

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

Helion

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie 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 właścicielami.

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_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-289-1265-6

Copyright © Adam Roman, Lucjan Stapp, Michaël Pilaeten 2024

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [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

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

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 testowymi273	
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 testowe329	
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 rozwiązania335	
ROZDZIAŁ 7. Odpowiedzi do pytań testowych337	
Rozdział 1.	337
Rozdział 2.	342
Rozdział 3.	345
Rozdział 4.	348
Rozdział 5.	358
Rozdział 6.	363
ROZDZIAŁ 8. Odpowiedzi do ćwiczeń365	
Rozdział 4.	365
Rozdział 5.	376

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 syllabus 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 syllabusów i egzaminów ISTQB®, ale duża ich część jest niestety złą jakością. Zdarza się nawet, że materiały znalezione w sieci zawierają poważne błędy. Ponadto, ze względu na istotne zmiany, jakie zaszły w syllabusie w stosunku do poprzedniej wersji (3.1.1), ilość dostępnych dla kandydatów materiałów opartych na nowym syllabusie jest wciąż niewielka.

Podręcznik rozszerza i uszczegółnia wiele kwestii, które w samym syllabusie opisane są zdawkowo lub ogólnie. Zgodnie z wytycznymi ISTQB® dla szkoleń opartych na syllabusie 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, syllabus i egzamin poziomu podstawowego

Część I podręcznika zawiera oficjalne informacje dotyczące treści i układu syllabusa 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 syllabusa 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.

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, pozwalającym 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 uzasadnieniach tych odpowiedzi.

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.

O autorach

Wszyscy trzej autorzy są ekspertami ISTQB, zaangażowanymi w prace przy tworzeniu sylabusów i zadań egzaminacyjnych. W szczególności są współautorami sylabusa poziomu podstawowego w wersji 4.0.

dr Lucjan Stapp — emerytowany pracownik naukowo-dydaktyczny Politechniki Warszawskiej, gdzie przez wiele lat prowadził wykłady oraz seminaria z testowania oprogramowania i zapewniania jakości. Autor ponad 40 publikacji, w tym 12 o różnych problemach związanych z testowaniem i zapewnianiem jakości. Jako tester przeszedł całą ścieżkę kariery, od junior testera do kierownika zespołu testerskiego w kilkunastu projektach. Współorganizator i prelegent wielu konferencji testerskich (m.in. TestWarez). Członek założyciel, a obecnie wiceprezes Stowarzyszenia Jakości Systemów Informatycznych. Certyfikowany tester (m.in. ISTQB CTAL-TM, ISTQB CTAL-TA, Agile Tester, Tester Akceptacyjny).

dr hab. Adam Roman, prof. UJ — pracownik naukowo-dydaktyczny Instytutu Informatyki i Matematyki Komputerowej na Uniwersytecie Jagiellońskim, gdzie od wielu lat prowadzi wykłady oraz seminaria z testowania oprogramowania i zapewniania jakości. Współtwórca studiów podyplomowych „Testowanie oprogramowania” na Uniwersytecie Jagiellońskim. Jego zainteresowania naukowe obejmują m.in. badania nad metrykami oprogramowania, modelami predykcji defektów oraz efektywnymi technikami projektowania testów. W ramach Polskiego Komitetu Normalizacyjnego współpracował nad międzynarodową normą ISO/IEEE 29119 Software Testing Standard. Autor monografii: *Testowanie i jakość oprogramowania. Modele, techniki, narzędzia, Thinking-Driven Testing, A Study Guide to the ISTQB Foundation Level 2018 Syllabus. Test Techniques and Sample Mock Exams* oraz wielu publikacji naukowych i popularyzatorskich w obszarze testowania. Prelegent na wielu krajowych i międzynarodowych konferencjach testerskich (m.in. EuroSTAR, TestWell, TestingCup, TestWarez). Członek Stowarzyszenia Jakości Systemów Informatycznych. Posiada m.in. certyfikaty: ASQ Certified Software Quality Engineer, ISTQB Full Advanced Level, ISTQB Expert Level — Improving the Test Process.

Michaël Pilaeten — łamanie systemu, pomoc w jego odbudowie oraz porady i wskaźówki, jak uniknąć problemów. To właśnie Michaël w skrócie. Posiadając prawie dwudziestoletnie doświadczenie w konsultacjach testowych w różnych środowiskach, widział to, co najlepsze (i najgorsze) w rozwoju oprogramowania. W swojej obecnej roli jako Learning & Development Manager jest odpowiedzialny za prowadzenie konsultantów, partnerów i klientów na ich osobistej i zawodowej ścieżce w kierunku jakości i doskonałości. Jest przewodniczącym grupy roboczej ISTQB Agile oraz Product Ownerem sylabusa ISTQB CTFL 4.0. Członek BNTQB (Belgium & The Netherlands Testing Qualification Board), akredytowany szkoleniowiec dla większości szkoleń ISTQB i IREB oraz międzynarodowy mówca i moderator warsztatów.

CZĘŚĆ I

**CERTYFIKAT, SYLABUS
I EGZAMIN POZIOMU
PODSTAWOWEGO**

Certyfikat poziomu podstawowego

Okoliczności powstania i historia certyfikatu podstawowego

Niezależna certyfikacja testerów oprogramowania rozpoczęła się w 1998 roku w Wielkiej Brytanii. Pod auspicjami działającej w ramach British Computer Society rady ds. egzaminów informatycznych (Information Systems Examination Board — ISEB) została wówczas powołana specjalna jednostka ds. testowania oprogramowania — Software Testing Board (www.bcs.org.uk/iseb). W 2002 roku własny system kwalifikacji testerów stworzyła również niemiecka organizacja ASQF (www.asqf.de). Sylabus poziomu podstawowego powstał na bazie sylabusów ISEB i ASQF, przy czym zawarte w nim informacje zostały przeorganizowane, zaktualizowane i uzupełnione. Główny nacisk położono na zagadnienia zapewniające testerom największe praktyczne wsparcie.

Dotychczasowe certyfikaty podstawowe w dziedzinie testowania oprogramowania (np. certyfikaty wydawane przez ISEB, ASQF lub rady krajowe ISTQB®), które zostały przyznane przed powaniem certyfikatu międzynarodowego, są uznawane za równoważne temu certyfikatowi. Certyfikat podstawowy nie wygasa i nie wymaga odnowienia. Data przyznania jest umieszczona na certyfikacie.

Uwarunkowania lokalne w poszczególnych krajach uczestniczących w programie leżą w gestii rad krajowych ISTQB®. Obowiązki rad krajowych określa ISTQB®, natomiast realizację tych obowiązków pozostawiono poszczególnym organizacjom członkowskim. Do obowiązków rad krajowych należą zwykle akredytacja dostawców szkoleń i wyznaczanie terminów egzaminów.

Ścieżki kariery dla testerów

System stworzony przez ISTQB® umożliwia definiowanie ścieżek kariery dla osób zawodowo zajmujących się testowaniem na podstawie trójpoziomowego programu certyfikacji, który obejmuje poziom podstawowy, poziom zaawansowany i poziom ekspercki. Punktem wejścia jest sylabus poziomu podstawowego opisany w niniejszej książce. Posiadanie certyfikatu poziomu podstawowego jest warunkiem koniecznym zdobywania kolejnych certyfikatów.

Posiadacz certyfikatu na poziomie podstawowym może poszerzyć posiadaną wiedzę w dziedzinie testowania poprzez uzyskanie kwalifikacji na poziomie zaawansowanym.

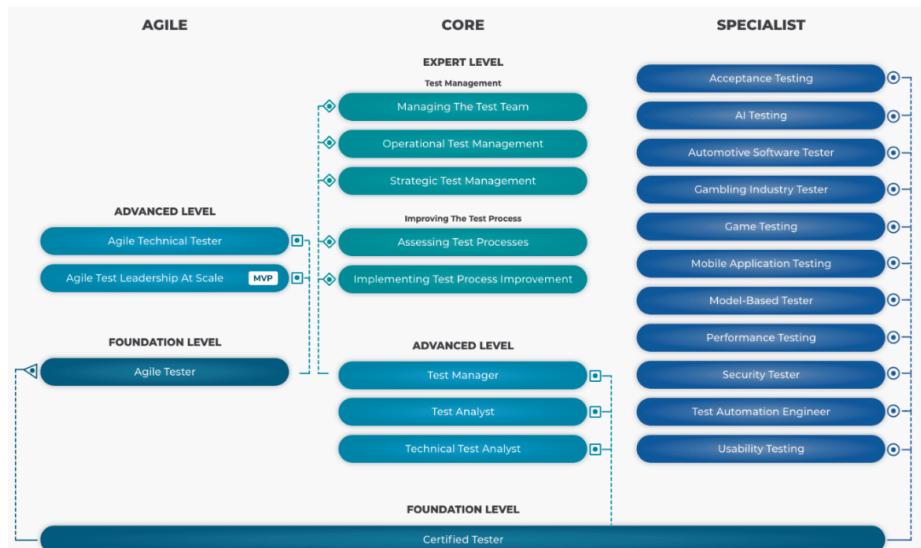
Na tym poziomie ISTQB® oferuje szereg programów kształcenia. W zakresie ścieżki głównej są to trzy programy:

- **techniczny analityk testów** (zorientowanie na technologię, testowanie nie-funkcjonalne, analizę statyczną, białośkrzynkowe techniki testowania, pracę z kodem źródłowym);
- **analityk testów** (zorientowanie na pracę z klientem, zrozumienie biznesu, testy funkcyjne, techniki testowania czarnoskrzynkowe i oparte na doświadczeniu);
- **kierownik testów** (zorientowanie na zarządzanie procesem testowym i zespołem testerskim).

Poziom zaawansowany stanowi punkt wyjścia do zdobywania dalszej wiedzy i umiejętności na poziomie eksperckim. Osoba, która zdobyła już doświadczenie na stanowisku kierownika testów, może na przykład zdecydować się na dalsze rozwijanie kariery testera poprzez uzyskanie certyfikatów na poziomie eksperckim w dziedzinach **zarządzania testami** oraz **doskonalenia procesu testowego**.

Ponadto, poza ścieżką główną, ISTQB® oferuje także specjalistyczne programy kształcenia dotyczące takich zagadnień jak: testowanie akceptacyjne, testowanie sztucznej inteligencji, testowanie w branży automotive, testowanie maszyn do gier hazardowych, testowanie gier, testowanie aplikacji mobilnych, testowanie oparte na modelu, testowanie wydajności, testowanie zabezpieczeń, automatyzacja testów czy testowanie użyteczności. W zakresie metodik zwinnych jest to techniczny tester zwinnego lub Agile Test Leadership at Scale.

Rysunek 0.1 przedstawia schemat certyfikacyjny oferowany przez ISTQB® na dzień 3 lipca 2023 roku. Najnowsza wersja przeglądu ścieżek kariery ISTQB® jest dostępna w witrynie www.istqb.org.



RYSUNEK 0.1. Oficjalny schemat certyfikacji ISTQB® (źródło: www.istqb.org)

Docelowi odbiorcy

Kwalifikacja na poziomie podstawowym jest przeznaczona dla wszystkich osób zainteresowanych testowaniem oprogramowania. Mogą to być między innymi: testerzy, analitycy testów, inżynierowie testów, konsultanci ds. testów, kierownicy testów, użytkownicy wykonujący testowanie akceptacyjne, członkowie zespołów zwinnych oraz programiści. Ponadto kwalifikacja na poziomie podstawowym jest odpowiednia dla osób chcących zdobyć podstawową wiedzę w dziedzinie testowania oprogramowania, takich jak: kierownicy projektu, kierownicy ds. jakości, kierownicy ds. tworzenia oprogramowania, analitycy biznesowi, dyrektorzy ds. informatyki oraz konsultanci w dziedzinie zarządzania.

Cele certyfikatu podstawowego

- Podkreślenie znaczenia testowania jako jednej z podstawowych specjalizacji zawodowych w ramach inżynierii oprogramowania.
- Stworzenie standardowych ram rozwoju zawodowego testerów.
- Stworzenie systemu umożliwiającego uznawanie kwalifikacji zawodowych testerów przez pracodawców, klientów i innych testerów oraz podnoszenie statusu testerów.
- Promowanie spójnych, dobrych praktyk testowania we wszystkich dyscyplinach inżynierii oprogramowania.
- Wskazanie zagadnień związanych z testowaniem, które są istotne i wartościowe dla całej branży IT.
- Stworzenie dostawcom oprogramowania możliwości zatrudniania certyfikowanych testerów i uzyskiwania przewagi handlowej nad konkurencją poprzez reklamowanie przyjętej polityki rekrutacji testerów.
- Stworzenie testerom i osobom zainteresowanym testowaniem okazji do zdobycia uznawanych na całym świecie kwalifikacji w tej dziedzinie.

Cele międzynarodowego systemu uzyskiwania kwalifikacji

- Stworzenie podstaw do porównywania wiedzy w dziedzinie testowania w różnych krajach.
- Ułatwienie testerom znalezienia pracy w innych krajach.
- Zapewnienie wspólnego rozumienia zagadnień związanych z testowaniem w ramach projektów międzynarodowych.
- Zwiększenie liczby wykwalifikowanych testerów na całym świecie.

- Stworzenie inicjatywy o charakterze międzynarodowym, która zapewni większe korzyści i silniejsze oddziaływanie niż inicjatywy realizowane na szczeblu krajowym.
- Opracowanie wspólnego, międzynarodowego zbioru informacji i wiedzy na temat testowania na bazie sylabusów i słownika pojęć związanych z testowaniem oraz podnoszenie poziomu wiedzy na temat testowania u pracowników branży IT.
- Promowanie zawodu testera w większej liczbie krajów.
- Umożliwienie testerom uzyskania powszechnie uznawanych kwalifikacji w języku ojczystym.
- Stworzenie testerom z różnych krajów warunków do dzielenia się wiedzą i zasobami.
- Zapewnienie międzynarodowego uznawania statusu testerów i niniejszej kwalifikacji.

Cele biznesowe

Z każdym sylabusem ISTQB związany jest zbiór tzw. celów biznesowych (ang. *business outcomes*). Cel biznesowy to zwięzły, zdefiniowany i możliwy do zaobserwowania rezultat lub zmiana w działaniu przedsiębiorstwa, poparty konkretną miarą. W tabeli 0.1 wymieniono 14 celów biznesowych, do których realizacji powinien przyczyniać się kandydat otrzymujący certyfikat na poziomie podstawowym.

TABELA 0.1. Cele realizowane przez certyfikowanego testera na poziomie podstawowym

FL-BO1	Znajomość istoty testowania i wynikających z niego korzyści
FL-BO2	Znajomość podstawowych pojęć związanych z testowaniem oprogramowania
FL-BO3	Identyfikowanie podejścia do testowania i czynności testowych, które mają być realizowane w zależności od kontekstu testowania
FL-BO4	Dokonywanie oceny i podnoszenie jakości dokumentacji
FL-BO5	Zwiększanie skuteczności i efektywności testowania
FL-BO6	Dopasowywanie procesu testowego do cyklu wytwarzania oprogramowania
FL-BO7	Znajomość zasad zarządzania testami
FL-BO8	Sporządzanie i udostępnianie przejrzystych, zrozumiałych raportów o defektach
FL-BO9	Znajomość czynników wpływających na priorytety i pracochłonność testowania
FL-BO10	Praca w zespole interdyscyplinarnym
FL-BO11	Znajomość ryzyk i korzyści związanych z automatyzacją testów
FL-BO12	Identyfikowanie niezbędnych umiejętności wymaganych w związku z testowaniem
FL-BO13	Znajomość wpływu ryzyka na testowanie
FL-BO14	Sprawne raportowanie na temat postępu i jakości testów

Cele nauczania

Treść każdego sylabusa tworzona jest tak, aby pokryć zbiór ustalonych dla tego sylabusa celów nauczania (ang. *learning objectives*). Cele nauczania wspomagają osiąganie celów biznesowych i służą do tworzenia egzaminów certyfikacyjnych dla certyfikowanych testerów na poziomie podstawowym. Rozumienie tego, czym są cele nauczania, oraz znajomość tych celów dla poszczególnych sekcji sylabusa są kluczowe w efektywnym przygotowaniu się do egzaminu certyfikacyjnego.

Wszystkie cele nauczania są w sylabusie zdefiniowane w taki sposób, aby każdy z nich stanowił niepodzielną całość. Cele nauczania zdefiniowane są na początku każdego rozdziału sylabusa. Każda sekcja sylabusa dotyczy dokładnie jednego celu nauczania. Dzięki temu można jednoznacznie powiązać każdy cel nauczania (i pytania egzaminacyjne) ze ścisłe określonymi partiami materiału.

W kolejnym rozdziale przedstawiono cele nauczania obowiązujące w przypadku sylabusa poziomu podstawowego. Znajomość każdego z zagadnień poruszonych w sylabusie będzie sprawdzana na egzaminie zgodnie z przypisanym celem nauczania. Każdemu celowi nauczania przypisany jest tzw. poziom wiedzy, zwany również poziomem kognitywnym (lub poziomem K): K1, K2 lub K3, który określa stopień, w jakim należy przyswoić daną partię materiału. Poziomy wiedzy dla poszczególnych celów nauczania przedstawione są przy każdym celu nauczania wymienionym na początku każdego rozdziału sylabusa.

Poziom K1 — zapamiętać

Kandydat rozpoznaje, pamięta lub umie sobie przypomnieć dany termin lub dane pojęcie.

Słowa kluczowe: pamiętać, rozpoznać, określić, wskazać.

Przykłady:

- „Kandydat wskazuje typowe cele testów”.
- „Kandydat pamięta pojęcia związane z piramidą testów”.
- „Kandydat rozpoznaje, jaki jest wkład testera w planowanie iteracji i wydań”.

Poziom K2 — zrozumieć

Kandydat potrafi uzasadnić lub wyjaśnić stwierdzenia dotyczące danego zagadnienia, a także podsumować, porównać i sklasyfikować pojęcia z zakresu testowania oraz podać odpowiednie przykłady.

Słowa kluczowe: sklasyfikować, porównać, zestawić ze sobą, objaśnić, wyjaśnić, omówić, rozróżnić, odróżnić, wyjaśnić, podać przykłady, podsumować.

Przykłady:

- „Kandydat klasyfikuje różne sposoby pisania kryteriów akceptacji”.
- „Kandydat porównuje poszczególne role występujące w testowaniu”.

- „Kandydat rozróżnia ryzyka projektowe i produktowe”.
- „Kandydat omawia na przykładach cel i treść planu testów”.
- „Kandydat wyjaśnia wpływ wybranego modelu cyklu wytwarzania oprogramowania na testowanie”.
- „Kandydat podsumowuje czynności wykonywane w ramach procesu przeglądu”.

Poziom K3 — zastosować

Kandydat potrafi wybrać prawidłowe zastosowanie pojęcia lub techniki i zastosować je w danym kontekście.

Słowa kluczowe: zastosować, obliczyć, sporządzić, użyć.

Przykłady:

- „Kandydat stosuje priorytetyzację przypadków testowych”.
- „Kandydat sporządza raport o defekcie”.
- „Kandydat używa techniki analizy wartości brzegowych, aby zaprojektować przypadki testowe”.

Co do zasady przedmiotem egzaminu — poza samymi celami nauczania — może być cała treść syllabusa na poziomie K1, z wyjątkiem wstępu i załączników. Oznacza to, że od kandydata może być wymagane rozpoznanie, zapamiętanie lub przypomnienie sobie *słowa kluczowego* wymienionego w każdym z sześciu rozdziałów dokumentu.

Wymagania stawiane kandydatom

Od osób przystępujących do egzaminu ISTQB® na certyfikowanego testera na poziomie podstawowym wymaga się jedynie zainteresowania tematyką testowania oprogramowania. Jednakże zdecydowanie zaleca się, aby kandydaci:

- posiadali co najmniej podstawowe doświadczenie w dziedzinie wytwarzania lub testowania oprogramowania, na przykład sześciomiesięczne doświadczenie na stanowisku testera wykonującego testy systemowe lub akceptacyjne bądź na stanowisku programisty;
- ukończyli kurs ISTQB® (akredytowany przez jedną z rad krajowych ISTQB® zgodnie ze standardami ISTQB®).

Nie są to jednak wymogi formalne, choć ich spełnienie znaczco ułatwia przygotowanie się i zdanie egzaminu certyfikacyjnego. Do egzaminu może przystąpić każdy, niezależnie od zainteresowań i doświadczenia w pracy jako tester.

Sylabus i egzamin poziomu podstawowego

Odniesienia do norm i standardów

Sylabus zawiera odniesienia do norm i standardów IEEE, ISO itd. Celem tych odniesień jest stworzenie ram pojęciowych lub odesłanie Czytelnika do źródła, z którego może skorzystać w celu uzyskania dodatkowych informacji. Należy jednak zaznaczyć, że przedmiotem egzaminu mogą być wyłącznie te zapisy przywoływanych norm lub standardów, do których odwołują się konkretne omawiane fragmenty sylabusa. Treść norm i standardów nie jest przedmiotem egzaminu, a odwołania do tych dokumentów mają wyłącznie charakter informacyjny.

Obecna wersja sylabusa (4.0) z 2023 roku odwołuje się do następujących standardów:

ISO/IEC/IEEE 29119 — Software Testing Standard. Standard ten składa się z kilku części, z których najważniejszymi z punktu widzenia sylabusa są: część 1. (ogólne pojęcia), część 2. (procesy testowe), część 3. (dokumentacja testowa) oraz część 4. (techniki testowania). Część 3. tej normy zastąpiła wycofany standard IEEE 829.

ISO/IEC 25010 — System and Software Quality Requirements and Evaluation (tzw. SQuaRE) — *System and software quality models*. Norma ta opisuje model jakości oprogramowania i zastąpiła wycofany standard ISO 9126.

ISO/IEC 20246 — Work Product Reviews. Norma ta opisuje kwestie związane z przeglądami produktów prac. Zastąpiła wycofany standard IEEE 1028.

ISO 31000 — Risk Management — Principles and guidelines. Standard ten opisuje proces zarządzania ryzykiem.

Ciągła aktualizacja

W branży IT zachodzą dynamiczne zmiany. Aby uwzględnić zmieniającą się sytuację i zapewnić interesariuszom dostęp do przydatnych, aktualnych informacji, grupy robocze ISTQB® stworzyły listę odsyłaczy do dokumentów pomocniczych i zmian w normach/standardach, która jest dostępna w witrynie www.istqb.org. Powyższe informacje nie są przedmiotem egzaminu dotyczącego sylabusa poziomu podstawowego.

Nota wydania dla sylabusa w wersji 4.0

Sylabus poziomu podstawowego w wersji 4.0 zawiera najlepsze praktyki i techniki, które wytrzymały próbę czasu, ale w stosunku do poprzedniej wersji (3.1) z 2018 roku dokonano w nim szeregu istotnych zmian mających na celu bardziej nowoczesne przedstawienie materiału, lepsze jego dostosowanie do poziomu podstawowego, a także uwzględnienie zmian, jakie dokonały się w inżynierii oprogramowania w ostatnich latach. W szczególności:

- położono większy nacisk na metody i praktyki stosowane w zwinnych modelach wytwarzania oprogramowania (podejście „cały zespół”, podejście „przesunięcie w lewo” (ang. *shift-left*), planowanie iteracji i planowanie wydania, piramida testów, kwadranty testowe, praktyki „najpierw test”: TDD, BDD, ATDD);
- rozszerzono i pogłębiono sekcję dotyczącą umiejętności testerskich, w szczególności umiejętności miękkich;
- przeorganizowano i ustrukturyzowano sekcję dotyczącą zarządzania ryzykiem;
- dodano sekcję omawiającą podejście DevOps;
- dodano sekcję omawiającą szczegółowo niektóre techniki szacowania testów;
- technikę testowania decyzji zastąpiono techniką testowania gałęzi (ang. *branch testing*);
- usunięto sekcję opisującą szczegółowe techniki przeglądu;
- usunięto omówienie techniki testowania opartego na przypadkach użycia (jest ona omawiana w sylabusie poziomu zaawansowanego „Analyst testów”);
- usunięto sekcję omawiającą przykładowe strategie testów;
- usunięto niektóre zagadnienia związane z narzędziami, m.in. zagadnienie wprowadzania narzędzia do organizacji czy przeprowadzania projektu pilotażowego.

Zawartość sylabusa

Rozdział 1. Podstawy testowania

- Kandydat poznaje podstawowe zasady związane z testowaniem, powody, dla których testowanie jest niezbędne, oraz cele testów.
- Kandydat poznaje proces testowy oraz najważniejsze czynności testowe i testalia.
- Kandydat dowiaduje się, jakie umiejętności są niezbędne w testowaniu.

Cele nauczania

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.

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 zapewnieniem jakości.

FL-1.2.3 (K2) Kandydat odróżnia podstawową przyczynę, pomyłkę, defekt i awarię.

1.3. Zasady testowania

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

1.4. Czynności testowe, testalia i role związane z testami

FL-1.4.1 (K2) Kandydat podsumowuje poszczególne czynności i zadania testowe.

FL-1.4.2 (K2) Kandydat wyjaśnia wpływ kontekstu na proces testowy.

FL-1.4.3 (K2) Kandydat rozróżnia testalia wspomagające czynności testowe.

FL-1.4.4 (K2) Kandydat wyjaśnia korzyści wynikające ze śledzenia powiązań.

FL-1.4.5 (K2) Kandydat porównuje poszczególne role występujące w testowaniu.

1.5. Niezbędne umiejętności i dobre praktyki w dziedzinie testowania

FL-1.5.1 (K2) Kandydat podaje przykłady ogólnych umiejętności wymaganych w testowaniu.

FL-1.5.2 (K1) Kandydat pamięta, jakie są zalety podejścia „cały zespół”.

FL-1.5.3 (K2) Kandydat omawia korzyści i wady niezależności testowania.

Rozdział 2. Testowanie w cyklu wytwarzania oprogramowania

- Kandydat dowiaduje się, w jaki sposób testowanie jest uwzględniane w różnych podejściach do wytwarzania oprogramowania.
- Kandydat poznaje pojęcia związane z podejściem „najpierw test” i metodyką DevOps.
- Kandydat uzyskuje wiedzę na temat poszczególnych poziomów testów i typów testów oraz testowania pielęgnacyjnego.

Cele nauczania

2.1. Testowanie w kontekście cyklu wytwarzania oprogramowania

FL-2.1.1 (K2) Kandydat wyjaśnia wpływ wybranego modelu cyklu wytwarzania oprogramowania na testowanie.

FL-2.1.2 (K1) Kandydat pamięta dobre praktyki testowania mające zastosowanie do wszystkich modeli cyklu wytwarzania oprogramowania.

FL-2.1.3 (K1) Kandydat podaje przykłady podejścia typu „najpierw test” w kontekście wytwarzania oprogramowania.

- FL-2.1.4 (K2) Kandydat podsumowuje, w jaki sposób metodyka DevOps może wpływać na testowanie.
- FL-2.1.5 (K2) Kandydat wyjaśnia, na czym polega przesunięcie w lewo.
- FL-2.1.6 (K2) Kandydat wyjaśnia, w jaki sposób retrospektywy mogą posłużyć jako mechanizmy doskonalenia procesów.

2.2. Poziomy testów i typy testów

- FL-2.2.1 (K2) Kandydat rozróżnia poszczególne poziomy testów.
- FL-2.2.2 (K2) Kandydat rozróżnia poszczególne typy testów.
- FL-2.2.3 (K2) Kandydat odróżnia testowanie potwierdzające od testowania regresji.

2.3. Testowanie pielęgnacyjne

- FL-2.3.1 (K2) Kandydat podsumowuje testowanie pielęgnacyjne i zdarzenia je wyzwalające.

Rozdział 3. Testowanie statyczne

- Kandydat poznaje podstawy testowania statycznego oraz proces uzyskiwania informacji zwrotnych i przeprowadzania przeglądu.

Cele nauczania

3.1. Podstawy testowania statycznego

- FL-3.1.1 (K1) Kandydat rozpoznaje typy produktów, które mogą być badane przy użyciu poszczególnych technik testowania statycznego.
- FL-3.1.2 (K2) Kandydat wyjaśnia korzyści wynikające z testowania statycznego.
- FL-3.1.3 (K2) Kandydat porównuje i zestawia ze sobą testowanie statyczne i dynamiczne.

3.2. Informacje zwrotne i proces przeglądu

- FL-3.2.1 (K1) Kandydat pamięta korzyści wynikające z wczesnego i częstego otrzymywania informacji zwrotnych od interesariuszy.
- FL-3.2.2 (K2) Kandydat podsumowuje czynności wykonywane w ramach procesu przeglądu.
- FL-3.2.3 (K1) Kandydat pamięta, jakie obowiązki są przypisane do najważniejszych ról w trakcie wykonywania przeglądów.
- FL-3.2.4 (K2) Kandydat porównuje i zestawia ze sobą różne typy przeglądów.
- FL-3.2.5 (K1) Kandydat pamięta, jakie czynniki decydują o powodzeniu przeglądu.

Rozdział 4. Analiza i projektowanie testów

- Kandydat dowiaduje się, jak należy stosować techniki czarnoskrzynkowe, białośkrzynkowe i oparte na doświadczeniu, aby tworzyć przypadki testowe na podstawie różnych produktów pracy związań z oprogramowaniem.
- Kandydat poznaje podejście do testowania oparte na współpracy.

Cele nauczania

4.1 Ogólna charakterystyka technik testowania

FL-4.1.1 (K2) Kandydat rozróżnia czarnoskrzynkowe i białośkrzynkowe techniki testowania oraz techniki testowania oparte na doświadczeniu.

