ADAM ROMAN
LUCJAN STAPP

Certyfikowany tester ISTQB®

POZIOM PODSTAWOWY

Podręcznik do samodzielnej nauki na podstawie nowego sylabusa z 2018 roku

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Jan Paluch

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

ISBN: 978-83-8322-185-4

Gliwice 2020, 2022

Printed in Poland.

Spis treści

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 testu — 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, system lub proces spełnia określone wymagania i/lub spełnia potrzeby i oczekiwania klienta lub użytkownika.

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

nadzór nad testami — działalność, która rozwija i stosuje 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 dla analizy i projektowania testów.

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

pokrycie — wyrażony w procentach stopień, w jakim określone elementy pokrycia zostały określone lub sprawdzone przez zestaw testowy. *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 ustawienia warunków wstępnych i wszelkich czynności podsumowujących po wykonaniu.

proces testowy — zbiór powiązanych ze sobą działań obejmujący planowanie i monitorowanie testów, ich analizę, projektowanie, implementację i zakończenie.

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

przedmiot testów — moduł lub system podlegający testowaniu.

przypadek testowy — zestaw warunków wstępnych, danych wejściowych, akcji (w stosownych przypadkach), oczekiwanych rezultatów i warunków końcowych opracowany na podstawie warunków testowych.

śledzenie — stopień, w jakim można ustalić relację pomiędzy dwoma produktami prac lub ich większą liczbą.

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 życia, zarówno statycznych, jak i dynamicznych, skoncentrowany na planowaniu, przygotowaniu i ewaluacji oprogramowania oraz powiązanych produktów w celu określenia, czy spełniają one wyspecyfikowane wymagania, oraz wykazania, że są one dopasowane do swoich celów oraz do 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 — egzaminowanie 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.

wyrocznia testowa — źródło określające oczekiwane wyniki w celu porównania z faktycznym wynikiem testowanego systemu. *Synonim*: wyrocznia.

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

zestaw testowy — zestaw wykonywanych przypadków testowych lub procedur testowych. *Synonimy*: zestaw przypadków testowych, zestaw testów.

1.1. Co to jest testowanie

FL-1.1.1 (K1)	Kandydat potrafi wskazać typowe cele testowania.
FL-1.1.2 (K2)	Kandydat potrafi odróżnić 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), a kończąc na urządzeniach dla konsumenta (samochody), rozrywce (gry komputerowe) czy komunikacji. Używanie oprogramowania z błędami może:

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

Dlatego dobre testowanie jest niezbędne. Problemem jest to, że wiele osób (także pracujących w IT) za testowanie uważa tylko wykonywanie testów, to jest uruchamianie oprogramowania w celu znalezienia defektów. Jednak wykonywanie testów stanowi tylko część testowania. Istnieją jeszcze inne czynności związane z testowaniem. Czynności te są wykonywane zarówno przed wykonaniem testów (punkty 1. – 5. poniżej), jak i po ich wykonaniu (punkt 7. poniżej):

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

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.

Co więcej, często testowanie jest postrzegane jako czynność skupiona wyłącznie na **weryfikacji** wymagań, historyjek użytkownika lub innych form specyfikacji (tzn. sprawdzeniu, czy system

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

Warto pamiętać, że testowanie to *technologiczne badanie* pozwalające otrzymać informacje o jakości testowanego produktu:

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

Podstawowe **cele testowania** to:

- dokonywanie oceny produktów pracy, takich jak wymagania, historyjki użytkownika, projekt, kod;
- sprawdzanie, czy zostały spełnione wszystkie wyspecyfikowane wymagania;
- sprawdzanie, czy przedmiot testów jest kompletny i działa zgodnie z oczekiwaniami użytkowników i innych interesariuszy;
- budowanie zaufania do poziomu jakości przedmiotu testów;
- wykrywanie defektów i awarii;
- zapobieganie awariom;
- dostarczanie interesariuszom informacji niezbędnych do podejmowania świadomych decyzji dotyczących zwłaszcza poziomu jakości przedmiotu testów;
- obniżanie poziomu ryzyka związanego z jakością oprogramowania (ryzyka wystąpienia niewykrytych wcześniej awarii podczas eksploatacji);
- przestrzeganie wymagań wynikających z umów, przepisów prawa i norm/standardów;
- sprawdzanie, czy obiekt testów jest zgodny z tymi wymaganiami lub standardami.

Różne cele wymagają różnych strategii testowania. W przypadku testowania modułowego — tzn. testowania pojedynczych fragmentów aplikacji/systemu (więcej o testowaniu modułowym w podrozdziale 2.2 „Poziomy testów") — 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 (więcej o testowaniu akceptacyjnym także w podrozdziale 2.2 „Poziomy testów") 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.

Część osób uważa, że testowanie polega na debugowaniu. Należy pamiętać, że testowanie a debugowanie to dwie odmienne aktywności. Testowanie (zwłaszcza dynamiczne) ma ujawnić awarie spowodowane defektami. **Debugowanie** natomiast jest czynnością programistyczną wykonywaną w celu

zidentyfikowania przyczyny defektu, poprawienia kodu i sprawdzenia, czy defekt został poprawnie naprawiony. Późniejsze testowanie potwierdzające (retest) wykonywane przez testera ma zapewnić, że poprawka rzeczywiście usunęła awarię. Testerzy są odpowiedzialni za przeprowadzenie testu początkowego oraz końcowego testu potwierdzającego, programiści — za przeprowadzenie debugowania oraz związanego z nim testowania modułowego. Jednakże w przypadku zwinnego wytwarzania oprogramowania i niektórych innych cykli życia testerzy mogą być również zaangażowani w debugowanie i testowanie modułowe. Należy jednak pamiętać, że podstawową zasadą jest: „Testerzy testują, a programiści debugują".

1.2. Dlaczego testowanie jest niezbędne

FL-1.2.1 (K2)	Kandydat potrafi podać przykłady wskazujące, dlaczego testowanie jest niezbędne.
FL-1.2.2 (K2)	Kandydat potrafi opisać relację między testowaniem a zapewnieniem jakości oraz podać przykłady wskazujące, w jaki sposób testowanie przyczynia się do podnoszenia jakości.
FL-1.2.3 (K2)	Kandydat potrafi rozróżnić pomyłkę, defekt i awarię.
FL-1.2.4 (K2)	Kandydat potrafi odróżnić podstawową przyczynę od skutków defektu.

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.

1. Gra 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 malejąco. Gdy doprowadzi się do sytuacji jak na rysunku 1.1 — na stole jest 9 kart, w grupach jest 30 (co daje w sumie $3 \cdot 13$ — 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 koszt naprawy tej sytuacji?
- jakie jest prawdopodobieństwo jej zajścia?

W tym wypadku koszt jest 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 — jest rzędu 10^{-6}. Gdy koszt jest bliski zera, a prawdopodobieństwo minimalne, defektów — zwłaszcza w systemach niekrytycznych — na ogół się nie usuwa.

Rysunek 1.1. *Gra pasjans „Pająk”*

2. Przypadek Ariane 5.

4 czerwca 1996 roku rakieta Ariane 5, przygotowana przez Europejska 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 błąd 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.

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

3. Przypadek rakiety Patriot.

Każda bateria rakiet systemu Patriot składa się z radaru, stanowiska dowodzenia (komputera) i mobilnych wyrzutni (patrz rysunek 1.2). 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.

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.

Rysunek 1.2. *Bateria rakiet Patriot*

Był to błąd arytmetyczny. System mierzył czas w dziesiątych częściach sekundy, używając 24-bitowego stałoprzecinkowego rejestru (ang. *24 bit fixed point register*). Po około 100 godzinach pracy błąd numeryczny 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 Scud jest poza jego zasięgiem.

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.

Rygorystyczne testowanie systemów i dokumentacji może zredukować ryzyko wystąpienia problemów (awarii) w środowisku produkcyjnym i przyczynić się do osiągnięcia wysokiej jakości 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 życia oprogramowania — częstotliwość występowania tego rodzaju problemów można jednak ograniczyć. Przykładami mogą tu być:

- Zaangażowanie testerów w przeglądy wymagań lub w doprecyzowywanie historyjek użytkownika pomaga wykryć defekty w powyższych produktach pracy.
- Ścisła współpraca testerów z projektantami systemu na etapie prac projektowych umożliwia obu stronom lepsze zrozumienie projektu.
- Ścisła współpraca testerów z programistami na etapie tworzenia kodu pozwala obu stronom lepiej zrozumieć kod.

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.

Testowanie jest często 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*). **Zarządzanie jakością** obejmuje wszystkie czynności mające na celu kierowanie działaniami organizacji w dziedzinie jakości i ich nadzorowanie.

Elementami zarządzania jakością są między innymi:

- zapewnienie jakości,
- kontrola jakości.

Zapewnienie jakości skupia się na przestrzeganiu właściwych procesów w celu uzyskania pewności, że zostaną osiągnięte odpowiednie poziomy jakości. Jeśli procesy są wykonywane prawidłowo, powstające w ich wyniku produkty pracy mają zwykle wyższą jakość, co przyczynia się do zapobiegania defektom. Duże znaczenie dla skutecznego zapewnienia jakości ma również wykorzystanie procesu analizy przyczyny podstawowej do wykrywania i usuwania przyczyn defektów oraz prawidłowe stosowanie wniosków ze spotkań retrospektywnych w celu doskonalenia procesów.

