnoskrzynkowe, białoskrzynkowe i oparte na doświadczeniu) do wyprowadzania przypadków testowych i jest szczegółowo opisane w rozdziale 4.

Testowanie to nie tylko czynność techniczna. Proces testowy musi być również odpowiednio zaplanowany, zarządzany, szacowany, monitorowany i kontrolowany (patrz rozdział 5.). Testerzy w swojej codziennej pracy intensywnie korzystają z różnego rodzaju narzędzi (patrz rozdział 6.), ale należy pamiętać, że testowanie jest w dużej mierze czynnością intelektualną, wymagającą od testerów specjalistycznej wiedzy, umiejętności analitycznych oraz myślenia krytycznego i systemowego [Myers 2011, Roman 2018].

Norma ISO/IEC/IEEE 29119-1 zawiera dodatkowe informacje na temat koncepcji testowania oprogramowania.

Warto pamiętać, że testowanie to technologiczne badanie pozwalające otrzymać informacje o jakości przedmiotu testów:

- *technologiczne* — bo używamy inżynierskiego podejścia, wykorzystując eksperyment, doświadczenie, matematykę, logikę, narzędzia (programy wspomagające), pomiary itp.;
- *badanie* — bo jest to zorganizowane poszukiwanie informacji.

1.1.1. Cele testów

Testowanie umożliwia wykrywanie awarii bądź defektów w testowanym produkcie pracy. Ta fundamentalna własność testowania umożliwia realizację szeregu celów. Podstawowe **cele testów** (ang. *test objective*) to:

- dokonywanie oceny produktów pracy, takich jak wymagania, historyjki użytkownika, projekt, kod;
- wykrywanie awarii i znajdowanie defektów;
- zapewnienie uzyskania odpowiedniego pokrycia przedmiotu testów;

Kup książkę Poleć książkę

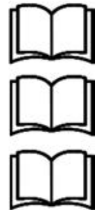
- obniżanie poziomu ryzyka związanego z niewystarczającą jakością oprogramowania (ryzyka wystąpienia niewykrytych wcześniej awarii podczas eksploatacji);
- sprawdzanie, czy zostały spełnione wszystkie wyspecyfikowane wymagania;
- sprawdzanie, czy przedmiot testów jest zgodny z wymaganiami wynikającymi z umów, przepisów prawa oraz norm/standardów;
- dostarczanie interesariuszom informacji niezbędnych do podejmowania świadomych decyzji dotyczących wytwarzanego produktu;
- budowanie zaufania do poziomu jakości przedmiotu testów;
- sprawdzanie, czy przedmiot testów jest kompletny i działa zgodnie z oczekiwaniami użytkowników i innych interesariuszy.

Różne cele wymagają różnych strategii testowania. W przypadku testowania modułowego — tzn. testowania pojedynczych fragmentów aplikacji/systemu (patrz podrozdział 2.2) — celem może być wykrycie jak największej liczby awarii, by w rezultacie wcześnie zidentyfikować i usunąć powodujące je defekty. Można dążyć też do zwiększenia pokrycia kodu przez testy modułowe. Natomiast w testowaniu akceptacyjnym (patrz podrozdział 2.2) celami mogą być:

- potwierdzenie, że system działa zgodnie z oczekiwaniami i spełnia stawiane mu wymagania;
- dostarczenie interesariuszom informacji na temat ryzyka, jakie wiąże się z przekazaniem systemu do eksploatacji w danym momencie.

W testach akceptacyjnych (zwłaszcza w testach akceptacyjnych użytkownika (ang. *UAT*) nie oczekujemy wykrycia dużej liczby awarii (defektów), bo może to prowadzić do utraty zaufania przez przyszłych użytkowników (patrz punkt 2.2.1.4). Awarie (defekty) te powinny być wykryte na wcześniejszych etapach testowania.

1.1.2. Testowanie a debugowanie



Część osób uważa, że testowanie polega na debugowaniu. Należy jednak pamiętać, że testowanie a debugowanie to dwie odmienne aktywności. Testowanie (zwłaszcza dynamiczne) ma ujawnić **awarie** (ang. *failure*) spowodowane defektami. **Debugowanie** (ang. *debugging*) natomiast jest czynnością programistyczną wykonywaną w celu zidentyfikowania przyczyny **defektu** (ang. *defect*, *fault*), poprawienia kodu i sprawdzenia, czy defekt został poprawnie naprawiony.

Gdy testy dynamiczne wykryją awarię, typowy proces debugowania będzie składał się z następujących czynności:

- reprodukcja awarii (w celu upewnienia się, że awaria rzeczywiście występuje, oraz aby możliwe było jej kontrolowane wywołanie w dalszym procesie debugowania);
- analiza (znajdowanie przyczyny awarii, np. lokalizacja defektu odpowiedzialnego za wystąpienie tej awarii);
- naprawa (eliminacja przyczyny awarii, np. usunięcie defektu w kodzie).

Kup ksi k Pole ksi k

Późniejsze testowanie potwierdzające (retest) wykonywane przez testera ma zapewnić, że poprawka rzeczywiście usunęła awarię. Najczęściej testowanie potwierdzające wykonywane jest przez tę samą osobę, która wykonała oryginalny test ujawniający problem. Po naprawie można również wykonać testy regresji w celu sprawdzenia, czy poprawka w kodzie nie spowodowała niepoprawnego działania oprogramowania w innych miejscach. Testy potwierdzające i retesty omówione są szczegółowo w punkcie 2.2.3.

Gdy testowanie statyczne odkryje defekt, proces debugowania polega po prostu na eliminacji tego defektu. Nie trzeba, tak jak w przypadku wykrycia awarii w testach dynamicznych, przeprowadzać reprodukcji awarii oraz analizy, ponieważ w testowaniu statycznym testowany produkt pracy nie jest uruchamiany. Testowanie statyczne nie znajduje bowiem awarii, lecz bezpośrednio identyfikuje defekty. Testowaniu statycznemu poświęcony jest rozdział 3.

Przykład. Rozważmy uproszczoną wersję problemu opisanego przez Myersa [Myers 2011]. Testujemy program, który na wejściu otrzymuje trzy liczby naturalne *a*, *b*, *c*, a na wyjściu odpowiada „tak” lub „nie”, w zależności od tego, czy z odcinków o bokach o długości *a*, *b*, *c* da się zbudować trójkąt. Program ma postać uruchamialnego pliku `trojkat.exe` i pobiera wartości wejściowe z klawiatury.

Tester przygotował kilka przypadków testowych. W szczególności uruchomił program dla danych wejściowych a = b = c = 16 500 (przypadek testowy polegający na wpisaniu bardzo dużych wartości danych wejściowych) i otrzymał następujący wynik (rysunek 1.1):

RYSUNEK 1.1. Awaria oprogramowania

Program odpowiedział „nie”, tzn. stwierdził, że nie da się zbudować trójkąta z trzech boków o długości 16 500 każdy. Jest to awaria, ponieważ oczekiwanym wynikiem jest odpowiedź „tak” — z takich odcinków da się zbudować trójkąt równoboczny. Tester zgłosił ten defekt deweloperowi. Programista powtórzył przypadek testowy i otrzymał taki sam wynik. Programista rozpoczął analizę kodu, który wygląda tak:

```
int _tmain(int argc, _TCHAR* argv[])
{
  short a,b,c,d;
  scanf("%d",&a);
  scanf("%d",&b);
  scanf("%d",&c);
  d=a+b;
  if (abs(a-b)<c && c<d)
    printf ("tak\n");
  else
    printf("nie\n");
  return 0;
}
```

Kup książkę Poleć książkę

Programista stwierdził, że warunek w instrukcji `if` jest poprawny, więc defekt musi być w innym miejscu. Programista zauważył, że w miejscu, gdzie do zmiennej `d` przypisywana jest suma zmiennych `a` i `b` może być problem z przepełnieniem rejestru. Zmienne `a`, `b`, `d` są bowiem tego samego typu (short), a więc mieszczą się w zakresie od –32 768 do 32 767. Jednak suma 16 500 + 16 500 wynosi 33 000, więcej niż maksymalna wartość zmiennej short. Operacja d = a + b powoduje „przekręcenie licznika" zmiennej `d`, w rezultacie czego przyjmuje ona wartość ujemną. W tym momencie warunek w instrukcji `if` jest fałszywy, ponieważ nie jest prawdą, że c < d.