4.2. Czarnoskrzynkowe techniki testowania

FL-4.2.1 (K3) Kandydat używa techniki podziału na klasy równoważności, aby zaprojektować przypadki testowe.

FL-4.2.2 (K3) Kandydat używa techniki analizy wartości brzegowych, aby zaprojektować przypadki testowe.

FL-4.2.3 (K3) Kandydat używa techniki testowania w oparciu o tablicę decyzyjną, aby zaprojektować przypadki testowe.

FL-4.2.4 (K3) Kandydat używa techniki testowania przejść pomiędzy stanami, aby zaprojektować przypadki testowe.

4.3. Białośkrzynkowe techniki testowania

FL-4.3.1 (K2) Kandydat wyjaśnia pojęcie testowanie instrukcji.

FL-4.3.2 (K2) Kandydat wyjaśnia pojęcie testowanie gałęzi.

FL-4.3.3 (K2) Kandydat wyjaśnia korzyści wynikające z testowania białośkrzynkowego.

4.4. Techniki testowania oparte na doświadczeniu

FL-4.4.1 (K2) Kandydat wyjaśnia pojęcie zgadywanie błędów.

FL-4.4.2 (K2) Kandydat wyjaśnia pojęcie testowanie eksploracyjne.

FL-4.4.3 (K2) Kandydat wyjaśnia pojęcie testowanie w oparciu o listę kontrolną.

4.5. Podejścia do testowania oparte na współpracy

FL-4.5.1 (K2) Kandydat wyjaśnia, w jaki sposób należy pisać historyjki użytkownika we współpracy z programistami i przedstawicielami jednostek biznesowych.

FL-4.5.2 (K2) Kandydat klasyfikuje różne sposoby pisania kryteriów akceptacji.

FL-4.5.3 (K3) Kandydat używa metody wytwarzania sterowanego testami akceptacyjnymi (ATDD), aby zaprojektować przypadki testowe.

Rozdział 5. Zarządzanie czynnościami testowymi

- Kandydat poznaje ogólne zasady planowania testów i szacowania ich pracochności.
- Kandydat dowiaduje się, jak ryzyka mogą wpływać na zakres testowania.
- Kandydat dowiaduje się, jak należy monitorować i nadzorować czynności testowe.
- Kandydat dowiaduje się, w jaki sposób zarządzanie konfiguracją wspomaga testowanie.
- Kandydat uczy się zgłaszania defektów w przejrzysty i zrozumiały sposób.

Cele nauczania

5.1. Planowanie testów

- FL-5.1.1 (K2) Kandydat omawia na przykładach cel i treść planu testów.
- FL-5.1.2 (K1) Kandydat rozpoznaje, jaki jest wkład testera w planowanie iteracji i wydań.
- FL-5.1.3 (K2) Kandydat porównuje i zestawia ze sobą kryteria wejścia i kryteria wyjścia.
- FL-5.1.4 (K3) Kandydat oblicza pracochność testowania przy użyciu technik szacowania.
- FL-5.1.5 (K3) Kandydat stosuje priorytetyzację przypadków testowych.
- FL-5.1.6 (K1) Kandydat pamięta pojęcia związane z piramidą testów.
- FL-5.1.7 (K2) Kandydat podsumowuje kwadranty testowe oraz ich relację do poziomów testów i typów testów.

5.2. Zarządzanie ryzykiem

- FL-5.2.1 (K1) Kandydat określa poziom ryzyka na podstawie prawdopodobieństwa ryzyka i wpływu ryzyka.
- FL-5.2.2 (K2) Kandydat rozróżnia ryzyka projektowe i produktowe.
- FL-5.2.3 (K2) Kandydat wyjaśnia potencjalny wpływ analizy ryzyka produktowego na staranność i zakres testowania.
- FL-5.2.4 (K2) Kandydat wyjaśnia, jakie środki można podjąć w odpowiedzi na przeanalizowane ryzyka produktowe.

5.3. Monitorowanie testów, nadzór nad testami i ukończenie testów

- FL-5.3.1 (K1) Kandydat pamięta metryki stosowane w odniesieniu do testowania.
- FL-5.3.2 (K2) Kandydat podsumowuje cele i treść raportów z testów oraz wskazuje ich odbiorców.
- FL-5.3.3 (K2) Kandydat omawia na przykładach sposób przekazywania informacji o statusie testowania.

5.4. Zarządzanie konfiguracją

FL-5.4.1 (K2) Kandydat podsumowuje, w jaki sposób zarządzanie konfiguracją wspomaga testowanie.

5.5. Zarządzanie defektami

FL-5.5.1 (K3) Kandydat sporządza raport o defekcie.

Rozdział 6. Narzędzia testowe

- Kandydat uczy się klasyfikować narzędzia oraz poznaje ryzyka i korzyści wynikające z automatyzacji testów.

Cele nauczania

6.1. Narzędzia wspomagające testowanie

FL-6.1.1 (K2) Kandydat wyjaśnia, w jaki sposób różnego typu narzędzia testowe wspomagają testowanie.

6.2. Korzyści i ryzyka związane z automatyzacją testowania

FL-6.2.1 (K1) Kandydat pamięta korzyści i ryzyka związane z automatyzacją testowania.

Struktura egzaminu

Opis egzaminu certyfikacyjnego na poziomie podstawowym zdefiniowano w dokumencie zatytułowanym *Struktura i zasady tworzenia egzaminów CTFL* (ang. *Exam Structure Rules*), który jest dostępny w witrynie www.sjsi.org (w wersji ang. www.istqb.org).

Egzamin ma formę testu wielokrotnego wyboru i składa się z **40 pytań**. Do zdania egzaminu niezbędne jest udzielenie poprawnej odpowiedzi na co najmniej **65%** pytań (tj. **26 pytań**).

Egzaminy można zdawać w ramach akredytowanego szkolenia lub samodzielnie (na przykład w ośrodku egzaminacyjnym lub w ramach egzaminu publicznego). Ukończenie akredytowanego kursu nie jest warunkiem przystąpienia do egzaminu, ale uczestnictwo w takim kursie jest zalecane, gdyż pozwala lepiej zrozumieć materiał oraz znacząco zwiększa szansę zdania egzaminu. W przypadku niezdania egzaminu można podchodzić do niego ponownie dowolną liczbę razy.

Reguły egzaminu

- Egzaminy na Certyfikowanego Testera — Poziom Podstawowy są oparte na sylabusie Certyfikowany Tester — Poziom Podstawowy [ISTQB FL 2023].
- Odpowiedź na pytanie egzaminacyjne może wymagać wykorzystania materiału z więcej niż jednej sekcji sylabusa.
- Wszystkie cele nauczania zawarte w sylabusie (o poziomach kognitywnych od K1 do K3) podlegają egzaminowaniu.
- Wszystkie definicje słów kluczowych w sylabusie podlegają egzaminowaniu (na poziomie K1). Słownik w języku polskim dostępny jest na stronie www.sjsi.org. Słownik online (również w wersji polskojęzycznej) dostępny jest także na stronie www.glossary.istqb.org.
- Każdy egzamin na Certyfikowanego Testera — Poziom Podstawowy składa się z zestawu pytań wielokrotnego wyboru na podstawie celów nauczania dla tego konkretnego sylabusa. Poziom pokrycia i rozkład pytań został oparty na celach nauczania, ich poziomach K i ich poziomie ważności według oceny ISTQB®. Szczegółowe informacje na temat struktury każdego modułu egzaminacyjnego znajdują się poniżej, w podrozdziale „Rozkład pytań na egzaminie”.
- Ogólnie rzecz biorąc, oczekuje się, że czas na przeczytanie, analizę i odpowiedź na pytania na poziomach K1 i K2 nie powinien przekroczyć 1 minuty, a na poziomie K3 może zająć 3 minuty. Autor egzaminu powinien jednak pamiętać, że jest to tylko wskazówka odnośnie do przeciętnego czasu i że prawdopodobnie niektóre pytania będą wymagały więcej, a inne mniej czasu na udzielenie odpowiedzi.
- Egzamin składa się z 40 pytań wielokrotnego wyboru. Każda poprawna odpowiedź jest warta jeden punkt (niezależnie od poziomu K celu nauczania, którego pytanie dotyczy). Maksymalna możliwa ocena za egzamin to 40 punktów.
- Czas przeznaczony na egzamin wynosi dokładnie 60 minut. Jeśli język ojczysty kandydata nie jest językiem egzaminacyjnym, kandydatowi przysługuje dodatkowe 25% czasu (w przypadku egzaminu na poziomie podstawowym oznacza to, że egzamin będzie trwał 75 minut).
- Do zaliczenia wymagane jest uzyskanie co najmniej 65% (26 punktów).
- Ogólny podział pytań według poziomu K jest przedstawiony w tabeli 0.2:

TABELA 0.2. Rozkład pytań egzaminacyjnych według poziomów K

POZIOM K	LICZBA PYTAŃ	CZAS NA PYTANIE [W MINUTACH]	ŚREDNI CZAS DLA POZIOMU K [W MINUTACH]
K1	8	1	8
K2	24	1	24
K3	8	3	24
Razem	40		56

Jeśli zsumować oczekiwany czas odpowiedzi na pytania według reguł podanych powyżej, to — uwzględniając rozkład pytań według poziomów K — okaże się, że odpowiedź na wszystkie pytania powinna zająć około 56 minut. Zostają więc 4 minuty zapasu.

Każde pytanie egzaminacyjne powinno sprawdzać co najmniej jeden cel nauczania (LO) z sylabusa Certyfikowany Tester — Poziom Podstawowy. Pytania mogą zawierać słowa kluczowe bez podania ich definicji, ponieważ oczekuje się ich znajomości przez kandydatów. W przypadku gdy pytania dotyczą więcej niż jednego LO, powinny odwoływać się (i być przyporządkowane) do celu nauczania o najwyższej wartości poziomu K.

Rozkład pytań na egzaminie

Struktura egzaminu Certyfikowany Tester — Poziom Podstawowy przedstawiona jest w tabeli 0.3. Każdy z egzaminów wymaga obowiązkowych pytań sprawdzających określone cele nauczania, a także określonej liczby pytań na podstawie wybranych celów nauczania. W przypadku, gdy liczba celów nauczania jest większa niż liczba pytań dla określonej grupy celów nauczania opisanej w tabeli, każde pytanie musi pokryć inny cel nauczania.

TABELA 0.3. Szczegółowy rozkład pytań egzaminacyjnych w podziale na poziomy K i rozdziały

ROZKŁAD PYTAŃ W ROZDZIALE 1.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA	
FL-1.1.1, FL-1.2.2	K1	1	1	W sumie 8 pytań wymaganych dla rozdziału 1.
FL-1.5.2	K1	1	1	
FL-1.1.2, FL-1.2.1, FL-1.2.3	K2	1	1	K1 = 2 K2 = 6
FL-1.3.1	K2	1	1	K3 = 0
FL-1.4.1, FL-1.4.2, FL-1.4.3, FL-1.4.4, FL-1.4.5	K2	3		Liczba punktów dla tego rozdziału = 8
FL-1.5.1, FL-1.5.3	K2	1	1	
ROZKŁAD PYTAŃ W ROZDZIALE 2.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA	
FL-2.1.2	K1	1	1	W sumie 6 pytań wymaganych dla rozdziału 2.
FL-2.1.3	K1	1	1	
FL-2.2.1, FL-2.2.2	K2	1	1	K1 = 2 K2 = 4
FL-2.2.3, FL-2.3.1	K2	1	1	K3 = 0
FL-2.1.1, FL-2.1.6	K2	1	1	Liczba punktów dla tego
FL-2.1.4, FL-2.1.5	K2	1	1	rozdziału = 6

TABELA 0.3. Szczegółowy rozkład pytań egzaminacyjnych w podziale na poziomy K i rozdziały – ciąg dalszy

ROZKŁAD PYTAŃ W ROZDZIALE 3.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA
FL-3.1.1, FL-3.2.1, FL-3.2.3, FL-3.2.5	K1	2	1 W sumie 4 pytania wymagane dla rozdziału 3. K1 = 2 K2 = 2 K3 = 0 Liczba punktów dla tego rozdziału = 4
FL-3.1.2, FL-3.1.3	K2	1	1
FL-3.2.2, FL-3.2.4	K2	1	1

ROZKŁAD PYTAŃ W ROZDZIALE 4.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA
FL-4.1.1	K1	1	1 W sumie 11 pytań wymaganych dla rozdziału 4. K1 = 1 K2 = 5 K3 = 5
FL-4.3.1, FL-4.3.2, FL-4.3.3	K2	2	1
FL-4.4.1, FL-4.4.2, FL-4.4.3	K2	2	1
FL-4.5.1, FL-4.5.2	K2	1	1 Liczba punktów dla tego rozdziału = 11
FL-4.2.1, FL-4.2.2, FL-4.2.3, FL-4.2.4, FL-4.5.3	K3	5	1

ROZKŁAD PYTAŃ W ROZDZIALE 5.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA
FL-5.1.2, FL-5.1.6, FL-5.2.1, FL-5.3.1	K1	1	1 W sumie 9 pytań wymaganych dla rozdziału 5. K1 = 1 K2 = 5 K3 = 3
FL-5.1.1, FL-5.1.3	K2	1	1
FL-5.1.7	K2	1	1
FL-5.2.2, FL-5.2.3, FL-5.2.4	K2	1	1 Liczba punktów dla tego rozdziału = 9
FL-5.3.2, FL-5.3.3	K2	1	1
FL-5.4.1	K2	1	1
FL-5.1.4, FL-5.1.5, FL-5.5.1	K3	3	1

TABELA 0.3. Szczegółowy rozkład pytań egzaminacyjnych w podziale na poziomy K i rozdziały – ciąg dalszy

ROZKŁAD PYTAŃ W ROZDZIALE 6.	POZIOM K	LICZBA PYTAŃ Z WYBRANEJ GRUPY CELÓW NAUCZANIA	PUNKTACJA POJEDYNCZEGO PYTANIA
FL-6.1.1	K2	1	1 W sumie 2 pytania wymagane dla rozdziału 6.
FL-6.2.1	K1	1	1 K1 = 1 K2 = 1 K3 = 0 Liczba punktów dla tego rozdziału = 2
CERTYFIKOWANY TESTER — POZIOM PODSTAWOWY PODSUMOWANIE		40 PUNKTÓW, 60 MINUT W SUMIE 40 PYTAŃ	

Z analizy powyższej tabeli wynika, że na egzaminie *na pewno* pojawi się 17 następujących pytań:

- pięć pytań na poziomie K1 (FL-1.5.2, FL-2.1.2, FL-2.1.3, FL-4.1.1, FL-6.2.1);
- cztery pytania na poziomie K2 (FL-1.3.1, FL-5.1.7, FL-5.4.1, FL-6.1.1);
- osiem pytań pokrywających wszystkie osiem celów nauczania na poziomie K3.

Każde z pozostałych 23 pytań będzie wybrane z grupy dwóch lub więcej celów nauczania. Ponieważ nie wiadomo, które z tych celów nauczania będą pokryte, kandydat i tak musi opanować cały materiał zawarty w syllabusie (wszystkie cele nauczania).

Wskazówki — przed egzaminem i w jego trakcie

Aby skutecznie zdać egzamin, należy przede wszystkim uważnie przeczytać syllabus i słownik pojęć, których znajomość jest wymagana na poziomie podstawowym. Egzamin jest bowiem ściśle oparty na tych dwóch dokumentach. Wskazane jest, aby rozwiązywać również pytania testowe oraz przykładowe egzaminy — na stronach ISTQB® (www.istqb.org) można znaleźć przykładowe, oficjalne zestawy egzaminacyjne w języku angielskim, na stronie SJSI (www.sjsi.org) — w językach polskim i angielskim. Niniejsza publikacja, poza szeregiem przykładowych pytań dla każdego rozdziału syllabusa, zawiera również oficjalny przykładowy egzamin ISTQB®.

W trakcie samego egzaminu należy:

- Uważnie czytać pytania — czasem jedno słowo zmienia cały sens pytania lub jest wskazówką pozwalającą na udzielenie poprawnej odpowiedzi!
- Zwracać uwagę na słowa kluczowe (na przykład w jakim modelu cyklu wytwarzania oprogramowania jest prowadzony projekt).

- Starać się dopasować pytanie do celu nauczania — wtedy łatwiej będzie zrozumieć ideę pytania oraz uzasadnić poprawność i niepoprawność poszczególnych odpowiedzi.
- Uważać na pytania zawierające negację (na przykład „które z poniższych NIE jest...”) — w takich pytaniach trzy odpowiedzi będą poprawne, a jedna niepoprawna. Należy wskazać odpowiedź *niepoprawną*.
- Wybierać odpowiedź, która bezpośrednio odpowiada na pytanie. Niektóre odpowiedzi mogą być zupełnie poprawnymi zdaniami, ale nie odpowiadają na zadane pytanie — na przykład pytanie dotyczy ryzyka związanego z automatyzacją, a jedna z odpowiedzi wymienia jakąś korzyść automatyzacji.
- Strzelać, jeśli nie wiadomo, którą odpowiedź wybrać — nie ma punktów ujemnych, więc nie opłaca się pozostawiać pytań bez odpowiedzi.
- Pamiętać, że odpowiedzi z silnymi, kategorycznymi sformułowaniami zazwyczaj są niepoprawne (na przykład „zawsze”, „musi być”, „nigdy”, „w każdym przypadku”) — choć nie jest to regułą.

CZĘŚĆ II

**OMÓWIENIE
TREŚCI SYLABUSA**

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 oczekiwanej 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.

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 — testowały 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.

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).



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, białośkrzynkowe 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;



- 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 oczekiwaniemi 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 wcześniej 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 oczekiwaniemi 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).

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 retests 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):

```
d:\pw\c#\trojkat\debug\Trojkat.exe
16500
16500
16500
nie
Press any key to continue...
```