Natomiast **kontrola jakości** obejmuje cały szereg czynności, także 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 testowania) jest istotny z punktu widzenia zapewnienia jakości, w związku z czym zapewnienie jakości wspiera właściwe testowanie.

W podejściu ISTQB® rozróżniamy trzy poziomy powstania nieprawidłowego wyniku:

- **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 człowieka w kodzie oprogramowania lub w innym związanym z nim produkcie pracy może powstać defekt. Uruchomienie tego fragmentu kodu, w którym jest defekt, może (ale nie musi!) spowodować awarię. 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 — mówimy: „w tym miejscu w kodzie jest błąd". Z punktu widzenia terminologii ISTQB® jest to niepoprawne. W programie znajdować się może *defekt*. *Błąd* zawsze dotyczy pomyłki ludzkiej.

Przykład. Rozważmy 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]:

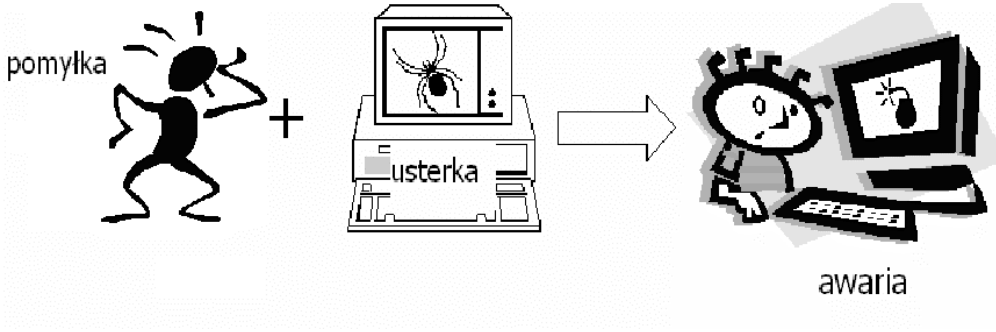
```
program Zlicz
wejście: tablica T (o elementach T[0], T[1], …, T[n])
wyjście: liczba komórek T zawierających zero


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) zależeć będzie od tego, czy komórka T[0] zawiera zero, czy inną liczbę. W pierwszym przypadku 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 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 występować bardzo rzadko lub nigdy. Tylko równoczesne zaistnienie trzech czynników (błąd — defekt — awaria) powoduje obserwowane nieprawidłowe działanie testowanego produktu (rysunek 1.3).



Rysunek 1.3. *Pomyłka, usterka, awaria*

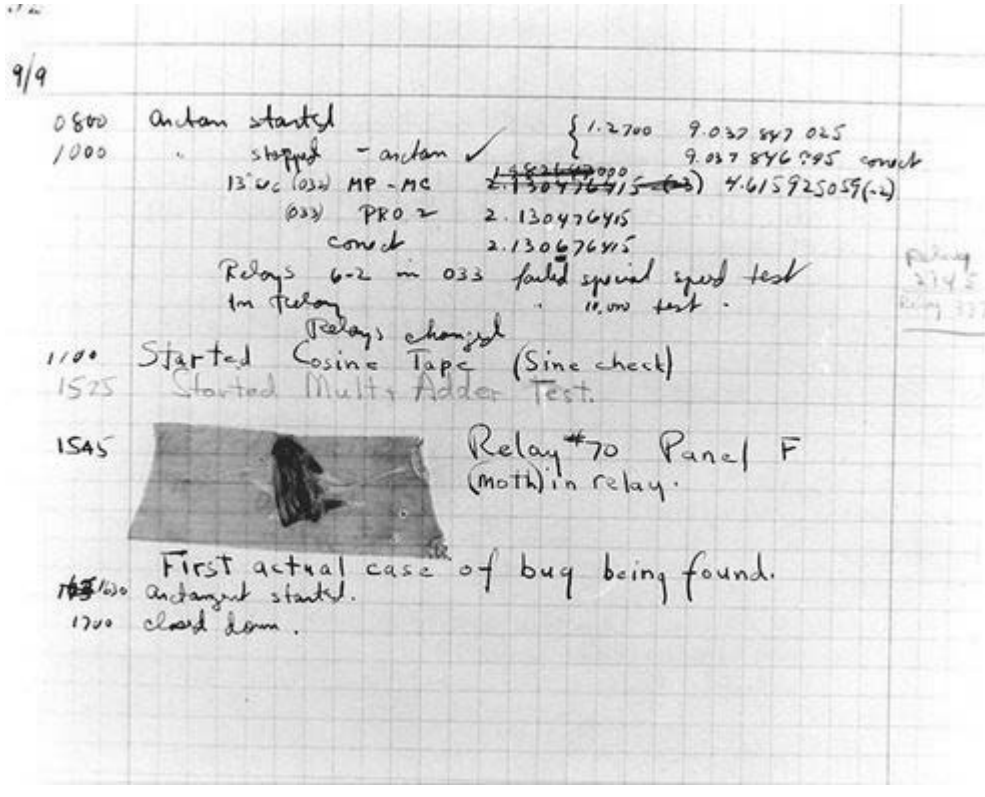
Pomyłki (błędy) mogą pojawiać się z wielu powodów:

- 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 1.4. *Pierwsza „pluskwa" w historii (www.atlasobscura.com/places/grace-hoppers-bug)*

Pierwsza pluskwa (ćma) w historii została znaleziona przez admirał Hopper w 1947 roku. Rysunek 1.4 przedstawia fragment oryginalnego raportu Hopper, zawierającego… rzeczywistą ćmę — przyczynę awarii.

Nie wszystkie nieoczekiwane wyniki testów oznaczają awarie. Wynik **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 — wyniku **fałszywie negatywnego**, czyli sytuacji, w której testy nie wykrywają defektu, który powinny wykryć. Więcej na temat wyników fałszywie pozytywnych i fałszywie negatywnych czytelnik znajdzie w punkcie 2.2.3 „Testowanie systemowe".

Dlatego ważnym czynnikiem jest analiza **podstawowej przyczyny** defektu: pierwotnego powodu, w wyniku którego defekt ten powstał. 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.

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. Siedem zasad testowania

FL-1.3.1 (K2) Kandydat potrafi objaśnić siedem zasad testowania.

Istnieje wiele różnych „praw" czy „zasad" związanych z testowaniem, które są prawdziwe niezależnie od kontekstu projektowego czy rodzaju wytwarzanego produktu. Sylabus poziomu podstawowego opisuje siedem takich zasad. Te podstawowe zasady testowania to:

1. Testowanie ujawnia usterki, ale nie może dowieść ich braku.
2. Testowanie gruntowne jest niemożliwe.
3. Wczesne testowanie oszczędza czas i pieniądze.
4. Kumulowanie się defektów.
5. Paradoks pestycydów.
6. Testowanie jest zależne od kontekstu.
7. Przekonanie o braku błędów[1] 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". 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 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.

[1] Chodzi oczywiście o przekonanie o braku defektów, nie błędów. Zostajemy tu przy słowie „błąd" ze względów czysto stylistycznych.

2. Testowanie gruntowne jest niemożliwe.

Rozpatrzmy następujący przykład [Myers 2005]. Mamy do przetestowania fragment kodu z czterema warunkami (A, B, C, D) i sześcioma instrukcjami (I1, I2, I3, I4, I5, I6), przedstawiony na rysunku 1.5, 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):

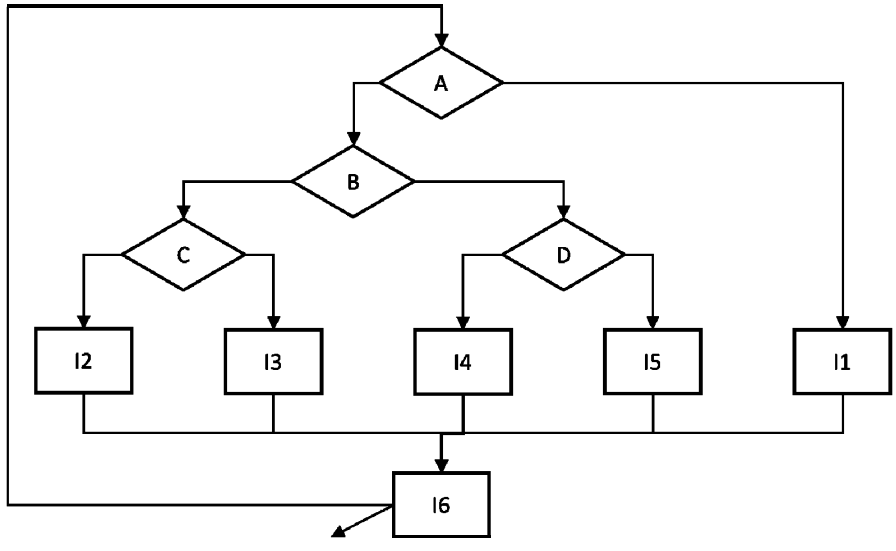
S1. A I_1 I_6

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 1.5. *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 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, 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). 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.

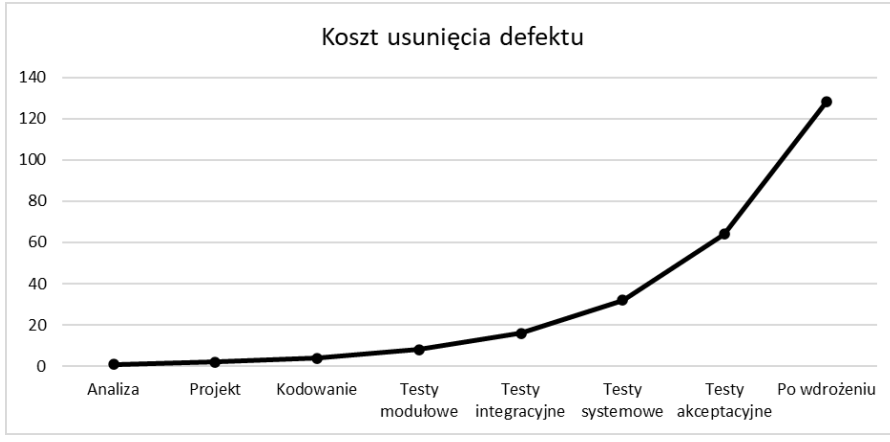
3. Wczesne testowanie oszczędza czas i pieniądze.

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. 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 1.6 przedstawiona jest słynna tzw. krzywa Boehma, która ilustruje zależność kosztu usunięcia defektu od momentu 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.

4. Kumulowanie się defektów.

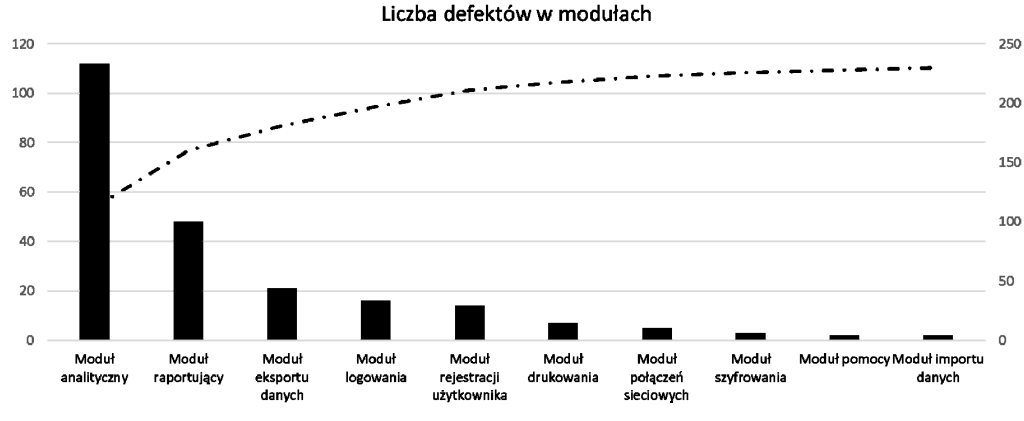
Defekty nie rozkładają się równomiernie ani w oprogramowaniu, ani w czasie. Większość defektów znalezionych podczas testowania przed wypuszczeniem oprogramowania lub powodujących awarie produkcyjne znajduje się w małej liczbie modułów. W rezultacie przewidywane skupiska defektów i skupiska defektów faktycznie 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.



Rysunek 1.6. *Koszt usunięcia defektu w funkcji czasu*

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 1.7 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).



Rysunek 1.7. *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. Paradoks pestycydów.

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

Niezmieniane testy tracą z czasem zdolność do wykrywania defektów, podobnie jak pestycydy po pewnym czasie nie są zdolne do eliminowania szkodników. Czasami — na przykład przy automatycznym przeprowadzaniu testowania regresji — paradoks pestycydów może być korzystny, ponieważ pozwala potwierdzić, że liczba defektów związanych z regresją jest niewielka.

6. Testowanie jest zależne 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 (dwugłowy koń na rysunku 1.8).

Rysunek 1.8. *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 itp.

7. Przekonanie o braku błędów 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. Możemy stworzyć perfekcyjny, wolny od defektów produkt, który będzie z punktu widzenia użytkownika zupełnie bezużyteczny.

1.4. Proces testowy

FL-1.4.1 (K2)	Kandydat potrafi wyjaśnić wpływ kontekstu na proces testowy.
FL-1.4.2 (K2)	Kandydat potrafi opisać czynności testowe i odpowiadające im zadania w ramach procesu testowego.
FL-1.4.3 (K2)	Kandydat potrafi rozróżnić produkty pracy wspomagające proces testowy.
FL-1.4.4 (K2)	Kandydat potrafi wyjaśnić korzyści wynikające ze śledzenia powiązań między podstawą testów a produktami pracy związanymi z testowaniem.