Programista stwierdza, że najprostszą metodą naprawienia kodu jest eliminacja zmiennej `d`, a liczenie sumy a + b będzie następowało „w locie". Poprawiony kod wygląda tak:

```
int _tmain(int argc, _TCHAR* argv[])
{
  short a,b,c;
  scanf("%d",&a);
  scanf("%d",&b);
  scanf("%d",&c);
  if (abs(a-b)<c && c<a+b)
    printf ("tak\n");
  else
    printf("nie\n");
  return 0;
}
```

Programista sprawdza ponownie test dla a = b = c = 16 500 i teraz program działa poprawnie. Rozwiązanie działa, ponieważ w momencie, gdy kompilator ma obliczyć „w locie" sumę a + b, automatycznie przeznacza na tę operację tyle pamięci, ile jest potrzebne. Defekt w poprzedniej wersji kodu polegał na tym, że suma ta była wpisywana do zmiennej `d`, która miała ściśle określony typ, a więc miała ograniczenie na górną wartość. Programista informuje testera o naprawieniu defektu, tester ponownie wykonuje test i stwierdza, że program tym razem zwraca poprawną odpowiedź. Tester zamyka zgłoszenie o defekcie, uznając, że defekt został naprawiony.

W powyższym przykładzie wszystkie działania testera wchodziły w zakres testowania, natomiast działania programisty (poza wykonaniem testu) wchodziły w zakres czynności debugowania.

1.2. Dlaczego testowanie jest niezbędne?

FL-1.2.1 (K2)	Kandydat podaje przykłady wskazujące, dlaczego testowanie jest niezbędne.
FL-1.2.2 (K1)	Kandydat pamięta, jaka jest relacja między testowaniem a zapewnianiem jakości.
FL-1.2.3 (K2)	Kandydat odróżnia podstawową przyczynę, pomyłkę, defekt i awarię.

Testowanie modułów, systemów i związanej z nimi dokumentacji wspomaga identyfikację defektów w oprogramowaniu. Testowanie wykrywa również luki i inne

Kup ksi k Pole ksi k

braki w specyfikacji oprogramowania. Stąd testowanie może pomóc w zmniejszeniu ryzyka wystąpienia awarii w trakcie eksploatacji. Kiedy wykryte defekty są naprawiane, przyczynia się to do poprawy jakości obiektu testów. Ponadto testowanie oprogramowania może być konieczne w celu spełnienia wymagań umownych lub prawnych albo w celu spełnienia norm regulacyjnych.

Większość z nas zetknęła się z oprogramowaniem, które nie działało tak, jak powinno — także poza pracą zawodową. Poniżej przytaczamy trzy przykłady — mimo że niektóre miały miejsce dość dawno temu, są wciąż aktualne, gdyż opisywane przez nas typy awarii zdarzają się także w produkowanym obecnie oprogramowaniu.

Przypadek gry — pasjans „Pająk”

Gracz gra 104 kartami, z których 54 rozłożone są w 10 stosach, 50 leży z prawej strony stołu w rzędach po 10 kart. Celem gry jest ułożenie kart w stosy od króla do asa w porządku malejącym. Gdy doprowadzi się do sytuacji jak na rysunku 1.2 — na stole jest 9 kart, w grupach jest 30 kart (co daje w sumie 3 · 13 — 39 kart — 3 pełne kolory), pojawia się komunikat: *Jeżeli jakieś kolumny są puste, nie możesz rozdać nowego rzędu*.

Gdy pojawia się nieprawidłowość w aplikacji, trzeba zawsze odpowiedzieć na dwa pytania:

- jaki jest wpływ tej awarii na użytkownika?
- jakie jest prawdopodobieństwo zajścia tej awarii?

W tym wypadku wpływ jest praktycznie zerowy: gracz może albo zrezygnować z dalszej gry, albo dokonać obejścia — przycisk *Cofnij* powoduje, że ostatni stos z powrotem jest na stole i można go rozłożyć na wolne kolumny. Prawdopodobieństwo takiej sytuacji — gdy gra się wszystkimi czterema kolorami, a nie tylko pikami, jak na rysunku 1.2 — jest rzędu 10^{-6}. Oznacza to, że awaria pojawi się mniej więcej raz na milion okazji do jej wystąpienia. Gdy koszt wystąpienia awarii jest bliski zeru, a prawdopodobieństwo jej wystąpienia minimalne, defektów — zwłaszcza w systemach niekrytycznych — na ogół się nie usuwa.

RYSUNEK 1.2. Gra pasjans „Pająk”

Przypadek rakiety Ariane 5

4 czerwca 1996 roku rakieta Ariane 5, przygotowana przez Europejską Agencję Kosmiczną (European Space Agency), eksplodowała 40 sekund po starcie w Kourou w Gujanie Francuskiej. Był to pierwszy lot tej rakiety, po dekadzie przygotowań, które kosztowały 7 miliardów USD. Rakieta i jej ładunek były warte 500 milionów USD.

Po dwutygodniowym śledztwie okazało się, że problem tkwił w oprogramowaniu. W ramach unowocześniania rakiety zastąpiono 16-bitowy procesor w Ariane 4 procesorem 64-bitowym. Gdy pionowa prędkość rakiety przekroczyła 32 767 ($2^{15} - 1$) jednostek, wartość odpowiedniej zmiennej została odczytana jako liczba ujemna. System sterujący stwierdził, że rakieta przestała się wznosić (spada), uruchomiona została więc procedura awaryjna, co doprowadziło do eksplozji rakiety w wyniku zastosowania mechanizmu autodestrukcji.

Wniosek: w ramach unowocześnienia systemu nie zostały przeprowadzone odpowiednie testy regresji.

Przypadek rakiety Patriot

Każda bateria rakiet systemu Patriot składa się z radaru, stanowiska dowodzenia (komputera) i mobilnych wyrzutni (patrz rysunek 1.3). Każda bateria ma przydzielony obszar, który ma chronić; innymi słowy, jeżeli namierzona rakieta celuje w ten obszar, bateria strzela; jeżeli cel rakiety jest poza obszarem, bateria nie strzela.

RYSUNEK 1.3. Bateria rakiet Patriot

25 lutego 1991 roku w Dhahran w Arabii Saudyjskiej podczas pierwszej wojny w Zatoce Perskiej amerykańska rakieta Patriot nie przechwyciła irackiego Scuda, który trafił w barak armii amerykańskiej, zabijając 28 i raniąc ponad 100 osób. Raport General Accounting Office, GAO/IMTEC-92-26, zatytułowany *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, podaje przyczynę tego zdarzenia.

Kup ksi k Pole ksi k

Był to błąd arytmetyczny. System mierzył czas w dziesiątych częściach sekundy, używając 24-bitowego stałoprzecinkowego rejestru. Ułamek $^1/_{10}$ w systemie dziesiętnym ma skończoną reprezentację, ale w systemie dwójkowym jest ułamkiem o nieskończonym rozwinięciu. Ponieważ komputer reprezentuje liczby w skończonych rejestrach, jakaś część ułamka musi zostać ucięta, co będzie powodowało minimalny błąd zaokrągleń. Po około 100 godzinach pracy ten błąd zaokrągleń skumulował się do 0,34 sekundy, co przy prędkości Scuda ponad 1676 metrów na sekundę dawało odległość ponad 0,5 kilometra. System śledzący Patriot uznał więc, że iracka rakieta jest poza jego zasięgiem, i nie odpalił rakiety w celu zniszczenia wrogiego Scuda.

W raporcie podano także, że defekt był znany — jego eliminacja wymagała restartu systemu mniej więcej co 4 godziny przy czasie restartu około 5 minut; podane to było w instrukcji obsługi systemu. Jest to do dziś typowa sytuacja — użytkownicy nie zapoznają się z instrukcją obsługi systemu (o ile ona w ogóle istnieje). Co więcej, defekt był znany i został usunięty w większości miejsc w systemie. Oznacza to, że programiści wkopiowali ten sam kod w wielu miejscach. Zdarza się to i dziś — naprawiony już defekt musi być wtedy ponownie analizowany i usuwany.

Dziś prawdopodobieństwo wystąpienia tak katastroficznej sytuacji jest mniejsze, ale w dalszym ciągu możemy się spotkać z oprogramowaniem działającym nieprawidłowo.

1.2.1. Znaczenie testowania dla powodzenia projektu

Testowanie pomaga w osiągnięciu uzgodnionych celów w ustalonym zakresie, czasie, określonej jakości i budżecie. Wkład testowania w sukces projektu może być rozpatrywany w kategoriach:

- jakości produktu,
- jakości procesu,
- celów projektowych,
- umiejętności ludzi.

Jakość produktu

Rygorystyczne testowanie systemów i dokumentacji może zredukować ryzyko wystąpienia problemów (awarii) w środowisku produkcyjnym i przyczynić się do osiągnięcia wysokiej **jakości** (ang. *quality*) systemu. Wykrycie, a następnie usunięcie defektów przyczynia się do podniesienia jakości modułów lub systemów. Odpowiedni poziom testowania może być również wymagany przez zapisy kontraktowe, wymogi prawne lub standardy przemysłowe.