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 oczekiwany 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;
}
```

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 mieścią 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. Rozwiążanie 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 miałaścieli 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

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.

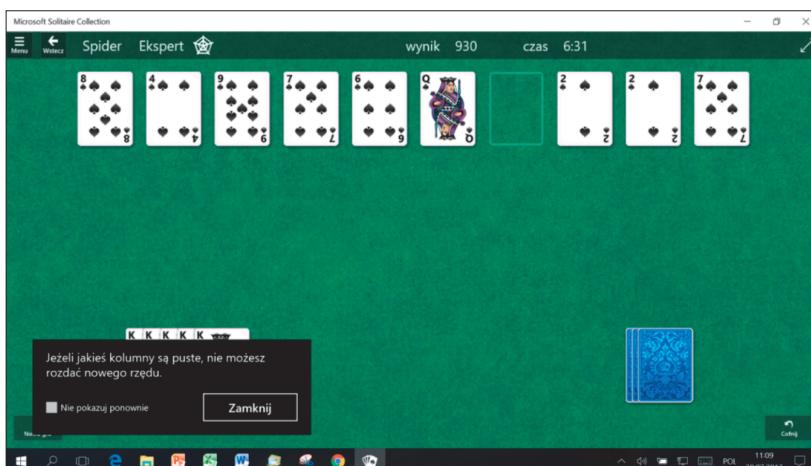
Przypadek gry — pasjans „Pająk”

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 $3 \cdot 13 = 39$ kart — 3 pełne kolory), pojawią 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零, 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 startie w Kourou w Gujanie Francuskiej. Był to pierwszy lot tej rakiety, po dekadzie przygotowań, które kosztowały 7 miliardów USD. Rakietą 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 wznieść (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 przehvyciła 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.

Był to błąd arytmetyczny. System mierzył czas w dziesiątych częściach sekundy, używając 24-bitowego stałoprzecinkowego rejestru. Ułamek $\frac{1}{10}$ w systemie dziesiętnym ma skończoną reprezentację, ale w systemie dwójkowym jest ułamkiem o nieskończonym rozwinięciu. Ponieważ komputer reprezentuje liczby w skończonych rejestrach, jakąś 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 wkopili 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 przemyślowe.



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 zaspokoiliły 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.

Weryfikacja i walidacja oprogramowania przez testerów przed przekazaniem go do eksploatacji umożliwia wykrycie awarii, ułatwia usuwanie związkanych 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 zwiększoną 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 (choć z powiązaniem 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.

Zapewnienie jakości

Zapewnienie jakości (ang. *quality assurance, QA*) skupia się na ustalaniu, 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 pielegnacji 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	ZAPEWNIEŃ JAKOŚCI	KONTROLA JAKOŚCI
Ogólny opis	Wdrażanie procesów, metodyk i standardów, które zapewniają, że wytwarzany produkt spełni wymagane standardy jakościowe	Wykonywanie czynności, których celem jest weryfikacja, że wytwarzany produkt spełnia wymagane standardy jakościowe
Cel	Doskonalenie procesu wytwarzczego	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

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 znajdująca 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. Założymy, 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 liczzonego 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. Założymy 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) 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 ($T[0] \neq 0$) 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 działać tak w przypadku każdego defektu. Niektóre defekty powodują awarię na przykład tylko po wprowadzeniu ścisłe 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;
- myślność 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 wewnętrz 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 Hoppera, zawierającego... rzeczywistą ćmę — przyczynę awarii komputera Mark II.

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 pozytywny¹ 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 oczekiwanej. 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.

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

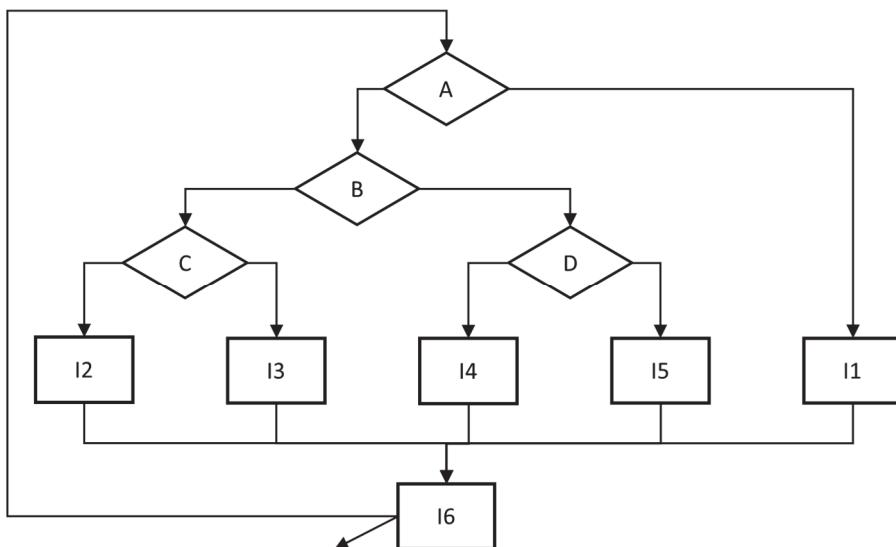
Ta słynna zasada została sformułowana przez Edsgera Dijkstrę, duńskiego informatyka, na konferencji „Software Engineering Techniques” w Rzymie w 1969 roku. Brzmi ona: „Testowanie pokazuje obecność, a nie brak usterek” [Buxton 1970]. Testowanie może wykazać, że istnieją defekty, natomiast nie jesteśmy 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).

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 I₁ I₆
- S2. !A B D I₅ I₆
- S3. !A B !D I₄ I₆
- S4. !A !B C I₃ I₆
- S5. !A !B !C I₂ I₆



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 $5 + 5^2 + 5^3 + \dots + 5^{20} = 119209289550780 > 10^{14}$.

Jeśli przyjmiemy, że potrzebujemy 0,001 sekundy do sprawdzenia jednej ścieżki, to do przetestowania wszystkich ścieżek potrzeba nam będzie około 3860 lat. Nie-stety 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 wytwarzczym. 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.



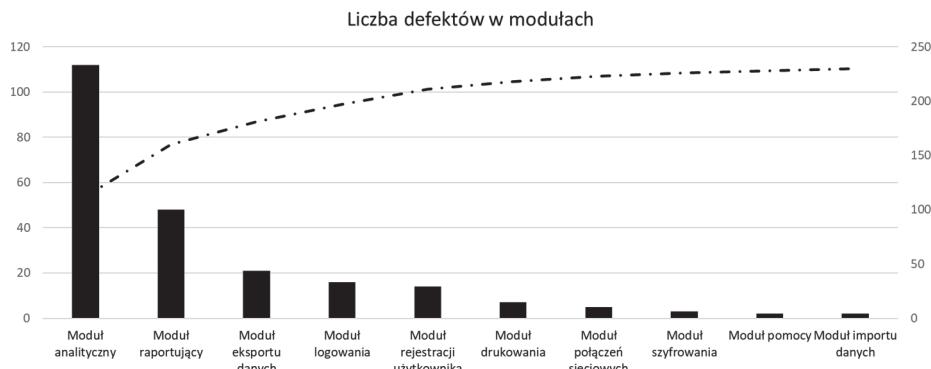
RYSUNEK 1.7. Koszt usunięcia defektu w funkcji czasu

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).



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 przewyciężyć 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).



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 nieraz 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 spełnić 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.

1.4. Czynności testowe, testalia i role związane z testami

FL-1.4.1 (K2)	Kandydat podsumowuje poszczególne czynności i zadania testowe.
FL-1.4.2 (K2)	Kandydat wyjaśnia wpływ kontekstu na proces testowy.
FL-1.4.3 (K2)	Kandydat rozróżnia testalia wspomagające czynności testowe.
FL-1.4.4 (K2)	Kandydat wyjaśnia korzyści wynikające ze śledzenia powiązań.
FL-1.4.5 (K2)	Kandydat porównuje poszczególne role występujące w testowaniu.

1.4.1. Czynności i zadania testowe

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. To, które czynności testowe są uwzględnione w tym procesie testowym, w jaki sposób są wdrażane i kiedy mają miejsce, jest zwykle określane w ramach planowania testów dla konkretnej sytuacji (patrz rozdział 5.).

Dobór procesu testowego do konkretnego oprogramowania zależy od wielu czynników, takich jak:

- wykorzystywany model cyklu wytwarzania oprogramowania i metodyki projektowe;
- rozważane poziomy testów i typy testów;
- ryzyka produktowe i projektowe;
- 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** (ang. *coverage*) 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).



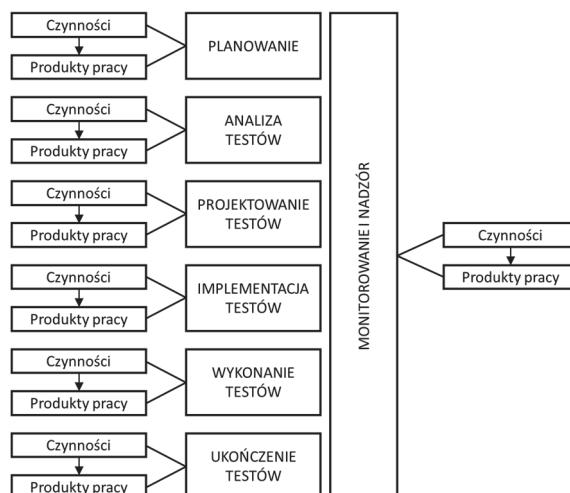
Proces testowy może — ale nie musi — być zdefiniowany formalnie. W typowych sytuacjach składa się z następujących grup czynności:

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

Na przykład rozwój oprogramowania w metodykach zwinnych (ang. *Agile*) obejmuje małe iteracje projektowania, budowania i testowania oprogramowania, które odbywają się w sposób ciągły, wspierany przez bieżące planowanie. Dlatego testowanie odbywa się również w sposób iteracyjny, ciągły, w ramach tego podejścia wytwórczego. Nawet w rozwoju sekwencyjnym grupy procesu testowego — przedstawione powyżej jako sekwencyjne — w rzeczywistym procesie mogą występować iteracyjnie, zazębiać się, występować jednocześnie (np. w testowaniu eksploracyjnym) albo być pomijane. Zależy to od konkretnego projektu. Czynniki kontekstowe, które wpływają na proces testowy w organizacji, obejmują:

- stosowany model cyklu wytwarzania i metodologie projektowe;
- rozważane poziomy i typy testów;
- ryzyko produktowe i projektowe;
- dziedzinę biznesową;
- ograniczenia operacyjne, np. budżet, zasoby, ramy czasowe, złożoność, wymagania umowne i regulacyjne;
- politykę i strategię w organizacji;
- standardy wewnętrzne i zewnętrzne;
- doświadczenie członków zespołu.

Poniżej omówione zostaną przedstawione na rysunku 1.10 czynności w ramach poszczególnych grup procesu testowego. W punkcie 1.4.3 omówione z kolei zostaną produkty prac wytwarzane w ramach tych czynności.



RYSUNEK 1.10. Czynności i produkty pracy w procesie testowym

Planowanie testów — czynności

Planowanie testów (ang. *test planning*) obejmuje zdefiniowanie celów testów i podejścia testowego do ich realizacji w ramach ograniczeń narzuconych przez kontekst. Planowanie testów jest szerzej wyjaśnione w podrozdziale 5.1.



Typowe czynności w ramach planowania testów obejmują:

- 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 wykonania testów, który umożliwi dotrzymanie wyznaczonego terminu;
- zdefiniowanie miar.

Plany testów mogą być zmieniane na podstawie informacji zwrotnych z monitorowania testów i działań kontrolnych. Planowanie testów powinno być działaniem ciągłym.

Monitorowanie testów i nadzór nad testami — czynności

Monitorowanie testów (ang. *test monitoring*) to ciągłe porównywanie rzeczywistego i planowanego postępu testowania przy użyciu miar specjalnie w tym celu zdefiniowanych w planie testów. **Nadzór nad testami** (ang. *test control*) 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. Monitorowanie i nadzór są szerzej wyjaśnione w podrozdziale 5.3.



Postęp prac w stosunku do planu testów jest komunikowany interesariuszom w pisemnych lub ustnych raportach z postępów testów (patrz punkt 5.3.2), które zawierają wszelkie godne uwagi odchylenia od planu oraz informacje o czynnikach utrudniających testowanie i związane z nimi obejścia.

Elementem wspomagającym monitorowanie testów i nadzór nad nimi jest 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 do tego czasu 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.

Analiza testów — czynności

Zadaniem **analizy testów** (ang. *test analysis*) jest zapoznanie się z **podstawą testów** (ang. *test basis*) i przeanalizowanie jej w celu zidentyfikowania testowalnych cech oraz zdefiniowania związanych z nimi warunków testowych, jak również ustalenia, „co” należy przetestować (w kategoriach mierzalnych kryteriów pokrycia). Ogólne *cele testowania* zostają przekształcone w konkretne *warunki testowe i projekty testów*.



Warunek testowy (ang. *test condition*) to wszelkiego rodzaju własność, cecha czy atrybut oprogramowania, które można sprawdzić za pomocą testów. Na przykład w przypadku testowania bankomatu można zdefiniować następujące warunki testowe: „sprawdź, czy bankomat poprawnie rozpoznaje karty płatnicze”; „sprawdź, czy bankomat poprawnie weryfikuje numer PIN wpisywany przez użytkownika”; „sprawdź, czy użytkownik ma możliwość wydrukowania salda konta” itp. Warunki testowe są podstawą do wyprowadzania przypadków testowych i danych testowych.



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.

Typowe czynności w ramach analizy testów obejmują:

- zapoznanie się z podstawą testów: specyfikacją wymagań, wymaganiami biznesowymi, wymaganiami funkcjonalnymi, wymaganiami systemowymi, historijkami użytkownika, opowieściami (ang. *epic*)², przypadkami użycia, podobnymi produktami 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;

² Czasami spotykanym tłumaczeniem słowa „epic” jest „epika”, jednak jest to tłumaczenie niefortunne, ponieważ epika to gatunek literacki, a słowo „epic” oznacza poemat, długą opowieść, epos.

- 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.

Stosowanie technik czarnoskrzynkowych, biało-skrzynkowych i opartych na doświadczeniu może być przydatne w ramach analizy testów w celu zmniejszenia prawdopodobieństwa pominięcia ważnych warunków testowych oraz zdefiniowania bardziej precyzyjnych i dokładnych warunków testowych (patrz rozdział 4.). Bardziej formalne warunki testowe są często przedstawiane jako tzw. modele testowe (np. diagramy przejść między stanami, tabele decyzyjne, wykresy przepływu sterowania).

Identyfikacja defektów w podstawie testów w wyniku przeprowadzenia analizy testów jest ważną korzyścią, zwłaszcza gdy nie jest przeprowadzany oddzielny przegląd podstawy testów. Czynności związane z analizą testów mogą nie tylko zweryfikować, czy wymagania są spójne, właściwie wyrażone i kompletne, ale także zweryfikować, czy wymagania właściwie uwzględniają potrzeby klientów, użytkowników i innych interesariuszy.

Projektowanie testów — czynności

Podczas **projektowania testów** (ang. *test design*) warunki testowe są przekształcane w **przypadki testowe** (ang. *test case*) wysokiego poziomu, zbiory takich przypadków testowych oraz w inne testalia. Projektowanie testów odpowiada na pytanie „jak należy testować”.



Opracowywanie warunków testowych w przypadkach testowych często wiąże się z identyfikacją *elementów pokrycia* testowego oraz z użyciem technik testowych (patrz rozdział 4.). Pozycje pokrycia testowego służą również jako wskazówki do określania danych wejściowych przypadków testowych, a w niektórych sytuacjach mogą nawet definiować dane wejściowe (np. w analizie wartości brzegowych).

Projektowanie testów poprzedza implementację testów lub czasami mogą być wykonywane jednocześnie, ale ważne jest, aby rozróżnić te dwie czynności. Projektując testy przed ich wdrożeniem, można zidentyfikować defekty w projekcie testów, zmniejszając w ten sposób ryzyko marnowania czasu i wysiłku na implementację.

Czynności w ramach projektowania testów 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.

Implementacja testów — czynności

W ramach **implementacji testów** (ang. *test implementation*) tworzone i/lub kończone są testalia niezbędne do wykonania testów, w tym m.in. szeregowanie przypadków testowych w ramach **procedur testowych** (ang. *test procedure*), tworzenie zautomatyzowanych skryptów testowych, **danych testowych** (ang. *test data*), często też tworzone jest środowisko testowe. 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?”.



Czynności fazy implementacji obejmują:

- opracowanie procedur testowych i określenie ich priorytetów;
- 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.

Wykonywanie testów — czynności

Podczas **wykonywania testów** (ang. *test execution*) zestawy testów są uruchamiane zgodnie z harmonogramem wykonania testów. W ramach tej grupy wykonywane są następujące czynności:



- zarejestrowanie danych identyfikacyjnych i wersji elementów testowych bądź przedmiotu testów, narzędzi testowych i testaliów;
- wykonywanie testów manualnie lub przy użyciu narzędzi;

- porównanie rzeczywistych wyników testów z oczekiwanyymi;
- przeanalizowanie anomalii w celu ustalenia ich prawdopodobnych przyczyn (np. 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.

Ukończenie testów — czynności

W fazie **ukończenia testów** (ang. *test completion*) 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 zwanego (np. w ramach spotkania retrospektynego);
- ukończenie poziomu testów;
- zakończenie prac nad wydaniem pielęgnacyjnym.

W ramach grupy ukończenia testów wykonywane są poniższe czynności:

- sprawdzanie, czy wszystkie raporty o defektach są zamknięte, i tworzenie żądań zmian lub pozycji rejestru produktu dla wszelkich nierozwiązań defektów;
- identyfikowanie i archiwizowanie wszelkich testów, które mogą być przydatne w przyszłości;
- przekazanie testaliów zespołowi zajmującemu się pielęgnacją, innym zespołem projektowym lub innym interesariuszom, którzy mogliby skorzystać z ich użycia;
- sprowadzenie środowiska testowego do uzgodnionego stanu;
- analizowanie zakończonych czynności testowych w celu zidentyfikowania wyciągniętych wniosków i zidentyfikowania ulepszeń dla przyszłych iteracji, wydań lub projektów;
- stworzenie raportu z zakończenia testów do przekazania interesariuszom.

1.4.2. Proces testowy w kontekście

Testowanie nie jest przeprowadzane w izolacji. Wspiera ono proces wytwórczy, ustalony w danej organizacji. Testowanie jest również procesem sponsorowanym przez interesariuszy, z których każdy ma wobec końcowego produktu wymagania

czy oczekiwania. Dlatego proces testowy musi być dostosowany do przyjętego w organizacji procesu wytwarzczego, a sposób jego szczegółowej konstrukcji będzie zależał od szeregu czynników kontekstowych. Czynniki te obejmują w szczególności:

- interesariuszy (ich potrzeby, oczekiwania, wymagania, w tym wymagania biznesowe wobec produktu, chęć do współpracy z zespołem testowym itp.);
- członków zespołu (ich umiejętności, wiedzę, poziom doświadczenia, dostępność, potrzeby szkoleniowe, relacje z innymi członkami zespołu itp.);
- dziedzinę biznesową (zidentyfikowane ryzyka produktowe, potrzeby rynku, specyficzne uwarunkowania prawne itp.);
- czynniki techniczne i technologiczne (architekturę projektu, wykorzystywaną technologię itp.);
- ograniczenia projektowe (zakres projektu, czas, dostępny budżet, dostępne zasoby, ryzyka projektowe itp.);
- czynniki organizacyjne (strukturę organizacyjną, istniejące polityki, w tym politykę testów, stosowane praktyki itp.);
- model cyklu wytwarzania (techniki inżynierskie, praktyki deweloperskie, metody wytwarzania oprogramowania itp.);
- narzędzia (ich dostępność, trudność w użyciu itp.).

Powyższe czynniki będą istotnie wpływać na sposób organizacji i przeprowadzenia procesu testowego w projekcie. W szczególności będą wpływać na:

- strategię testową;
- wykorzystywane techniki testowania;
- stopień automatyzacji testów;
- wymagany poziom pokrycia testowego w odniesieniu do wymagań oraz zidentyfikowanych ryzyk;
- poziom szczególowości oraz rodzaj wytwarzanej dokumentacji testowej;
- sposób i poziom szczególowości raportowania postępu testowania;
- sposób i poziom szczególowości raportowania defektów.

1.4.3. Testalia

W ramach procesu testowego powstają **testalia** (ang. *testware*), czyli produkty pracy związane z testowaniem (patrz rysunek 1.10). 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.



Planowanie testów — produkty pracy

Typowymi produktami pracy planowania są:

- plan testów,
- rejestr ryzyk
- kryteria wejścia i kryteria wyjścia.

Rejestr ryzyk to lista zidentyfikowanych przez zespół ryzyk, wraz z informacją o ich prawdopodobieństwie, wpływie oraz sposobie ich łagodzenia (zob. podrozdział 5.2). Rejestr ryzyk i kryteria wejścia oraz wyjścia są zwykle częścią planu testów. W większych projektach może występować więcej niż jeden plan testów, np. kilka planów odnoszących się do poszczególnych poziomów testowania (plan testów integracyjnych, plan testów akceptacyjnych itp.).

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 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 wykonania 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).

Więcej informacji na temat celu i treści planu testów znajduje się w punkcie 5.1.1.

Monitorowanie testów i nadzór nad testami — produkty pracy

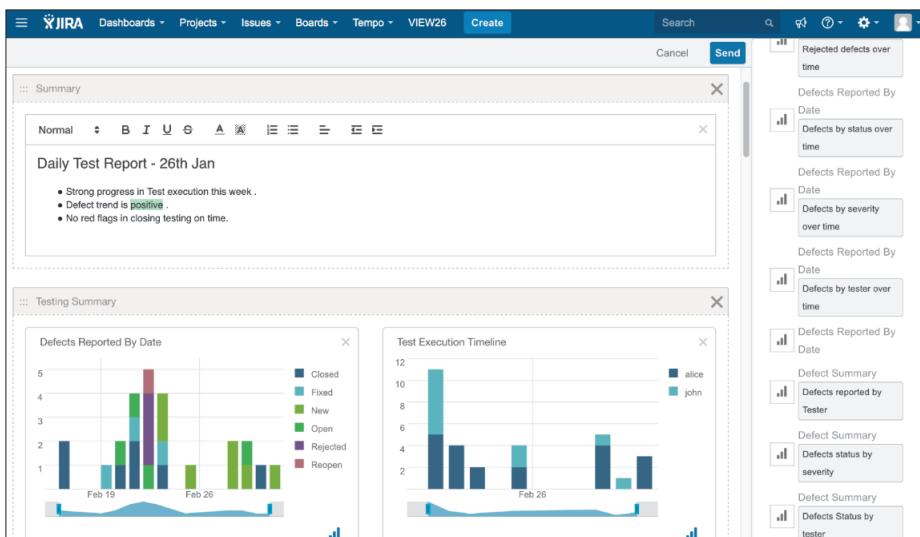
Zasadniczymi produktami pracy w tej grupie są:

- raporty o postępie testów (zob. punkt 5.3.2);
- dokumentacja dyrektyw nadzorczych (zob. podrozdział 5.3);
- informacja o ryzyku (zob. podrozdział 5.2).

Raporty o postępie testów są tworzone na bieżąco i/lub w regularnych odstępach czasu. Wszystkie raporty z testów powinny zawierać istotne dla odbiorców szczegółowe informacje o bieżącym postępie testów, w tym podsumowanie wyników ich wykonania, gdy staną się one dostępne. Informacje o ryzyku mogą być przechowywane w rejestrze ryzyka projektu lub lokalnie w planie testów.

Raport o postępie testów 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 raport o postępie testów przedstawiony jest na rysunku 1.11. Pokazuje on dzienny raport z testów w porównaniu z dniami poprzednimi. 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 1.11. Przykładowy raport dzienny z testów (źródło: acommunity.atlassian.com)

Analiza testów — produkty pracy

Typowymi produktami pracy fazy analizy są:

- (priorytetyzowane) warunki testowe;
- kryteria akceptacji (zob. punkt 4.5.2);
- raporty o defektach w podstavie testów (jeśli nie są naprawiane bezpośrednio).

Warunki testowe zazwyczaj definiuje się i szereguje według priorytetów. Kryteria akceptacji mogą być traktowane jak odpowiedniki warunków testowych w metodykach zwinnego wytwarzania oprogramowania. W przypadku testowania eksploracyjnego

produktem pracy mogą być karty opisu testów. Analiza testów może również skutkować wykryciem i zgłoszeniem defektów w podstawie testów, co oznacza, że produkty pracy analizy testów to również raporty o defektach.

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 — produkty pracy

W wyniku projektowania testów powstają w szczególności:

- przypadki testowe;
- elementy pokrycia;
- wymagania dotyczące danych testowych;
- projekt środowiska testowego.

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 <i>Enter</i> 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 <i>Loguj</i>
Wciśnij <i>Enter</i> .	Logowanie do serwisu
Warunki wyjścia: użytkownik zalogowany, fakt logowania zapisany do bazy wraz z czasem logowania	

Implementacja testów — produkty pracy

W ramach tej grupy czynności tworzone są następujące produkty pracy:

- procedury testowe (wraz z kolejnością ich wykonywania);
- automatyczne skrypty testowe;
- zestawy testowe;
- dane testowe;
- harmonogram wykonania testów;
- elementy środowiska testowego.

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.

Przykładami elementów środowiska testowego są:

- obiekty imitacji (np. zaślepki, sterowniki);
- symulatory;
- wirtualizacja usług.

W testowaniu eksploracyjnym niektóre produkty pracy związane z projektowaniem i implementacją testów mogą być tworzone podczas wykonywania testów, chociaż zakres dokumentowania testów eksploracyjnych i ich identyfikowalności z określonymi elementami podstawy testów są bardzo zróżnicowane.

Czasami produktem pracy tej grupy jest opis wykorzystania narzędzi (np. do wirtualizacji usług). Najbardziej typowymi produktami pracy tej grupy są natomiast skrypty testów automatycznych.

Przykład. Założymy, że testujemy funkcję `mnoż(x, y)`, która przyjmuje na wejściu dwie liczby całkowite i zwraca ich iloczyn. W fazie analizy testów zdefiniowano następujący warunek testowy: „sprawdzić poprawność mnożenia w przypadku występowania wartości równej零”. W trakcie projektowania testów ten warunek testowy przekształcono w 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 jest 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 (sługującej 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.

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

public class MyTests {
    @Test
    public void mnozenieLiczbowCalkowitychPrzez0Daje0() {
        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");
    }
}
```

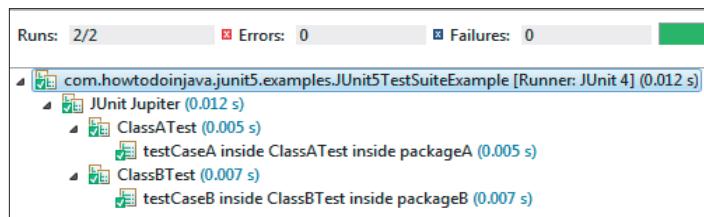
Wykonywanie testów — produkty pracy

Typowymi produktami pracy wykonywania testów są:

- dzienniki (logi) testów;
- dokumentacja statusu poszczególnych przypadków testowych;
- raporty o defektach (patrz punkt 5.5.1);
- dokumentacja wskazująca, co zostało wykorzystane w ramach testowania (np. obiekty testów, narzędzia testowe i testalia).

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 (patrz punkt 1.4.4). Można np. wskazać, które wymagania zostały całkowicie przetestowane i pozytywnie 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 1.12 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 1.12. Wynik wykonania testów jednostkowych w JUnit (źródło: howtodoinjava.com)

Ukończenie testów — produkty pracy

Typowymi produktami prac w ramach fazy ukończenia testów są:

- sumaryczny raport z testów (zob. punkt 5.3.2);
- elementy działań w celu poprawy kolejnych projektów lub iteracji;
- żądania zmian (np. jako elementy backlogu produktu).

1.4.4. Śledzenie powiązań między podstawą testów a testaliami

W celu zapewnienia skutecznego monitorowania i nadzoru istotne jest stworzenie i utrzymanie mechanizmu śledzenia powiązań (ang. *traceability*) między każdym elementem podstawy testów a odpowiadającymi mu produktami pracy związanymi z testowaniem (np. warunkami testowymi, przypadkami testowymi, ryzykami) 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 (wymagań, 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.

Kryteria pokrycia mogą funkcjonować jako kluczowe wskaźniki wydajności (ang. *Key Performance Indicators*, KPI) demonstrujące osiągnięcie celów testu (patrz punkt 1.1.1). Na przykład poprzez wykorzystanie możliwości śledzenia od:

- przypadków testowych do wymagań można zweryfikować, czy spełnione jest pokrycie wymagań przez przypadki testowe;
- wyników przypadków testowych do ryzyk można ocenić poziom ryzyka rezydualnego w obiekcie testowym.

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

Znak X oznacza, że dany przypadek testowy pokrywa dane ryzyko. Na przykład PT 001 pokrywa ryzyko 1 i 2, PT 003 — ryzyko 3 itd. Założymy, że PT 001, PT 002 i PT 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 związań z nim przypadków testowych, które zakończyły się sukcesem).

1.4.5. Role w procesie testowania

Syllabus poziomu podstawowego wyróżnia dwie zasadnicze role w testowaniu: rolę zarządczą (kierowniczą, menedżerską) oraz rolę testera (techniczną). Czynności i zadania przypisane do tych ról zależą od wielu czynników, takich jak kontekst projektu czy produktu, przyjęty model cyklu wytwarzania, umiejętności ludzi czy organizacja pracy w zespole.

Osoba (lub zespół) będąca w roli zarządzania testami ponosi odpowiedzialność za realizację procesu testowego, organizację pracy zespołu testowego i kierowanie działaniami testowymi. Zadania zarządzania testami koncentrują się głównie na czynnościach związanych z:

- planowaniem testów;
- monitorowaniem i nadzorem testów;
- ukończeniem testów.

Osoba w roli testera ponosi ogólną odpowiedzialność za inżynierski (techniczny) aspekt testowania. Zadania związane z testowaniem koncentrują się głównie na:

- analizie testów;
- projektowaniu testów;
- implementacji testów;
- wykonywaniu testów.

Sposób, w jaki rola zarządcza jest zdefiniowana i rozumiana, zależy w szczególności od przyjętego modelu cyklu wytwarzania. Czasami rolę tę pełni pojedyncza osoba, zwana najczęściej kierownikiem testów. Ale na przykład w projektach prowadzonych na bazie metodyk zwinnych (Agile) niektóre zadania zarządcze mogą pozostać w gestii całego (samoorganizującego się) zespołu deweloperskiego. Zadania dotyczące wielu zespołów lub całej organizacji mogą być z kolei wykonywane przez kierowników testów spoza zespołu deweloperskiego.

Należy też odróżniać role od stanowisk pełnionych w firmie. Stanowisko jest zazwyczaj przypisane na stałe do konkretnej osoby: każdy członek zespołu jest zwykle zatrudniony na określonym stanowisku. Natomiast każdy członek zespołu w różnych okresach może pełnić różne role. W szczególności rola zarządcza w testowaniu może być pełniona przez lidera zespołu (ang. *team leader*), kierownika projektu, kierownika działu jakości itp. W większych projektach kierownik lub koordynator testów może mieć pod sobą kilka zespołów testowych, którymi kierują liderzy testów lub testerzy prowadzący.

Z kolei rolę testera pełni każdy, kto w danym momencie przeprowadza jakiekolwiek czynności związane z testowaniem. W tym sensie np. programista w momencie przeprowadzania testów modułowych czy klient wykonujący testy akceptacyjne wchodzą w rolę testera. Jest również możliwe, że jedna osoba pełni w jednym momencie obie role — testerską oraz zarządczą.

Poniżej omówione zostaną szczegółowo dwie role: kierownika testów i testera.

Rola kierownika testów

Sposób wykonywania obowiązków kierownika testów zależy od cyklu wytwarzania oprogramowania. Typowe zadania kierownika testów to:

- opracowywanie lub dokonywanie przeglądu strategii testów i polityki testów;
- planowanie projektu testowania z uwzględnieniem kontekstu (patrz podrozdział 5.1):
 - tworzenie i aktualizowanie planów testów,
 - planowanie iteracji i wydania w projektach zwinnych,
 - wybór podejścia do testowania,
 - definiowanie kryteriów wejścia i wyjścia,
 - wprowadzanie odpowiednich miar służących do mierzenia postępu testów oraz oceniania jakości testowania i produktu,
 - definiowanie poziomów i cykli testowych,
 - szacowanie czasu, pracochronności i kosztów testowania,
 - priorytetyzacja testów;
- zarządzanie ryzykiem (patrz podrozdział 5.2);
- monitorowanie rezultatów testów, sprawdzanie statusu kryteriów wyjścia (definicji ukończenia) i sprawowanie nadzoru nad testami np. poprzez dostosowywanie planów do rezultatów i postępu testów (patrz podrozdział 5.3);

- nadzór nad procesami:
 - zarządzania konfiguracją (patrz podrozdział 5.4),
 - zarządzania defektami (patrz podrozdział 5.5);
- pozyskiwanie zasobów;
- koordynowanie strategii testów i planu testów z kierownikami projektu i innymi interesariuszami;
- prezentowanie punktu widzenia testerów w ramach innych czynności projektowych, np. w planowaniu integracji;
- inicjowanie procesów analizy, projektowania, implementacji i wykonywania testów;
- raportowanie postępu testów, tworzenie raportu sumarycznego z testów;
- wspieranie zespołu przy korzystaniu z narzędzi służących do realizacji procesu testowego (np. pozyskiwanie środków na zakup narzędzia, zakup licencji, nadzór nad wdrożeniem narzędzia w organizacji);
- podejmowanie decyzji o implementacji środowisk testowych;
- promowanie testerów i zespołu testowego oraz reprezentowanie ich punktu widzenia w organizacji;
- rozwijanie umiejętności i kariery testerów poprzez plan szkoleń, ewaluacje osiągnięć, coaching.

Rola testera

Typowe zadania testera to:

- dokonywanie przeglądu planów testów i uczestniczenie w ich opracowywaniu;
- analizowanie, dokonywanie przeglądu i ocenianie podstawy testów (tj. wymagań, historyjek użytkownika i kryteriów akceptacji, specyfikacji oraz modeli) pod kątem testowalności;
- identyfikowanie i dokumentowanie warunków testowych oraz rejestrowanie powiązań między przypadkami testowymi, warunkami testowymi i podstawą testów;
- projektowanie, konfigurowanie i weryfikowanie środowisk testowych (na ogół w porozumieniu z administratorami systemu i sieci);
- projektowanie i implementowanie przypadków testowych i skryptów testowych;
- przygotowywanie i pozyskiwanie danych testowych;
- współtworzenie harmonogramu wykonania testów;
- wykonywanie testów, ocenianie rezultatów i dokumentowanie odchyleń od oczekiwanych rezultatów;
- korzystanie z odpowiednich narzędzi usprawniających proces testowy (np. narzędzia do automatyzacji testów);
- ocena i pomiar charakterystyk niefunkcjonalnych oprogramowania (patrz punkt 2.2.2);
- współpraca w zespole (np. przeglądanie testów zaprojektowanych przez innych);

- używanie — w razie potrzeby — narzędzi do administrowania lub zarządzania testami;
- automatyzacja testów (np. tworzenie, uruchamianie i modyfikacja skryptów testowych).

Należy pamiętać, by osoby pracujące w zespole testowym (i ogólniej — testujące), a więc zajmujące się analizą testów, projektowaniem testów, konkretnymi typami testów czy automatyzacją, były specjalistami w swoich rolach. Tak jak wspomniano wcześniej, w zależności od poziomu testów oraz ryzyka produktowego i projektowego rolę testera mogą pełnić różne osoby:

- na poziomach modułowym i integracyjnym zwykle są to *programiści*;
- na poziomie testów systemowych zwykle są to *testerzy*, członkowie niezależnego zespołu testerskiego;
- na poziomie testów akceptacyjnych testerami mogą być *eksperci biznesowi* oraz *użytkownicy*;
- testerami w produkcyjnych testach akceptacyjnych mogą być *operatorzy systemów* oraz *administratorzy systemów*.

1.5. Niezbędne umiejętności i dobre praktyki w dziedzinie testowania

FL-1.5.1 (K2)	Kandydat podaje przykłady ogólnych umiejętności wymaganych w testowaniu.
FL-1.5.2 (K1)	Kandydat pamięta, jakie są zalety podejścia „cały zespół”.
FL-1.5.3 (K2)	Kandydat omawia korzyści i wady niezależności testowania.

1.5.1. Ogólne umiejętności wymagane w związku z testowaniem

Proces wytwarzania oprogramowania, w tym jego testowanie, jest realizowany głównie przez ludzi, w związku z czym duże znaczenie dla przebiegu testowania mają umiejętności poszczególnych testerów oraz uwarunkowania psychologiczne.

Zasadnicze umiejętności wymagane w testowaniu

Dobry tester powinien cechować się szeregiem cech, które sprawią, że jego praca będzie efektywna i wydajna. W szczególności dobry tester posiada następujące cechy (zob. też [Stray 2021]):

- dokładność, staranność, ciekawość, zwracanie uwagi na szczegółowe, metodyczność (w celu zidentyfikowania różnych rodzajów wad i defektów, zwłaszcza tych trudnych do znalezienia);
- dobre umiejętności komunikacyjne, aktywne słuchanie, bycie graczem zespołowym (aby efektywnie współpracować ze wszystkimi interesariuszami, przekazywać informacje innym, być zrozumianym, zgłaszać i omawiać defekty);

- myślenie analityczne, myślenie krytyczne, kreatywność (w celu zwiększenia efektywności testów);
- posiadanie wiedzy technicznej (w celu zwiększenia wydajności testowania, np. poprzez wykorzystanie narzędzi testowych);
- znajomość technik szacowania (aby dokładniej oszacować wysiłek związany z testowaniem);
- posiadanie wiedzy dziedzinowej (aby być w stanie zrozumieć użytkowników końcowych i komunikować się z nimi).

Aspekty psychologiczne w testowaniu

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*) — jest to tendencja do preferowania informacji, które potwierdzają wcześniejsze oczekiwania i hipotezy, niezależnie od tego, czy te informacje są prawdziwe. Może to utrudniać zaakceptowanie informacji sprzecznych z dotychczasowymi przekonaniami. Powoduje to, że ludzie poszukują informacji i zapamiętują je w sposób selektywny, błędnie je interpretując. 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 wyjaśnili 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 zgłasiane przez nich awarie są efektem defektów w kodzie; występuje u nich efekt potwierdzenia, przez który trudno jest im zaakceptować sytuację, 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. Mają oni 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 posiadać 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/rozwiązań).
- 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.

1.5.2. Podejście „cały zespół”

Jedną z ważnych umiejętności testowania jest bycie „graczem zespołowym”, posiadanie umiejętności efektywnej pracy w zespole i wnoszenie pozytywnego wkładu w osiągnięcie celu zespołu. Na tej umiejętności opiera się podejście „cały zespół” (ang. *whole team approach*).

Podejście „cały zespół” charakteryzuje się następującymi atrybutami:

- angażowanie wszystkich osób posiadających niezbędną wiedzę i umiejętności, aby zapewnić sukces projektu;
- udział stosunkowo małych zespołów liczących kilka osób;
- dzielenie tej samej przestrzeni roboczej, co znacznie ułatwia komunikację i interakcję.

W zwinnym tworzeniu oprogramowania podejście „cały zespół”:

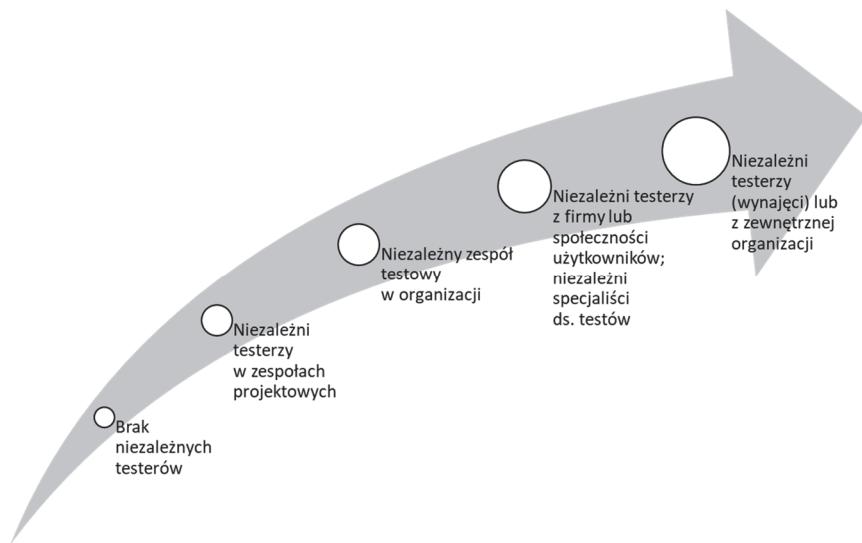
- zapewnia, że w zespole są przedstawiciele klienta i innych interesariuszy biznesowych, którzy decydują o cechach produktu;
- jest wspierane poprzez codzienne spotkania (ang. *stand-up meeting*) z udziałem wszystkich członków zespołu, podczas których to spotkań komunikowany jest postęp prac i wskazywane są wszelkie przeszkody w postępie;
- promuje bardziej efektywną i wydajną dynamikę zespołu;
- poprawia komunikację i współpracę w zespole;
- umożliwia wykorzystanie różnych umiejętności członków zespołu z korzyścią dla projektu;
- sprawia, że wszyscy są odpowiedzialni za jakość.