Nie istnieje jeden uniwersalny proces testowania oprogramowania. Istnieją natomiast typowe czynności testowe, bez stosowania których w testowaniu trudno by było osiągnąć ustalone cele. Czynności te tworzą **proces testowy**. Organizacja może w swojej strategii testowej zdefiniować, które czynności testowe wchodzą w skład tego procesu, jak czynności testowe mają być zaimplementowane oraz kiedy powinny mieć miejsce. Dobór procesu testowego do konkretnego oprogramowania zależy od wielu czynników, takich jak:

- wykorzystywany model cyklu życia oprogramowania i metodyki projektowe,
- rozważane poziomy testów i typy testów,
- czynniki ryzyka produktowego i projektowego,
- dziedzina biznesowa,
- wymagania wynikające z umów i przepisów,

- ograniczenia operacyjne:
  - budżety i zasoby,
  - harmonogramy;
- złożoność dziedziny,
- polityka testów i praktyki obowiązujące w organizacji,
- wymagane normy/standardy wewnętrzne i zewnętrzne.

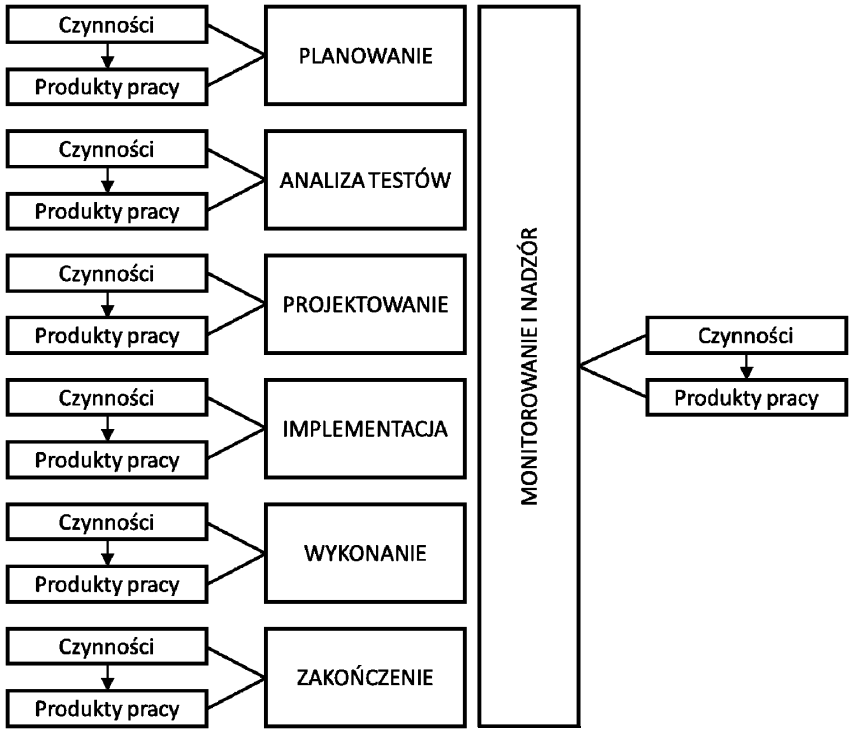
Dobrą praktyką jest zdefiniowanie mierzalnych **kryteriów pokrycia** dotyczących podstawy testów (w odniesieniu do każdego rozważanego poziomu lub typu testów). W praktyce mogą one pełnić funkcję tzw. kluczowych wskaźników wydajności (ang. *Key Performance Indicators*, KPI) sprzyjających wykonywaniu określonych czynności i pozwalających wykazać osiągnięcie celów testowania (np. kryteria pokrycia mogą wymagać co najmniej jednego testu dla każdego elementu podstawy testów).

Proces testowy może — ale nie musi — być zdefiniowany formalnie. W typowych sytuacjach składa się z następujących grup czynności:

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

Grupy procesu testowego przedstawione powyżej są logicznie sekwencyjne, ale w rzeczywistym procesie mogą występować iteracyjnie (np. w podejściu zwinnym), zazębiać się, występować jednocześnie (np. w testowaniu eksploracyjnym) albo być pomijane. Zależy to od konkretnego projektu.

W ramach procesu testowego powstają **produkty pracy** (patrz rysunek 1.9). W różnych organizacjach mogą mieć różne typy i różne nazwy. Dobrym punktem odniesienia dla produktów pracy związanych z testowaniem jest standard międzynarodowy ISO/IEC/IEEE 29119-3. Warto zauważyć, że wiele produktów pracy związanych z testowaniem może być tworzonych i zarządzanych przy użyciu narzędzi do zarządzania testami oraz narzędzi do zarządzania defektami.



Rysunek 1.9. *Czynności i produkty pracy w procesie testowym*

Poniżej omówione zostaną przedstawione na rysunku 1.9 czynności i produkty pracy w ramach poszczególnych grup procesu testowego.

Planowanie testów — czynności:

- zdefiniowanie celów testowania,
- określenie czynności testowych potrzebnych do wypełnienia misji i zrealizowania celów testowania,
- określenie podejścia do osiągania celów testowania w granicach wyznaczonych przez kontekst,
- określenie odpowiednich technik testowania i zadań testowych,
- sformułowanie harmonogramu testów, który umożliwi dotrzymanie wyznaczonego terminu,
- zdefiniowanie miar.

Planowanie testów — produkty pracy:

- plan testów (czasami kilka).

Plan testów zawiera informacje na temat podstawy testów, z którą zostaną powiązane za pośrednictwem informacji śledzenia inne produkty pracy związane z testowaniem. Określa się w nim kryteria wyjścia (definicję ukończenia), które będą stosowane w ramach monitorowania testów i nadzoru nad nimi. Plany testów mogą być korygowane na podstawie informacji zwrotnych z monitorowania i nadzoru. Należy pamiętać, że proces planowania jest procesem ciągłym, nie kończy się w początkowej fazie projektu. Korekta planów może nastąpić w dowolnym momencie cyklu życia wytwarzania oprogramowania.

Przykład. Szablon planu testów akceptacyjnych (na podstawie [Web1]).

Plan testów akceptacyjnych systemu XYZ

Nr ID, autor, data, historia zmian

1. Wprowadzenie (cel dokumentu, podstawa jego opracowania, opis systemu).

2. Strategia testów.

2.1. Założenia wstępne.

2.1.1. Warunki wejścia dla rozpoczęcia testów (dostępna i wymagana dokumentacja, zaakceptowany harmonogram testów, warunki logistyczno-organizacyjne, wykonane testy wewnętrzne, dostępna dokumentacja z testów wewnętrznych).

2.1.2. Warunki wejścia dla kolejnych iteracji testów (naprawione defekty z poprzednich iteracji, ewentualnie brak defektów o odpowiednio wysokim priorytecie, dokonanie koniecznych modyfikacji w dokumentacji testowej w zakresie wynikającym z poprzedniej iteracji, osiągnięcie odpowiedniego pokrycia).

2.1.3. Rodzaje testów akceptacyjnych (opis rodzajów wykonywanych testów akceptacyjnych, określenie miejsca ich przeprowadzenia, określenie, kto ma je przeprowadzić, np. testy alfa, testy beta, testy akceptacyjne użytkownika, operacyjne testy akceptacyjne, określenie ról interesariuszy w przeprowadzaniu tych testów — np. rola wykonawcy, rola zamawiającego).

2.2. Organizacja testów.

2.2.1. Zasoby osobowe (wymagane role, kwalifikacje, składy zespołów, opis ról).

2.2.2. Procedura testowania (ogólne i szczegółowe procedury opisujące przebieg testów akceptacyjnych: warunki wstępne, sposób przekazywania informacji, np. danych testowych przez wykonawcę).

2.2.3. Klasyfikacja defektów (opis klasyfikacji defektów, np. krytyczny/poważny/o niskim priorytecie, wraz z opisem działań w razie wystąpienia defektu danej kategorii).

2.2.4. Procedura zgłaszania uwag (proces zgłaszania defektów, używane narzędzia).

2.2.5. Obsługa defektów (proces obsługi zgłoszeń defektów i ich rozwiązywania).

2.2.6. Raportowanie postępów (opis zasad raportowania prac, rodzaje raportów).

2.2.7. Warunki akceptacji i odbioru poszczególnych rodzajów testów.

2.3. Środowiska testowe (opis środowisk, infrastruktury oraz danych testowych, sposobu ich pozyskiwania i utrzymywania; podział danych na słownikowe i operacyjne, opis procedur postępowania z danymi).

2.4. Harmonogram testów.

2.5. Typy i poziomy przeprowadzanych testów (np. testy regresji, testy scenariuszowe, testy eksploracyjne).

3. Plan testów

3.1. Sekwencja realizacji testów (kolejność realizacji poszczególnych działań testerskich).

3.2. Przypadki testowe (hierarchiczna lista powiązanych ze sobą przypadków testowych ze śledzeniem do scenariuszy testowych, ryzyk i innych istotnych elementów; mogą być zawarte w odrębnych dokumentach).

3.3. Scenariusze testowe (specyfikacja scenariuszy; może być zawarta w odrębnym dokumencie).

4. Załączniki (np. wyciągi z umów z klientem, wzory raportów i inna niezbędna dokumentacja).

Monitorowanie i nadzór — czynności

Monitorowanie testów to ciągłe porównywanie rzeczywistego i planowanego postępu testowania przy użyciu miar specjalnie w tym celu zdefiniowanych w planie testów. Nadzór nad testami to aktywne podejmowanie działań, które są niezbędne do osiągnięcia celów wyznaczonych w planie testów (z uwzględnieniem jego ewentualnych aktualizacji). Działania te podejmowane są na podstawie informacji pochodzących z monitorowania.