Znanych jest wiele przykładów oprogramowania i systemów (patrz powyżej), które zostały przekazane do eksploatacji, ale na skutek defektów uległy awarii lub z innych powodów nie zaspokoiły potrzeb interesariuszy. Dzięki odpowiednim technikom testowania — stosowanym w sposób fachowy na odpowiednich poziomach testów i w odpowiednich fazach cyklu wytwarzania oprogramowania — częstotliwość występowania tego rodzaju problemów można jednak ograniczyć.

Kup ksi k Pole ksi k

Weryfikacja i walidacja oprogramowania przez testerów przed przekazaniem go do eksploatacji umożliwia wykrycie awarii, ułatwia usuwanie związanych z nimi defektów (debugowanie). Ułatwia to testowanie, zmniejsza się ryzyko i rośnie prawdopodobieństwo, że oprogramowanie zaspokoi potrzeby interesariuszy i spełni stawiane mu wymagania. Tym samym rośnie prawdopodobieństwo powodzenia projektu.

Jakość procesu

Testowanie może pośrednio przyczynić się do zwiększenia jakości procesu wytwórczego. Jest to bardzo istotne, ponieważ wiadomo, że na końcową jakość produktu bardzo duży wpływ ma jakość procesu, w ramach którego produkt jest wytwarzany. Jakość procesu może być zwiększona na różne sposoby. Na przykład wprowadzenie automatyzacji testów poprawia efektywność procesu wydania systemu. Zastosowanie testowania opartego na ryzyku optymalizuje nakłady na testowanie. Dane zbierane w ramach monitorowania testów (patrz podrozdział 5.3) pomagają w ustaleniu miejsc w procesie wytwórczym, które wymagają poprawy (np. zbyt długiego czasu naprawy defektów, zbyt dużej liczby defektów wprowadzanych w określonej fazie cyklu wytwarzania itp.).

Cele projektowe

Testowanie może przyczynić się do zwiększenia prawdopodobieństwa osiągnięcia założonych celów projektowych. Na przykład stosowanie testów statycznych na wczesnym etapie projektu zmniejsza koszty utrzymania oprogramowania i poprawia efektywność pracy programistów poprzez zmniejszenie czasu przeznaczonego na usuwanie defektów. Dzięki temu jest większa szansa, że produkt zostanie ukończony w założonym czasie i w założonym budżecie.

Umiejętności ludzi

„Efektem ubocznym” praktyk stosowanych w testowaniu jest zwiększenie umiejętności członków zespołu oraz innych interesariuszy. Na przykład wykonywanie przeglądów kodu (np. w formie przejrzenia, patrz punkt 3.2.4) zwiększa zrozumienie kodu i pozwala mniej doświadczonym programistom poprawić swoje umiejętności programowania i projektowania). Ścisła współpraca testerów z projektantami systemu na etapie prac projektowych umożliwia obu stronom lepsze zrozumienie projektu.

1.2.2. Testowanie a zapewnienie jakości

Testowanie jest często błędnie utożsamiane z zapewnieniem jakości (ang. *quality assurance*). Są to dwa oddzielne (chociaż powiązane ze sobą) procesy, które zawierają się w szerszym pojęciu „zarządzanie jakością” (ang. *quality management*, QM). Zarządzanie jakością obejmuje wszystkie czynności mające na celu kierowanie działaniami organizacji w dziedzinie jakości i ich nadzorowanie.

Dwoma podstawowymi elementami zarządzania jakością są:

- zapewnienie jakości,
- kontrola jakości.

Kup ksi k Pole ksi k

Zapewnienie jakości

Zapewnienie jakości (ang. *quality assurance*, *QA*) skupia się na ustanawianiu, wprowadzaniu, monitorowaniu, doskonaleniu i przestrzeganiu procesów związanych z jakością. Kiedy odpowiednie procesy są realizowane prawidłowo, przyczynia się to do zapobiegania defektom i zwiększa pewność, że odpowiednie poziomy jakości produktów pracy zostaną osiągnięte. Zapewnienie jakości, gdy jest stosowane do rozwoju i utrzymania oprogramowania, powinno być również stosowane do testowania oprogramowania, które jest częścią każdego z tych działań. Ponadto stosowanie analizy przyczyn źródłowych w celu wykrycia przyczyn defektów oraz stosowanie wniosków ze spotkań retrospektywnych w celu poprawy procesów są ważne dla efektywnego zapewnienia jakości.

Kontrola jakości

Kontrola jakości (ang. *quality control*, QC) obejmuje cały szereg czynności, w tym czynności testowe, które wspierają osiągnięcie odpowiednich poziomów jakości. Czynności testowe są ważnym elementem ogólnego procesu wytwarzania lub pielęgnacji oprogramowania. Prawidłowy przebieg tego procesu (w tym procesu testowania) jest istotny z punktu widzenia zapewnienia jakości, w związku z czym zapewnienie jakości wspiera właściwe testowanie.

W tabeli 1.1 podsumowano zasadnicze różnice pomiędzy procesami zapewnienia i kontroli jakości.

TABELA 1.1. Porównanie procesów zapewnienia jakości i kontroli jakości

KATEGORIA	ZAPEWNIENIE JAKOŚCI	KONTROLA JAKOŚCI
Ogólny opis	Wdrażanie procesów, metodyk i standardów, które zapewnią, że wytwarzany produkt spełni wymagane standardy jakościowe	Wykonywanie czynności, których celem jest weryfikacja, że wytwarzany produkt spełnia wymagane standardy jakościowe
Cel	Doskonalenie procesu wytwórczego	Doskonalenie produktu poprzez wykrywanie awarii i defektów
Rodzaj procesu	Prewencyjny (zapobieganie defektom), proaktywny	Kontrolny (wykrywanie defektów), reaktywny
Przykłady działań	Wdrażanie procesów, np. zarządzania defektami, zarządzania zmianą, wydawaniem oprogramowania; audyty jakości; pomiary procesu i produktu; weryfikacja poprawności implementacji i wykonania procesów; szkolenia członków zespołu; wybór narzędzi	Analiza statyczna dokumentacji projektowej; przeglądy kodu; analiza, projektowanie, implementacja przypadków testowych; wykonanie testów dynamicznych; pisanie i wykonywanie skryptów testowych; raportowanie defektów; używanie narzędzi wspomagających testowanie

Kup ksi k Pole ksi k

1.2.3. Pomyłki, defekty, awarie i podstawowe przyczyny

W podejściu ISTQB® rozróżnia się trzy etapy prowadzące do powstania nieprawidłowego wyniku, związane z trzema bardzo ważnymi pojęciami, którymi są:

- **pomyłka** (zwana także błędem) — działanie człowieka powodujące powstanie nieprawidłowego rezultatu;
- **defekt** (pluskwa, usterka) — niedoskonałość lub wada produktu pracy, polegająca na niespełnieniu wymagań;
- **awaria** — zdarzenie, w którym moduł lub system nie wykonuje wymaganej funkcji w określonym zakresie.

Na skutek **pomyłki** (ang. *error*) człowieka w kodzie oprogramowania lub w innym związanym z nim produkcie pracy może powstać defekt. Uwaga! W trakcie egzaminu należy zwracać uwagę na tę terminologię, ponieważ jest ona nie do końca zgodna z intuicją, a wchodzi w zakres słownictwa, którego znajomość obowiązuje na egzaminie. Zazwyczaj, w mowie potocznej, defekt nazywamy błędem — często mówimy: „w tym miejscu w kodzie jest błąd". Z punktu widzenia terminologii ISTQB® jest to niepoprawne. W programie znajdować się może *defekt*, czyli wada produkcyjna. *Błąd* zawsze dotyczy *pomyłki ludzkiej*. Uruchomienie fragmentu kodu, w którym jest defekt, może, ale nie musi spowodować awarię.

Przykład. Rozważmy prosty przykład pokazujący, dlaczego wykonanie linii programu zawierającej defekt *nie musi* prowadzić do wystąpienia awarii. Załóżmy, że pewien fragment kodu oblicza średnią wartość pomiarów, dzieląc ich sumę przez liczbę dokonanych pomiarów. W kodzie służy do tego instrukcja:

```
Srednia := SumaPomiarow/LiczbaPomiarow
```

W tej linii znajduje się defekt polegający na tym, że program nie kontroluje, czy mianownik liczonego ułamka jest zerem. Dzielenie przez zero jest bowiem niedozwolone i wykonanie takiej operacji może skutkować zakończeniem pracy programu. W sytuacji, gdy liczba pomiarów jest dodatnia, wykonanie powyższej instrukcji nie spowoduje żadnych problemów — program zadziała perfekcyjnie i zwróci prawidłowy wynik. Test, który wymusi wykonanie tej instrukcji z dodatnią liczbą pomiarów, zostanie zaliczony i nie zauważymy żadnego objawu awarii. Jeśli jednak linia ta zostanie wykonana w sytuacji, gdy jest zero pomiarów (wartość zmiennej `LiczbaPomiarow` wynosi 0), to awaria wystąpi i ją zauważymy.