Istota podejścia „cały zespół” polega na tym, że programiści, przedstawiciele biznesu i testerzy współpracują ze sobą na każdym etapie procesu rozwoju oprogramowania. Ścisła współpraca tych trzech grup ma na celu zapewnienie osiągnięcia pożądanych poziomów jakości. Obejmuje ona współpracę z przedstawicielami biznesu, aby pomóc im w tworzeniu odpowiednich testów akceptacyjnych, oraz współpracę z programistami w celu uzgodnienia strategii testowania i podjęcia decyzji o podejściu do automatyzacji testów. Testerzy mogą także przekazywać wiedzę o testowaniu innym członkom zespołu oraz wpływać na rozwój produktu.

Cały zespół jest zaangażowany we wszelkie konsultacje czy spotkania, podczas których ustalane są, analizowane lub szacowane cechy produktu. Koncepcja angażowania testerów, programistów i przedstawicieli biznesu we wszystkie dyskusje na temat funkcji tworzonego produktu jest znana jako „siła trzech” [Crispin 2008].

1.5.3. Niezależność testowania

Zadania związane z testowaniem mogą wykonywać zarówno osoby pełniące konkretne role w procesie testowym, jak i osoby pełniące inne role (np. klienci). Z jednej strony pewien stopień niezależności często zwiększa skuteczność wykrywania defektów, ponieważ autor i tester mogą być obarczeni różnymi błędami poznawczymi (patrz punkt 1.5.1). Z drugiej strony niezależność nie zastępuje znajomości produktu, a programiści mogą efektywnie wykrywać wiele defektów w tworzonym przez siebie kodzie. Należy jednak pamiętać, że w testowaniu autorskim (programiści) autor często nie dostrzega własnych błędów. Programista, tworząc kod, czyni co do niego pewne założenia. Gdy następnie ma ów kod przetestować, może — nawet nieświadomie — pisać testy tak, aby zweryfikowały jego założenia. W rezultacie testy pisane przez programistę na ogół wykryją niewiele problemów w jego własnym kodzie.



RYSUNEK 1.13. Poziomy niezależności testowania

Niezależność testowania (ang. *independence of testing*) wykonywanego przez niezależny zespół testowy posiada szereg zalet, w szczególności:

- zwiększa nacisk na testy;
- zwiększa skuteczność w znajdowaniu defektów i awarii;
- zapewnia dodatkowe korzyści, takie jak niezależne spojrzenie wyszkolonego i profesjonalnego zespołu testowego.

Niezależność testowania może mieć miejsce na każdym poziomie testowania. Istnieją różne poziomy niezależności (poniżej uporządkowano je od najniższego do najwyższo- szego; patrz też rysunek 1.13):

- brak niezależnych testerów — programiści testują własny kod;
- niska niezależność — niezależni programiści lub testerzy pracujący w ramach zespołu programistów; programiści mogą testować produkty tworzone przez współpracowników w ramach tzw. testowania koleżeńskiego;
- wysoka niezależność — testy przeprowadzane przez niezależny zespół testowy działający w ramach organizacji, podlegający kierownictwu projektu lub dyrekcji, bądź przez przedstawicieli innych działów organizacji;
- bardzo wysoka niezależność — niezależni testerzy spoza organizacji, pracujący u klienta (ang. *on-site*) lub poza siedzibą firmy (ang. *outsourcing*).

Rozwiązaniem optymalnym (dla dużych, złożonych projektów lub w zespołach tworzących produkty krytyczne pod względem bezpieczeństwa) jest testowanie na wielu poziomach:

- programiści na niższych poziomach — odpowiedzialni za testy modułowe i integracji modułów (mogą w ten sposób kontrolować jakość swojej pracy; należy jednak pamiętać, że brak obiektywności ogranicza ich skuteczność);
- niezależni testerzy na wyższych poziomach — odpowiedzialni za testy integracji systemów, systemowe oraz akceptacyjne.

Przy rozważaniu poziomów niezależności należy brać pod uwagę także model cyklu wytwarzania oprogramowania. W podejściu zwinnym przyjmuje się, że testerzy mogą zostać przydzieleni do pracy w ramach zespołu programistów, ale czasami mogą być uznawani za członków szerszego niezależnego zespołu testowego. Często właściciele produktu wykonują — na zakończenie każdej iteracji — testowanie akceptacyjne mające na celu walidację historyjek użytkownika.

Niezależny tester ma inne spojrzenie na testowany produkt; dostrzega często inne błędy niżauważone przez programistów ze względu na różne doświadczenia, techniczne punkty widzenia i błędy poznawcze; może zweryfikować poprawność (przeglądy), zakwestionować lub obalić założenia przyjęte przez interesariuszy zarówno w zakresie specyfikacji, jak i architektury systemu. Podchodzi też do obiektu testów bez gotowych oczekiwania i uprzedzeń.

Niezależność testowania niesie ze sobą nie tylko korzyści, ale i pewne zagrożenia. Należy tu wymienić przede wszystkim:

- odizolowanie testerów od zespołu wykonawczego, co może powodować brak komunikacji, opóźnienia w dostarczaniu informacji zwrotnych czy złe relacje z zespołem wytwórczym (*tworzymy jeden zespół*): występuje to najczęściej w wypadku pełnej niezależności testerów od zespołu wytwórczego);
- utratę przez programistów poczucia odpowiedzialności za jakość („skoro produkt testują niezależni eksperci, to ja już nie muszę dbać o jego jakość”);
- możliwość wystąpienia tzw. wąskiego gardła w końcówce projektu i obciążenie testerów odpowiedzialnością za nieterminowe przekazanie produktu do eksploatacji;
- ryzyko, że niezależni testerzy nie będą dysponowali ważnymi informacjami (np. na temat przedmiotu testów).

Pytania testowe do rozdziału 1.

Pytanie 1.1

(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 oprogramowania.
- D. Budowanie zaufania do poziomu jakości testowanego systemu.

Wybierz jedną odpowiedź.

Pytanie 1.2

(FL-1.1.2, K2)

Poniżej podana jest lista czynności związanych z defektami i awariami w oprogramowaniu:

- i. znajdowanie defektów w kodzie
- ii. ujawnianie awarii
- iii. analizowanie znalezionych defektów
- iv. wykonywanie retestów

Które z tych czynności są zadaniem debugowania, a które testowania?

- A. (ii), (iv) — zadania testowania, (i), (iii) — zadania debugowania.
- B. (i), (iv) — zadania testowania, (ii), (iii) — zadania debugowania.
- C. (ii), (iii) — zadania testowania, (i), (iv) — zadania debugowania.
- D. (i), (iii), (iv) — zadania testowania, (ii) — zadanie debugowania.

Wybierz jedną odpowiedź.

Pytanie 1.3

(FL-1.2.1, K2)

Jesteś testerem w projekcie tworzącym grę opartą na mitach greckich. Tworzysz obecnie kryterium akceptacyjne dla następującej historyjki użytkownika:

Jako gracz poziomu 4. chcę móc użyć róźdżki Midasa, by zamienić stojący przed mną przedmiot w złoto i powiększyć moje zasoby finansowe

Zauważłeś(-łaś), że brak jest informacji dotyczącej czasu zamiany przedmiotu w złoto.

Które z poniższych stwierdzeń NAJLEPIEJ obrazuje wkład testowania w sukces projektu?

- A. Poinformowanie zespołu, 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. Polepszenie podcharakterystyki wydajności, jaką jest zachowanie się w czasie.

Wybierz jedną odpowiedź.

Pytanie 1.4

(FL-1.2.2, K1)

Rozważ następujące sformułowania dotyczące testowania i zapewnienia jakości.

- i. Zapewnienie jakości koncentruje się na zapobieganiu awarii poprzez sprawdzenie, czy wymagania jakościowe są spełnione.
- ii. Zapewnienie jakości koncentruje się na kontroli jakości tworzonego produktu.
- iii. Testowanie koncentruje się na ocenie oprogramowania oraz powiązanych produktów w celu określenia, czy spełniają one wyspecyfikowane wymagania.
- iv. Testowanie koncentruje się na usuwaniu defektów z oprogramowania.

Które z tych sformułowań są prawdziwe?

- A. (i) oraz (iii) są prawdziwe; (ii) oraz (iv) są nieprawdziwe.
- B. (ii) oraz (iv) są prawdziwe; (i) oraz (iii) są nieprawdziwe.
- C. (i) oraz (iv) są prawdziwe; (ii) oraz (iii) są nieprawdziwe.
- D. (ii) oraz (iii) są prawdziwe; (i) oraz (iv) są nieprawdziwe.

Wybierz jedną odpowiedź.

Pytanie 1.5

(FL-1.2.3, 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 systemu na błędne dane.

Wybierz jedną odpowiedź.

Pytanie 1.6

(FL-1.3.1, K2)

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

- A. Wczesne testowanie oszczędza czas i pieniądze.
- B. Testowanie zależy od kontekstu.
- C. Testy się zużywają.
- D. Kumulowanie się defektów.

Wybierz jedną odpowiedź.

Pytanie 1.7

(FL-1.4.1, K2)

Podczas której fazy procesu testowego sprawdzana jest testowalność podstawy testów?

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

Wybierz jedną odpowiedź.

Pytanie 1.8

(FL-1.4.2, K2)

Które z poniższych ma NAJMIEJSZY wpływ na proces testowy w organizacji?

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

Wybierz jedną odpowiedź.

Pytanie 1.9

(FL-1.4.3, K2)

Który z poniższych NIE jest produktem pracy powstającym podczas monitorowania testów i nadzoru nad testami?

- A. Raport o postępie testów.
- B. Informacja o bieżącym poziomie ryzyka w produkcie.
- C. Dokumentacja opisująca podjęte działania nadzorcze.
- D. Sumaryczny raport końcowy z testów.

Wybierz jedną odpowiedź.

Pytanie 1.10

(FL-1.4.4, K2)

Która z poniższych czynności jest możliwa do wykonania dzięki mechanizmowi śledzenia powiązań między elementami podstawy testów a odpowiadającymi im produktami pracy związanymi z testowaniem?

- A. Obliczenie poziomu ryzyka w produkcie na podstawie wyników testów.
- B. Zdefiniowanie akceptowalnego poziomu pokrycia kodu testami modułowymi.

- C. Użycie wyroczni testowej do automatycznego określenia oczekiwanej wyników dla przypadku testowego.
- D. Wyprowadzenie danych testowych pokrywających klasy równoważności.

Wybierz jedną odpowiedź.

Pytanie 1.11

(FL 1.4.5, K2)

Które z poniższych stanowią typowe zadania osoby pełniącej rolę związaną z zarządzaniem testami, a które osoby pełniącej rolę związaną z testowaniem?

- i. koordynowanie realizacji strategii testów i planu testów
 - ii. definiowanie warunków testowych
 - iii. tworzenie sumarycznego raportu z testów
 - iv. automatyzowanie testowania
 - v. decydowanie o implementacji środowisk testowych
 - vi. weryfikowanie środowisk testowych
-
- A. (i), (v) oraz (vi) to zadania osoby pełniącej rolę związaną z zarządzaniem testami, a (ii), (iii) oraz (iv) — osoby pełniącej rolę związaną z testowaniem
 - B. (ii), (iii) oraz (vi) to zadania osoby pełniącej rolę związaną z zarządzaniem testami, a (i), (iv) oraz (v) — osoby pełniącej rolę zowaną z testowaniem
 - C. (i), (ii) oraz (v) to zadania osoby pełniącej rolę zowaną z zarządzaniem testami, a (iii), (iv) oraz (vi) — osoby pełniącej rolę zowaną z testowaniem
 - D. (i), (iii) oraz (v) to zadania osoby pełniącej rolę zowaną z zarządzaniem testami, a (ii), (iv) oraz (vi) — osoby pełniącej rolę zowaną z testowaniem

Wybierz jedną odpowiedź.

Pytanie 1.12

(FL-1.5.1, K2)

Która z poniższych cech testera jest NAJMNIEJ krytyczna?

- A. Analytyczne myślenie.
- B. Wiedza dziedzinowa.
- C. Umiejętność programowania.
- D. Umiejętność komunikowania się.

Wybierz jedną odpowiedź.

Pytanie 1.13

(FL-1.5.2, K1)

Które DWIE z poniższych czynności NAJLEPIEJ odpowiadają obowiązkom pełnionym w ramach podejścia „cały zespół”?

- A. Testerzy są odpowiedzialni za implementację testów modułowych, które przekazują programistom do wykonania.
- B. Testerzy współpracują z przedstawicielami klienta i programistami podczas tworzenia testów akceptacyjnych.
- C. Przedstawiciele klienta wybierają narzędzia, których programiści i testerzy będą używali podczas trwania projektu.
- D. Testy niefunkcjonalne są projektowane przez klienta, a wykonywane przez testerów i programistów.
- E. Za jakość produktu odpowiadają nie tylko testerzy, ale też programiści i przedstawiciele klienta.

Wybierz **DWIE** odpowiedzi.

Pytanie 1.14

(FL 1.5.3, K2)

Dlaczego testowanie jest zazwyczaj przeprowadzane przez niezależnych testerów?

- A. Ponieważ pomaga zwiększyć nacisk na testy oraz pozwala uzyskać niezależną opinię zawodowych testerów.
- B. Ponieważ programiści nie mają zdolności do testowania kodu.
- C. Aby wykryte awarie były zgłasiane w sposób konstruktywny.
- D. Aby znajdowanie defektów nie było postrzegane jako krytyka wobec programistów.

Wybierz jedną odpowiedź.

ROZDZIAŁ 2.

Testowanie w cyklu wytwarzania oprogramowania

Słowa kluczowe

poziom testów — specyficzna instancja procesu testowego. *Synonim:* etap testów.

przedmiot testów — moduł lub system podlegający testowaniu.

przesunięcie w lewo (ang. *shift-left*) — podejście do wykonywania czynności związanych z testowaniem i zapewnianiem jakości na jak najwcześniejszym etapie cyklu rozwoju oprogramowania.

testowanie akceptacyjne — poziom testów zorientowany na ustalenie, czy zaakceptować system.

testowanie białośkrzynkowe — testowanie oparte na analizie wewnętrznej struktury modułu lub systemu. *Synonim:* testowanie przezroczysto-skrzynkowe, testowanie na podstawie kodu, testowanie szklano-skrzynkowe, testowanie pokrycia logiki, testowanie sterowane logiką, testowanie strukturalne, testowanie w oparciu o strukturę.

testowanie czarnoskrzynkowe — testowanie, funkcjonalne lub niefunkcjonalne, bez odwoływania się do wewnętrznej struktury modułu lub systemu.

testowanie funkcjonalne — testowanie wykonywane, by ocenić, czy moduł lub system spełnia wymagania funkcjonalne.

testowanie integracji modułów — testy wykonywane w celu wykrycia usterek w interfejsach i interakcjach pomiędzy integrowanymi modułami. *Synonim:* testowanie połączenia.

testowanie integracji systemów — poziom testów koncentrujący się na interakcjach między modułami lub systemami.

testowanie integracyjne — testowanie wykonywane w celu wykrycia defektów w interfejsach i interakcjach pomiędzy modułami lub systemami.

testowanie modułowe — poziom testów, który koncentruje się na poszczególnych modułach sprzętowych lub programowych.

testowanie niefunkcjonalne — testowanie przeprowadzane w celu sprawdzenia zgodności modułu lub systemu z wymaganiami niefunkcjonalnymi.

testowanie pielęgnacyjne — testowanie zmian we wdrożonym systemie lub testowanie wpływu zmienionego środowiska na wdrożony system.

testowanie potwierdzające — rodzaj testu związanego ze zmianą, przeprowadzonego po naprawieniu defektu w celu potwierdzenia, że awaria spowodowana przez ten defekt już nie występuje. *Synonim:* retestowanie.

testowanie regresji — rodzaj testu związanego ze zmianami mający na celu wykrycie, czy defekty zostały wprowadzone lub odkryte w niezmienionych obszarach oprogramowania.

testowanie systemowe — poziom testów, który koncentruje się na sprawdzeniu, czy system jako całość spełnia określone wymagania.

typ testów — grupa czynności w testowaniu ukierunkowana na określone charakterystyki modułu lub systemu, oparta na specyficznych celach testowania.

2.1. Testowanie w kontekście modelu cyklu wytwarzania oprogramowania

FL-2.1.1 (K2)	Kandydat wyjaśnia wpływ wybranego modelu cyklu wytwarzania oprogramowania na testowanie.
FL-2.1.2 (K1)	Kandydat pamięta dobre praktyki testowania mające zastosowanie do wszystkich modeli cyklu wytwarzania oprogramowania.
FL-2.1.3 (K1)	Kandydat podaje przykłady podejść typu „najpierw test” w kontekście wytwarzania oprogramowania.
FL-2.1.4 (K2)	Kandydat podsumowuje, w jaki sposób metodyka DevOps może wpływać na testowanie.
FL-2.1.5 (K2)	Kandydat wyjaśnia, na czym polega przesunięcie w lewo.
FL-2.1.6 (K2)	Kandydat wyjaśnia, w jaki sposób retrospektywy mogą posłużyć jako mechanizmy doskonalenia procesów.

Model cyklu wytwarzania oprogramowania to abstrakcyjna, wysokopoziomowa reprezentacja procesu wytwarzania oprogramowania. Model ten opisuje czynności wchodzące w skład procesu wytwarzania oprogramowania oraz relacje pomiędzy nimi, zarówno logiczne, jak i czasowe. Modele cyklu wytwarzania wykorzystuje się, aby ustalić sposób, w jaki oprogramowanie będzie wytwarzane. Ułatwiają one również zrozumienie następstwa różnych faz procesu wytwórczego.

Przykłady klasycznych modeli cyklu wytwarzania obejmują:

- modele sekwencyjne (np. model kaskadowy [ang. *waterfall model*], model V);
- modele iteracyjne (np. model spiralny Boehma, prototypowanie);
- modele przyrostowe (np. *Unified Process*).

Niektóre aktywności w ramach procesu wytwarzania oprogramowania mogą być również opisywane przez bardziej szczegółowe modele, metodyki wytwórcze czy praktyki zwinne. Przykłady takiego podejścia to:

- Scrum;
- Kanban;
- Lean IT (wytwarzanie oparte na tzw. szczupłym zarządzaniu);
- programowanie ekstremalne (XP — ang. *eXtreme Programming*);
- wytwarzanie sterowane testami (TDD — ang. *Test-Driven Development*);
- wytwarzanie sterowane testami akceptacyjnymi (ATDD — ang. *Acceptance Test-Driven Development*);
- wytwarzanie sterowane zachowaniem (BDD — ang. *Behavior-Driven Development*);
- projektowanie sterowane dziedziną biznesową (DDD — ang. *Domain-Driven Design*);
- wytwarzanie oparte na cechach (FDD — ang. *Feature-Driven Development*).

Więcej informacji na temat modeli cyklu wytwarzania w kontekście inżynierii oprogramowania można znaleźć np. w [Pressman 2019].

2.1.1. Wpływ cyklu wytwarzania oprogramowania na testowanie

Testowanie, aby było skuteczne, musi być zintegrowane z obowiązującym w projekcie modelem cyklu wytwarzania oprogramowania. Wybór modelu cyklu wytwarzania oprogramowania ma wpływ na następujące zagadnienia związane z testowaniem:

- zakres i czas przeprowadzania czynności testowych;
- wybór i umiejscowienie w czasie poziomów i typów testów;
- poziom szczegółowości dokumentacji testowej;
- wybór technik i praktyk testowania;
- zakres automatyzacji testów.

Modele sekwencyjne

Modele sekwencyjne cyklu wytwarzania zakładają wykonywanie poszczególnych czynności wytwórczych jedna po drugiej, liniowo. Konsekwencją przyjęcia takiego modelu wytwarzania oprogramowania jest to, że dana faza nie może się rozpocząć, dopóki nie zakończy się faza poprzednia. W praktyce może się czasami zdarzyć, że

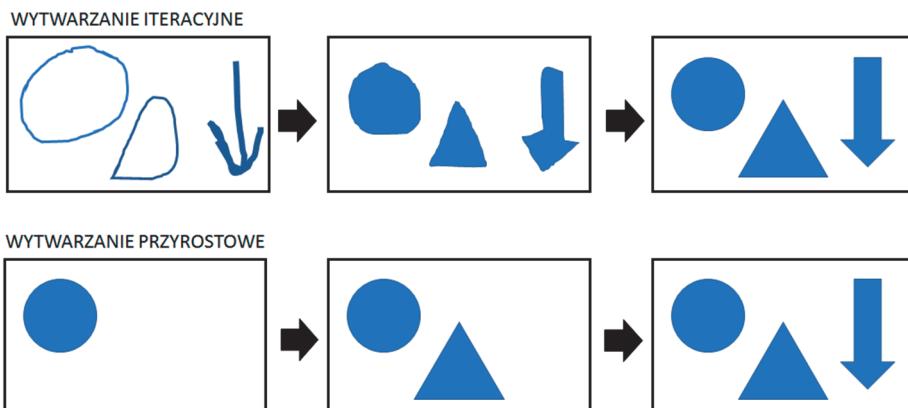
fazy na siebie zachodzą. Jednak modele sekwencyjne zakładają, przynajmniej w teorii, że zanim rozpocznie się kolejna faza, następuje moment weryfikacji tego, czy wszystkie czynności fazy poprzedzającej zostały wykonane poprawnie.

W początkowych fazach projektu prowadzonego w modelach sekwencyjnych tester zazwyczaj uczestniczy w przeglądach wymagań i w projektowaniu testów. Produkt w wykonywalnej formie jest zazwyczaj dostarczany w późniejszych fazach, zatem najczęściej testy dynamiczne nie mogą być przeprowadzane wcześnie w cyklu wytwarzania.

Przykłady najważniejszych modeli sekwencyjnych omówione są poniżej.

Modele iteracyjne i przyrostowe

Wszystkie modele z rodziny modeli iteracyjnych i przyrostowych oparte są na jednej, fundamentalnej zasadzie. Zasada ta mówi, że oprogramowanie wytwarzane jest w cyklach. Pomiędzy tymi dwoma typami modeli istnieją pewne zasadnicze różnice. Symbolicznie pokazane są one na rysunku 2.1.



RYSUNEK 2.1. Symboliczna różnica między wytwarzaniem iteracyjnym i przyrostowym

Przyrostowe wytwarzanie oprogramowania to proces polegający na określaniu wymagań oraz projektowaniu, budowaniu i testowaniu systemu częściami, co oznacza, że funkcjonalność oprogramowania rośnie przyrostowo. Wielkość poszczególnych przyrostów funkcjonalności zależy od konkretnego modelu cyklu wytwarzania: niektóre modele przewidują podział na większe fragmenty, a inne — na mniejsze. Jednorazowy przyrost funkcjonalności może ograniczać się nawet do wprowadzenia pojedynczej zmiany na ekranie interfejsu użytkownika lub nowej opcji zapytania.

Dolina część rysunku 2.1 przedstawia to właśnie podejście. Założymy, że naszym produktem jest obraz złożony z trzech elementów: koła, trójkąta i strzałki. W modelu przyrostowym tworzymy fragmenty tego obrazu po kolei, np. każdą figurę w kolejnym przyroście. Oznacza to w szczególności, że przyrostowo dodawane funkcjonalności powinny mieć już taką postać, jaką będą miały w docelowym, końcowym produkcie. Oczywiście zmiany są nieuniknione, jednak jest to generalna zasada odróżniająca

modele przyrostowe od iteracyjnych. Pojedynczy przyrost może być prowadzony w modelu sekwencyjnym lub iteracyjnym.

Wytwarzanie iteracyjne polega z kolei na specyfikowaniu, projektowaniu, budowaniu i testowaniu wspólnie grup funkcjonalności w serii cykli, często o ściśle określonym czasie trwania. Iteracje mogą zawierać zmiany funkcjonalności wytworzonych we wcześniejszych iteracjach, wspólnie ze zmianami w zakresie projektu. Każda iteracja dostarcza działające oprogramowanie, które stanowi rosnący podzbiór całkowitego zbioru funkcjonalności aż do momentu, w którym końcowa wersja oprogramowania zostaje dostarczona lub wytwarzanie zostaje zatrzymane.

Górna część rysunku 2.1 prezentuje podejście iteracyjne. Budowę naszego obrazu realizujemy niejako w szkicach, z których każdy obejmuje całość końcowego pomysłu, jednak całość ta przedstawiana jest na różnych poziomach szczegółowości: najpierw mamy jakiś początkowy rys, zarys koncepcji obrazu. W kolejnych iteracjach dopracowujemy tę koncepcję, „uwyróżniając” coraz więcej szczegółów. Oczywiście podejście to nie musi dotyczyć całego produktu, lecz np. jakiejś wyróżnionej grupy funkcjonalności, jak to opisano wcześniej.

W niektórych modelach iteracyjnych i przyrostowych zakłada się, że każda iteracja lub przyrost kończy się działającym produktem. Oznacza to, że w każdej iteracji (przyroście) zarówno testowanie statyczne, jak i dynamiczne może być przeprowadzane na wszystkich poziomach testów. Częste dostarczanie działających fragmentów oprogramowania wymaga szybkiej informacji zwrotnej i szeroko zakrojonych testów regresji.

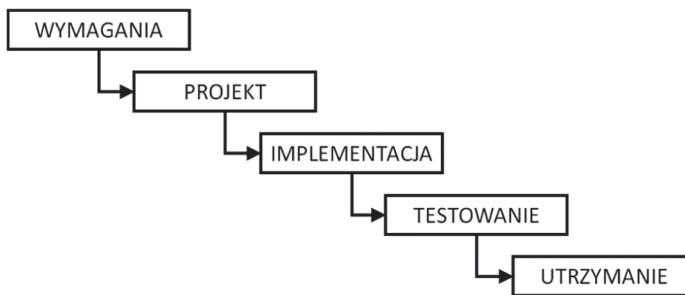
Zwinne metody wytwarzania oprogramowania zakładają, że zmiana może nastąpić w dowolnym momencie projektu. Takie założenie powoduje, że w metodach tych preferuje się lekką formę dokumentacji oraz szeroką automatyzację testów, aby w szczególności testy regresji były łatwiejsze do przeprowadzenia. Ponadto duża część testów manualnych przeprowadzana jest z wykorzystaniem technik testowania opartych na doświadczeniu, które nie wymagają długotrwałego, dokładnego, uprzedniego planowania (zob. podrozdział 4.4).

Omówimy teraz kilka najważniejszych modeli sekwencyjnych i iteracyjnych.

MODELE SEKWENCYJNE

Model kaskadowy

W modelu kaskadowym (ang. *waterfall*, patrz rysunek 2.2) czynności związane z wytwarzaniem oprogramowania (takie jak: analiza wymagań, projektowanie, tworzenie kodu czy testowanie) wykonuje się jedna po drugiej. Zgodnie z założeniami tego modelu czynności testowe następują dopiero, gdy wszystkie inne czynności wytwórcze zostaną ukończone. Jest to z punktu widzenia testera sytuacja problematyczna. W rozdziale pierwszym zauważliśmy, że testowanie powinno rozpoczęć się tak szybko, jak to tylko możliwe, ponieważ im później to nastąpi, tym droższa będzie naprawa znalezionych defektów.

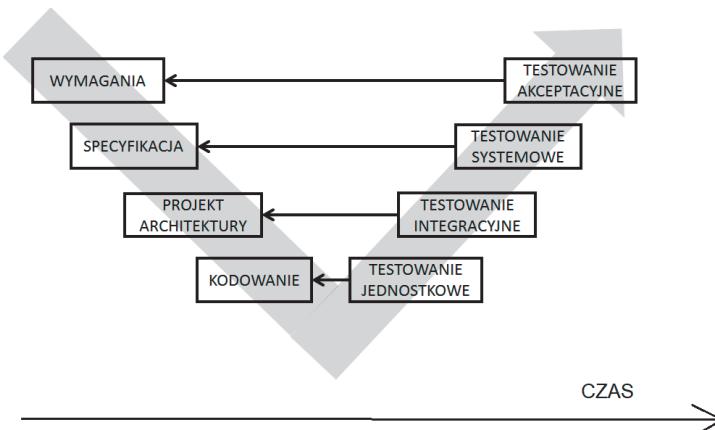


RYSUNEK 2.2. Model kaskadowy (wodospadowy)

Model kaskadowy uniemożliwia nam wczesne rozpoczęcie testowania. Jaki zatem jest zysk ze stosowania tego podejścia? Otóż sprawdza się ono w projektach, w których wymagania są dobrze znane i dobrze opracowane, a zespół może być pewien (lub prawie pewien), że nie zmienią się one w trakcie czynności wytwórczych. Model kaskadowy jest dobrym rozwiązaniem w projektach typowych, „seryjnych” (np. projektach wdrożeniowych, wymagających bardziej skomplikowanej konfiguracji) czy w takich, w których występują bardzo dobrze zidentyfikowane wymagania i ryzyko ich zmiany jest niewielkie lub wręcz go nie ma (np. wymagania wynikające wprost z przepisów prawa).

Model V

Model V (ang. *V model*) jest zmodyfikowanym podejściem kaskadowym. W tym modelu (patrz rysunek 2.3) zakłada się integrację procesu testowania z całym procesem wytwarzania oprogramowania, a tym samym wprowadza w życie zasadę wczesnego testowania. Ponadto model V obejmuje poziomy testowania powiązane z poszczególnymi odpowiadającymi im fazami wytwarzania oprogramowania, co dodatkowo sprzyja wczesnemu testowaniu (patrz podrozdział 2.2). W tym modelu wykonanie testów powiązanych ze wszystkimi poziomami testów następuje sekwencyjnie, jednak w niektórych przypadkach może następować nachodzenie faz na siebie.