Element wspomagający monitorowanie testów i nadzór nad nimi to ocena kryteriów wyjścia (w podejściu zwinnym często nazywanych definicją ukończenia) z planu testów, która obejmuje:

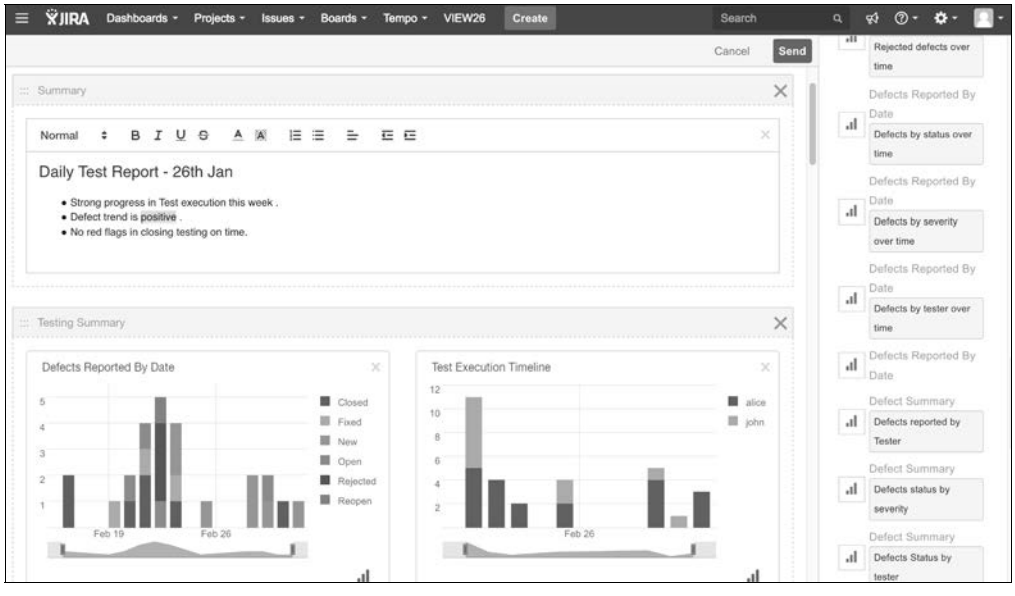
- sprawdzenie rezultatów i dziennika (logu) testów pod kątem określonych kryteriów pokrycia,
- oszacowanie poziomu jakości modułu lub systemu na podstawie rezultatów i dziennika (logu) testów,
- ustalenie, czy są konieczne dalsze testy (w przypadku nieosiągnięcia przez dotychczas wykonane testy pierwotnie założonego poziomu pokrycia ryzyka produktowego; wiąże się to z koniecznością napisania i wykonania dodatkowych testów),
- informowanie interesariuszy o postępie w realizacji planu testów,
- raporty o postępie testów.

Monitorowanie i nadzór — produkty pracy

Zasadniczymi produktami pracy w tej grupie są raporty z testów. Raporty o postępie testów są tworzone na bieżąco i/lub w regularnych odstępach czasu. Sumaryczne raporty końcowe z testów tworzone są w momencie osiągnięcia poszczególnych kamieni milowych. Powinny zawierać szczegółowe informacje na temat dotychczasowego przebiegu procesu testowego, podsumowanie rezultatów wykonywania testów oraz informacje o ewentualnych odchyleniach od planu. Raport zawsze powinien być dostosowany do potrzeb konkretnych odbiorców.

W raportach powinno uwzględniać się ponadto kwestie dotyczące zarządzania projektem, informacje o ukończonych zadaniach oraz o alokacji i zużyciu zasobów.

Przykład. Przykładowy, sumaryczny raport z testów przedstawiony jest na rysunku 1.10. Pokazuje on dzienny raport z testów w porównaniu z dniem poprzednim. Lewy wykres opisuje rozkład liczby defektów w określonych interwałach czasowych, w podziale na kategorie (*Closed* — zamknięty, *Fixed* — naprawiony, *New* — nowy, *Open* — otwarty, *Rejected* — odrzucony, *Reopen* — otwarty ponownie). Prawy wykres przedstawia raport z wykonania testów w podziale na dwóch testerów (Alice i John).



Rysunek 1.10. *Przykładowy raport dzienny z testów (źródło: acommunity.atlassian.com)*

Analiza testów — czynności

Zadaniem analizy testów jest zapoznanie się z podstawą testów i przeanalizowanie jej w celu zidentyfikowania testowalnych cech i zdefiniowania związanych z nimi warunków testowych oraz ustalenia, „co" należy przetestować (w kategoriach *mierzalnych* kryteriów pokrycia).

Ogólne **cele testowania** zostają następnie przekształcone w konkretne **warunki testowe** i **projekty testów**. Czynności wykonywane w ramach analizy testów pozwalają zweryfikować, czy wymagania:

- są spójne,
- są prawidłowo wyrażone,
- są kompletne,
- właściwie odzwierciedlają potrzeby klienta, użytkowników i innych interesariuszy.

Techniki takie jak wytwarzanie sterowane zachowaniem (ang. *Behavior Driven Development* — BDD) i wytwarzanie sterowane testami akceptacyjnymi (ang. *Acceptance Test Driven Development* — ATDD) obejmują generowanie warunków testowych i przypadków testowych na podstawie historyjek użytkownika i kryteriów akceptacji przed rozpoczęciem tworzenia kodu, przewidują weryfikację i walidację historyjek użytkowników i kryteriów akceptacji, wykrywanie występujących w nich defektów.

Czynności w grupie analizy obejmują:

- zapoznanie się z podstawą testów: specyfikacja wymagań: wymagania biznesowe, wymagania funkcjonalne, wymagania systemowe, historyjki użytkownika, opowieści (ang. *epic*), przypadki użycia, podobne produkty pracy; określają one pożądane zachowanie funkcjonalne i niefunkcjonalne modułu lub systemu;
- analizę informacji dotyczących projektu i implementacji takich jak diagramy lub dokumenty opisujące architekturę systemu lub oprogramowania, specyfikacje projektowe, przepływy wywołań, modele oprogramowania (diagramy UML [UML 2017] lub diagramy związków encji, specyfikacje interfejsów lub podobne produkty pracy); dokumenty te określają strukturę modułu lub systemu;
- analizę implementacji samego modułu lub systemu: kodu, metadanych, zapytań do bazy danych oraz interfejsów;
- analizę raportów z analizy ryzyka, które mogą dotyczyć aspektów funkcjonalnych, niefunkcjonalnych i strukturalnych modułu lub systemu;
- dokonanie oceny testowalności podstawy, by zidentyfikować często występujące typy defektów: niejednoznaczności, pominięcia, niespójności, nieścisłości, sprzeczności, nadmiarowe (zbędne) instrukcje; identyfikowanie defektów jest istotną korzyścią, gdy nie stosuje się żadnego innego procesu przeglądu i/lub gdy proces testowy jest ściśle powiązany z procesem przeglądu;
- zidentyfikowanie cech i zbiorów cech, które mają zostać przetestowane;
- zdefiniowanie warunków testowych w odniesieniu do poszczególnych cech oraz określenie ich priorytetów na bazie analizy podstawy testów — z uwzględnieniem parametrów funkcjonalnych, niefunkcjonalnych i strukturalnych, innych czynników biznesowych i technicznych oraz poziomów ryzyka;
- stworzenie możliwości dwukierunkowego śledzenia powiązań między elementami podstawy testów a związanymi z nimi warunkami testowymi.

Analiza testów — produkty pracy

Typowym produktem pracy fazy analizy są **warunki testowe**, zdefiniowane i uszeregowane według priorytetów. Ustala się ponadto dwukierunkowe **śledzenie** powiązań (macierz śledzenia) między tymi warunkami a pokrywanymi przez nie elementami podstawy testów. W przypadku testowania eksploracyjnego produktem pracy mogą być karty opisu testów. Innym produktem pracy mogą być zgłoszenia o wykryciu defektów w podstawie testów.

Przykład. Jeśli testujemy funkcjonalność logowania do serwisu, możemy na przykład zidentyfikować następujące warunki testowe:

- poprawne logowanie (istniejący użytkownik, poprawny login i hasło),
- niepoprawne logowanie (istniejący użytkownik, złe hasło),
- niepoprawne logowanie z blokadą konta (trzykrotne wpisanie złego hasła),
- niepoprawne logowanie (nieistniejący użytkownik).

Projektowanie testów — czynności

Podczas projektowania testów warunki testowe są przekształcane w **przypadki testowe wysokiego poziomu**, zbiory takich przypadków testowych oraz w inne testalia. Projektowanie testów odpowiada na pytanie „jak należy testować". Czynności tej grupy obejmują:

- zaprojektowanie (zbiorów) przypadków testowych i określenie ich priorytetów,
- zidentyfikowanie niezbędnych danych testowych,
- zaprojektowanie środowiska testowego,
- zidentyfikowanie wszelkich niezbędnych narzędzi i elementów infrastruktury,
- stworzenie możliwości dwukierunkowego śledzenia powiązań między podstawą testów, warunkami testowymi, przypadkami testowymi i procedurami testowymi (rozbudowanie macierzy śledzenia),
- identyfikację typowych defektów w podstawie testów.

Często używane są tu techniki testowania, omówione w rozdziale 4.