Przykład. Rozważmy teraz nieco bardziej wyrafinowaną, choć nadal względnie prostą sytuację. Dany niech będzie program, który zlicza, na ilu miejscach tablicy o nazwie T występują zera. Do elementów tablicy T odwołujemy się przez jej indeksy — na przykład T[3] oznacza element, który występuje w tablicy na pozycji nr 3. Załóżmy ponadto, że elementy tablicy (tak jak w wielu rzeczywistych językach programowania) indeksowane są od zera, zatem pierwszy element tablicy to T[0], kolejny to T[1] i tak dalej. Poniżej przedstawiony jest pseudokod naszego programu, zawierający defekt — pętla przechodząca po kolejnych elementach tablicy zaczyna przechodzenie od elementu T[1] zamiast od T[0]:

Kup ksi k Pole ksi k

```
program Zlicz
wejście: tablica T (o elementach T[0], T[1], …, T[n])
wyjście: liczba komórek T zawierających zero
liczba := 0
dla każdego i = 1, 2, …, n wykonaj
  jeżeli T[i] == 0 to liczba := liczba + 1
zwróć liczba
```

Zauważmy, że linia z defektem (pętla „dla każdego”) wykona się dla każdego uruchomienia kodu. Jednak wystąpienie awarii (zły wynik) będzie zależeć od tego, czy komórka T[0] zawiera zero, czy inną liczbę. W pierwszym przypadku (T[0] = 0) wynik będzie niepoprawny — program zliczy o jedną komórkę za mało. Na przykład jeśli T = (0, 3, 2, 0, 1), program zwróci 1 zamiast 2. Jednak w drugim przypadku (T[0] ≠ 0) wynik będzie całkowicie poprawny! Na przykład jeśli T = (5, 2, 0, 1, 0, 0), program zwróci 3, co jest poprawnym rezultatem pomimo uruchomienia linii z defektem. Projektowanie testów polega między innymi na tym, aby uwzględniać tego typu sytuacje i aby zestaw testowy był w stanie wykrywać podobne usterki w kodzie.

Błąd skutkujący wprowadzeniem defektu w jednym produkcie pracy może spowodować błąd powodujący wprowadzenie defektu w innym, powiązanym produkcie pracy. Wykonanie kodu zawierającego defekt może spowodować awarię, ale — jak zobaczyliśmy w powyższym przykładzie — nie musi dziać się tak w przypadku każdego defektu. Niektóre defekty powodują awarię na przykład tylko po wprowadzeniu ściśle określonych danych wejściowych bądź na skutek wystąpienia określonych warunków wstępnych, które mogą mieć miejsce bardzo rzadko lub nigdy (na przykład w instrukcji dzielenia jednej liczby przez drugą bez kontroli mianownika awaria wystąpi tylko wtedy, gdy mianownik będzie zerem). Tylko równoczesne zaistnienie trzech czynników (błąd — defekt — awaria) powoduje obserwowane nieprawidłowe działanie testowanego produktu (rysunek 1.4).

RYSUNEK 1.4. Pomyłka, usterka, awaria

Pomyłki (błędy) mogą pojawiać się z wielu powodów:

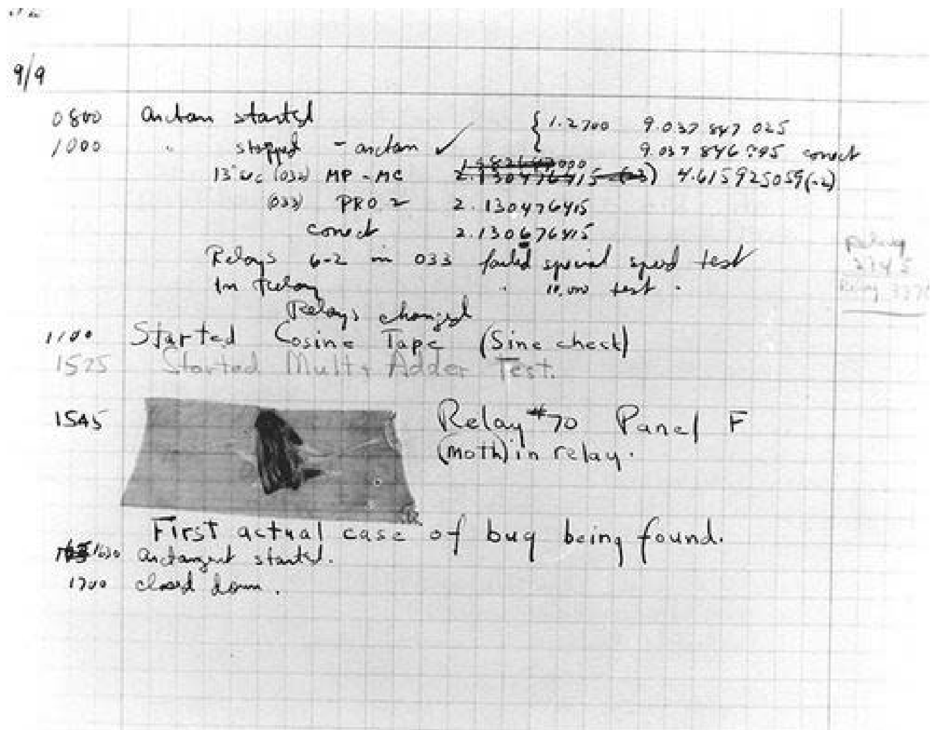
- presja czasu;
- omylność człowieka;
- brak doświadczenia lub niedostateczne umiejętności uczestników projektu;
- problemy z wymianą informacji między uczestnikami projektu;

- niejasności dotyczące rozumienia wymagań i dokumentacji projektowej;
- złożoność kodu, projektu, architektury, rozwiązywanego problemu i/lub wykorzystywanej technologii;
- nieporozumienia dotyczące interfejsów wewnątrz systemu i między systemami, zwłaszcza w przypadku dużej liczby tych systemów;
- stosowanie nowych, nieznanych technologii.

Awarie z kolei mogą być wywołane *niekoniecznie* przez błędy ludzkie, ale także przez czynniki środowiska, takie jak:

- promieniowanie,
- pole elektromagnetyczne,
- skażenie.

Czynniki te mogą powodować awarie w oprogramowaniu wbudowanym lub wpływać na działanie oprogramowania przez zmianę warunków działania sprzętu.



RYSUNEK 1.5. Pierwsza „pluskwa” w historii (www.atlasobscura.com/places/grace-hoppers-bug)

Pierwsza pluskwa (ang. *bug*) w historii została znaleziona przez admirał Hopper w 1947 roku. Rysunek 1.5 przedstawia fragment oryginalnego raportu Hopper, zawierającego… rzeczywistą ćmę — przyczynę awarii komputera Mark II.

Kup ksi k Pole ksi k

Nie wszystkie nieoczekiwane **wyniki testów** (ang. *test result*) oznaczają awarie. *Rezultat fałszywie pozytywny* może być skutkiem błędów związanych z wykonaniem testów, defektów w danych testowych, środowisku testowym, innych testaliach itp. Wyniki fałszywie pozytywne są raportowane jako defekty, których w rzeczywistości nie ma. Podobne problemy mogą być przyczyną sytuacji odwrotnej — *rezultatu fałszywie negatywnego*, czyli sytuacji, w której testy nie wykrywają defektu, który powinny wykryć (patrz tabela 1.2).

Formalnie rezultat fałszywie pozytywny[1] to pozytywny wynik testu (test niezdany), podczas gdy tak naprawdę test powinien zostać zaliczony. Przykładem takiej sytuacji jest np. złe zrozumienie przez testera specyfikacji i niepoprawne zdefiniowanie wyniku oczekiwanego. Rezultat fałszywie negatywny to negatywny wynik testu (test zdany), podczas gdy tak naprawdę test powinien zostać niezaliczony. Przykładem takiej sytuacji jest zmiana wymagań pomiędzy cyklami testów. W drugim cyklu zmienione wymaganie powinno spowodować niezdanie testu, jednak w teście wynik oczekiwany pozostał niezmieniony i test nadal jest zaliczany.