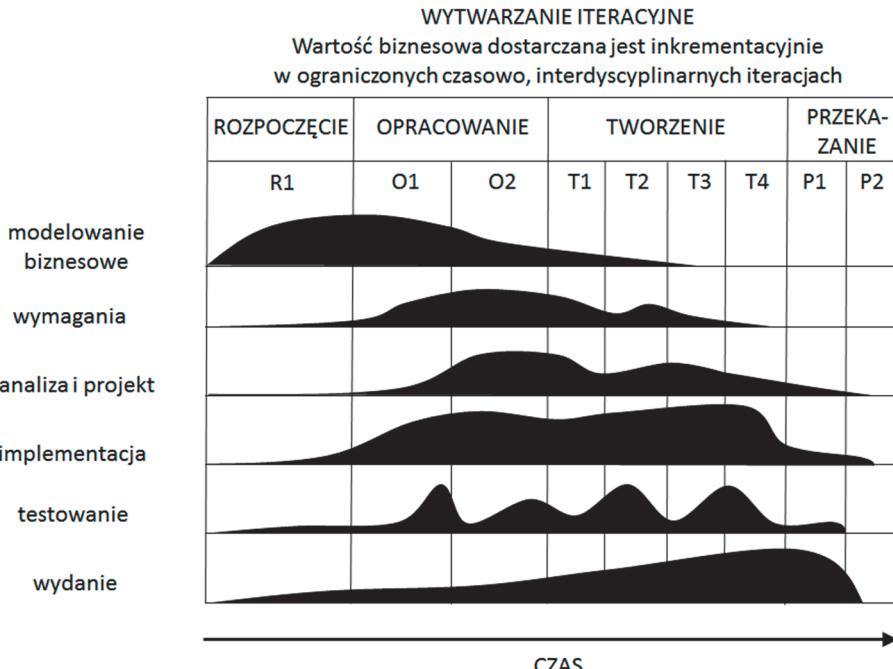
RYSUNEK 2.3. Model V

Model V jest więc próbą zaradzenia problemom, jakie stwarza model kaskadowy w kontekście testowania i zapewniania jakości. Zwraca on uwagę na czynności testowe, które następują w każdej fazie — nie tylko w fazie testowania (prawa gałąź modelu), ale także w fazach wytwórczych (lewa gałąź modelu). W tych ostatnich nie zawsze tester jest w stanie wykonać testy (np. w fazie projektowania nie ma jeszcze testowalnego produktu, który można uruchomić), ale zawsze może te testy zaplanować. W pewnej wersji modelu V, tzw. modelu W, proponuje się, aby w fazach wytwórczych czynności testerskie dotyczyły również testowania statycznego, tzn. aby prowadzić przeglądy produktów prac (np. przegląd wymagań, przegląd architektury, przegląd kodu).

MODELE ITERACYJNE I PRZYROSTOWE

Model UP (ang. Unified Process)

Model UP (patrz rysunek 2.4) jest modelem iteracyjno-przyrostowym. W zasadzie UP nie jest pojedynczym procesem, lecz elastyczną strukturą procesów (ang. *process framework*), w ramach której definiuje się poszczególne procesy, takie jak modelowanie biznesowe, wymagania, analiza i projekt, implementacja, testowanie czy wydanie. Procesy te można wybierać i dostosowywać do potrzeb projektu. Poszczególne迭代 trwają zwykle stosunkowo długo (np. dwa lub trzy miesiące), a przyrostowe części systemu są odpowiednio duże, obejmują na przykład dwie lub trzy grupy powiązanych funkcjonalności.

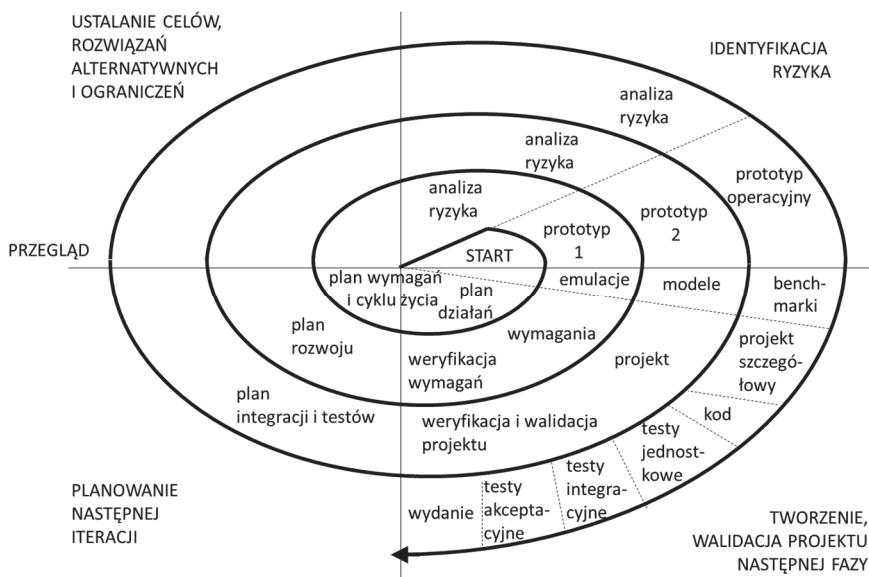


RYSUNEK 2.4. Unified Process

Model spiralny Boehma

Model spiralny (ang. *spiral model*) to podejście, w którym tworzy się eksperymentalne elementy przyrostowe, które następnie mogą zostać gruntownie przebudowane, a nawet porzucone na dalszych etapach wytwarzania oprogramowania (patrz rysunek 2.5). Komponenty lub systemy wykorzystujące powyższe modele często zawierają nachodzące na siebie i powtarzane w cyklu wytwarzania oprogramowania poziomy testów. W idealnym przypadku każda funkcjonalność jest testowana na wielu poziomach, zbliżając się do finalnego wydania. W niektórych sytuacjach zespoły korzystają z ciągłego dostarczania (ang. *continuous delivery*) lub ciągłego wdrażania oprogramowania, które to podejścia szeroko wykorzystują automatyzację na wielu poziomach testowania jako część potoku dostarczania oprogramowania. Ponadto wiele tego typu modeli uwzględnia koncepcję samoorganizujących się zespołów, która może zmienić sposób organizacji pracy w związku z testowaniem, oraz relacje między testerami a programistami.

Model spiralny został zaproponowany przez Barry'ego Boehma i — jak twierdzi sam Boehm — jest w zasadzie „modelem modeli”, ponieważ można go tak skonfigurować, aby otrzymać praktycznie każdy inny model cyklu wytwarzania (np. ograniczając go do ostatniej ćwiartki zewnętrznej spirali, otrzymujemy model kaskadowy). Jego charakterystyczną i jedną z najważniejszych cech jest analiza ryzyka przeprowadzana przed rozpoczęciem każdego kolejnego cyklu produkcyjnego.

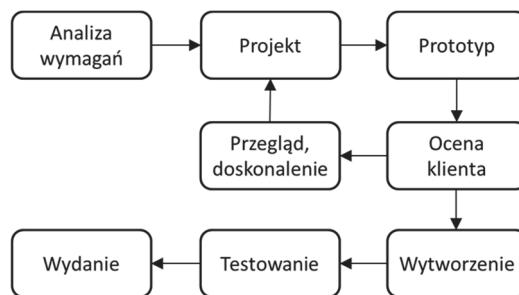


RYSUNEK 2.5. Model spiralny Boehma

Model prototypowania

Model prototypowania (ang. *prototyping*) często stosowany jest tam, gdzie cele projektowe nie są ściśle zdefiniowane czy określone z góry. Na przykład klient definiuje zestaw ogólnych celów dla oprogramowania, ale nie określa szczegółowych wymagań

funkcjonalnych; programista może nie mieć pewności co do skuteczności implementowanego algorytmu lub formy interakcji programu z użytkownikiem itp. W tego typu sytuacjach paradygmat prototypowania może oferować najlepsze podejście.



RYSUNEK 2.6. Model prototypowania

Prototypowanie pomaga interesariuszom lepiej zrozumieć, co ma być zbudowane, gdy wymagania są rozmyte. Proces (patrz rysunek 2.6) rozpoczyna się od komunikacji, w ramach której definiuje się ogólne cele (wymagania) dla oprogramowania. Po fazie planowania następuje szereg szybkich iteracji prototypowania. W każdej z nich zespół koncentruje się na reprezentacji tych aspektów oprogramowania, które będą widoczne dla użytkowników końcowych — kończy się to budową działającego prototypu. Prototyp jest wdrażany i oceniany przez interesariuszy, którzy przekazują informacje zwrotne. Informacje te są wykorzystywane do dalszego doskonalenia wymagań.

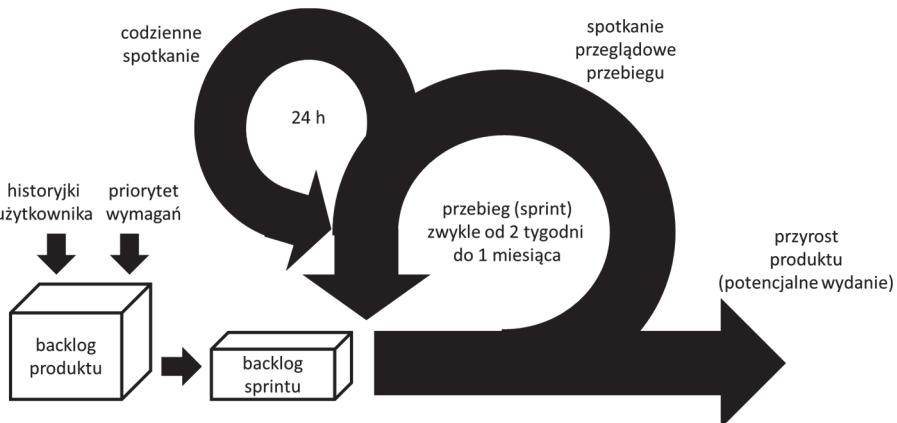
W idealnym przypadku prototyp służy jako mechanizm identyfikacji wymagań dotyczących oprogramowania. Jeśli ma być zbudowany działający prototyp, można wykorzystać istniejące fragmenty programu lub zastosować narzędzia, które umożliwiają szybkie generowanie programów roboczych (np. „dynamiczne” makiety graficznego interfejsu użytkownika).

METODYKI WYTÓWCZE I PRAKTYKI ZWINNE

Scrum

Scrum (patrz rysunek 2.7) jest obecnie jednym z najczęściej spotykanych modeli cyklu wytwarzania oprogramowania¹. Metodyka Scrum dzieli wytwarzanie oprogramowania na krótkie iteracje o tej samej długości (zazwyczaj 2 – 4 tygodnie), a przyrostowe części systemu są odpowiednio małe, czyli obejmują na przykład kilka udoskonaleń bądź nowych funkcjonalności.

¹ Często można spotkać się z opinią, że Scrum to nie metodyka, lecz framework. Czasami również stosuje się słowo „metodologia” zamiast „metodyka”. Nieporozumienia te wynikają z powszechnego, niestety, nieroźróżniania znaczenia słów: „metoda”, „metodyka”, „metodologia”, „struktura” (ang. *framework*). Metodyka (lub metoda) to zbiór zasad dotyczących wykonywania jakiejś pracy. Struktura zaś to układ i wzajemne relacje elementów stanowiących całość. Metodologia z kolei to nauka o metodach. Scrum jest zatem metodyką (lub metodą) i to wbrew pozorom bardzo preskryptywną (tzn. narzucającą określony sposób postępowania) co do wielu elementów tego modelu, np. długości sprintów czy czasu trwania tzw. codziennych spotkań (ang. *stand-up meeting*).



RYSUNEK 2.7. Scrum

W modelu takim jak Scrum testowanie staje się wyzwaniem ze względu na krótkie iteracje i częste zmiany. Dlatego zamiast dokładnego i długotrwałego procesu projektowania przypadków testowych częściej spotyka się tu praktyki takie jak testowanie eksploracyjne czy nacisk na automatyzowanie możliwie dużej liczby testów — dotyczy to zwłaszcza testów regresji, których liczba zazwyczaj szybko rośnie z każdą kolejną iteracją.

Scrum jest mieszanką systemu push oraz systemu pull, ponieważ przed każdą iteracją zespół wybiera („push”) z rejestru produktu do rejestru sprintu z góry określony zestaw zadań do wykonania w tej iteracji, a w ramach sprintu każdy członek zespołu zaczyna pracę nad kolejnym zadaniem, gdy poprzednie zakończy („pull”).

Kanban

Kanban umożliwia dostarczenie jednego udoskonalenia lub jednej funkcjonalności naraz (natychmiast po przygotowaniu) bądź zgrupowanie większej liczby funkcjonalności w celu równoczesnego przekazania do środowiska produkcyjnego. Oryginalnie metoda ta znalazła zastosowanie w firmach produkcyjnych takich jak Toyota, gdzie produkty wytwarzane są seryjnie. Można ją jednak również dostosować do projektów informatycznych. Kanban wykorzystuje tzw. tablicę Kanban do wizualizacji, planowania i nadzoru procesu wytwórczego (patrz rysunek 2.8).

Metoda Kanban oparta jest na tzw. kartach wyrobów i tak w jej ramach organizowany jest proces wytwórczy, aby każde stanowisko produkcyjne wytwarzalo w danej chwili dokładnie tyle, ile jest w danym momencie potrzebne. Ponadto każde stanowisko ma odgórnie ustalony limit pracy, jaką może wykonać w ramach określonej jednostki czasu (tzw. WIP — ang. *Work in Process Limit*). Taka organizacja pracy umożliwia sprawne prowadzenie produkcji, tak by nie powstawały żadne niepotrzebne zapasy, a „masa” zadań do wykonania „przepływała” równomiernie przez system produkcyjny, tym samym optymalizując zużycie zasobów oraz czas produkcji.

Kanban jest systemem typu „pull”, ponieważ pracownik na danym stanowisku „zaciąga” z poprzedniego stanowiska produkt, który na tamtym stanowisku został już ukończony i jest gotowy do przekazania dalej.

ANALIZA WYMAGAŃ	IMPLEMENTACJA		TESTOWANIE		GOTOWE
	Oczekujące (3)	W trakcie (3 / 3)	Oczekujące (2)	W trakcie (2 / 2)	
+dodaj zadanie	+dodaj zadanie	+dodaj zadanie	+dodaj zadanie	+dodaj zadanie	+dodaj zadanie
Przetestować nowe rozwiązanie do kompresji obrazu	Uaktualnić plik z defektami	Nowi użytkownicy mogą odszukać swoje pliki <i>właściciel: Michał</i>	Jako użytkownik chcę mieć dostęp do ostatnich zmian		Kompresja obrazu metodą LZW
			Funkcjonalność 3F - retest	<i>właściciel: Marek</i>	
	Poprawa defektów w wymaganiach 5F-7F		Nadanie nowych uprawnień użytkownikowi klasy I przez admina	Retesty funkcjonalności 1F, 4F, 8F <i>właściciel: Kasia</i>	
Dostarczyć kod promocyjny do sprzedawców		Edycja konta admina <i>właściciel: Magda</i>			Logowanie z kontrolą haseł
	Zadzwonić do Janka 226662137				

RYSUNEK 2.8. Przykładowa tablica Kanban (za: kanbantool.com/kanban-board-examples)

Zasady wyboru modelu cyklu wytwarzania

Modele cyklu wytwarzania oprogramowania należy dobierać i dopasowywać do kontekstu wynikającego z charakterystyki projektu i produktu. W związku z tym przy wyborze i dostosowaniu odpowiedniego modelu należy wziąć pod uwagę: cel projektu, typ wytwarzanego produktu, priorytety biznesowe (takie jak czas wprowadzenia produktu na rynek) i zidentyfikowane ryzyka produktowe i projektowe. Na przykład wytwarzanie i testowanie wewnętrznego systemu administracyjnego o niewielkim znaczeniu powinno przebiegać inaczej niż wytwarzanie i testowanie systemu krytycznego ze względów bezpieczeństwa, takiego jak układ sterowania hamulcami w samochodzie. Innym przykładem może być to, że w niektórych przypadkach kwestie organizacyjne i kulturowe mogą utrudniać komunikację między członkami zespołu, co może skutkować spowolnieniem wytwarzania oprogramowania w modelu iteracyjnym.

W przemyśle IT istnieje wiele modeli cyklu wytwarzania i często wybór tego właściwego może być trudny. Należy jednak pamiętać o wspomnianych powyżej kryteriach wyboru. Trzeba też mieć świadomość tego, że model opisujący metodykę wytwarzania oprogramowania nie jest nienaruszalną świętością. Jest jedynie propozycją, która podpowiada nam sposób organizacji procesu wytwarzania oprogramowania. Model taki można, a nawet należy dostosowywać do potrzeb i wymagań organizacji, w której wytwarzane jest oprogramowanie.

Z punktu widzenia testera najistotniejszy w ramach cyklu wytwarzania będzie oczywiście proces testowania (a ogólniej kontroli jakości). Zależnie od kontekstu projektu może być konieczne połączenie lub przeorganizowanie niektórych poziomów testów i/lub czynności testowych. Na przykład w przypadku integracji oprogramowania do powszechniej sprzedaży (COTS, od ang. *Commercial off-the-Shelf*) z większym

systemem nabywca może wykonywać testy współdziałania na poziomie testowania integracji systemów (np. w zakresie integracji z infrastrukturą i innymi systemami) oraz na poziomie testowania akceptacyjnego (testowanie funkcjonalne i niefunkcjonalne wraz z testowaniem akceptacyjnym przez użytkownika i produkcyjnymi testami akceptacyjnymi). Poziomy i typy testów opisane są dokładnie w podrozdziale 2.2.

Ponadto różne modele cyklu wytwarzania oprogramowania można łączyć. Przykładem może być zastosowanie modelu V do wytwarzania i testowania back-endowej części systemu oraz modelu zwanego do wytwarzania i testowania front-endu (interfejsu użytkownika). Innym wariantem jest zastosowanie modelu prototypowania na wczesnym etapie projektu, a następnie zastąpienie go modelem przyrostowym po zakończeniu fazy eksperymentalnej.

W przypadku systemów związanych z Internetem rzeczy (IoT, od ang. *Internet of Things*), które składają się z wielu różnych obiektów — takich jak: urządzenia, produkty i usługi — w odniesieniu do poszczególnych obiektów stosuje się zwykle oddzielne modele cyklu wytwarzania. Stanowi to duże wyzwanie zwłaszcza w zakresie wytwarzania poszczególnych wersji systemów IoT. Ponadto w przypadku powyższych obiektów większy nacisk kładzie się na późniejsze etapy cyklu wytwarzania oprogramowania, już po wprowadzeniu obiektów na produkcję (np. na fazy produkcyjną, aktualizacji i wycofania z użytku).

Proces wyboru właściwego modelu cyklu wytwarzania może przebiegać w następujący sposób. W pierwszym kroku definiujemy kryteria, według których będziemy oceniać przydatność poszczególnych modeli w naszym projekcie. Przykładowe kryteria, które można wziąć pod uwagę, to:

- wielkość i doświadczenie zespołu;
- wielkość, typ, poziom skomplikowania projektu;
- relacje z klientem, stabilność wymagań, częstość ich zmian;
- rozproszenie geograficzne zespołu;
- zgodność modelu ze strategią biznesową/organizacyjną firmy.

W kroku drugim porównujemy potencjalne modele cyklu wytwarzania oprogramowania, uwzględniając tak ich zalety, jak i wady. Należy pamiętać, że nie ma rozwiązań idealnych ani uniwersalnych. Każdy model cyklu wytwarzania oprogramowania będzie miał jakieś strony pozytywne, ale też będzie stawał nam pewne wyzwania. Na przykład wykorzystywanie modeli zwinnych, zwłaszcza w zespole stosującym dotychczas modele sekwencyjne, wymaga zmiany myślenia członków zespołu, co często jest bardzo trudnym zadaniem.

W kroku trzecim oceniamy potrzeby naszej organizacji. Model cyklu wytwarzania oprogramowania powinien odzwierciedlać naturę i sposób działania firmy bądź zespołu, w którym pracujemy. Produkcja oprogramowania dla przemysłu lotniczego, medycznego czy wojska będzie wymagała zupełnie innych procedur i podejść niż na przykład tworzenie gry na urządzenie mobilne we właśnie powstałym start-upie.

W ostatnim kroku stosujemy wybrane wcześniej kryteria w kontekście potrzeb naszej organizacji.

2.1.2. Model cyklu wytwarzania oprogramowania a dobre praktyki testowania

Znajomość modeli cyklu wytwarzania oprogramowania jest istotna z punktu widzenia testera, ponieważ to, jaki model został przyjęty w procesie wytwórczym, rzutuje na to, kiedy i w jaki sposób wykonywane będą czynności testerskie. W podrozdziale 2.1.1 opisaliśmy szereg przykładowych modeli cyklu wytwarzania oprogramowania. Niezależnie jednak od tego, w jakim modelu przyjdzie testerowi pracować, sylabus wymienia kilka zasad dobrego testowania, których powinno się przestrzegać niezależnie od przyjętego modelu:

- **Dla każdej czynności związanej z wytwarzaniem oprogramowania istnieje odpowiadająca jej czynność testowa.** Zasada ta zwraca uwagę na „totalność” czynności testerskich: tester powinien weryfikować każdy artefakt, który został wytworzony podczas tworzenia oprogramowania.
- **Każdemu poziomowi testów odpowiadają cele testowania odpowiednie do fazy lub grupy czynności w ramach cyklu wytwarzania oprogramowania.** Zasada ta zwraca uwagę na to, że cele testowania mogą istotnie różnić się od siebie (patrz podrozdział 1.1 sylabusa); na jednym poziomie będzie nas interesować głównie wykrycie jak największej liczby defektów, a na innym — raczej walidacja (potwierdzenie) tego, że system spełnia wymogi i potrzeby klienta.
- **Analizę i projektowanie testów na potrzeby danego poziomu testów należy rozpocząć już podczas wykonywania odpowiadającej mu czynności związanej z wytwarzaniem oprogramowania.** Reguła ta związana jest z zasadą wczesnego testowania (podrozdział 1.3), która zakłada, że czynności testowe rozpoczynają się możliwie najwcześniej, aby jak najszybciej wykryć problemy, których usunięcie we wczesnych fazach jest tanie, a w późniejszych — często bardzo drogie. Należy zwrócić uwagę, że czasami defekty znajduje się nie poprzez fizyczne testowanie aplikacji, ale poprzez czynności projektowania testów (patrz rozdział 4.).
- **Testerzy powinni uczestniczyć w przeglądach produktów pracy (np. wymagań, projektu czy historyjek użytkownika) natychmiast po udostępnieniu wersji roboczych odpowiednich dokumentów.** Zasada ta uwypukla rolę testera jako specjalisty w zakresie jakości oprogramowania i jest przykładem realizacji zasady „shift-left” (patrz punkt 2.1.5). Testerzy bardzo często w trakcie tego typu dyskusji są w stanie znaleźć mnóstwo niejasności, błędów czy niedopowiedzeń, które — niezauważone — mogłyby skutkować poważnymi problemami w przyszłości.

Przykład. Organizacja na zamówienie Ministerstwa Sprawiedliwości wytwarza produkt SędzioLotek, czyli system do losowego przydzielania spraw sędziom. W tabeli 2.1 zebrano przykłady czynności wytwórczych i odpowiadających im czynności testowych.

TABELA 2.1. Przykładowe czynności wytwarzające i odpowiadające im czynności testerskie

FAZA	CZYNNOŚĆ WYTWÓRCZA	CZYNNOŚĆ TESTERSKA
Wymagania	Zbieranie wymagań systemu, tworzenie dokumentu wymagań	Inspekcja wymagań pod kątem testowalności oraz ich zgodności z wymaganiami biznesowymi, np. wymogami ustawy o sądach powszechnych
Projektowanie	Projekt bazy danych	Inspekcja projektu bazy danych pod kątem jej zgodności z wymaganiami systemu
Implementacja	Implementacja systemu front-endowego (webowego)	Testy użyteczności, testy integracji systemu z bazą danych, testy systemowe
Wdrożenie	Wdrożenie systemu w ministerstwie	Testy akceptacyjne (beta) systemu w środowisku docelowym przeprowadzone przez prezesów sądów

2.1.3. Testowanie jako czynnik określający sposób wytwarzania oprogramowania

Wytwarzanie sterowane testami (TDD — ang. *Test-Driven Development*), wytwarzanie sterowane testami akceptacyjnymi (ATDD — ang. *Acceptance Test-Driven Development*) oraz wytwarzanie sterowane zachowaniem (BDD — ang. *Behavior-Driven Development*) to trzy popularne metody wytwarzania oprogramowania oparte na podejściu „najpierw test” (ang. *test first approach*). Podejście to zakłada, że *zanim* napisany zostanie kod źródłowy realizujący określoną funkcję, tworzony jest test dla tej funkcji. Jest ono przykładem tego, jak testowanie może stanowić czynnik kierujący *wytwarzaniem* oprogramowania.

Podejście „najpierw test” wywodzi się z metodyki programowania ekstremalnego (XP — ang. *eXtreme Programming*) i stanowi jedną podstawowych praktyk zwanego wytwarzania oprogramowania. Podejście to może wydawać się sprzeczne z intuicją (zazwyczaj najpierw coś tworzymy, a potem to sprawdzamy), ale przynosi pewne istotne korzyści [Linz 2014]:

- Testowanie zastępuje metodę prób i błędów — zamiast np. sprawdzać metodą prób i błędów, czy zaimplementowana funkcja działa poprawnie, tworzy się zestaw sensownych przypadków testowych ze ścisłe określonymi wartościami wejścia i wyjścia, a następnie pisze się kod, który ma te testy przejść.
- Przypadki testowe dostarczają obiektywnej informacji zwrotnej o postępie — każdy zdany test jest obiektywnym dowodem na to, że projekt posuwa się naprzód.
- Przypadki testowe zastępują specyfikację — ponieważ testy pisane są, zanim zostanie stworzony kod, nie są jedynie zbiorem kontrolnych procedur, ale także definiują przykładowe zachowanie obiektu testów, stając się przykładem „żywej dokumentacji” [Adzic 2011]; zmiana wymagań wymaga zmiany testów, a więc będzie również wymuszać zmianę dokumentacji, czyniąc ją aktualną w każdym momencie projektu.

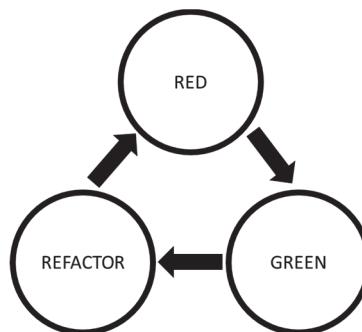
- Podejście „najpierw test” polepsza jakość publicznych interfejsów (API) — testy modułowe (jednostkowe) wykorzystują metody publiczne testowanych klas, a skoro testy są pisane, zanim stworzony zostanie kod, testy definiują nazwy metod publicznych testowanej klasy, ich parametry oraz przykłady użycia metod. Projekt testów staje się praktycznie definicją interfejsu.
- Podejście „najpierw test” polepsza testowalność tworzonego kodu. Gdy testy są już zaimplementowane, programista musi następnie napisać kod, który je zda. To oznacza, że kod musi odwoływać się do interfejsów wymaganych przez testy. Zamiast więc pisać testy, które sprawdzają istniejący kod, programista musi napisać kod, który jest „zgodny” z napisanymi wcześniej testami. Jednocześnie, ponieważ kod pisany jest w celu zdania określonych testów, programista lepiej kontroluje i łatwiej osiąga wyższy poziom pokrycia kodu testami.

Zasadnicza różnica pomiędzy TDD, BDD i ADD polega na tym, że TDD koncentruje się na testowaniu komponentów, BDD — na testowaniu zachowania systemu, natomiast ATDD — na akceptacji systemu. Można więc powiedzieć, że TDD działa na poziomie testów modułowych i integracji modułów, BDD — na poziomie testów systemowych i integracji systemów, a ATDD — na poziomie testów akceptacyjnych (patrz punkt 2.2.1). Wynikowe przypadki testowe powinny nadawać się do wykorzystania w zautomatyzowanych testach regresyjnych. Każde z tych podejść realizuje zasadę testowania „Wczesne testowanie oszczędza czas i pieniądze” (patrz podrozdział 1.3) i podąża za podejściem „shift left” (patrz punkt 2.1.5), ponieważ testy są definiowane przed napisaniem kodu.

Wytwarzanie sterowane testami (TDD)

TDD wykorzystuje zautomatyzowane przypadki testów komponentów do kierowania rozwojem kodu i jest zwykle wykonywane przez programistę. Proces TDD przebiega następująco:

- Programista tworzy nowy przypadek testowy, odpowiadający określonymu wymaganiu wobec kodu.
- Wszystkie istniejące testy są uruchamiane. Nowy test powinien być niezdany, ponieważ nie ma jeszcze odpowiadającego mu kodu (faza „RED” na rysunku 2.9). Pierwsze uruchomienie nowego testu ma na celu sprawdzenie, czy test się w ogóle kompliluje. Z kolei jeśli test ten zostanie zdany przy pierwszym uruchomieniu, oznacza to najprawdopodobniej, że jest jakiś problem z samym testem. Uruchomione pozostałe testy pełnią rolę testów regresji (patrz punkt 2.2.3).
- Programista pisze minimalną ilość kodu wystarczającą do przejścia nowego testu. Wszystkie wcześniej napisane testy również muszą przejść (faza „GREEN” na rysunku 2.9).
- Programista refaktoryzuje kod (jeśli to konieczne), aby utrzymać jego wysoką jakość, ponieważ TDD odbywa się w wielu krótkich cyklach „test-kod do testu”, co może wpływać na pogorszenie jakości samego kodu (faza „RE-FACTOR” na rysunku 2.9). Po refaktoryzacji programista ponownie uruchamia wszystkie testy, aby upewnić się, że refaktoryzacja niczego nie zepsuła.
- Powyższe kroki są powtarzane, dopóki wciąż pozostał jakiś kod do napisania.