Projektowanie testów — produkty pracy

W wyniku projektowania testów powstają **przypadki testowe** i **zbiory testów** zdefiniowane na etapie analizy testów.

Często dobrą praktyką jest projektowanie przypadków testowych wysokiego poziomu, które nie zawierają konkretnych wartości danych wejściowych i oczekiwanych rezultatów. Wysokopoziomowe przypadki testowe można wykorzystywać wielokrotnie w różnych cyklach testowych z użyciem różnych danych szczegółowych, należycie dokumentując przy tym zakres przypadku testowego. W idealnej sytuacji dla każdego przypadku testowego istnieje możliwość dwukierunkowego śledzenia powiązań między tym przypadkiem a pokrywanym przez niego warunkiem testowym (lub warunkami testowymi).

Wśród rezultatów etapu projektowania testów można również wymienić zaprojektowanie i/lub zidentyfikowanie niezbędnych danych testowych, zaprojektowanie środowiska testowego oraz zidentyfikowanie infrastruktury i narzędzi, przy czym stopień udokumentowania powyższych rezultatów bywa bardzo zróżnicowany.

Warunki testowe zdefiniowane w fazie analizy mogą być doprecyzowane podczas projektowania testów.

Przykład. Poniższy przykład pokazuje przypadek testowy wraz z krokami do wykonania (ang. *test steps*).

PT nr 20.002.11	Autor: Jan Kowalski
Priorytet: średni	Data: 11.07.2020
Funkcja: logowanie	
Opis: próba logowania z poprawnym loginem i hasłem przy użyciu klawisza *Enter* zamiast klikania na guzik logowania	
Warunki wstępne: użytkownik dysponuje loginem i hasłem	
Krok/Dane testowe.	Oczekiwany wynik
Otwórz stronę logowania.	Strona załadowana
Wpisz login „jankowalski”.	Login przyjęty
Wpisz hasło „haslo123”.	Hasło przyjęte, domyślny aktywny guzik *Loguj*
Wciśnij *Enter*.	Logowanie do serwisu
Warunki wyjścia: użytkownik zalogowany, fakt logowania zapisany do bazy wraz z czasem logowania	

Implementacja testów — czynności

W tej grupie tworzone i/lub kończone są testalia niezbędne do wykonania testów, w tym szeregowanie przypadków testowych w ramach **procedur testowych**. W związku z tym, o ile projektowanie testów odpowiada na pytanie „jak testować”, o tyle implementacja testów odpowiada na pytanie „czy mamy wszystko, co jest potrzebne do uruchomienia testów?”.

Zadania związane z projektowaniem i implementacją testów są często łączone. Etapy projektowania i implementacji testów mogą być realizowane i dokumentowane w ramach wykonywania testów zwłaszcza w przypadku testowania eksploracyjnego:

- testowanie eksploracyjne może odbywać się na podstawie kart opisu testów (sporządzanych w ramach analizy testów);
- testy eksploracyjne są wykonywane natychmiast po zaprojektowaniu i zaimplementowaniu.

Ta praktyka ma zastosowanie także w przypadku innych typów testowania opartego na doświadczeniu.

Czynności fazy implementacji obejmują:

- opracowanie procedur testowych i określenie ich priorytetów;
- utworzenie skryptów testów automatycznych;

- utworzenie zestawów testowych (na podstawie procedur testowych) oraz skryptów testów automatycznych (jeśli używane są automaty testowe);
- uporządkowanie zestawów testowych w harmonogram wykonywania testów, tak by zapewnić efektywny przebieg całego procesu;
- zbudowanie środowiska testowego, włączając w to — jeśli to konieczne — jarzma testowe, (sterowniki (ang. *driver*) i zaślepki (ang. *stub*)), wirtualizację usług, symulatory i inne elementy infrastrukturalne, oraz sprawdzenie, czy zostało ono poprawnie skonfigurowane;
- przygotowanie danych testowych i sprawdzenie, że zostały one poprawnie załadowane do środowiska testowego;
- zweryfikowanie i zaktualizowanie możliwości dwukierunkowego śledzenia powiązań między podstawą testów, warunkami testowymi, przypadkami testowymi, procedurami testowymi i zestawami testowymi.

Implementacja testów — produkty pracy

W ramach tej grupy czynności tworzone są następujące produkty pracy:

- procedury testowe oraz kolejność ich wykonywania,
- zestawy testowe,
- harmonogram wykonania testów,
- dane testowe.

Dane testowe służą do przypisywania konkretnych wartości do danych wejściowych i oczekiwanych rezultatów przypadków testowych. Te konkretne wartości, wraz z konkretnymi wskazówkami ich użycia, przekształcają przypadki testowe wysokiego poziomu w wykonywalne **przypadki testowe niskiego poziomu**. Ten sam przypadek testowy wysokiego poziomu można wykonywać z użyciem różnych danych testowych w odniesieniu do różnych wersji przedmiotu testów. Konkretne wyniki oczekiwane związane z określonymi danymi testowymi identyfikuje się przy użyciu wyroczni testowej.

W idealnym przypadku można wykazać spełnienie kryteriów pokrycia określonych w planie testów dzięki możliwości dwukierunkowego śledzenia powiązań pomiędzy procedurami testowymi a elementami podstawy testów.

Czasami produktem pracy tej grupy jest opis wykorzystania narzędzi (np. do wirtualizacji usług) czy skryptów testów automatycznych. W przypadku testowania eksploracyjnego niektóre produkty pracy mogą powstawać już w trakcie wykonywania testów, a stopień udokumentowania testów eksploracyjnych i możliwości śledzenia powiązań do określonych elementów podstawy testów może być bardzo różny.

Przykład. Załóżmy, że testujemy funkcję `mnóż(x, y)`, która przyjmuje na wejściu dwie liczby całkowite i zwraca ich iloczyn. W trakcie analizy testów zdefiniowano m.in. trzy przypadki testowe związane z mnożeniem przez zero:

- PT1: pierwszy parametr jest zerem, a drugi jest różny od zera;
- PT2: drugi parametr jest zerem, a pierwszy różny od zera;
- PT3: oba parametry są równe zeru.

Dla przypadków tych zdefiniowano następujące dane wejściowe i oczekiwane wyjścia:

- PT1: x = 0, y = 10, oczekiwane wyjście: 0;
- PT2: x = 10, y = 0, oczekiwane wyjście: 0;
- PT3: x = 0, y = 0, oczekiwane wyjście: 0.

Poniższy automatyczny skrypt testowy jest przykładem implementacji trzech przypadków testowych: PT1, PT2, PT3 w języku Java z użyciem biblioteki JUnit (służącej do wspomagania tworzenia testów jednostkowych). Instrukcja `assertEquals` sprawdza, czy dwa pierwsze jej parametry mają tę samą wartość. W naszym przypadku sprawdzamy, czy wyniki iloczynów $10 \cdot 0$, $0 \cdot 10$ oraz $0 \cdot 0$ będą rzeczywiście równe 0.

Poniższy kod został zaadaptowany ze strony *www.vogella.com/tutorials/JUnit/article.html*.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyTests {
    @Test
    public void mnozenieLiczbCalkowitychPrzezODaje0() {
        MyClass testy = new MyClass(); // testy klasy MyClass

        // asercje implementujące przypadki PT1, PT2, PT3
        assertEquals(0, testy.mnóż(10, 0), "10 x 0 musi być 0");
        assertEquals(0, testy.mnóż(0, 10), "0 x 10 musi być 0");
        assertEquals(0, testy.mnóż(0, 0), "0 x 0 musi być 0");
    }
}
```