TABELA 1.2. Możliwe wyniki testów w kontekście ich poprawności

MOŻLIWE WYNIKI TESTÓW		WYNIK ZINTERPRETOWANY	
		TEST ZDANY	TEST NIEZDANY
PRAWDZIWY WYNIK TESTU	TEST ZDANY	Wynik poprawny (negatywny)	Wynik fałszywie pozytywny
	TEST NIEZDANY	Wynik fałszywie negatywny	Wynik poprawny (pozytywny)

Przypadki testowe należy projektować w taki sposób, aby uniknąć maskowania defektów, czyli sytuacji, w których wystąpienie jednego defektu uniemożliwia wykrycie innego defektu lub wystąpienie dwóch defektów znosi ich wzajemny efekt. Istnieje kilka dobrych praktyk stosowanych w projektowaniu testów, które pomagają testerowi uniknąć takich sytuacji (więcej o tym w rozdziale 4.), ale na ogół zamaskowane defekty są trudne do wykrycia.

W związku z powyższymi rozważaniami ważnym czynnikiem jest analiza **podstawowej przyczyny** (ang. *root cause*) defektu: pierwotnego powodu, w wyniku którego defekt ten powstał. Powodem tym może być zaistnienie określonej sytuacji lub wystąpienie ludzkiej pomyłki. Przeanalizowanie defektu w celu zidentyfikowania podstawowej przyczyny pozwala zredukować wystąpienia podobnych defektów w przyszłości. Analiza przyczyny podstawowej, skupiająca się na najważniejszych, pierwotnych przyczynach defektów, może prowadzić do udoskonalenia procesów, co może z kolei przełożyć się na dalsze zmniejszenie liczby defektów w przyszłości.

[1] Terminologia „wynik fałszywie pozytywny" i „wynik fałszywie negatywny" pochodzi z obszaru analityki medycznej. Wynik negatywny oznacza brak obecności np. wirusa w organizmie, a pozytywny — jego obecność. Analogia z testowaniem jest więc następująca: test jest pozytywny, gdy wykrywa awarię (obecność defektu), a negatywny — gdy jej nie wykrywa.

Kup ksi k Pole ksi k

Przykład. Defekt w kodzie powoduje nieprawidłowe wyliczanie rabatu przy zakupie hurtowym w e-sklepie, co powoduje reklamacje klientów. Wadliwy kod został napisany na podstawie historyjki użytkownika — właściciel produktu źle zrozumiał zasady naliczania rabatów i źle napisał historyjkę. Reklamacje klientów to *skutki*, nieprawidłowe wyliczenie rabatów to *awaria*, *defekt* zaś to nieprawidłowe obliczenia wykonywane przez kod, a *podstawowa przyczyna* to braki wiedzy właściciela produktu, wskutek czego popełnił on *pomyłkę* przy pisaniu historyjki użytkownika.

1.3. Zasady testowania

FL-1.3.1 (K2)	Kandydat objaśnia siedem zasad testowania.

Istnieje wiele różnych „praw" czy „zasad" związanych z testowaniem, które są prawdziwe niezależne od kontekstu projektowego czy rodzaju wytwarzanego produktu, a które powstały w ciągu ostatnich 60 lat. Sylabus poziomu podstawowego opisuje siedem takich zasad. Oto podstawowe zasady testowania:

1. Testowanie ujawnia usterki, ale nie może dowieść ich braku.
2. Testowanie gruntowne jest niemożliwe.
3. Wczesne testowanie oszczędza czas i pieniądze.
4. Defekty mogą się kumulować.
5. Testy ulegają zużyciu.
6. Testowanie zależy od kontekstu.
7. Przekonanie o braku defektów jest błędem.

1. Testowanie ujawnia usterki, ale nie może dowieść ich braku

Ta słynna zasada została sformułowana przez Edsgera Dijkstrę, duńskiego informatyka, na konferencji „Software Engineering Techniques" w Rzymie w 1969 roku Brzmi ona: „Testowanie pokazuje obecność, a nie brak usterek" [Buxton 1970]. Testowanie może wykazać, że istnieją defekty, natomiast nie jesteśmy w stanie dowieść, że w testowanym programie nie ma usterek. Tym samym testowanie jedynie zmniejsza prawdopodobieństwo, że w oprogramowaniu pozostaną niezidentyfikowane defekty. To, że nie wykryto defektów, nie jest dowodem na poprawność testowanego programu. Zasada ta ma poważne konsekwencje: testowanie ma charakter negatywny, tzn. pokazuje, że coś nie działa, a nie, że wszystko jest w porządku. Niesie to za sobą pewne istotne implikacje natury psychologicznej, o których będzie mowa później.

Ciekawostką jest to, że stwierdzenie Dijkstry można sformułować jako ścisłe i precyzyjne twierdzenie matematyczne — związane jest to z faktem, że pewne problemy informatyczne, takie jak tzw. problem stopu (który może być traktowany jako defekt algorytmu), są problemami nierozstrzygalnymi. Nie da się, w ogólności, odpowiedzieć na pytanie: czy dany program zatrzyma się dla każdego możliwego wejścia? Zatem nie jesteśmy w stanie w każdym przypadku wykryć tego typu usterki (możliwości zapętlenia się programu).

Kup ksi k Pole ksi k

2. Testowanie gruntowne jest niemożliwe

Rozpatrzmy następujący przykład [Myers 2011]. Mamy do przetestowania fragment kodu z czterema warunkami (A, B, C, D) i sześcioma instrukcjami (I1, I2, I3, I4, I5, I6), przedstawiony na rysunku 1.6, przy czym kod ten wykonujemy w pętli, wykonywanej co najwyżej 20 razy. W pierwszym przypadku mamy pięć możliwych ścieżek S1 – S5 (symbol „!" oznacza fałszywość warunku, na przykład !A oznacza, że warunek w decyzji A jest fałszywy; w każdej decyzji prawdziwość warunku oznacza przejście prawą, a fałszywość — lewą strzałką):

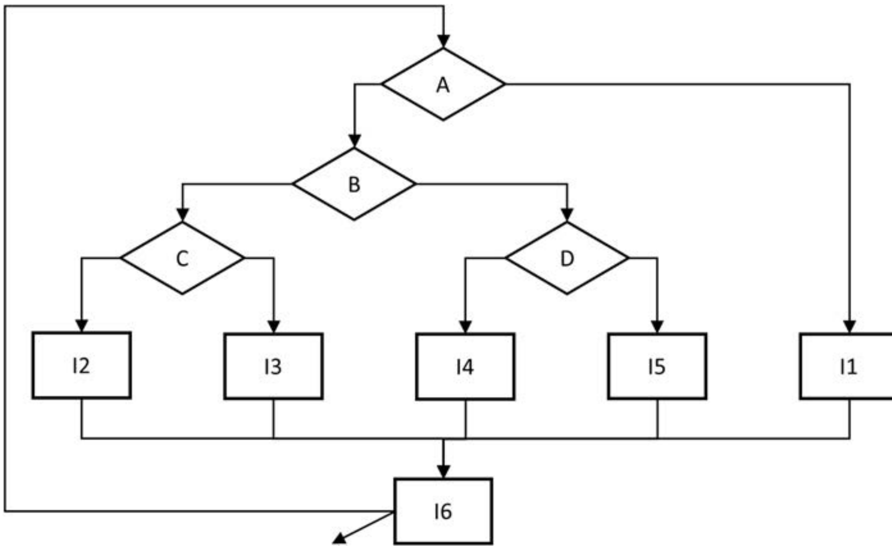
S1. A I_1 I_6

S2. !A B D I_5 I_6

S3. !A B !D I_4 I_6

S4. !A !B C I_3 I_6

S5. !A !B !C I_2 I_6



RYSUNEK 1.6. Graf przepływu sterowania fragmentu kodu do przetestowania

Dwa przebiegi pętli mogą realizować już 25 możliwych ścieżek:

S1 S1; S1 S2; S1 S3; S1 S4; S1 S5;

S2 S1; S2 S2; S2 S3; S2 S4; S2 S5;

S3 S1; S3 S2; S3 S3; S3 S4; S3 S5;

S4 S1; S4 S2; S4 S3; S4 S4; S4 S5;

S5 S1; S5 S2; S5 S3; S5 S4; S5 S5.

Trzy przebiegi pętli dają już $5^3 = 125$ możliwych ścieżek. Ogólnie przy *n*-krotnym przebiegu pętli mamy 5^n możliwych realizacji ścieżek. Przy założeniu, że pętla wykona się co najwyżej 20 razy, liczba różnych możliwych realizacji ścieżek wyniesie $5 + 5^2 + 5^3 + \ldots + 5^{20} = 119209289550780 > 10^{14}$.

Jeśli przyjmiemy, że potrzebujemy 0,001 sekundy do sprawdzenia jednej ścieżki, to do przetestowania wszystkich ścieżek potrzeba nam będzie około 3860 lat. Niestety nie mamy tyle czasu!

By w pełni (gruntownie) przetestować daną aplikację, należałoby sprawdzić:

- każdą możliwą wartość wejściową dla każdej zmiennej (włączając w to zmienne wynikowe i robocze);
- każdą możliwą sekwencję wykonania programu;
- każdą konfigurację sprzętu (hardware)/oprogramowania (software), włączając konfiguracje na przykład serwerów, gdzie nic nie możemy zmodyfikować;
- wszystkie możliwe — ale na ogół *niewyobrażalne* — użycia testowanego produktu przez użytkownika końcowego.