RYSUNEK 2.9. Proces wytwarzania sterowanego testami

W podejściu TDD zwykle używa się framework'a testowego typu xUnit do wspierania automatycznych testów. Pojedynczy cykl „test-kod do testu” jest często bardzo krótki i może trwać nawet mniej niż minutę.

Stosując podejście TDD, programista zyskuje pewną niezależność testowania. Nie ma poczucia emocjonalnej więzi z kodem, bo go jeszcze nie stworzył. Dzięki temu, poprzez sam fakt projektowania testów, deweloper może przetestować bądź ustalić założenia implementacyjne dla danego modułu. Jego testy będą w związku z tym silniejsze niż w przypadku, gdyby je pisał po zaimplementowaniu modułu.

Wytwarzanie sterowane zachowaniem (BDD)

W TDD testy mogą odnosić się nawet do kilku pojedynczych linii kodu. W wytwarzaniu sterowanym zachowaniem testy skupiają się na oczekiwanyim zachowaniu systemu [Chelimsky 2010], a więc operują na nieco wyższym poziomie niż testy modułowe w podejściu TDD. BDD wykorzystuje podejście oparte na współpracy w celu wygenerowania kryteriów akceptacji w prostym języku, zwykle jako część historyjki użytkownika, które to kryteria mogą być zrozumiane przez wszystkich interesariuszy (patrz podrozdział 4.5).

Kryteria akceptacji pisze się zwykle przy użyciu frameworków. Typowym formatem kryteriów akceptacji jest format Given/When/Then (Mając/Kiedy/Wtedy). Framework, na podstawie historyjki użytkownika oraz związanych z nią kryteriów akceptacji, automatycznie generuje kod przypadków testowych i wykonuje je.

Zasada działania frameworków do BDD przedstawiona jest poniżej. Test może mieć postać czytelnego i zrozumiałego dla interesariuszy dokumentu tekstowego napisanego w języku Gherkin, na przykład takiego (test wraz z poniższym kodem jest lekko zmodyfikowanym przykładem ze strony <https://automationrhapsody.com/introduction-to-cucumber-and-bdd-with-examples/>):

```

Feature: wyszukaj hasło w Wikipedi
  Scenario: bezpośrednie wyszukiwanie artykułu
    Given Wprowadź do wyszukania hasło 'testowanie'
    When Kliknij przycisk Szukaj
    Then Pojawia się strona zawierająca frazę 'testowanie'
  
```

Test ten sprawdza, czy jeśli użytkownik wpisze w wyszukiwarce Wikipedii hasło *testowanie*, to w sytuacji, gdy kliknie przycisk *Szukaj*, system zwróci stronę zawierającą tekst *testowanie*. Te kroki przypadku testowego implementowane są fizycznie przez kod korzystający z framework'a BDD. Fragment takiego kodu przedstawiony jest na poniższym listingu. W wierszu z anotacją @Given programista każe wyszukać w powyższym skrypcie gherkinowym frazę „Wprowadź do wyszukania hasło X” w linii zaczynającej się od słowa Given, gdzie wartość X jest automatycznie przypisywana do zmiennej searchTerm, będącej parametrem metody searchFor, stwarzanej z linijką Given. Framework identyfikuje ten fragment poprzez użycie nawiasów w łańcuchu znaków definiującym tzw. wyrażenie regularne, które przypasowuje się do przeszukiwanego tekstu, znajdując żądaną ciąg znaków.

Jeśli ciąg znaków zostanie dopasowany do wyrażenia regularnego, metoda searchFor jest uruchamiana z parametrem X (w naszym przypadku — ciągiem znaków *testowanie*). Wykonuje ona fizycznie to, co trzeba — wyszukuje na stronie WWW pole tekstowe do wyszukiwania tekstu i przesyła do niego wartość zmiennej X (czyli słowo „*testowanie*”), wykorzystując odpowiednie instrukcje biblioteki Selenium WebDriver, czyli narzędzi do automatycznego testowania stron internetowych. Analogicznie, metody stwarzyszone z anotacjami @When i @Then realizują odpowiednie akcje: kliknięcia przycisku *Szukaj* oraz weryfikacji, czy na stronie pojawia się określony napis.

```
package com.automationrhapsody.cucumber.parallel.tests.wikipedia;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

import cucumber.api.java.After;
import cucumber.api.java.Before;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;

import static junit.framework.Assert.assertTrue;
import static junit.framework.TestCase.assertFalse;
import static org.junit.Assert.assertEquals;

public class WikipediaSteps {
    private WebDriver driver;

    @Before
    public void before() {
        driver = new FirefoxDriver();
        driver.navigate().to("http://en.wikipedia.org");
    }

    @After
    public void after() {
        driver.quit();
    }

    @Given("^Wprowadź do wyszukania hasło '(.*?)'$")
    public void searchFor(String searchTerm) {
        WebElement searchField = driver.findElement(By.id("searchInput"));
    }
}
```

```
    searchField.sendKeys(searchTerm);
}

@When("^Kliknij przycisk szukaj$")
public void clickSearchButton() {
    WebElement searchButton = driver.findElement(By.id("searchButton"));
    searchButton.click();
}

@Then("^Pojawia się strona zawierająca frazę '(.*?)'$")
public void assertSingleResult(String expectedResult) {
    WebElement results = driver
        .findElement(By.cssSelector("div#mw-content-text.mw-content-ltr p"));
    assertFalse(results.getText().contains(expectedResult + "may refer to:"));
    assertTrue(results.getText().startsWith(expectedResult));
}
}
```

BDD kieruje się koncepcją „żywej dokumentacji”, ponieważ specyfikacja jest wykonywalna w testach automatycznych. Specyfikacja wtedy zwykle podąża za zasadą „specyfikacji opartej na przykładach” [Adzic 2011], która opisuje wymaganie raczej przez typowe (testowalne) przykłady, zamiast wyjaśniać wszystkie ewentualne potrzeby i wyjątki.

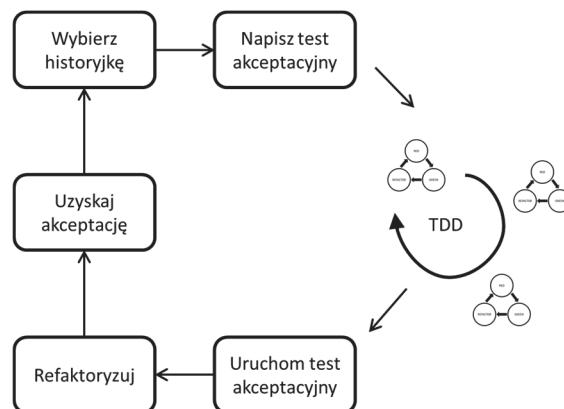
Wytwarzanie sterowane testami akceptacyjnymi (ATDD)

ATDD podąża za tą samą procedurą, która jest zdefiniowana dla TDD i BDD, ale z automatycznymi przypadkami testowymi, które są na odpowiednim poziomie dla testów akceptacyjnych [Gärtner 2011]. Te przypadki testów akceptacyjnych pochodzą z kryteriów akceptacji, które są generowane wspólnie przez programistów, testerów i przedstawicieli biznesu. Kryteria akceptacji są pisane z perspektywy użytkownika i są zazwyczaj w łatwym do zrozumienia formacie (np. Given/When/Then znanym z podejścia BDD). ATDD zostało szczegółowo wyjaśnione w dalszej części podręcznika (patrz podrozdział 4.5).

Schemat wykonania ATDD (patrz rysunek 2.10) jest taki:

- wybierz historyjkę użytkownika;
- napisz test akceptacyjny;
- zaimplementuj historyjkę użytkownika (powstanie kodu);
- uruchom test akceptacyjny;
- przeprowadź refaktoryzację kodu, jeśli to konieczne;
- uzyskaj akceptację historyjki.

W ATDD do wspierania automatycznych testów akceptacyjnych używa się zwykle framework'a do testowania automatycznego (np. FitNesse). ATDD, podobnie jak BDD, kieruje się koncepcją „żywej dokumentacji”.

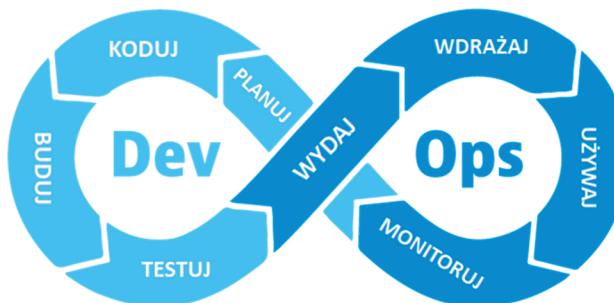


RYSUNEK 2.10. Proces wytwarzania sterowanego testami akceptacyjnymi

2.1.4. Metodyka DevOps a testowanie

Tradycyjnie obszary wytwarzania, testowania i operacji oprogramowania pozostały w oddzielnych silosach, z zupełnie odmiennymi priorytetami. Wytwarzanie skupiało się na tworzeniu i dostarczaniu oprogramowania, testowanie dotyczyło kontroli jakości wytworzzonego oprogramowania, natomiast operacje skupiały się na wdrażaniu i wspieraniu oprogramowania. Separacja tych trzech obszarów często powodowała konflikt pomiędzy nimi.

DevOps to podejście mające na celu stworzenie synergii poprzez współpracę wytwarzania, testowania i operacji w celu osiągnięcia wspólnych celów (rysunek 2.11). DevOps wymaga zmiany kulturowej w organizacji, aby zniwelować różnice pomiędzy wytwarzaniem, testowaniem i operacjami, traktując ich funkcje z równą wartością.

RYSUNEK 2.11. Proces wdrożeń DevOps (za: www.dynatrace.com)

DevOps opiera się na wykorzystaniu iteracyjnego/przyrostowego modelu cyklu wytwarzania oprogramowania wspieranego przez zasady Agile, takie jak autonomia zespołu, szybka informacja zwrotna i zintegrowane łańcuchy narzędzi, a także praktyki techniczne, takich jak ciągła integracja (CI — ang. *Continuous Integration*), ciągłe dostarczanie oprogramowania (ang. *Continuous Delivery*) czy ciągłe wdrażanie oprogramowania (ang. *Continuous Deployment*). Te zasady i praktyki pozwalają zespołowi

szycie budować, testować i wydawać wysokiej jakości kod poprzez tzw. potok wdrożeń DevOps [Kim 2016]. Aby pomóc w osiągnięciu wyższej jakości i szybszych dostaw, w potokach DevOps automatyzuje się czynności manualne. W idealnym przypadku zarządzanie konfiguracją, komplikacja, budowanie oprogramowania, wdrażanie i testowanie są ujęte w jeden, zautomatyzowany, powtarzalny potok wdrożeń.

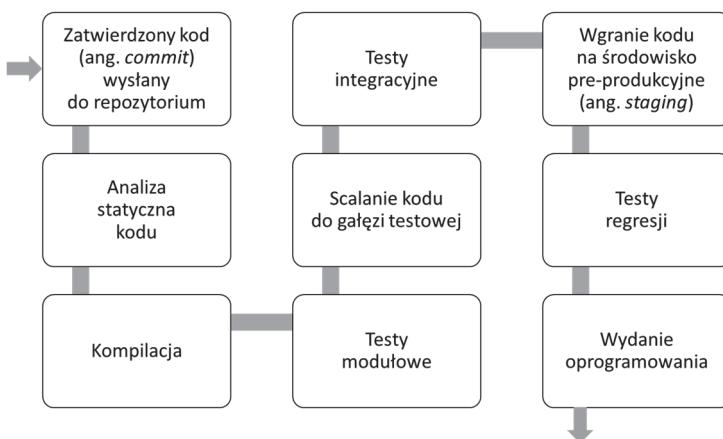
W praktyce efektywne wdrożenie potoku wdrożeń oznacza, że jak najwięcej testów musi być zautomatyzowanych, ponieważ w przeciwnym razie testowanie spowalnia dostarczanie kodu i potencjalnie umożliwia przeniknięcie defektów na produkcję. Idealnie byłoby, gdyby te zautomatyzowane testy zapewniały szybką informację zwrotną o wszelkich znalezionych defektach, a także wspierały ciągłą integrację, ciągłe dostarczanie i ciągłe wdrażanie.

Dzięki procesowi ciągłej integracji nowy kod przesyłany przez programistów do repozytorium kodu jest automatycznie łączony, komplikowany i budowany. W idealnym przypadku przesyłane są również testy modułowe (jednostkowe). Jest to zdecydowanie łatwiejsze, jeśli stosowane jest podejście TDD. Automatyczne testy modułowe są uruchamiane, aby upewnić się, że przesyłany kod przejdzie je. Dodatkowo na tym etapie może zostać przeprowadzona analiza statyczna kodu. Informacja zwrotna dla dewelopera jest praktycznie natychmiastowa, zwłaszcza w sytuacjach, jeśli kod nie skompiluje się, nie przejdzie testów dynamicznych lub zostaną wykryte anomalie przez narzędzie analizy statycznej. W zależności od dostępności środowisk testowych i czasu potrzebnego na przeprowadzenie testów może być również możliwe przeprowadzenie na tym etapie modułowych testów integracyjnych. Te testy będą mieszanką nowych testów dla nowej funkcjonalności i testów regresji, aby zapewnić, że istniejąca funkcjonalność nadal działa. Główną zaletą ciągłej integracji jest to, że zapewnia programistom szybką informację zwrotną, jeśli ich kod jest wadliwy, ale jeśli informacja zwrotna dociera do nich ze zbyt dużym opóźnieniem, to prawdopodobnie będą kontynuować rozwój w oparciu o wadliwy kod, co nie jest efektywne.

Ciągłe dostarczanie oprogramowania może być uważane za rozszerzenie ciągłej integracji i obejmuje kolejny poziom testów przeprowadzanych na środowisku testowym, które jest reprezentatywne dla środowiska produkcyjnego. Modułowe testy integracyjne, testy systemowe i niefunkcjonalne (z których wiele to testy regresji) mogą być uruchamiane jako część procesu ciągłego dostarczania. Jeśli wszystkie te testy przejdą, kod będzie uważany za gotowy do wdrożenia, ale decyzja o wdrożeniu jest podejmowana przez zespół DevOps. Zespół ten może wdrożyć kod automatycznie lub manualnie.

Ciągłe wdrażanie można uznać za rozszerzenie ciągłego dostarczania. Od tego ostatniego różni się tym, że zarówno decyzja o wdrożeniu, jak i samo wdrożenie są zautomatyzowane. Przy ciągłym wdrażaniu zespół DevOps stara się osiągnąć jak największą pewność, że automatyczne testy wykryją wszelkie defekty, które nie powinny przenikać do środowiska produkcyjnego. Organizacje zazwyczaj przechodzą na ciągłe wdrażanie z ciągłego dostarczania, gdy uzyskają wystarczające zaufanie do zautomatyzowanego potoku wdrożeń. Często stosowana przez zespoły DevOps jest również praktyka infrastruktury jako kodu (IaC — ang. *Infrastructure as Code*). Wspiera ona zautomatyzowaną konfigurację środowisk testowych i produkcyjnych w celu wsparcia ciągłego dostarczania i ciągłego wdrażania.

Przykładowy potok wdrożeń przedstawiony jest na rysunku 2.12.



RYSUNEK 2.12. Przykładowy potok wdrożeń

Po wysłaniu przez programistę kodu do repozytorium uruchamiany jest automatyczny proces, który — jeśli po drodze nie nastąpią żadne problemy — prowadzi do automatycznego wydania nowej wersji oprogramowania. Po przesłaniu kodu poddawany jest on analizie statycznej (np. sprawdzane jest, czy kod jest poprawnie sformatowany, czy nazewnictwo zmiennych jest zgodne z przyjętymi w organizacji regułami, czy kod ma odpowiednio niską złożoność cyklomatyczną² itp.). Następnie kod jest komplikowany i uruchamiane są dla niego testy jednostkowe. Po ich pomyślnym przejściu nowy kod jest scalany z już istniejącym i można przeprowadzić automatyczne modułowe testy integracyjne. Po ich pomyślnym przejściu kod wdrażany jest na środowisko pre-produkcyjne (podobne lub identyczne z docelowym środowiskiem klienta) i przeprowadzane są testy regresji, aby sprawdzić, czy nowy kod nie spowodował problemów w innych miejscach oprogramowania. Po pomyślnym przejściu testów regresji następuje wydanie kodu do klienta.

Jeśli w którymkolwiek momencie tego procesu coś pójdzie nie tak (np. nastąpi negatywny wynik analizy statycznej, błąd komplikacji, niezaliczony test), proces jest natychmiast zatrzymywany, a programista otrzymuje odpowiednią informację zwrotną. Może wtedy poprawić kod i wysłać go ponownie do repozytorium, uruchamiając jeszcze raz potok wdrożeń.

Bardzo dużą zaletą automatycznego potoku wdrożeń, wspieranego przez proces zarządzania (wersjonowania) kodem jest to, że w każdym momencie cyklu wytwarzania kod jest komplikowany, uruchamialny i gotowy do wydania. Każda zmiana w kodzie powoduje automatyczne uruchomienie potoku wdrożeń, a programiści otrzymują natychmiast informację zwrotną na temat poziomu jakości kodu. W przypadku wystąpienia jakiegokolwiek problemu kod wraca do (działającej) wersji sprzed zmian.

² Złożoność cyklomatyczna (ang. *cyclomatic complexity*), inaczej „złożoność McCabe'a”, to metryka oprogramowania opracowana przez Thomasa J. McCabe'a w 1976 roku, używana do pomiaru stopnia skomplikowania struktury programu. Podstawą do wyliczeń jest liczba punktów decyzyjnych w schemacie blokowym danego programu.

Z perspektywy testowania korzyści z podejścia DevOps są następujące:

- szybka informacja zwrotna na temat jakości kodu oraz informacja, czy zmiany wpływają negatywnie na istniejący kod;
- CI tworzy przesunięcie w lewo (*shift-left*) w testowaniu (patrz punkt 2.1.5) poprzez zachęcanie deweloperów do dostarczania wysokiej jakości kodu wraz z testami modułów;
- DevOps ułatwia tworzenie stabilnych środowisk testowych poprzez promowanie zautomatyzowanych procesów ciągłej integracji i ciągłego dostarczania oprogramowania (CI/CD);
- zwiększenie nacisku na niefunkcjonalne charakterystyki jakościowe (np. wydajność, niezawodność);
- automatyzacja poprzez potok wdrożeń zmniejsza potrzebę powtarzalnych testów manualnych;
- ryzyko regresji jest minimalizowane dzięki skali i zakresowi zautomatyzowanych testów regresji.

DevOps nie jest jednak pozbawiony wad i podlega zagrożeniom i wyzwaniom, w szczególności:

- potok wdrożeń DevOps musi być zdefiniowany, ustanowiony i utrzymywany;
- narzędzia ciągłej integracji muszą być wprowadzone i utrzymywane;
- automatyzacja testów wymaga dodatkowych zasobów i może być trudna do ustanowienia i utrzymania;
- zespoły czasami stają się nadmiernie zależne od testów modułowych i wykonują zbyt mało testów systemowych i akceptacyjnych.

2.1.5. Przesunięcie w lewo

Zasada „Wczesne testowanie oszczędza czas i pieniądze” (zobacz podrozdział 1.3) jest czasami nazywana **przesunięciem w lewo** (ang. *shift-left*), ponieważ jest to podejście, w którym testowanie jest wykonywane jak najwcześniej w danym cyklu wytwarzania oprogramowania. Przesunięcie w lewo zwykle sugeruje, że testowanie powinno być wykonane wcześniej (np. bez czekania na implementację kodu lub integrację modułów), ale nie oznacza to, że testowanie w późniejszym okresie cyklu wytwarzania powinno być zaniedbane.



Istnieje kilka dobrych praktyk, które ilustrują, jak osiągnąć „przesunięcie w lewo” w testowaniu. Są to:

- **Stosowanie przeglądów.** Przeglądy można wykonywać wcześnie w cyklu wytwarzania, jeszcze przed napisaniem kodu. Przegląd specyfikacji często znajduje potencjalne defekty w specyfikacji, takie jak niejednoznaczności, niekompletność i niespójności.
- **Stosowanie ciągłej integracji i ciągłego dostarczania oprogramowania** (patrz punkt 2.1.4) zapewnia szybką informację zwrotną o jakości kodu, a także wymusza tworzenie zautomatyzowanych testów modułowych dla kodu źródłowego, gdy jest on przekazywany do repozytorium kodu.

- **Wykonywanie analizy statycznej.** Analiza statyczna kodu źródłowego może być przeprowadzona przed testowaniem dynamicznym lub jako część zautomatyzowanego procesu, np. potoku wdrożeń (patrz punkt 2.1.4).
- **Wykonywanie testów niefunkcjonalnych na poziomie testowania modułowego, jeśli jest to możliwe.** Jest to forma przesunięcia w lewo, ponieważ testy niefunkcjonalne są zwykle wykonywane w późniejszym okresie cyklu wytwarzania oprogramowania, kiedy dostępny jest kompletny system i reprezentatywne środowisko testowe.
- **Stosowanie testowania opartego na modelu** (ang. *model-based testing*). W podejściu tym tester tworzy model oprogramowania, a specjalne narzędzie na podstawie tego modelu automatycznie tworzy przypadki testowe osiągające określone kryterium pokrycia. Model zazwyczaj może być utworzony przed rozpoczęciem prac implementacyjnych.

Przykłady stosowania podejścia „przesunięcie w lewo” widać dobrze w niektórych modelach cyklu wytwarzania czy praktykach wytwórczych:

- w modelu V czynności testowe (np. projektowanie lub tworzenie prototypów testów) odbywają się już w momencie rozpoczęcia odpowiadającej im fazy wytwórczej;
- w modelach iteracyjnych testowanie obecne jest zwykle w każdej iteracji, a więc rozpoczyna się we wczesnym momencie projektu;
- w wytwarzaniu sterowanym testami (podejście „najpierw test”) pisanie testów przed napisaniem kodu w podejściu TDD czy ATDD (patrz punkt 2.1.3) w naturalny sposób powoduje przesunięcie czynności testowych wcześnie w cyklu wytwarzania.

2.1.6. Retrospektwy i doskonalenie procesów

Retrospektwy (znane również jako spotkania *lessons learned*) odbywają się na końcu projektu lub po osiągnięciu w nim kamienia milowego, np. na końcu iteracji lub po ukończeniu poziomu testów. Podczas tych spotkań uczestnicy dyskutują o tym, co się udało (co było sukcesem), co się nie udało i można poprawić oraz jak wprowadzić ulepszenia i utrzymać sukcesy w przyszłości. Spotkania te obejmują w szczególności takie tematy jak:

- procesy i organizacja (np. czy stosowane procedury opóżniały, utrudniały, czy też raczej przyspieszały i ułatwiały wykonanie określonych zadań?);
- ludzie (np. czy członkowie zespołu dysponowali odpowiednią wiedzą i umiejętnościami, czy też występują w tym obszarze jakieś braki i jest konieczność przeprowadzenia szkoleń?);
- relacje (np. czy zespół współpracował ze sobą w sposób płynny? czy komunikacja była efektywna?);
- narzędzia (np. czy użycie określonych narzędzi zwiększyło efektywność lub wydajność testowania? czy pozyskanie określonego narzędzia pomogłoby w zwiększeniu efektywności lub wydajności?).

Zwykle jednak uczestnicy nie są ograniczeni do dyskusji w żadnych konkretnych obszarach. Jeśli mają miejsce odpowiednie działania naprawcze w kwestii problemów zidentyfikowanych podczas retrospektwy, przyczyniają się one do zwiększenia samoorganizacji zespołów i ciągłego doskonalenia rozwoju oraz testowania. Wyniki retrospektwy powinny być zapisane i mogą być częścią sumarycznego raportu z testów (patrz punkt 5.3.2).

Retrospektwy mogą skutkować decyzjami dotyczącymi ulepszeń związanych z testami, a tym samym przyczyniać się do doskonalenia procesu testowego. Udanie przeprowadzone retrospektwy dają w szczególności następujące korzyści dla testowania:

- zwiększenie efektywności testów (np. więcej wykrywanych defektów);
- zwiększenie wydajności testów (np. osiągnięcie tego samego celu mniejszym nakładem sił poprzez wykorzystanie narzędzi);
- zwiększenie jakości przypadków testowych (większe prawdopodobieństwo wykrywania defektów, mniejsze prawdopodobieństwo defektów w samych testach);
- zwiększenie zadowolenia zespołu testerskiego (podniesienie morale, polepszenie relacji w zespole, zwiększenie efektywności zespołu);
- polepszenie współpracy między deweloperami a testerami.

Retrospektwy mogą również dotyczyć testowalności aplikacji, historyjek użytkownika lub innych wymagań, funkcji bądź interfejsów systemowych. Analiza przyczyn źródłowych defektów może również prowadzić do ulepszeń w testowaniu i rozwoju. Ogólnie rzecz biorąc, zespoły powinny wdrażać tylko kilka usprawnień naraz (np. w iteracji), aby umożliwić ciągłe doskonalenie w stałym tempie.

Czas i organizacja spotkania zależą od konkretnego modelu cyklu wytwarzania oprogramowania. Na przykład w zwinnym rozwoju oprogramowania przedstawicielem biznesu i zespół zwinny biorą udział w każdej retrospektwie jako uczestnicy, podczas gdy moderator organizuje i prowadzi spotkanie. W niektórych przypadkach zespoły mogą zaprosić na spotkanie innych uczestników.

Testerzy powinni odgrywać ważną rolę w retrospektywach, ponieważ wnoszą swoją unikatową perspektywę (patrz punkt 1.5.3). Testowanie występuje w każdej iteracji lub wydaniu i przyczynia się do sukcesu projektu. Wszyscy członkowie zespołu są zachęcani do wnoszenia wkładu zarówno w testowanie, jak i w działania niezwiązane z nim.

Retrospektwy powinny odbywać się w atmosferze wzajemnego zaufania. Cechy udanej retrospektwy są takie same jak w przypadku przeglądów (patrz rozdział 3.).

2.2. Poziomy testów i typy testów

- | | |
|---------------|---|
| FL-2.2.1 (K2) | Kandydat rozróżnia poszczególne poziomy testów. |
| FL-2.2.2 (K2) | Kandydat rozróżnia poszczególne typy testów. |
| FL-2.2.3 (K2) | Kandydat odróżnia testowanie potwierdzające od testowania regresji. |

Poziomy testów (ang. *test level*) to grupy czynności testowych, które organizuje się i którymi zarządza się wspólnie. Każdy poziom testów jest instancją procesu testowego składającą się z czynności opisanych w podrozdziale 1.4, wykonywanych dla oprogramowania na danym poziomie wytwarzania, od pojedynczych modułów po kompletne systemy lub systemy systemów. Poziomy te są powiązane z innymi czynnościami wykonywanymi w ramach cyklu wytwarzania oprogramowania.



Poziomy testów są charakteryzowane w szczególności przez takie atrybuty jak:

- **przedmiot testów** (ang. *test object*);
- cel testu;
- podstawa testów;
- defekty i awarie;
- specyficzne podejścia i odpowiedzialności.



Testowanie związane z określona cechą obiektu testowego nazywane jest **typem testu** (ang. *test type*). Różnica między poziomami a typami testów jest taka, że poziomy odnoszą się do faz cyklu wytwarzania i stopnia zaawansowania w tworzeniu produktu, natomiast w typach testów bierze się pod uwagę cel, jaki przyświeca testom. Dlatego poziomy i typy testów są od siebie niezależne. Wróćmy jeszcze do tej kwestii.



2.2.1. Poziomy testów

Sylabus poziomu podstawowego opisuje pięć typowych, najczęściej spotykanych poziomów testów. Są to:

- testowanie modułowe;
- testowanie integracyjne modułów;
- testowanie systemowe;
- testowanie integracyjne systemów;
- testowanie akceptacyjne.

Należy jednak pamiętać, że jest to tylko przykładowa klasyfikacja. Organizacje wytwarzające oprogramowanie mogą wykorzystywać tylko część z tych poziomów lub posiadać inne, dodatkowe poziomy, specyficzne dla danej organizacji.

Każdy poziom testów wymaga odpowiedniego środowiska testowego. Dla testów modułowych środowiskiem tym jest zazwyczaj środowisko do przeprowadzania

testów jednostkowych, czyli biblioteki typu xUnit (np. JUnit), biblioteki służące do tworzenia tzw. *atrap obiektów* (np. EasyMock) itp. Do testowania akceptacyjnego z kolei idealnie nadaje się środowisko testowe podobne do środowiska produkcyjnego, ponieważ duża część defektów polowych (tzn. zgłaszanych przez użytkowników po wydaniu oprogramowania do klienta) to defekty powstałe w wyniku interakcji systemu i środowiska.

2.2.1.1. Testowanie modułowe

Cele testowania modułowego

Testowanie modułowe (ang. *component testing*), zwane także testowaniem jednostkowym, testowaniem komponentów lub testowaniem programu, skupia się na modułach, które można przetestować *oddzielnie*. Celami tego typu testowania są między innymi:



- zmniejszanie ryzyka w module;
- sprawdzanie zgodności zachowań funkcjonalnych i niefunkcjonalnych modułu z projektem i specyfikacjami;
- budowanie zaufania do jakości modułu;
- wykrywanie defektów w module;
- zapobieganie przedostawianiu się defektów na wyższe poziomy testowania.