Wykonanie testów — czynności

W tej grupie wykonywane są następujące czynności:

- zarejestrowanie danych identyfikacyjnych i wersji elementów testowych bądź przedmiotu testów, narzędzi testowych i testaliów,
- wykonywanie testów ręcznie lub przy użyciu narzędzi,
- porównanie rzeczywistych wyników testów z oczekiwanymi,
- przeanalizowanie anomalii w celu ustalenia ich prawdopodobnych przyczyn:
  - defekty w kodzie,
  - wyniki fałszywie pozytywne;
- raportowanie defektów oparte na obserwowanych awariach,
- zalogowanie wyniku wykonania testów (zaliczony, niezaliczony, test blokujący),

- powtórzenie czynności testowych (testowanie potwierdzające, wykonanie poprawionego testu, testowanie regresji),
- zweryfikowanie i zaktualizowanie możliwości dwukierunkowego śledzenia powiązań.

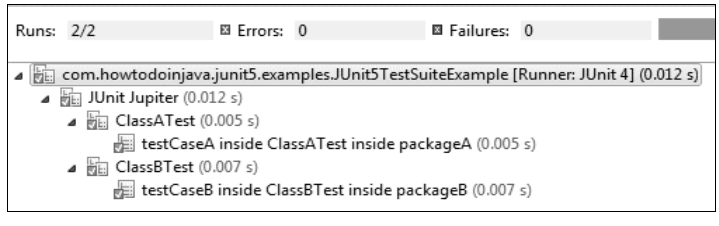
Wykonanie testów — produkty pracy

Typowym produktem pracy wykonania testów są:

- dokumentacja dotycząca statusu poszczególnych przypadków testowych lub procedur testowych (gotowy do wykonania, zaliczony, niezaliczony, zablokowany, celowo pominięty),
- raporty o defektach,
- dokumentacja wskazująca, co zostało wykorzystane w ramach testowania.

W sytuacjach idealnych można również raportować status poszczególnych elementów podstawy testów. Raporty na ten temat są wykonalne dzięki możliwości dwukierunkowego śledzenia powiązań z odpowiednimi procedurami testowymi. Można np. wskazać, które wymagania zostały całkowicie przetestowane i pomyślnie przeszły testy, które nie przeszły testów i/lub wiążą się z defektami, a które nie zostały jeszcze w pełni przetestowane. Umożliwia to sprawdzenie, czy zostały spełnione kryteria pokrycia testowego, i przedstawienie rezultatów testów w raportach w sposób zrozumiały dla interesariuszy.

Przykład. Na rysunku 1.11 pokazano ekran raportu z wykonania dwóch testów jednostkowych: testCaseA oraz testCaseB przy użyciu biblioteki JUnit5. Kolory ikon przy nazwach testów symbolizują rezultaty testów: kolor zielony oznacza, że test jest zdany, kolor czerwony — niezdany. W naszym przypadku oba testy są zdane (zaliczone).



Rysunek 1.11. *Wynik wykonania testów jednostkowych w JUnit (źródło: howtodoinjava.com)*

Ukończenie testów — czynności

W fazie ukończenia testów następuje zebranie danych pochodzących z wykonanych czynności testowych w celu skonsolidowania zdobytych doświadczeń, testaliów oraz innych stosownych informacji. Wykonywane jest to w momencie osiągnięcia kamieni milowych projektu, takich jak:

- przekazanie systemu oprogramowania do eksploatacji,
- zakończenie realizacji (lub anulowanie) projektu,
- zakończenie iteracji projektu zwinnego (np. w ramach spotkania retrospektywnego),
- ukończenie poziomu testów,
- zakończenie prac nad wydaniem pielęgnacyjnym.

W ramach grupy ukończenia wykonywane są poniższe czynności:

- sprawdzenie, czy wszystkie raporty o defektach są zamknięte;
- wprowadzenie żądań zmian lub pozycji do rejestru produktu w odniesieniu do wszelkich defektów, które nie zostały rozwiązane do momentu zakończenia wykonywania testów;
- utworzenie sumarycznego raportu z testów, który zostanie przekazany interesariuszom;
- sfinalizowanie i zarchiwizowanie środowiska testowego, danych testowych, infrastruktury testowej oraz innych testaliów do ponownego wykorzystania w przyszłości.

Ukończenie testów — produkty pracy

Typowymi produktami prac w ramach fazy ukończenia testów są **sumaryczne raporty** z ukończenia testów, opisujące czynności do wykonania mające na celu wprowadzenie udoskonaleń w kolejnych projektach lub iteracjach (np. po retrospektywie projektu zwinnego), żądania zmian, pozycje listy zaległości produktowych oraz sfinalizowane testalia.

Śledzenie powiązań

Należy pamiętać, że produkty pracy związane z testowaniem i ich nazwy mogą być bardzo różne. W celu zapewnienia skutecznego monitorowania i nadzoru istotne jest stworzenie i utrzymanie mechanizmu śledzenia powiązań między każdym elementem podstawy testów a odpowiadającymi mu produktami pracy związanymi z testowaniem przez cały czas trwania procesu testowego. Sprawne śledzenie umożliwia:

- ocenę pokrycia testowego,
- analizowanie wpływu zmian,
- przeprowadzanie audytu testów,
- spełnianie kryteriów związanych z zarządzaniem w obszarze IT,
- tworzenie zrozumiałych raportów o statusie testów i sumarycznych raportów z testów,
- uwzględnienie statusu elementów podstawy testów (wymagania, dla których testy zostały zaliczone, niezaliczone lub czekają na wykonanie),
- przekazywanie interesariuszom informacji o aspektach technicznych testowania w zrozumiałej dla nich formie,
- udzielanie informacji potrzebnych do oceny jakości produktów, możliwości procesów oraz postępu w realizacji projektu z punktu widzenia celów biznesowych.