Gruntowne testowanie musi oznaczać, że po zakończeniu testów wszyscy będą pewni, że nie nastąpią żadne awarie; jest to niemożliwe (poza trywialnymi przypadkami) [Manna 1978]. Zamiast testowania gruntownego do kierowania testami należy wykorzystać analizę ryzyka i priorytetyzację. Oznacza to jednak, że testerzy oprogramowania są niezbędni w jego cyklu wytwórczym. Ponadto testerzy muszą mieć opanowane pewne techniki testowania.

3. Wczesne testowanie oszczędza czas i pieniądze

Wczesne testowanie nazywa się niekiedy „przesunięciem w lewo" (ang. *shift-left*). Czynności testowe powinny rozpoczynać się najwcześniej, jak tylko jest to możliwe w przypadku danego oprogramowania. Oszczędza to czas i pieniądze, bo defekty, które zostaną usunięte na wczesnym etapie procesu, nie spowodują kolejnych usterek w pochodnych produktach pracy, takich jak projekt lub kod, co zwiększy produktywność programowania, zmniejszy koszty i czas rozwoju, a także może mieć pozytywny wpływ na wymagany wysiłek związany z testowaniem [Boehm 1981]. Całkowity koszt jakości zostanie obniżony, ponieważ później w cyklu wytwarzania wystąpi mniej awarii. Testowanie zawsze powinno być nakierowane na spełnienie dobrze określonych i formalnie zdefiniowanych celów. Nawet jeśli nie jesteśmy gotowi do wykonania testów dynamicznych (bo na przykład oprogramowanie nie zostało jeszcze napisane), możemy wykonywać testy statyczne, przeglądy dokumentacji, projektu itd.

Na rysunku 1.7 przedstawiona jest słynna tzw. krzywa Boehma, która ilustruje zależność kosztu usunięcia defektu od czasu, jaki upłynął do jego znalezienia. Krzywa ta jest wykładnicza, co oznacza, że im później znajdziemy defekt, tym większy będzie wzrost kosztów jego naprawy. Współczesne badania sugerują, że krzywa ta nie do końca rośnie aż tak szybko, niemniej jej wzrost w każdym przypadku jest znaczący, co wyraźnie sugeruje, że wczesne znajdowanie defektów jest bardzo opłacalne.

Kup ksi k Pole ksi k

RYSUNEK 1.7. Koszt usunięcia defektu w funkcji czasu

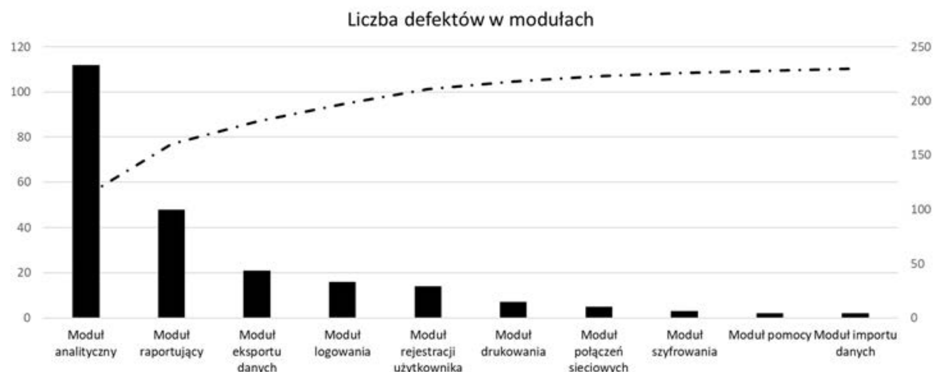
4. Defekty mogą się kumulować

Defekty nie rozkładają się równomiernie ani w oprogramowaniu, ani w czasie. Większość defektów znalezionych podczas testowania przed wypuszczeniem oprogramowania lub powodujących awarie produkcyjne znajduje się w małej liczbie modułów [Enders 1975]. W rezultacie przewidywane skupiska defektów i skupiska defektów faktycznie zaobserwowane na etapie testowania lub eksploatacji są ważnym elementem analizy ryzyka, którą przeprowadza się w celu odpowiedniego ukierunkowania wysiłków związanych z testowaniem. Nie oznacza to, że w pozostałych modułach jest mniejsza liczba defektów — po prostu w ramach testowania (lub w produkcji) koncentrujemy się na najistotniejszych z punktu widzenia użytkownika ścieżkach i tam znajdujemy większość defektów.

W przypadku kumulowania się defektów ma zastosowanie znana zasada, tzw. reguła Pareto, która mówi, że mała liczba przyczyn powoduje dużą liczbę skutków. W terminologii defektów można by ją przetłumaczyć na przykład tak: około 20% modułów zawiera około 80% defektów.

Na rysunku 1.8 pokazany jest typowy rozkład liczby defektów w modułach (histogram) oraz skumulowana liczba defektów (linia). Moduły posortowane są malejąco ze względu na liczbę defektów. Zazwyczaj jedynie kilka modułów ma bardzo dużo defektów, a pozostałe mają ich mało. Wykres pokazuje przykład zastosowania reguły Pareto do optymalizacji wysiłku. Jeśli znamy (na przykład poprzez szacowanie lub odwołanie do danych historycznych) rozkład przewidywanej liczby defektów, tak jak na wspomnianym wykresie, to możemy zająć się testowaniem małej liczby najbardziej defektogennych modułów, dzięki czemu w krótkim czasie wykryjemy większość defektów. Na przykład moduły analityczny i raportujący zawierają łącznie 160 defektów, czyli 20% modułów (2 z 10) zawiera ok. 70% wszystkich przewidywanych defektów (160 z 230).

Kup ksi k Pole ksi k



RYSUNEK 1.8. Przykład analizy Pareto

Jeśli w trakcie testowania widzimy, że przy porównywalnym wysiłku testowym w module A wykrywamy o wiele więcej defektów niż w module B, oznacza to, że prawdopodobnie w module A jest jeszcze więcej niewykrytych defektów. Racjonalne podejście oparte na zasadzie kumulowania się defektów wymagałoby w tej sytuacji skupienia się jeszcze bardziej na module A niż na module B.

5. Testy ulegają zużyciu (lub „paradoks pestycydów")

Jeżeli powtarzamy ciągle te same testy, to — po zmianach, które prowadzą do usunięcia wykrytych defektów — nie znajduje się już żadnych nowych usterek [Beizer 1990]. Żeby przezwyciężyć to zjawisko, przypadki testowe muszą być *regularnie przeglądane i modyfikowane*. Co więcej, by sprawdzić nowe bądź poprawione części testowanego programu, należy stworzyć nowe testy.

Niezmieniane testy tracą z czasem zdolność do wykrywania defektów. Czasami — na przykład przy automatycznym przeprowadzaniu testowania regresji — paradoks pestycydów może być korzystny, ponieważ pozwala potwierdzić, że liczba defektów związanych z regresją jest niewielka (w przypadku testów regresji raczej zależy nam na tym, aby testy te były zawsze zaliczane).

6. Testowanie zależy od kontekstu

Jest to dosyć oczywista zasada: testowanie powinno być wykonywane w różny sposób w różnych sytuacjach. Na co innego zwracamy uwagę, gdy testujemy systemy krytyczne ze względu na bezpieczeństwo (ang. *life-critical*), na co innego, gdy testujemy systemy bankowe — tutaj najważniejsza jest dokładność funkcjonalna (na przykład prawidłowe wyliczanie odsetek od kapitału), a jeszcze na co innego, gdy testujemy gry komputerowe — tu zapewne bardziej istotne będą atrybuty niefunkcjonalne, takie jak wydajność (płynność grania) czy użyteczność (interesujący interfejs gry). Niemniej jednak także w grach należy zwracać uwagę na poprawność funkcjonalną (np. dwugłowy koń na rysunku 1.9).

Kup ksi k Pole ksi k

RYSUNEK 1.9. Dwugłowy koń w grze

Wspomniany w zasadzie „kontekst" ma tu bardzo szeroki zakres. Dotyczy m.in. charakteru tworzonego oprogramowania, dziedziny biznesowej, w ramach której oprogramowanie jest tworzone, ograniczeń projektowych i produktowych, wymagań funkcjonalnych, charakterystyki grupy użytkowników docelowych, wszelkiego rodzaju ryzyka i jego konsekwencji związanych z niepoprawnym działaniem oprogramowania, regulacji prawnych, praktyk, norm i zwyczajów obowiązujących w danej dziedzinie.