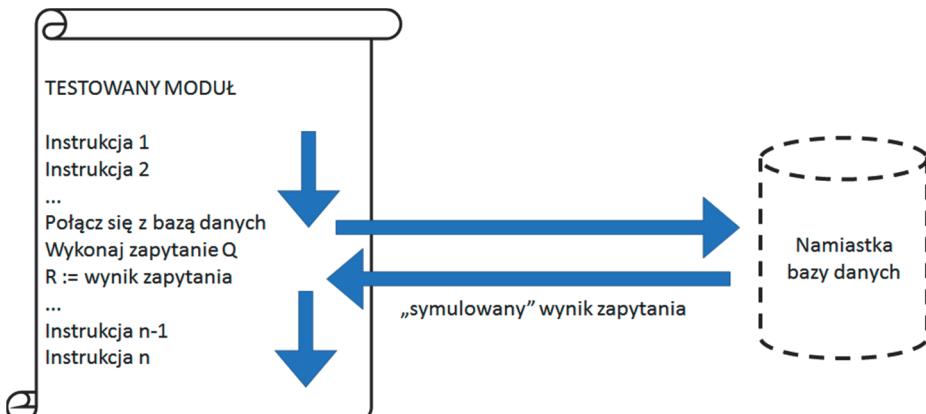
W niektórych przypadkach — zwłaszcza w przyrostowych i iteracyjnych modelach wytwarzania oprogramowania (np. modelu zwinnym), w których zmiany kodu mają charakter ciągły — automatyczne modułowe testy regresji są kluczowym elementem pozwalającym uzyskać pewność, że wprowadzone zmiany nie spowodowały nieprawidłowej pracy innych, istniejących już modułów.

Testy modułowe są najczęściej wykonywane w *izolacji* od reszty systemu (zależy to przede wszystkim od modelu cyklu wytwarzania oprogramowania i konkretnego systemu), przy czym w takiej sytuacji może być konieczne użycie atrap obiektów (ang. *mock object*), wirtualizacji usług, jarzm testowych, zaślepek bądź sterowników. Izolacja ta powoduje, że testy jednostkowe powinno dać się wykonać w dowolnej kolejności, a ich wynik powinien być taki sam niezależnie od tej kolejności.

Atrypy obiektów używane są w testach modułowych po to, aby umożliwić rzeczywiste wykonanie partii kodu odwołujących się do innych obiektów (np. do bazy danych), które jeszcze — w momencie wykonywania testów jednostkowych — nie istnieją. To częsta sytuacja, gdyż testy modułowe zazwyczaj wykonuje się wcześniej w fazach cyklu wytwarzania i wiele komponentów systemu może jeszcze nie być gotowych. Atrypy obiektów umożliwiają wykonanie kodu, a więc sprawdzenie jego własnej funkcjonalności — zauważmy, że nie interesuje nas tutaj sama komunikacja czy interakcja testowanego modułu z atrapą obiektu (np. atrapą bazy danych). Testowaniem takiej komunikacji zajmuje się dopiero testowanie integracyjne.

Rozważmy przykład z rysunku 2.13. Założymy, że testujemy skrypt, który w pewnym momencie ma za zadanie wysłać zapytanie do bazy danych, odebrać wynik tego zapytania i przetworzyć go w określony sposób. W teście modułowym nie będzie

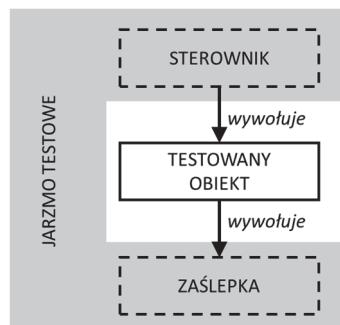
interesowała nas komunikacja skryptu z bazą, to, czy zapytanie zostało właściwie przekazane, zrozumiane, a wynik poprawnie dostarczony do skryptu. Tutaj interesuje nas tylko funkcjonalność samego modułu — czy robi to, co powinien robić według specyfikacji. Zatem namiastka bazy danych mogłaby nawet zwracać zawsze ten sam wynik zapytania, nie jest tu bowiem istotna postać tego wyniku, lecz to, co skrypt robi z otrzymanym zapytaniem.



RYSUNEK 2.13. Wykorzystanie atrapy obiektu

Ze względu na relację wywołujący-wywoływany można wyróżnić dwa typy atrap obiektów (ang. *mock object*): zaślepki (ang. *stub*) i sterowniki (ang. *driver*). Co do zasady obiekty te pełnią tę samą rolę, tzn. symulują działanie innych, nieistniejących jeszcze obiektów. Różnica dotyczy jedynie tego, czy atrapa obiektu jest przez nasz testowany moduł wywoływana, czy sama nasz moduł wywołuje. W pierwszym przypadku implementację takiej atrapy nazywamy zaślepką, w drugim — sterownikiem. Różnica ta pokazana jest symbolicznie na rysunku 2.14. Gdy do testowania używamy zarówno sterowników, jak i zaślepek, takie środowisko nazywamy jarzmem testowym.

Testowanie modułowe może obejmować funkcjonalność (np. poprawność obliczeń), ale również charakterystyki niefunkcjonalne oprogramowania (np. wydajność, wycieki pamięci) oraz właściwości strukturalne (np. testowanie decyzji).



RYSUNEK 2.14. Sterownik, zaślepka a jarzmo testowe

Podstawa testów

Przykładowymi produktami pracy, które mogą być wykorzystywane jako podstawa testów w ramach testowania modułowego, są między innymi:

- projekt szczegółowy;
- kod;
- model danych;
- specyfikacje modułów.

Przedmiot testów

Do typowych przedmiotów testów dla testów modułowych zaliczają się:

- moduły, jednostki lub komponenty;
- kod i struktury danych;
- klasy;
- moduły baz danych.

Typowe defekty i awarie

Przykładami typowych defektów i awarii wykrywanych w ramach testowania modułowego są:

- niepoprawna funkcjonalność (np. niezgodna ze specyfikacją projektu);
- problemy z przepływem danych;
- niepoprawne kod i logika.

Defekty są zwykle usuwane natychmiast po wykryciu, często bez formalnego zatrządzania nimi. Nie zbiera się zatem informacji np. o tym, ile razy programista poprawiał dany fragment kodu, jakiego rodzaju defekty były wykrywane, w jaki sposób je naprawiono, ile czasu trwała naprawa. Należy jednak zaznaczyć, że zgłaszając defekty, programiści dostarczają ważnych informacji na potrzeby analizy przyczyny podstawowej i doskonalenia procesów.

Konkretne podejście i odpowiedzialności

Testowanie modułowe jest zwykle wykonywane przez programistę — autora kodu, ponieważ zawsze wymaga dostępu do testowanego kodu. W związku z tym programiści mogą na przemian tworzyć moduły i wykrywać/usuwać defekty. Programiści często piszą i wykonują testy po napisaniu kodu danego modułu. Jednakże w niektórych sytuacjach — zwłaszcza w przypadku zwinnego tworzenia oprogramowania — automatyczne przypadki testowe do testowania modułowego można również tworzyć przed napisaniem kodu aplikacji.

2.2.1.2. Testowanie integracji modułów i testowanie integracji systemów

Testowanie integracji modułów oraz testowanie integracji systemów co do zasady są bardzo podobne, dlatego te dwa poziomy testów zostaną omówione łącznie.

Testowanie integracji modułów (ang. *component integration testing*) skupia się na interakcjach i interfejsach między integrowanymi modułami. Przez moduł można tu rozumieć klasę, plik, funkcję, procedurę, pakiet, komponent oprogramowania itp. Testy tego typu wykonuje się po zakończeniu testowania modułowego (zwykle są to testy automatyczne). W przypadku iteracyjnego wytwarzania oprogramowania testy integracyjne modułów są zwykle elementem procesu ciągłej integracji.



Testowanie integracji systemów (ang. *system integration testing*) skupia się na interakcjach i interfejsach między systemami, pakietami i mikrourługami. Testy tego typu mogą również obejmować interakcje z interfejsami dostarczonymi przez organizacje zewnętrzne (np. dostawców usług internetowych — ang. *web services*). W takim przypadku organizacja wytwarzająca oprogramowanie nie kontroluje interfejsów zewnętrznych, co może stwarzać cały szereg problemów w zakresie testowania (związań np. z usuwaniem defektów w kodzie tworzonym przez organizację zewnętrzną, które blokują testowanie, bądź z przygotowywaniem środowisk testowych). Testowanie integracji systemów może odbywać się po zakończeniu testowania systemowego lub równolegle do trwającego testowania systemowego (dotyczy to zarówno sekwencyjnego, jak i przystosowanego wytwarzania oprogramowania).



Cele testowania integracyjnego

Typowe cele **testowania integracyjnego** (ang. *integration testing*) to:

- zmniejszanie ryzyka wynikającego z interakcji modułów/systemów;
- sprawdzanie zgodności zachowań funkcjonalnych i niefunkcjonalnych interfejsów modułów/systemów z projektem i specyfikacjami;
- budowanie zaufania do jakości interfejsów wykorzystywanych przez moduły/systemy;
- wykrywanie defektów w interfejsach i protokołach komunikacyjnych;
- zapobieganie przedostawianiu się defektów na wyższe poziomy testowania.



Podstawa testów

Podstawą testów integracyjnych jest wszelkiego rodzaju dokumentacja opisująca interakcję czy współpracę poszczególnych modułów lub systemów. Przykładowymi produktami pracy, które mogą być wykorzystywane jako podstawa testów w ramach testowania integracyjnego, są:

- projekt oprogramowania i systemu;
- diagramy sekwencji;
- specyfikacje interfejsów;
- przypadki użycia;
- architektura na poziomie modułów i systemu;
- przepływy pracy;
- definicje interfejsów zewnętrznych.

Wymienienie jako podstawy testów integracyjnych przypadków użycia może wydawać się dziwne. Jednak przestanie takie być, jeśli uświadomimy sobie, że przypadki użycia opisują *interakcje* pomiędzy tzw. aktorami, czyli najczęściej pomiędzy użytkownikiem a systemem lub między dwoma systemami. Jest to więc forma testowania integracji.

Przedmiot testów

Do typowych przedmiotów testów zaliczają się:

- interfejsy programowania aplikacji (API) zapewniające komunikację między modułami;
- interfejsy zapewniające komunikację między systemami (np. system-system, system-baza danych, mikrousługa-mikrousługa);
- protokoły komunikacyjne pomiędzy modułami/systemami.

Typowe defekty i awarie

Przykładami typowych defektów i awarii wykrywanych w ramach testowania integracji modułów są:

- niepoprawne lub brakujące dane bądź niepoprawne kodowanie danych;
- niepoprawne uszeregowanie lub niepoprawna synchronizacja wywołań interfejsów;
- niezgodności interfejsów;
- błędy komunikacji między modułami;
- brak obsługi lub nieprawidłowa obsługa błędów komunikacji między modułami;
- niepoprawne założenia dotyczące znaczenia, jednostek lub granic danych przesyłanych między modułami.

Przykładami typowych defektów i awarii wykrywanych w ramach testowania integracji systemów są:

- niespójne struktury komunikatów przesyłanych między systemami;
- niepoprawne lub brakujące dane bądź niepoprawne kodowanie danych;
- niezgodność interfejsów;
- błędy komunikacji między systemami;

- brak obsługi lub nieprawidłowa obsługa błędów komunikacji między systemami;
- niepoprawne założenia dotyczące znaczenia, jednostek lub granic danych przesyłanych między systemami;
- nieprzestrzeganie bezwzględnie obowiązujących przepisów dotyczących zabezpieczeń.

Konkretnie podejścia i odpowiedzialności

Testy integracji modułów i testy integracji systemów powinny koncentrować się na samej integracji. Na przykład w przypadku integrowania modułu A z modułem B testy powinny skupiać się na komunikacji między tymi modułami, a nie na funkcjonalności każdego z nich (funkcjonalność ta powinna być uprzednio przedmiotem testowania modułowego). Analogicznie w przypadku integrowania systemu X z systemem Y: tu testy powinny skupiać się na komunikacji między tymi systemami, a nie na funkcjonalności każdego z nich (funkcjonalność ta powinna być przedmiotem testowania systemowego). Na tym poziomie testowania można stosować testy funkcjonalne, niefunkcjonalne i strukturalne.

Testowanie integracji modułów jest często obowiązkiem programistów, natomiast za testowanie integracji systemów (ze względu na wysokopoziomowy charakter testów) zwykle odpowiadają testerzy. O ile jest to możliwe, testerzy wykonujący testowanie integracji systemów powinni znać architekturę systemu, a ich uwagi powinny być brane pod uwagę na etapie planowania integracji.

Środowisko testowe, zwłaszcza w przypadku testowania integracji systemów, powinno w miarę możliwości odzwierciedlać specyfikę środowiska docelowego lub produkcyjnego. Jest tak dlatego, że bardzo duża liczba awarii powstaje na skutek interakcji systemu ze środowiskiem, w którym system jest posadowiony. Środowisko testowe identyczne lub podobne do docelowego pozwala wykryć większość z tych awarii i powodujących je defektów w fazie testów, przed wydaniem oprogramowania do klienta.

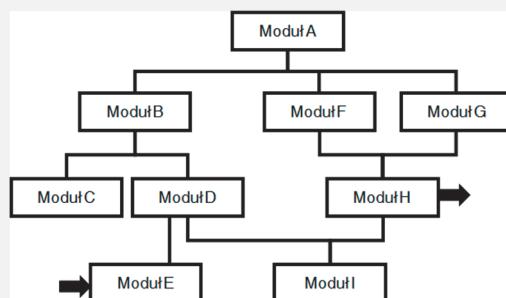
Podobnie jak w przypadku testowania modułowego istnieją sytuacje, w których automatyczne integracyjne testy regresji pozwalają uzyskać pewność, że wprowadzone zmiany nie spowodowały nieprawidłowej pracy dotychczasowych interfejsów.

Strategie integracji

Zaplanowanie testów integracyjnych i strategii integracji przed zbudowaniem modułów lub systemów umożliwia wytworzenie tych modułów lub systemów w sposób zapewniający maksymalną efektywność testowania. Istnieje szereg podejść w zakresie strategii przeprowadzania testów integracyjnych. Sylabus poziomu podstawowego wymienia kilka z nich. Głównie dotyczą integracji modułów i są to:

- **Strategie oparte na architekturze systemu**, czyli strategie **zstępujące** (ang. *top-down*) i **wstępujące** (ang. *bottom-up*). W strategii zstępującej w pierwszej kolejności testuje się integrację modułu głównego z modułami przezeń wywoływanymi, następnie integrację tych modułów z modułami wywoływanymi przez nie i tak dalej. Integracja następuje więc „poziomami” zagęśczenia modułów w hierarchii ich wywołań. W strategii wstępującej kierunek integracji jest odwrotny: zaczynamy od dołu i kolejno dołączamy moduły leżące na wyższych poziomach, wywołujące moduły już wywołane. Stosując strategię integracji opartej na architekturze systemu, często wykorzystuje się zaślepki (strategia zstępująca) i sterowniki (strategia wstępująca), ponieważ na ogół wywoływane lub wywołujące moduły nie są jeszcze napisane.
- **Strategie oparte na zadaniach funkcjonalnych**. Czasami, z pewnych względów, istotne jest dla nas, aby najpierw przetestować integrację grupy modułów, które łącznie odpowiadają za jakąś ważną dla nas w danym momencie funkcjonalność. Strategia ta polega więc najpierw na testach integracji modułów tworzących tę funkcjonalność, a potem na integracji pozostałych modułów.
- **Strategia oparta na sekwencjach przetwarzania transakcji**. Jeśli zależy nam przede wszystkim, aby najwcześniej przetestować np. ścieżkę przepływu informacji przez system (np. ścieżkę *określone wejście – określone wyjście*), najpierw przeprowadzamy testy integracji modułów leżących na tej właśnie ścieżce. W następnej kolejności przeprowadzamy testy integracji pozostałych modułów.
- **Strategia oparta na innych aspektach systemu**. Ostatnie dwa przykłady strategii integracji polegają na wyznaczeniu kolejności testów integracji ze względu na jakieś ważne dla nas kryteria. Można wyobrazić sobie wiele innych tego typu strategii, np. opartych na krytyczności modułów lub częstości wykorzystania komunikacji między modułami.

Przykład. Zobaczmy na przykładzie, jak w praktyce wyglądałyby powyższe strategie. Założmy, że nasz system posiada architekturę przedstawioną na rysunku 2.15. Dwa moduły połączone kreską oznaczają, że moduł leżący wyżej wywołuje moduł leżący niżej (zatem moduł A jest modułem głównym — wywołuje on moduły B, F i G; moduł B wywołuje moduły C i D itd.). Ponadto założmy, że moduły B, C, D, E tworzą jedną funkcjonalność, dotyczącą obsługi wejścia. Moduł E odpowiada za pobranie danych od użytkownika, moduł H — za raportowanie wyników.



RYSUNEK 2.15. Przykład architektury systemu

W strategii zstępującej kolejność testowania integracyjnego będzie następująca:

1. Integracja A-B, A-F, A-G (jeśli moduły B, F, G nie są jeszcze gotowe, należy użyć w testach zaślepek tych modułów).
2. Integracja B-C, B-D, F-H, G-H (analogicznie, może zachodzić konieczność użycia zaślepek dla C, D, H).
3. Integracja D-E, D-I, H-I (analogicznie, może zachodzić konieczność użycia zaślepek dla E, I).

W strategii wstępującej jest podobnie, tylko kolejność jest odwrotna:

1. Integracja D-E, D-I, H-I (może zachodzić konieczność użycia sterowników dla D, H).
2. Integracja B-C, B-D, F-H, G-H (może zachodzić konieczność użycia sterowników dla B, F, G).
3. Integracja A-B, A-F, A-G (może zachodzić konieczność użycia sterownika dla A).

W strategii opartej na funkcjonalności — przy założeniu, że funkcjonalność obsługi wejścia jest dla nas priorytetowa — przykładowa kolejność integracji może być taka:

1. Integracja B-C, B-D, D-E (funkcjonalność obsługi wejścia).
2. Integracja A-B, D-I (testy integracji obsługi wejścia z otoczeniem).
3. Integracja A-F, A-G, F-H, G-H, H-I (pozostałe połączenia).

Z kolei strategia oparta na sekwencji przetwarzanej transakcji może przybrać następującą postać:

1. Integracja E-D, D-B, B-A, A-F, F-H (ścieżka wejście-wyjście).
2. Integracja A-G, B-C, G-H, D-I, H-I (pozostałe połączenia).

Metody przyrostowe integracji — takie jak te przedstawione powyżej — pozwalają, w momencie wystąpienia awarii, na szybkie zlokalizowanie defektu. Jeśli w naszym przykładzie wykorzystujemy strategię zstępującą i awaria ma miejsce w momencie testowania F-H, to możemy być prawie pewni, że defekt dotyczy komunikacji F-H, a więc jest zlokalizowany w jednym z tych dwóch modułów, a nie np. w A czy B.

Nie zaleca się integracji metodą „wielkiego wybuchu” (ang. *big bang*), polegającej na zintegrowaniu wszystkich modułów lub systemów naraz, ponieważ w momencie wystąpienia awarii zwykle nie mamy pojęcia, czym mogła ona zostać spowodowana. W odpowiednim ukierunkowaniu testowania integracyjnego może również pomóc analiza ryzyka związanego z najbardziej złożonymi interfejsami.

Im szerszy jest zakres integracji, tym trudniej jest wskazać konkretny moduł lub system, w którym wystąpiły defekty, co może prowadzić do wzrostu ryzyka i wydłużenia czasu diagnozowania problemów. Jest to jeden z powodów, dla których

powszechnie stosuje się metodę ciągłej integracji (ang. *Continuous Integration, CI*), polegającą na integrowaniu oprogramowania moduł po module (np. integracja funkcjonalna). Elementem ciągłej integracji jest często automatyczne testowanie regresji, które w miarę możliwości powinno odbywać się na wielu poziomach testów.

2.2.1.3. Testowanie systemowe

Cele testowania systemowego

Poziom **testowania systemowego** (ang. *system testing*) to typowy poziom działania zawodowego testera (testowanie modułowe i integracyjne zazwyczaj przeprowadza programista, a akceptacyjne — klient). Testowanie systemowe skupia się na zachowaniu i możliwościach całego, już zintegrowanego systemu lub produktu, często z uwzględnieniem całokształtu zadań, jakie może on wykonywać, oraz zachowań niefunkcjonalnych, jakie wykazuje podczas wykonywania tych zadań. Celami testowania systemowego są przede wszystkim:



- zmniejszanie ryzyka niepoprawnego działania systemu;
- sprawdzanie zgodności zachowań funkcjonalnych i niefunkcjonalnych systemu z projektem i specyfikacjami;
- sprawdzanie kompletności systemu i prawidłowości jego działania;
- budowanie zaufania do jakości systemu jako całości;
- wykrywanie defektów w systemie;
- zapobieganie przedostawaniu się defektów na poziom testowania akceptacyjnego lub na produkcję.

W przypadku niektórych systemów celem testowania może być również zweryfikowanie jakości danych. Podobnie jak w przypadku testowania modułowego i testowania integracyjnego modułów i systemów, automatyczne systemowe testy regresji pozwalają uzyskać pewność, że wprowadzone zmiany nie spowodowały nieprawidłowej pracy dotychczasowych funkcji lub całościowej funkcjonalności. W wyniku testowania systemowego często powstają informacje, na podstawie których interesariusze podejmują decyzje o przekazaniu systemu do użycia produkcyjnego. Ponadto testowanie systemowe może być niezbędne do spełnienia wymagań wynikających z obowiązujących przepisów lub norm/standardów.

Tak jak w przypadku poziomu testów integracyjnych systemów, środowisko testowe powinno w miarę możliwości odzwierciedlać specyfikę środowiska docelowego lub produkcyjnego.

Na poziomie testów systemowych powstaje najwięcej informacji zwrotnych z procesu testowego. Dotyczą one liczby i typu defektów znalezionych w programie, czasu ich naprawy, momentu wprowadzenia defektu itp.

Podstawa testów

Testowanie systemowe odnosi się do w pełni zintegrowanego systemu, zatem podstawa testów powinna się odnosić właśnie do systemu jako całości. Przykładowe produkty pracy, które mogą być wykorzystywane jako podstawa testów w ramach testowania systemowego, to między innymi:

- specyfikacje wymagań (funkcjonalnych i niefunkcjonalnych) dotyczących systemu i oprogramowania;
- raporty z analizy ryzyka;
- przypadki użycia;
- opowieści i historyjki użytkownika;
- modele zachowania systemu;
- diagramy stanów;
- instrukcje obsługi systemu i podręczniki użytkownika.

Przedmioty testów

Do typowych przedmiotów testów dla testów systemowych zalicza się:

- system podlegający testowaniu (system jako całość);
- konfigurację i dane konfiguracyjne systemu.

Typowe defekty i awarie

Przykłady typowych defektów i awarii wykrywanych w ramach testowania systemowego to:

- niepoprawne obliczenia;
- niepoprawne lub nieoczekiwane zachowania funkcjonalne lub niefunkcjonalne systemu;
- niepoprawne przepływy sterowania i/lub przepływy danych w systemie;
- problemy z prawidłowym i kompletnym wykonywaniem całościowych zadań funkcjonalnych;
- problemy z prawidłowym działaniem systemu w środowisku produkcyjnym;
- niezgodność działania systemu z opisami zawartymi w instrukcji obsługi systemu i podręcznikach użytkownika.

Konkretnie podejścia i odpowiedzialności

Testowanie systemowe powinno koncentrować się na ogólnym, kompleksowym zachowaniu systemu jako całości na poziomie zarówno funkcjonalnym, jak i niefunkcjonalnym. W ramach testowania systemowego często stosuje się takie techniki testowania (patrz rozdział 4.), które są najbardziej odpowiednie dla danego aspektu (danych aspektów) systemu będącego przedmiotem testów. Na przykład w celu sprawdzenia, czy zachowanie funkcjonalne jest zgodne z opisem zawartym w regułach biznesowych, można wykorzystać tablicę decyzyjną.

Testowanie systemowe wykonują zwykle niezależni testerzy. Defekty w specyfikacjach (np. brakujące historyjki użytkownika lub niepoprawnie wyrażone wymagania biznesowe) mogą doprowadzić do nieporozumień lub sporów dotyczących oczekiwanej zachowania systemu. W konsekwencji mogą pojawić się rezultaty fałszywie pozytywne lub fałszywie negatywne (patrz tabela 1.2), które powodują, odpowiednio, stratę czasu lub spadek skuteczności wykrywania defektów. Między innymi dlatego — aby zmniejszyćczęstość występowania powyższych sytuacji — należy zaangażować testerów w przeglądy, doprecyzowywanie historyjek użytkownika i inne czynności testowe o charakterze statycznym na jak najwcześniejszym etapie prac.

Niektóre niefunkcjonalne testy systemowe mogą wymagać tego, aby przeprowadzać je w reprezentatywnym środowisku. Przykładami takich testów mogą być:

- testy wydajnościowe (np. środowisko powinno odzwierciedlać rzeczywistą przepustowość sieci, a komponenty nie powinny być zastępowane namiastkami, gdyż będzie to fałszować np. czas odpowiedzi modułu);
- testy zabezpieczeń (np. ataki na zabezpieczenia będą zależeć od konkretnej konfiguracji systemu i środowiska, w jakim system się znajduje);
- testy użyteczności (np. końcowe testy użyteczności interfejsu powinny być prowadzone na rzeczywistym interfejsie, a nie na jego atrapie czy makiecie).

2.2.1.4. Testowanie akceptacyjne

Cele testowania akceptacyjnego

Testowanie akceptacyjne (ang. *acceptance testing*) — podobnie jak testowanie systemowe — skupia się zwykle na zachowaniu i możliwościach całego systemu lub produktu. Jest ono jednak przeprowadzane z perspektywy użytkownika, a nie zespołu twórczego. Ma więc charakter walidacji, a nie weryfikacji systemu. Celami testowania akceptacyjnego są przede wszystkim:

- budowanie zaufania użytkownika do systemu;
- sprawdzanie kompletności systemu i jego prawidłowego działania z punktu widzenia realizacji celów biznesowych.



W wyniku testowania akceptacyjnego mogą powstawać informacje pozwalające ocenić gotowość systemu do wdrożenia i użytkowania przez klienta (użytkownika). Podczas testowania akceptacyjnego mogą też zostać wykryte defekty, ale ich wykrywanie najczęściej nie jest celem testów. Na tym etapie chodzi nam raczej o *walidację* systemu, tzn. o sprawdzenie, czy system rzeczywiście spełnia potrzeby biznesowe klienta. W niektórych przypadkach znalezienie dużej liczby defektów na etapie testowania akceptacyjnego może być nawet uznawane za istotne ryzyko projektowe. Jeśli bowiem klient ma sprawdzić, czy oprogramowanie rozwiązuje jego problem, a system „zawiesza się co drugie kliknięcie”, to testy takie nie mają raczej sensu — są jedynie oznaką, że w procesie kontroli jakości popełniono jakieś fundamentalne błędy, przez które mnóstwo defektów prześliznęło się przez wszystkie fazy twórcze aż do momentu wydania oprogramowania do klienta.

Ponadto testowanie akceptacyjne może być niezbędne do spełnienia wymagań wynikających z obowiązujących przepisów prawa lub norm/standardów.

Najczęściej występujące formy testowania akceptacyjnego to:

- testowanie akceptacyjne przez użytkownika (tzw. UAT — ang. *User Acceptance Testing*);
- operacyjne testy akceptacyjne (tzw. OAT — ang. *Operational Acceptance Testing*);
- testowanie akceptacyjne pod kątem zgodności z umową i zgodności z prawem.

Testowanie akceptacyjne można również podzielić ze względu na miejsce jego wykonywania:

- testowanie alfa — testowanie przez klienta w siedzibie producenta, na środowisku testowym;
- testowanie beta (inaczej: testowanie polowe) — testowanie przez klienta u siebie, na własnym środowisku docelowym.

Poszczególne formy testowania akceptacyjnego opisano w kolejnych podpunktach.

Testowanie akceptacyjne przez użytkownika

Testowanie akceptacyjne systemu przez użytkownika odbywa się w (symulowanym) środowisku produkcyjnym. Głównym celem jest budowanie pewności, że system umożliwi użytkownikom spełnienie potrzeb, postawionych przed nim wymagań i wykona proces biznesowy z minimalną liczbą problemów, kosztem i ryzykiem.

Operacyjne testy akceptacyjne

Testowanie akceptacyjne systemu przez operatorów lub administratorów odbywa się zwykle w (symulowanym) środowisku produkcyjnym. Testy skupiają się na aspektach operacyjnych i mogą obejmować:

- testowanie mechanizmów tworzenia kopii zapasowych i odtwarzania danych;
- instalowanie, odinstalowywanie i aktualizowanie oprogramowania;
- usuwanie skutków awarii;
- zarządzanie użytkownikami;
- wykonywanie czynności pielęgnacyjnych;
- wykonywanie czynności związanych z ładowaniem i migracją danych;
- sprawdzanie, czy występują podatności zabezpieczeń;
- wykonywanie testów wydajnościowych.

Głównym celem produkcyjnych testów akceptacyjnych jest uzyskanie pewności, że operatorzy lub administratorzy systemu będą w stanie zapewnić użytkownikom prawidłową pracę systemu w środowisku produkcyjnym, nawet w wyjątkowych i trudnych warunkach.

Testowanie akceptacyjne zgodności z umową i zgodności z prawem

Testowanie akceptacyjne zgodności z umową odbywa się według kryteriów akceptacji zapisanych w umowie dotyczącej wytworzenia oprogramowania na zlecenie. Powyższe kryteria akceptacji powinny zostać określone w momencie uzgadniania przez strony treści umowy. Tego rodzaju testowanie akceptacyjne wykonują często użytkownicy lub niezależni testerzy.