Znak X oznacza, że dany przypadek testowy pokrywa dane ryzyko. Na przykład PT 001 pokrywa ryzyka 1 i 2, PT 003 — ryzyko 3 itd. Załóżmy, że PT 001, 002 i 004 zakończyły się sukcesem, a PT 003 jest niezdany. Może to oznaczać, że ryzyko 3 nie zostało pokryte (jeśli wymagamy, by pokrycie ryzyka oznaczało zdanie *wszystkich* powiązanych z nim testów) lub na przykład, że zostało pokryte w 50% (jeśli definiujemy pokrycie ryzyka jako odsetek związanych z nim przypadków testowych, które zakończyły się sukcesem).

Przykład. Rozważmy następującą macierz śledzenia przypadków testowych do zidentyfikowanych ryzyk:

	Ryzyko 1	Ryzyko 2	Ryzyko 3	Ryzyko 4
PT 001	X	X		
PT 002		X		
PT 003			X	
PT 004	X		X	X

1.5. Psychologia testowania

FL-1.5.1 (K1)	Kandydat potrafi wskazać czynniki psychologiczne wpływające na powodzenie testowania.
FL-1.5.2 (K2)	Kandydat potrafi wyjaśnić różnice w sposobie myślenia testerów i programistów.

Proces wytwarzania oprogramowania, w tym jego testowanie, jest realizowany głównie przez ludzi, w związku z czym duże znaczenie dla przebiegu testowania mają uwarunkowania psychologiczne. Identyfikowanie defektów podczas testowania statycznego lub identyfikowanie awarii w trakcie testowania dynamicznego może być odbierane jako krytyka produktu lub jego autora. Istnieje zjawisko psychologiczne zwane **efektem potwierdzenia** lub błędem konfirmacji (ang. *confirmation bias*) — tendencja do preferowania informacji, które potwierdzają wcześniejsze oczekiwania i hipotezy, niezależnie od tego, czy te informacje są prawdziwe, czy nie, może utrudniać zaakceptowanie informacji sprzecznych z dotychczasowymi przekonaniami. Powoduje, że ludzie poszukują informacji i zapamiętują je w sposób selektywny, interpretując je w błędny sposób. Efekt ten jest szczególnie silny w przypadku zagadnień wywołujących duże emocje i dotyczących mocno ugruntowanych opinii. Przykładowo, czytając o polityce dostępu do broni, ludzie zwykle preferują źródła, które potwierdzają to, co sami na ten temat sądzą. Mają również tendencję do interpretowania niejednoznacznych dowodów jako potwierdzających ich własne zdanie.

Serie eksperymentów prowadzonych w latach 60. XX wieku pokazały, że ludzie mają tendencję do poszukiwania potwierdzeń dla swoich wcześniejszych przeświadczeń. Późniejsze badania wyjaśniły to jako wynik tendencji do testowania hipotez bardzo selektywnie, skupiania się na jednej możliwości i ignorowania alternatywy. W połączeniu z innymi efektami taka strategia wpływa na konkluzje, do jakich ludzie dochodzą. Błąd ten może wynikać z ograniczonych możliwości przetwarzania informacji przez ludzki mózg lub z ewolucyjnej optymalizacji, gdy szacowane koszty tkwienia w błędzie nie są większe niż koszty analizy prowadzonej w obiektywny, naukowy sposób.

Przykład. Testerzy są przekonani, że zgłaszane przez nich awarie są efektem defektów w kodzie; występuje u nich efekt potwierdzenia, przez który trudno jest im zaakceptować sytuacje, gdy nie ma defektu, a awaria powstała na skutek nieprawidłowego wykonania testu (wynik fałszywie pozytywny).

Występują również inne błędy poznawcze, które utrudniają ludziom zrozumienie czy zaakceptowanie informacji uzyskanych w wyniku testowania. Ludzie mają często skłonność do obwiniania osoby przynoszącej złe wiadomości, a takie sytuacje często wynikają z testowania. Co więcej, niektórzy postrzegają testowanie jako czynność destrukcyjną, nawet jeśli przyczynia się ono wydatnie do postępu w realizacji projektu i podnoszenia jakości produktów. Aby ograniczyć podobne reakcje, należy przekazywać informacje o defektach i awariach w sposób jak najbardziej konstruktywny. Należy dążyć do zmniejszenia napięcia między testerami a analitykami biznesowymi, właścicielami produktów, projektantami i programistami. Ma to zastosowanie zarówno w testowaniu statycznym, jak i w testowaniu dynamicznym.

Testerzy i kierownicy testów muszą mieć duże umiejętności interpersonalne, aby sprawnie przekazywać informacje na temat defektów, awarii, rezultatów testów, postępu testowania czy ryzyka i budować pozytywne relacje ze współpracownikami. W związku z tym sugeruje się następujące zasady postępowania:

- Współpracuj, a nie walcz.
- Podkreślaj korzyści wynikające z testowania (autorzy mogą wykorzystać informacje o defektach do doskonalenia produktów pracy i rozwijania umiejętności, a dla organizacji wykrycie i usunięcie defektów podczas testowania oznacza oszczędność czasu i pieniędzy oraz zmniejszenie ogólnego ryzyka związanego z jakością produktów).
- Przekazuj informacje na temat rezultatów testów i inne wnioski w sposób neutralny (koncentruj się na faktach, nie krytykuj autorów wadliwego elementu/rozwiązania).
- Twórz raporty o defektach i wnioski z przeglądu w sposób obiektywny i opierając się na faktach.
- Próbuj zrozumieć, dlaczego druga osoba negatywnie reaguje na podane informacje.
- Upewnij się, że rozmówca rozumie przekazywane informacje i *vice versa*. Ma to istotne znaczenie, zwłaszcza gdy pracujesz w projekcie rozproszonym.

Jednoznaczne zdefiniowanie właściwego zbioru celów testów ma istotne implikacje psychologiczne, bo większość ludzi ma skłonność do dopasowywania swoich planów i zachowań do celów określonych przez zespół, kierownictwo i innych interesariuszy. Należy dążyć, by testerzy przestrzegali przyjętych założeń, a ich osobiste nastawienie w jak najmniejszym stopniu wpływało na wykonywaną pracę. Trzeba także pamiętać, że sposób myślenia danej osoby wyznaczają przyjmowane przez nią założenia oraz preferowane przez nią sposoby podejmowania decyzji i rozwiązywania problemów.

Programiści i testerzy prezentują często różny sposób myślenia. Podstawowym celem prac programistycznych jest zaprojektowanie i wykonanie produktu, natomiast podstawowe cele testowania obejmują weryfikację i walidację produktu oraz wykrycie defektów przed przekazaniem produktu do eksploatacji.

Typowe cechy osobowościowe programisty to: zainteresowanie projektowaniem i budowaniem rozwiązań, wyraźny efekt potwierdzenia, natomiast rzadziej występuje analizowanie problemów w przyjętych rozwiązaniach. Programiści mający odpowiednie nastawienie mogą testować własny kod.

Pożądane cechy osobowościowe testera to: ciekawość, „zawodowy pesymizm", umiejętność krytycznego spojrzenia na wykonywane czynności, dbałość o szczegóły, motywacja do utrzymywania sprawnych, pozytywnych relacji zawodowych.

Sposób myślenia testera często ewoluuje i dojrzewa wraz z nabieraniem przez niego doświadczenia. Niezależni testerzy, wykonując czynności testowe, zwiększają skuteczność wykrywania defektów. Mają inny punkt widzenia, różny od punktów widzenia autorów produktów pracy (tj. analityków biznesowych, właścicieli produktów, projektantów i programistów), ponieważ mają inne **uprzedzenia poznawcze** (ang. *cognitive biases*).

Pytania testowe do rozdziału 1.

Pytanie 1.1

(FL-1.x, K1)

Usterka to:

A. Działanie człowieka powodujące powstawanie nieprawidłowego wyniku.
B. Materializacja w kodzie pomyłki twórcy oprogramowania.
C. Odchylenie od spodziewanego zachowania oprogramowania.
D. Przypadek testowy sprawdzający reakcję na błędne dane.

Wybierz jedną odpowiedź.

Pytanie 1.2

(FL-1.1.1, K1)

Które z poniższych NIE jest typowym celem testowania?

A. Sprawdzanie, czy przedmiot testów jest kompletny.
B. Wykrycie jak największej liczby awarii w czasie testów akceptacyjnych.
C. Obniżanie poziomu ryzyka wystąpienia niewykrytych wcześniej awarii podczas eksploatacji.
D. Budowanie zaufania do poziomu jakości testowanego systemu.

Wybierz jedną odpowiedź.

Pytanie 1.3

(FL-1.1.2, K2)

Poniżej podana jest lista czynności będących skutkiem defektów w oprogramowaniu:

i. znajdowanie defektów w kodzie,
ii. ujawnianie awarii,
iii. usuwanie znalezionych defektów,
iv. wykonywanie testów potwierdzających.

Które z tych czynności są zadaniami dewelopera, a które testera w przypadku tradycyjnych modeli wytwarzania oprogramowania?

A. (i), (ii), (iv) — tester, (iii) — programista.
B. (ii), (iv) — tester, (i), (iii) — programista.
C. (ii), (iii) — tester, (i), (iv) — programista.
D. (ii), (iii), (iv) — tester, (i) — programista.