Konsekwencją tej zasady jest to, że nie ma jednego, uniwersalnego podejścia do testowania [Kaner 2011]. Testowanie ze swojej natury jest procesem intelektualnym, wymagającym wiedzy, umiejętności, a nierzadko — sporej dawki intuicji czy kreatywności.

7. Przekonanie o braku defektów jest błędem

Ciągle jeszcze niektóre organizacje oczekują, że testerzy będą w stanie uruchomić wszystkie możliwe testy i wykryć wszystkie możliwe defekty, ale powyższe zasady 1. i 2. pokazują, że jest to niemożliwe. Błędne jest też przekonanie, że samo znalezienie i naprawienie dużej liczby defektów zapewni pomyślne wdrożenie systemu, bowiem nawet aplikacja wolna od defektów (poprawna weryfikacja może nie spełniać wymagań użytkownika (niepoprawna walidacja).

Zasada ta mówi w gruncie rzeczy o tym, że w ramach procesu testowego sama weryfikacja nie wystarczy — potrzebna jest jeszcze walidacja, dzięki której upewnimy się, że program spełnia wymagania klienta, a nie tylko techniczne założenia, jakie poczynił zespół projektowy na podstawie wymagań [Boehm 1981]. Możemy bowiem stworzyć perfekcyjny, wolny od defektów produkt, który będzie z punktu widzenia użytkownika zupełnie bezużyteczny.

Skorowidz

A

akceptacja, 253, 256
analityk testów, 16
analiza
  Pareto, 55
  problemów, 161
  ryzyka, risk analysis, 273, 303
  statyczna, 147, 148, 156
  testów, test analysis, 35, 60, 67
  wartości brzegowych, AWB, 187, *Patrz także* technika testowania AWB
  wpływu, impact analysis, 142
anomalia, anomaly, 147, 161
architektura systemu, 118
aspekty psychologiczne, 76
atak
  DDoS, 156
  SQL injection, 156
  XSS, 156
ATDD, Acceptance Test-Driven Development, 100, 104, 188, 255
atrybuty ryzyka, 300
automatyzacja testów, test automation, 329
autor, 165
awarie, failure, 35, 39, 47, 114, 116, 121, 126

B

BDD, Behavior-Driven Development, 100, 102
białoskrzynkowa technika testowania, 87, 133, 187, 190, 231, 238, *Patrz także* testowanie
biblioteka JUnit5, 71

C

cele
  biznesowe, 18
  certyfikatu podstawowego, 17
  międzynarodowego systemu uzyskiwania kwalifikacji, 17
  nauczania, 19
  projektowe, 45
  testowania
    akceptacyjnego, 122
    integracyjnego, 115
    modułowego, 112
  testów, test objective, 35, 38
certyfikat poziomu podstawowego, 15
ciągła aktualizacja, 21
cykl wytwarzania, *Patrz* model cyklu wytwarzania
czarnoskrzynkowa technika testowania, 87, 134, 187, 190, 191, 195
czytanie oparte na perspektywie, 180, 181

D

dane testowe, test data, 35, 62
debugowanie, debugging, 39
defekt, defect, fault, 35, 39, 47, 54, 114, 121, 126
  w projekcie, 155
  w wymaganiach, 155
  zarządzanie, 317
defekty
  pasjans „Pająk", 42
  rakieta Ariane 5, 43
  rakieta Patriot 43, 44
definicja gotowości, definition of Ready, 280
diagram
  przejść, 372, 373
  przejść między stanami, 221
    budowa, 218
  stanów, 271
dobre praktyki testowania, 75, 99
doskonalenie procesów, 109
dziennik błędów, 162

Kup książkę Poleć książkę

E

efektywność spotkań przeglądowych, 163
egzamin, 381–397
  dodatkowe pytania, 399–409
  odpowiedzi, 411–436
  poziomu podstawowego, 21
  reguły, 28
  rozkład pytań, 29
  struktura, 27
  wskazówki, 31
ekstrapolacja, 285
elementy pokrycia, coverage item, 187, 200

F

facylitator, 165
formularz błędów, 163

G

gałąź, branch, 234

H

harmonogram wykonania testów, test execution schedule, 291
heurystyka Nielsena, 248
hipoteza błędu, 238
historyjka użytkownika, user story, 250, 252, 256

I

identyfikacja ryzyka, risk identification, 273, 303
implementacja testów, test implementation, 35, 62, 69
informacje
  o problemach, 161
  zwrotne, 157
inspekcje, inspections, 147, 171

J

jakość, 35
  procesu, 45
  produktu, 44

K

karta, card, 250
kategorie technik testowania, 189
kierownictwo, 164
kierownik testów, 16, 73
klasy równoważności, KR, 195, *Patrz także* technika testowania
  maskowanie defektów, 201
  podział dziedziny, 197
  podział klas, 199
  pokrycie, 200
  wykrywanie problemów, 203
  zastosowanie, 196
kluczowe wskaźniki wydajności, KPI, 57, 71
kontekst testowania, 276
kontrola
  jakości, quality control, QC, 46
  ryzyka, risk control, 273, 306
konwersacja, conversation, 251
koszt usunięcia defektu, 54
KPI, Key Performance Indicators, 57, 71
kryteria
  akceptacji, acceptance criteria, 187, 253, 256
    w postaci reguł, 254
    w postaci scenariusza, 253
  pokrycia, coverage, 57
  wejścia, entry criteria, 273, 280
  wyjścia, exit criteria, 273, 281
kwadranty testowe, testing quadrants, 273, 297

L

lider przeglądu, 166
listy kontrolne, 246
  inspekcja kodu, 249
  pokrycie, 248
  rodzaje, 247
  zastosowanie, 247

Ł

łagodzenie ryzyka, risk mitigation, 273, 306

M

maskowanie defektów, 201
maszyna stanowa
  formy reprezentacji, 221
  zdarzenia, 372

MCR, Modern Code Review, 164
metodyka DevOps, 105
metodyki wytwórcze, 95
metryki, 132, 133
  defektów, 310
  jakości produktu, 310
  kosztów, 310
  pokrycia, 310
  projektowe, 310
  ryzyka, 310
migracja, 141
minimalizacja tablicy decyzyjnej, 216
model cyklu wytwarzania, 89, 97, 99
  kaskadowy, 91
  prototypowania, 94
  spiralny Boehma, 94
  UP, Unified Process, 93
  V, V model, 92
modele
  iteracyjne i przyrostowe, 90, 93
  sekwencyjne, 89, 91
modyfikacja, 141
monitorowanie
  ryzyka, risk monitoring, 273, 308
  testów, test monitoring, 35, 59, 66, 273, 309

N

nadzór nad testami, test control, 35, 59, 66, 273, 309
narzędzia testowe, 27, 329
niepoprawne specyfikacje interfejsów, 155
niezależność testowania, 78
nowoczesne przeglądy kodu, MCR, 164

O

OAT, Operational Acceptance Testing, 123
ocena ryzyka, risk assessment, 273, 304
odchylenia od standardów, 155
odpowiedzialności, 114, 117, 121
operacyjne testy akceptacyjne, OAT, 123

P

paradoks pestycydów, 55
perspektywy, 181
piramida testów, test pyramid, 273, 296
planowanie, 159
  iteracji, 279
  testów, test planning, 35, 59, 65, 274–277
  wydania, 279
pluskwa, bug, 49
podejście
  „cały zespół”, 77
  „najpierw test”, 101
  do testowania, test approach, 274–276
podklasy, 199
podstawa testów, test basis, 22, 35, 60, 114, 121, 125
podstawowa przyczyna defektu, 35, 50
podział
  dziedziny, 197
  klas na podklasy, 199
  na klasy równoważności, 187, 195
pokrycie, coverage, 36, 187, 192
  „each choice”, 202
  gałęzi, branch coverage, 187, 235
  instrukcji kodu, 187
  przejść poprawnych, valid transitions coverage, 223
  wszystkich przejść, all transitions coverage, 223
  wszystkich stanów, all states coverage, 222
pomiary inspekcji, 172
pomyłka, error, 36, 47, 48
potwierdzenie, confirmation, 250, 251
poziomy
  K1, 19
  K2, 19
  K3, 20
  ryzyka, risk level, 274, 300
  testów, 87, 111, 128, 131, 134
praktyki zwinne, 95
priorytetyzacja, 138
  oparta na pokryciu, 292
  oparta na ryzyku, 292
  oparta na wymaganiach, 292
procedury testowe, test procedure, 36, 62
proces
  biznesowy, 176
  inspekcji, 174
  przeglądu, 157, 159
  testowy, 58
  testowy w kontekście, 63
programowanie ekstremalne, XP, 100

projektowanie testów, test design, 25, 36, 61, 68, 187
protokolant, 165
przedmiot testów, test object, 36, 87, 114, 116, 121, 125
przegląd, 147, 157, 159
  ad hoc, 178
  formalny, formal review, 147, 160
  indywidualny, 160
  koleżeński, 168
  nieformalny, informal review, 147, 168
  oparty na liście kontrolnej, 178
  oparty na rolach, role-based review, 180, 181
  techniczny, technical review, 147, 170
przeglądający, 166
przeglądy, reviews, 147, 157, 159
  czynniki sukcesu, 176
  kolejność, 167
  porównanie, 169
  przebiegi próbne, 179
  scenariusze, 179
  techniki, 178
  typy, 166
  typy defektów, 168
przejrzenie, walkthrough, 147, 170
przepływ sterowania kodu, 52
przesunięcie w lewo, shift-left, 87, 108
przypadek
  testowy, test case, 36, 61, 100, 291
  użycia, use case, 227
    budowa, 228
    pokrycie, 230

R

raport
  dzienny z testów, 67
  o defekcie, defect report, 274, 317, 378
  o postępie testów, test progress report, 274, 310
  sumaryczny z testów, test summary report, 310
raportowanie, 162
redukcja, 138
retrospektywy, 109
rola
  kierownika testów, 73
  testera, 74
role
  w procesie testowania, 72
  w przeglądach, 164
ryzyka, risk, 274, 300
  produktowe, product risk, 274, 302
  projektowe, project risk, 274, 301

S

schemat certyfikacji ISTQB, 16
Scrum, 95
spójność, 155
sprzężenie, 155
standard
  ISO 31000, 21
  ISO/IEC 20246, 21
  ISO/IEC 25010, 21
  ISO/IEC/IEEE 29119, 21
  ISO/IEC/IEEE 29119-3, 312, 318
strategie
  integracji, 117
  mieszane, 138
  oparte na
    architekturze systemu, 118
    innych aspektach systemu, 118
    sekwencjach przetwarzania transakcji, 118
    zadaniach funkcjonalnych, 118
  testów
    analityczna, 278
    kierowana, 278
    metodyczna, 278
    minimalizująca regresję, 278
    oparta na modelu, 278
    reaktywna, 278
    zgodna z procesem, 278
  wstępujące, bottom-up, 118
  zstępujące, top-down, 118
suche przebiegi, dry runs, 179
sumaryczny raport z testów, 274
sylabus 4.0, 22
szacowanie, 282, 284
  trójpunktowe, 287
  wysiłku testowego, 321
szerokopasmowa metoda delficka, 285

Ś

śledzenie powiązań, traceability, 71

T

tabela przejść między stanami, 221
tablica decyzyjna, 212
  budowa, 213
  kombinacje nieosiągalne, 216
  minimalizacja, 216
  możliwe wartości, 214
  notacja, 214
  pokrycie, 217
  przypadki testowe, 214
  testowanie statyczne, 218
  wyznaczanie kombinacji warunków, 214
  zastosowanie, 212
tablice Kanban, 97
TDD, Test-Driven Development, 100, 101
techniczny analityk testów, 16
technika testowania, test technique, 187, 189, *Patrz także* testowanie
  AWB, 206
    pokrycie, 212
    wersja dwupunktowa, 207
    wersja trójpunktowa, 207
    wyznaczanie wartości brzegowych, 210
    zastosowanie, 210
  KR, 195
    maskowanie defektów, 201
    podział dziedziny, 197
    podział klas, 199
    pokrycie, 200
    wykrywanie problemów, 203
    zastosowanie, 196
  zgadywanie błędów, 188, 240
techniki testowania
  charakterystyka, 188
  kategorie, 189
  poziom formalizacji, 194
  w sylabusie, 192
  wybór, 193
techniki przeglądu, 178
testalia, testware, 36, 64
tester, 74
testowanie, 36, 37, 41, 44, 45, 51
  akceptacyjne, acceptance testing, 87, 122
    operacyjne, OAT, 123
    przez użytkownika, UAT, 123
    zgodności z prawem, 124
    zgodności z umową, 124
  alfa, 124
  beta, 124
  białoskrzynkowe, white-box testing, 87, 133, 187, 190, 231, 238
  czarnoskrzynkowe, black-box testing, 87, 134, 187, 190, 191, 195
  dobre praktyki, 75, 99
  dynamiczne, dynamic testing, 147, 149, 154
  eksploracyjne, exploratory testing, 187, 243
    stosowanie, 244
    w sesjach, 244
  funkcjonalne, functional testing, 87, 128
  gałęzi, 235
    hipoteza błędu, 238
    pokrycie, 234, 235
  gruntowne, 52
  instrukcji
    hipoteza błędu, 238
    pokrycie, 232, 239
  integracyjne, integration testing, 87, 115
    modułów, component integration testing, 87, 115, 117
    systemów, system integration testing, 87, 115, 117
  MC/DC, 232
  modułowe, component testing, 87, 112, 114
  narzędzia wspomagające, 329
  niefunkcjonalne, non-functional testing, 88, 130, 131
  oparte na
    doświadczeniu, 187, 191, 240
    przypadkach użycia, 227
    ryzyku, risk-based testing, 274, 299
    współpracy, collaboration-based test approach, 187, 250
  pętli, 232
  pielęgnacyjne, maintenance testing, 88, 140
  podstawy, 35
  potwierdzające, confirmation testing, 88, 137
  pracochłonność, 290
  przejść między stanami, state transition testing, 188, 218
    pokrycie, 222
  przekazywanie informacji, 313
  regresji, regression testing, 88, 138
  statyczne, static testing, 24, 147–150, 154
  stosowane metryki, 309
  systemowe, system testing, 88, 120, 121

testowanie
  ścieżek liniowo niezależnych, 232
  w cyklu wytwarzania oprogramowania, 23, 88
  w oparciu o
    listę kontrolną, checklist-based testing, 188, 246
    tablicę decyzyjną, decision table testing, 188, 212, 218
  warunków wielokrotnych, 232
  wczesne, 53
  zależne od kontekstu, 55
  zasady, 51
typy
  defektów, 155, 168
  przeglądów, 166
  testów, 88, 111, 127, 134

U

UAT, User Acceptance Testing, 123
ukończenie testów, test completion, 36, 63
umiejętności, 45, 75
usterka, 48
usuwanie defektów, 153, 162

W

walidacja, validation, 36, 38
warunek testowy, test condition, 36, 60
weryfikacja, verification, 36, 37
wycofanie, 141
wykonywanie testu, test execution, 36, 62
wymagania, 20
wyniki testów, test result, 36, 50
wysiłek testowy, test effort, 282, 321
wytwarzanie oprogramowania, 88
  sterowane testami, TDD, 100, 101
  sterowane testami akceptacyjnymi, ATDD, 100, 104, 188, 255
  sterowane zachowaniem, BDD, 100, 102

X

XP, eXtreme Programming, 100

Z

zadania testowe, 57
zapewnienie jakości, quality assurance, QA, 36, 46
zarządzanie
  czynnościami testowymi, 26
  defektami, defect management, 274, 317
  jakością, quality management, QM, 45
  ryzykiem, risk management, 274, 299
zasady testowania, 51
zgadywanie błędów, error guessing, 188, 240
złożoność
  cyklomatyczna, cyclomatic complexity, 107
  cyklomatyczna modułu, 156

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA Helion

Testerem być!

Testowanie oprogramowania to jedna z najdynamiczniej rozwijających się dziedzin inżynierii oprogramowania. Zarobki doświadczonych testerów są porównywalne z wynagrodzeniem, jakie otrzymują dobrzy programiści. Aby rozpocząć karierę w tym zawodzie, trzeba się legitymować odpowiednimi umiejętnościami. I dlatego warto uzyskać certyfikat ISTQB®: Certyfikowany tester — poziom podstawowy. Jest to uznawany na całym świecie dokument świadczący o opanowaniu najważniejszych kompetencji z zakresu kontroli jakości oprogramowania.

Powszechnie dostępne w sieci materiały dotyczące sylabusa, a także egzaminu umożliwiającego zdobycie certyfikatu ISTQB® są niestety często niepełne lub nieaktualne, zwłaszcza w odniesieniu do najnowszego sylabusa w wersji 4.0 z 2023 roku, w znaczący sposób różniącej się od poprzedniej, pochodzącej sprzed pięciu lat. Aby uniknąć niepotrzebnej straty czasu i frustracji związanej z wielokrotnym podchodzeniem do egzaminu, warto sięgnąć po rzetelne źródło wiedzy. Ta publikacja — napisana przez współautorów sylabusa w wersji 4.0 — pozwala szybko opanować zagadnienia wymagane na egzaminie, tak aby bez stresu poradzić sobie z uzyskaniem certyfikatu.

Sięgnij po najlepsze źródło wiedzy!

Patroni:

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-0439-2

Cena: 109,00 zł