Wybierz jedną odpowiedź.

Pytanie 1.4

(FL-1.2.1, K2)

Jesteś testerem w projekcie tworzącym grę na podstawie mitów greckich. Podczas tworzenia kryterium akceptacyjnego dla następującej historyjki użytkownika:

> *Jako* gracz poziomu 4.
>
> *chcę* móc użyć różdżki Midasa,
>
> *by* zamienić stojący przede mną przedmiot w złoto i powiększyć swoje zasoby finansowe.

zauważyłeś, że brak jest informacji dotyczącej czasu zamiany przedmiotu w złoto.

Na czym polegał Twój wkład w osiągnięcie powodzenia projektu?

A. Wskazanie, że twórca historyjki użytkownika niepoprawnie wykonał swoje zadanie.
B. Zmniejszenie ryzyka wytworzenia niepoprawnej lub nietestowalnej cechy jakościowej oprogramowania.
C. Wymuszenie, by właściciel produktu uzupełnił natychmiast brakujące dane.
D. Zapewnienie komfortu działania przyszłym graczom.

Wybierz jedną odpowiedź.

Pytanie 1.5

(FL-1.2.2, K2)

Testowanie podnosi jakość oprogramowania:

A. Tylko, jeśli wykryte defekty zostaną naprawione.
B. Poprzez zwiększanie zaufania do stabilności oprogramowania.
C. Poprzez mierzenie jakości oprogramowania.
D. Poprzez dokumentowanie ukrytych i zamaskowanych usterek w oprogramowaniu.

Wybierz jedną odpowiedź.

Pytanie 1.6

(FL-1.2.3, K2)

Które z poniższych zdań NIE jest prawdziwe?

A. Awarie mogą być spowodowane warunkami środowiskowymi.
B. Stosowanie nowych, nieznanych technologii może być powodem awarii
C. W czasie testów statycznych znajdujemy głównie awarie.
D. Defekty pojawiają się z powodu omylności człowieka.

Wybierz jedną odpowiedź.

Pytanie 1.7

(FL-1.2.4, K2)

Pracujesz w projekcie, w ramach którego tworzona jest nowa wersja oprogramowania dla automatów do kawy. Twój kierownik projektu bardzo podkreślał, że automat ma być oszczędny i zużywać tak mało prądu elektrycznego, jak to jest tylko możliwe. W czasie testowania stwierdziłeś, że automat przygotowuje zimną kawę, gdy wybrano cappuccino.

Która z poniższych odpowiedzi opisuje podstawową przyczynę powstania tej sytuacji?

A. Nacisk kierownika projektu na oszczędność prądu przez automat.
B. Błąd przy wyliczaniu czasu potrzebnego na podgrzanie wody.
C. Brak wiedzy inżyniera w kwestii wymagań dotyczących zasad działania podgrzewacza wody w tego typu urządzeniach.
D. Nieprawidłowo zaprojektowany interfejs użytkownika.

Wybierz jedną odpowiedź.

Pytanie 1.8

(FL-1.3.1, K2)

Zgodnie z zasadą Pareto większość problemów spowodowana jest małą liczbą przyczyn. Jest to podstawą jednej z zasad testowania. Której?

A. Wczesne testowanie oszczędza czas i pieniądze.
B. Kumulowanie się defektów.
C. Paradoks pestycydów.
D. Testowanie zależy od kontekstu.

Wybierz jedną odpowiedź.

Pytanie 1.9

(FL-1.4.1, K2)

Co NIE ma istotnego wpływu na proces testowy w organizacji?

A. Budżet.
B. Normy i standardy zewnętrzne.
C. Liczba certyfikowanych testerów zatrudnionych w organizacji.
D. Znajomość dziedziny biznesowej.

Wybierz jedną odpowiedź.

Pytanie 1.10

(FL-1.4.2, K2)

Podczas której fazy procesu testowego sprawdzana jest testowalność podstawy testów?

A. Planowanie testów.
B. Analiza testów.
C. Projektowanie testów.
D. Implementacja testów.

Wybierz jedną odpowiedź.

Pytanie 1.11

(FL-1.4.3, K2)

Który z poniższych NIE jest produktem pracy powstającym podczas monitorowania i nadzorowania testów?

A. Raport o postępie testów.
B. Sumaryczny raport końcowy z testów.
C. Raport o defektach.
D. Raport o alokacji i zużyciu zasobów.

Wybierz jedną odpowiedź.

Pytanie 1.12

(FL-1.4.4, K2)

Dlaczego tak ważne jest stworzenie i utrzymywanie mechanizmu śledzenia powiązań między każdym elementem podstawy testów a odpowiadającymi mu produktami pracy związanymi z testowaniem przez cały czas trwania procesu testowego?

i. umożliwia przeprowadzanie audytu testów;
ii. umożliwia udzielania informacji potrzebnych do oceny jakości produktów;
iii. umożliwia obliczenie nakładu pracy potrzebnego do wykonania zmian w przypadkach testowych wskazanych przez analizę wpływu po dokonaniu zmian w wymaganiach;
iv. umożliwia efektywne monitorowanie testów, ułatwiając obliczenie pozostałego ryzyka;
v. umożliwia obliczenie pokrycia testami strukturalnymi.

A. (i) oraz (iii) jest prawdą, (ii), (iv) oraz (v) jest fałszem.
B. (ii), (iii) oraz (iv) jest prawdą, (i) oraz (v) jest fałszem.
C. (i), (ii) oraz (iv) jest prawdą, (iii) oraz (v) jest fałszem.
D. (i), (iii) oraz (iv) jest prawdą, (ii) oraz (v) jest fałszem.

Wybierz jedną odpowiedź.

Zadanie 1.13

(FL-1.5.1, K1)

Jakkolwiek testowanie jest czynnością konstruktywną, wiele osób uważa ją za destruktywną. Jakie jest uzasadnienie tego fenomenu?

A. Niezależne testowanie jest postrzegane jako działanie „korkujące" proces wytwórczy.
B. Testowanie jest postrzegane jako kosztowny proces nieprzynoszący żadnej wartości.
C. Deweloperzy postrzegają znajdowanie defektów jako krytykę ich działalności.
D. W zespołach zwinnych testerzy bez umiejętności programowania — zwłaszcza automatów testowych — są postrzegani jako bezwartościowi członkowie zespołu.

Wybierz jedną odpowiedź.

Pytanie 1.14

(FL-1.5.2, K2)

Która z poniższych cech testera jest najmniej krytyczna?

A. Umiejętność programowania.
B. Dbałość o szczegóły.
C. Umiejętność krytycznego spojrzenia.
D. Dobra komunikacja z otoczeniem.

Wybierz jedną odpowiedź.

Skorowidz

R

S

Ś

T

U

W

Z

Zostań certyfikowanym testerem ISTQB®!

- Poznaj sylabus
- Zdaj egzamin
- Zdobądź certyfikat
- Zostań testerem

Branża IT kusi licznymi ofertami pracy, atrakcyjnymi zarobkami i ciekawymi ścieżkami rozwoju nie tylko zawodowych programistów, lecz także osoby bez wykształcenia informatycznego czy doświadczenia w tej dziedzinie. Te ostatnie zwykle zaczynają karierę od roli testera aplikacji. Jednym z podstawowych wymogów stawianych początkującym testerom jest posiadanie certyfikatu ISTQB®, uznawanego na całym świecie dokumentu świadczącego o opanowaniu najważniejszych kompetencji z zakresu kontroli jakości oprogramowania komputerowego.

W sieci dostępnych jest sporo informacji na temat sylabusa i egzaminu umożliwiającego zdobycie podstawowego certyfikatu ISTQB®, materiały te jednak często są niepełne lub nieaktualne. Aby uniknąć niepotrzebnej straty czasu oraz frustracji związanej z wielokrotnym podchodzeniem do egzaminu, warto sięgnąć po rzetelne źródło wiedzy. Książka *Certyfikowany tester ISTQB®. Poziom podstawowy* pozwoli Ci w krótkim czasie opanować materiał wystarczający, aby bez stresu poradzić sobie z procesem certyfikacji. Oprócz dokładnego omówienia treści nowego sylabusa w wersji 2018 v.3.1 znajdziesz tu również zestawy przykładowych pytań, które pomogą Ci sprawdzić swoje kompetencje i utrwalić zdobytą wiedzę. Znajdziesz tutaj:

- Omówienie struktury i zasad przeprowadzania egzaminu
- Zestaw praktycznych rad pomocnych przy zdawaniu egzaminu
- Dokładne omówienie treści sylabusa z licznymi praktycznymi przykładami
- Definicje pojęć (słów kluczowych) wymaganych na egzaminie
- 73 oryginalne pytania testowe pokrywające wszystkie cele nauczania z sylabusa
- 10 oryginalnych zadań, zgodnych z wymogami szkoleniowymi
- 3 oficjalne przykładowe egzaminy ISTQB® (120 pytań testowych)
- Poprawne odpowiedzi do wszystkich pytań i zadań wraz z obszernym uzasadnieniem

Rozpocznij karierę profesjonalnego testera oprogramowania!

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 978-83-8322-185-4
9 788383 221854
Cena: 89,00 zł