Testowanie akceptacyjne zgodności z prawem jest wykonywane w kontekście obowiązujących aktów prawnych, takich jak ustawy, rozporządzenia czy normy bezpieczeństwa. Tego rodzaju testowanie akceptacyjne wykonują często użytkownicy lub niezależni testerzy, a rezultaty mogą być obserwowane lub kontrolowane przez organy nadzoru. Głównym celem testowania akceptacyjnego zgodności z umową i zgodności z prawem jest uzyskanie pewności, że osiągnięto zgodność z wymaganiami wynikającymi z obowiązujących umów lub przepisów.

Dobrym przykładem standardu narzucającego konkretne kryteria pokrycia jest standard DO-178C obowiązujący w lotnictwie. W normie tej oprogramowanie klasyfikuje się ze względu na stopień ryzyka (poziomy od A do E), a następnie dla każdej klasyfikacji określa się w szczególności wymogi dotyczące spełnienia konkretnych kryteriów pokrycia (patrz tabela 2.2). Kryteria pokrycia: MC/DC, decyzji i instrukcji występujące w tym standardzie odnoszą się do konkretnych białoskrzyinkowych kryteriów pokryć. To ostatnie, a także kryterium pokrycia gałęzi — podobne do pokrycia decyzji — omówione są w sylabusie (patrz punkty 4.3.1 i 4.3.2 niniejszego podręcznika). Kryterium MC/DC omówione jest w sylabusie poziomu zaawansowanego — Techniczny Analityk Testów.

TABELA 2.2. Wymagania normy DO-178C dotyczące pokrycia testami w zależności od poziomu ryzyka

POZIOM KRYTYCZNOŚCI	RODZAJ AWARII	WYMOGI DOTYCZĄCE TESTOWANIA
A	katastroficzna	wymagane pełne pokrycie MC/DC (zmodyfikowane pokrycie decyzji/instrukcji)
B	niebezpieczna	wymagane pełne pokrycie decyzji
C	poważna	wymagane pokrycie wymagań niskopoziomowych; testowanie właściwej obsługi danych i sterowania; wymagane pełne pokrycie instrukcji
D	mała	wymagane pokrycie wymagań wysokopoziomowych
E	bez efektu	brak

Testowanie alfa i beta

Twórcy oprogramowania przeznaczonego do powszechnej sprzedaży (COTS) często chcą uzyskać informacje zwrotne od potencjalnych lub obecnych klientów, zanim oprogramowanie trafi na rynek. Służą do tego testy alfa i beta.

Testowanie alfa jest wykonywane w siedzibie organizacji wytwarzającej oprogramowanie, ale zamiast zespołu wytwórczego wykonują testy potencjalni lub obecni

klienci i/lub operatorzy bądź niezależni testerzy. Z kolei testowanie beta wykonują obecni lub potencjalni klienci we własnych lokalizacjach. Testy beta mogą, ale nie muszą być poprzedzone testami alfa.

Jednym z celów testów alfa i beta jest budowanie zaufania potencjalnych i aktualnych klientów i/lub operatorów co do tego, że mogą oni używać systemu w normalnych warunkach, w środowisku produkcyjnym, aby osiągnąć swoje cele przy minimalnym wysiłku, koszcie i ryzyku. Innym celem może być wykrywanie błędów związanych z warunkami i środowiskiem (środowiskami), w których system będzie używany, szczególnie wtedy, gdy takie warunki są trudne do odtworzenia dla zespołu projektowego. Jeśli testy beta prowadzone są na reprezentatywnej grupie użytkowników, to ponieważ przeprowadzane są w środowiskach używanych przez poszczególnych testerów, testy będą pokrywać w reprezentatywny sposób środowiska, w jakich system będzie działał. Na przykład jeśli 80% użytkowników posiada system Windows 11, a 20% — system Windows 10, to w losowej, reprezentatywnej grupie testerów beta około 80% z nich powinno testować produkt zainstalowany pod systemem Windows 11, a około 20% — pod systemem Windows 10.

Podstawa testów

Przykładowymi produktami pracy, które mogą być wykorzystywane jako podstawa testów w ramach dowolnego typu testowania akceptacyjnego, są między innymi:

- procesy biznesowe;
- wymagania użytkowników lub wymagania biznesowe;
- przepisy, umowy, normy i standardy;
- przypadki użycia;
- dokumentacja systemu lub podręczniki dla użytkowników;
- procedury instalacji;
- raporty z analizy ryzyka.

Ponadto podstawą testów, z której wyprowadzane są przypadki testowe na potrzeby produkcyjnych testów akceptacyjnych, mogą być następujące produkty pracy:

- procedury tworzenia kopii zapasowych i odtwarzania danych;
- procedury usuwania skutków awarii;
- dokumentacja operacyjna;
- instrukcje wdrażania i instalacji;
- założenia wydajnościowe;
- normy, standardy lub przepisy w dziedzinie zabezpieczeń.

Typowe przedmioty testów

Do typowych przedmiotów testów dowolnego typu testów akceptacyjnych zaliczają się:

- system podlegający testowaniu;
- konfiguracja i dane konfiguracyjne systemu;
- procesy biznesowe wykonywane na całkowicie zintegrowanym systemie;
- systemy rezerwowe i ośrodki zastępcze (ang. *hot site*) (do testowania mechanizmów zapewnienia ciągłości biznesowej i usuwania skutków awarii);
- procesy związane z użyciem produkcyjnym i utrzymaniem;
- formularze;
- raporty;
- istniejące i skonwertowane dane produkcyjne.

Typowe defekty i awarie

Przykłady typowych defektów wykrywanych w ramach różnych form testowania akceptacyjnego to:

- systemowe przepływy pracy niezgodne z wymaganiami biznesowymi lub wymaganiami użytkowników;
- niepoprawnie zaimplementowane reguły biznesowe;
- niespełnienie przez system wymagań umownych lub prawnych;
- awarie niefunkcjonalne, takie jak podatności zabezpieczeń, niedostateczna wydajność pod dużym obciążeniem bądź nieprawidłowe działanie na obsługiwanej platformie.

Konkretne podejście i obowiązki

Testowanie akceptacyjne często spoczywa na klientach, użytkownikach biznesowych, właścicielach produktów lub operatorach systemów, ale w proces ten mogą być również zaangażowani inni interesariusze. Testowanie akceptacyjne często uznaje się za ostatni poziom sekwencyjnego cyklu wytwarzania oprogramowania, ale może ono również odbywać się na innych etapach, na przykład:

- testowanie akceptacyjne oprogramowania do powszechniej sprzedaży może odbywać się podczas jego instalowania lub integrowania;
- testowanie akceptacyjne nowego udoskonalenia funkcjonalnego może mieć miejsce przed rozpoczęciem testowania systemowego.

W iteracyjnych modelach wytwarzania oprogramowania zespoły projektowe mogą stosować na zakończenie każdej iteracji różne formy testowania akceptacyjnego, takie jak testy skupiające się na weryfikacji zgodności nowej funkcjonalności z kryteriami akceptacji bądź testy skupiające się na walidacji nowej funkcjonalności z punktu widzenia potrzeb użytkowników. Ponadto na końcu każdej iteracji, po ukończeniu każdej iteracji lub po wykonaniu serii iteracji mogą być wykonywane testy alfa i beta, a także testy akceptacyjne wykonywane przez użytkownika, produkcyjne testy akceptacyjne oraz testy akceptacyjne zgodności z umową i zgodności z prawem.

Przykład. Firma tworzy oprogramowanie webowe dla urzędu skarbowego służące do wpisywania przez urzędnika danych z formularzy PIT-11 na potrzeby wygenerowania końcowego formularza PIT-37 dla konkretnego podatnika.

Przykładem testu modułowego będzie przetestowanie poprawności działania formularza do wpisywania danych np. pod kątem tego, jak pola walutowe zachowają się, gdy wpisze się do nich wartość znakową czy kwotę o niepoprawnej dokładności.

Przykładem testu integracyjnego modułów będzie interakcja pomiędzy funkcją logowania do systemu a funkcją nadającą określone uprawnienia użytkownikowi.

Przykładem testu systemowego będzie przeprowadzenie przez testera całego procesu wpisywania danych z kilku formularzy PIT-11, a następnie zweryfikowanie, czy system generuje poprawny PIT-37.

Przykładem testu integracyjnego systemów będzie przetestowanie poprawności zaciągania danych podatnika z bazy PESEL na podstawie wpisanego przez użytkownika numeru PESEL podatnika.

Przykładem testu akceptacyjnego (beta) będzie walidacja systemu przez urzędników urzędu skarbowego w siedzibie urzędu, na sprzęcie i w środowisku docelowym. Walidacja może sprawdzać poprawność merytoryczną (np. zgodność działania programu z przepisami prawa), ale również kwestie użyteczności (np. czy interfejsy są czytelne) czy to, jak dobrze system integruje się z infrastrukturą IT urzędu skarbowego.

2.2.2. Typy testów

Typ testów to grupa dynamicznych czynności testowych wykonywanych z myślą o przetestowaniu określonych charakterystyk systemu oprogramowania (lub jego części) zgodnie z określonymi celami testów. Zatem w przeciwieństwie do poziomów testów typ testu odnosi się nie do kwestii procesowych, ale do celu, jaki mamy na uwadze, przeprowadzając dany test.

Celem testów może być między innymi:

- ocena funkcjonalnych charakterystyk jakościowych takich jak: kompletność, prawidłowość i adekwatność;
- ocena niefunkcjonalnych charakterystyk jakościowych, w tym parametrów takich jak: niezawodność, wydajność, bezpieczeństwo, kompatybilność czy użyteczność;
- ustalenie, czy struktura lub architektura komponentu lub systemu jest poprawna, kompletna i zgodna ze specyfikacjami.

Sylabus poziomu podstawowego wyróżnia cztery podstawowe typy testów:

- testowanie funkcjonalne;
- testowanie niefunkcjonalne;
- testowanie białoskrzynkowe;
- testowanie czarnoskrzynkowe.

Należy jednak podkreślić, że można wyróżnić o wiele więcej typów testów. Przykładem mogą być tzw. testy dymne (ang. *smoke tests*), których celem jest sprawdzenie poprawności działania podstawowych funkcjonalności systemu przed rozpoczęciem wykonywania bardziej szczegółowych testów.

Omówimy teraz po kolejni każdy z czterech powyższych typów testów.

2.2.2.1. Testowanie funkcjonalne

Testowanie funkcjonalne (ang. *functional testing*) systemu polega na wykonaniu testów, które oceniają funkcje, jakie system ten powinien realizować. Testowanie funkcjonalne sprawdza zatem, „co” system powinien robić. Wymagania funkcjonalne mogą być opisane w takich produktach pracy jak: specyfikacje wymagań biznesowych, opowieści, historyjki użytkownika, przypadki użycia lub specyfikacje funkcjonalne, ale zdarza się również, że występują w postaci nieudokumentowanej.



Głównym celem testów funkcjonalnych jest sprawdzenie funkcjonalnej poprawności (ang. *functional correctness*), funkcjonalnej odpowiedniości (ang. *functional appropriateness*) oraz funkcjonalnej zupełności (ang. *functional completeness*). Są to trzy podcharakterystyki funkcjonalności zdefiniowane w modelu jakości ISO/IEC/IEEE 25010.

Testy funkcjonalne to najczęściej kojarzony z testowaniem typ testów, ponieważ powszechnie utożsamia się testowanie ze sprawdzeniem, czy system robi to, co powinien robić na rzecz użytkownika. Nie można jednak zapominać, że inne typy testów, także te omówione w kolejnych punktach, są równie istotne.

Testy funkcjonalne można (i należy) wykonywać na wszystkich poziomach testów (np. testy dotyczące modułów mogą opierać się na specyfikacjach modułów), jednak z zastrzeżeniem, że testy wykonywane na poszczególnych poziomach skupią się na różnych zagadnieniach (patrz punkt 2.2.1). Tabela 2.3 podaje przykładowe czynności wykonywane przez testy funkcjonalne na różnych poziomach testowania dla wspomnianego wcześniej systemu do wprowadzania danych przez urzędy skarbowe.

TABELA 2.3. Poziomy testów a testy funkcjonalne

POZIOM TESTU	PRZYKŁADOWA PODSTAWA TESTU	PRZYKŁADOWY TEST
Modułowe	Specyfikacja modułu	Sprawdzenie poprawności obliczenia przez moduł kwoty podatku
Integracyjne modułów	Diagram komponentów (projekt architektury)	Sprawdzenie poprawności przesyłania danych pomiędzy modułem logowania a modułem nadawania uprawnień
Systemowe	Historyjka użytkownika	Sprawdzenie poprawności realizacji procesu biznesowego „zapłać podatek”

TABELA 2.3. Poziomy testów a testy funkcjonalne – ciąg dalszy

POZIOM TESTU	PRZYKŁADOWA PODSTAWA TESTU	PRZYKŁADOWY TEST
Integracyjne systemów	Specyfikacja interfejsu system-baza danych	Sprawdzenie poprawności przesłania zapytania do bazy danych i odebrania jego wyniku
Akceptacyjne	Podręcznik użytkownika	Sprawdzenie, czy opis realizacji funkcjonalności zawarty w podręczniku jest zgodny z rzeczywistym procesem aplikacji

Testowanie funkcjonalne uwzględnia zachowanie oprogramowania, które najczęściej opisane jest w dokumentach zewnętrznych wobec systemu: specyfikacji wymagań, historyjkach użytkownika itp. W związku z tym w przypadku testów funkcjonalnych do wyprowadzania warunków testowych i przypadków testowych dotyczących funkcjonalności komponentu lub systemu używa się zazwyczaj technik czarnoskrzynkowych (patrz podrozdział 4.2).

Staranność testowania funkcjonalnego można zmierzyć na podstawie pokrycia funkcjonalnego. Termin „pokrycie funkcjonalne” oznacza stopień, w jakim został przetestowany określony typ elementu funkcjonalnego, wyrażony jako procent elementów danego typu pokrytych przez testy. Dzięki możliwości śledzenia powiązań między testami a wymaganiami funkcjonalnymi można na przykład obliczyć, jaki procent wymagań został uwzględniony w ramach testowania, a w rezultacie zidentyfikować ewentualne luki w pokryciu.

Przykład. Tabela 2.4 przedstawia śledzenie powiązań między testami a wymaganiami funkcjonalnymi. Każdemu testowi przyporządkowano jego ważność (związaną z ryzykiem, jakie pokrywa) w skali od 1 (mało istotny) do 5 (krytyczny).

TABELA 2.4. Śledzenie powiązań wymagań z testami i informacja o ważności testów

TEST \ WYMAGANIE	REQ 1	REQ 2	REQ 3
Test1 (ważność 2)	X		
Test2 (ważność 5)	X	X	
Test3 (ważność 1)		X	X
Test4 (ważność 1)		X	
Test5 (ważność 4)		X	
Test6 (ważność 2)			X

Załóżmy teraz, że wykonano wszystkie sześć testów. Założymy też, że testy 1, 2, 4 są zdane, a 3, 5, 6 — niezdane.

Przymijmy następującą definicję pokrycia wymagania: Pokrycie wymagania W to suma ważności testów związanych z wymaganiem W, które są zdane, w stosunku do sumy ważności wszystkich testów związanych z W.

Przykładowo, z wymaganiem Req 1 związane są testy 1 i 2 o sumie ważności $2 + 5 = 7$. Ponieważ oba testy są zdane, pokrycie wymagania Req 1 wynosi $7/7 = 100\%$. Pokrycia wymagań obliczamy zatem następująco:

$$\text{Req 1: pokrycie} = (2 + 5)/(2 + 5) = 7/7 = 100\%$$

$$\text{Req 2: pokrycie} = (1)/(1 + 1 + 4) = 1/6 = 16,6\%$$

$$\text{Req 3: pokrycie} = (5)/(5 + 1 + 2) = 5/8 = 62,5\%$$

Oczywiście to, jak dokładnie wyglądać będzie wynik pokrycia wymagań testami, zależy od przyjętej definicji pokrycia. Równie dobrze moglibyśmy np. zdefiniować je jako stosunek testów zdanych (związkowych z wymaganiem W) do wszystkich testów związanych z wymaganiem W. Wtedy pokrycie wymagań Req 1, Req 2 i Req 3 wynosiłoby, odpowiednio: $2/2 = 100\%$, $1/3 = 33,3\%$, $1/3 = 33,3\%$.

Do projektowania i wykonywania testów funkcjonalnych mogą być potrzebne specjalne umiejętności lub specjalistyczna wiedza, taka jak znajomość konkretnego problemu biznesowego rozwiązywanego przez dane oprogramowanie (np. oprogramowanie do modelowania geologicznego dla przemysłu naftowego i gazowego) bądź konkretnej roli, jaką spełnia dane oprogramowanie (np. gra komputerowa zapewniająca interaktywną rozrywkę). Testy funkcjonalne dotyczą bowiem *funkcji*, które dostarcza system, a te zazwyczaj osadzone są w konkretnym kontekście biznesowym. Dlatego np. testerzy aplikacji finansowych powinni przynajmniej znać podstawy finansów.

2.2.2.2. Testowanie niefunkcjonalne

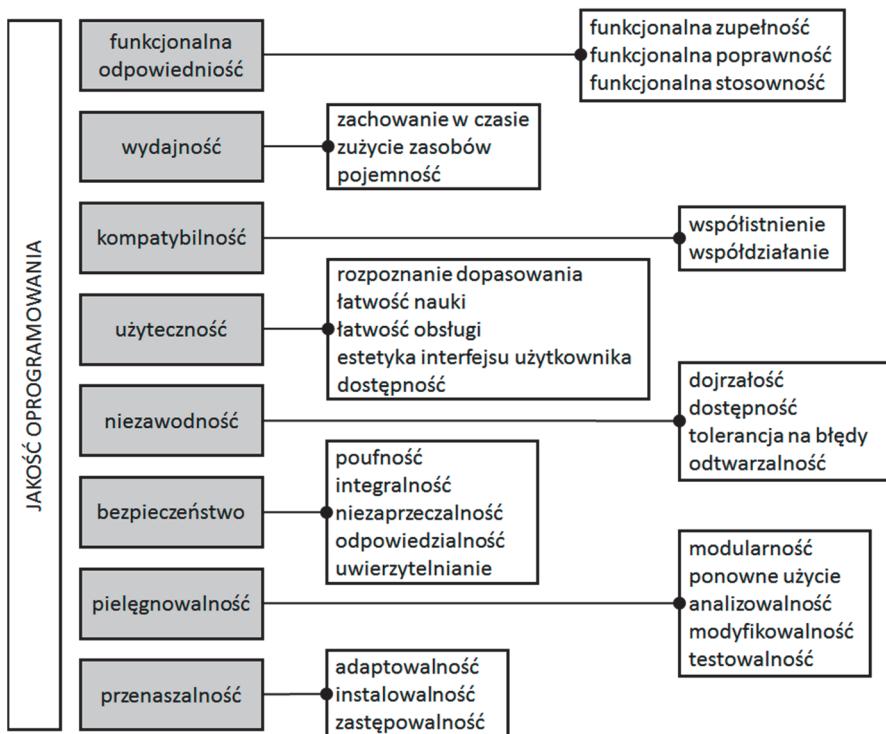
Celem **testowania niefunkcjonalnego** (ang. *non-functional testing*) jest dokonanie oceny charakterystyk systemów i oprogramowania, takich jak: użyteczność, wydajność, bezpieczeństwo. Klasifikację charakterystyk jakościowych oprogramowania zawiera standard ISO/IEC 25010 (Systems and software Quality Requirements and Evaluation (SQuaRE)).



Testowanie niefunkcjonalne sprawdza to, jak zachowuje się dany system.

Na rysunku 2.16 przedstawiony jest model jakości według normy ISO/IEC 25010. Wyróżnia on osiem charakterystyk jakościowych (szare prostokąty), a do każdej z nich przyporządkowuje odpowiednie podcharakterystyki. Z punktu widzenia systemu poziomu podstawowego charakterystykami niefunkcjonalnymi są wszystkie poza funkcjonalną odpowiedniością. Modele jakości, takie jak ISO/IEC 25010, mogą być pomocne w określeniu, jakie parametry niefunkcjonalne naszego systemu powinny zostać poddane testom.

Wbrew mylnemu przekonaniu testowanie niefunkcjonalne można i często należy wykonywać na wszystkich poziomach testów. Ponadto powinno ono odbywać się na jak najwcześniejjszym etapie, ponieważ zbyt późne wykrycie defektów niefunkcjonalnych może być bardzo dużym zagrożeniem dla powodzenia projektu.



RYSUNEK 2.16. Model jakości ISO/IEC 25010 Systems and software Quality Requirements and Evaluation (SQuaRE)

Tabela 2.5 pokazuje przykładowe testy niefunkcjonalne wspomnianego wcześniej systemu do wprowadzania danych przez urzędy skarbowe, które to testy można prowadzić na poszczególnych poziomach testów.

TABELA 2.5. Poziomy testów a testy niefunkcjonalne

POZIOM TESTU	PRZYKŁADOWY TEST
Modułowe	Sprawdzenie, czy moduł da się w prosty sposób wymienić na inny o równoważnej funkcjonalności (przenaszalność: zastępowałość)
Integracyjne systemów	Sprawdzenie, czy połączenie i komunikacja między systemem klienckim a serwerem centralnym są odpowiednio zabezpieczone i nienarażone na ewentualne ataki hakerskie (bezpieczeństwo: poufność)
Systemowe	Sprawdzenie, czy system dostatecznie szybko generuje raport zbiorczy na podstawie 100 000 formularzy PIT (wydajność: zachowanie w czasie)
Akceptacyjne	Sprawdzenie, czy system dostosowany jest do potrzeb osób niedowidzących (użyteczność: dostępność)

Do wyprowadzania warunków testowych i przypadków testowych na potrzeby testowania niefunkcjonalnego można używać technik czarnoskrzynekowych (patrz podrozdział 4.2). Przykładem może być zastosowanie analizy wartości brzegowych do zdefiniowania warunków skrajnych dotyczących testów wydajnościowych.

Staranność testowania niefunkcjonalnego można zmierzyć na podstawie pokrycia niefunkcjonalnego. Termin „pokrycie niefunkcjonalne” oznacza stopień, w jakim został przetestowany określony typ elementu niefunkcjonalnego, wyrażony jako procent elementów danego typu pokrytych przez testy. Dzięki możliwości śledzenia powiązań między testami a typami urządzeń obsługiwanych przez aplikację mobilną można na przykład obliczyć, jaki procent urządzeń został uwzględniony w ramach testowania kompatybilności, a w rezultacie zidentyfikować ewentualne luki w pokryciu.

Należy pamiętać o tym, że wyniki testów niefunkcjonalnych muszą być „konkretnie”, tzn. powinny dać się wyrazić w terminach jakichś dobrze zdefiniowanych miar. Jest to spowodowane tym, że w trakcie wykonania testów musimy dokonać porównania wyniku rzeczywistego z wynikiem oczekiwany i stwierdzić *jednoznacznie*, czy test został zdany, czy też nie. W tabeli 2.6 podajemy przykłady metryk dla różnych charakterystyk jakościowych według modelu ISO/IEC 25010.

TABELA 2.6. Przykładowe metryki dla charakterystyk niefunkcjonalnych

CHARAKTERYSTYKA	METRYKA	WZÓR	OPIS
Niezawodność	Gęstość awarii na liczbę testów	$M = A/B$	A = liczba wykrytych awarii B = liczba wykonanych testów
	Dostępność	$M = A/(A + B)$	A = średni czas do awarii (MTTF) B = średni czas do naprawy (MTTR)
Użyteczność	Łatwość nauki funkcji	$M = T$	T = czas nauki
	Dostępność pomocy	$M = A/B$	A = liczba zadań, dla których istnieje poprawny temat pomocy B = całkowita liczba przetestowanych zadań
Wydajność	Atrakcyjność interakcji	kwestionariusz	Metryka mierzona za pomocą kwestionariusza po użyciu oprogramowania
	Czas odpowiedzi	$M = T_1 - T_2$	T1 = czas ukończenia zadania T2 = czas ukończenia wydawania polecenia wykonania zadania
	Przepustowość	$M = A$	A = liczba wykonanych zadań w ustalonej jednostce czasu
Pielegnowalność	Zdolność do analizy awarii	$M = 1 - A/B$	A = liczba awarii, których przyczyna jest wciąż nieznana B = liczba wszystkich zaobserwowanych awarii

TABELA 2.6. Przykładowe metryki dla charakterystyk niefunkcjonalnych – ciąg dalszy

CHARAKTERYSTYKA	METRYKA	WZÓR	OPIS
Przenaszałość	Możliwość współdziałania	$M = A/T$	A = liczba nieoczekiwanych problemów bądź awarii podczas równoległego użytkowania innego oprogramowania T = całkowity czas, w którym użytkowno równolegle inne oprogramowanie

Do projektowania i wykonywania testów niefunkcjonalnych mogą być potrzebne specjalne umiejętności lub specjalna wiedza — taka jak znajomość słabych punktów charakterystycznych dla danego projektu lub danej technologii (np. podatności na zabezpieczenia związanych z określonymi językami programowania) bądź konkretnej grupy użytkowników (np. profili użytkowników systemów do zarządzania placówkami opieki zdrowotnej). Dlatego częstą praktyką jest np. wynajmowanie specjalistycznych, zewnętrznych organizacji, które specjalizują się w przeprowadzaniu testów określonego typu, np. testów bezpieczeństwa czy wydajności.

Szczegółowe informacje na temat testowania niefunkcjonalnych charakterystyk jakościowych zawierają syllabusy [ISTQB TA 2021, ISTQB TTA 2021, ISTQB SEC 2016] i inne specjalistyczne syllabusy ISTQB®.

2.2.2.3. Testowanie białośkrzynkowe

W przypadku **testowania białośkrzynkowego** (ang. *white-box testing*) testy wyprowadza się na podstawie struktury wewnętrznej lub implementacji danego systemu. Struktura wewnętrzna może obejmować kod, architekturę, przepływy pracy i/lub przepływy danych w obrębie systemu (patrz podrozdział 4.3, gdzie dokładnie opisujemy techniki i kryteria pokryć białośkrzynkowych obowiązujących na egzaminie poziomu podstawowego).



Staranność testowania białośkrzynkowego można zmierzyć na podstawie pokrycia strukturalnego. Termin „pokrycie strukturalne” oznacza stopień, w jakim został przetestowany określony typ elementu strukturalnego, wyrażony jako procent elementów danego typu pokrytych przez testy. Ogólna metryka pokrycia wyraża się zatem wzorem:

$$\text{pokrycie} = A/B,$$

gdzie A = liczba strukturalnych elementów pokrytych przez testy, B = liczba wszystkich elementów strukturalnych. Przykładem może być metryka pokrycia instrukcji, która jest ilorazem liczby pokrytych testami instrukcji wykonywalnych oraz wszystkich wykonywalnych instrukcji w danym kodzie.

Metryka pokrycia zdefiniowana tak jak powyżej przyjmuje wartości z przedziału od 0 do 1. Częstszą formą prezentacji tej metryki jest jej postać procentowa:

$$\text{pokrycie} = (A/B) \cdot 100\%.$$

Na przykład jeśli pokryliśmy naszymi testami 5 spośród 20 wykonywalnych instrukcji kodu, to pokrycie = $5/20 = 0,25$ albo inaczej: $(5/20) \cdot 100\% = 25\%$.

Na poziomie testowania modułów pokrycie kodu określa się na podstawie procentowej części kodu danego modułu, która została przetestowana. Wartość tę można mierzyć w kategoriach różnych aspektów kodu (przedmiotów pokrycia), takich jak procent instrukcji wykonywalnych lub procent wyników decyzji przetestowanych w danym module. Powyższe rodzaje pokrycia nazywa się zbiorczo pokryciem kodu. Na poziomie testowania integracji modułów testowanie białośkrzynkowe może odbywać się na podstawie architektury systemu (np. interfejsów między modułami), a pokrycie strukturalne może być mierzone w kategoriach procentowego udziału przetestowanych interfejsów. Tabela 2.7 przedstawia przykładowe struktury, które można wykorzystać do przeprowadzenia testów białośkrzynkowych na różnych poziomach testów.

TABELA 2.7. Poziomy testów a testy białośkrzynkowe

POZIOM TESTÓW	PRZYKŁADOWA STRUKTURA DO POKRYCIA
Modułowe	Kod źródłowy (np. instrukcje, decyzje)
Integracyjne modułów	Graf wywołań (tzn. zdarzenia wywołania jednej funkcji przez inną), zestaw funkcji API
Systemowe	Model procesu biznesowego
Integracyjne systemów	Graf wywołań na poziomie usług oferowanych przez komunikujące się ze sobą systemy
Akceptacyjne	Model struktury menu

Do projektowania i wykonywania testów białośkrzynkowych mogą być potrzebne specjalne umiejętności lub specjalna wiedza, np. znajomość budowy kodu (umożliwiająca np. użycie narzędzi do pomiaru pokrycia kodu), wiedza o sposobie przechowywania danych (pozwalająca np. ocenić zapytania do baz danych) bądź sposób korzystania z narzędzi do pomiaru pokrycia i poprawnego interpretowania generowanych przez nie rezultatów.

2.2.2.4. Testowanie czarnoskrzynkowe

Testowanie czarnoskrzynkowe (ang. *black-box testing*) jest oparte na specyfikacji (czyli informacji zewnętrznej wobec przedmiotu testów). Głównym celem testowania czarnoskrzynkowego jest sprawdzenie, czy zachowanie systemu jest zgodne z zachowaniem opisanym w specyfikacji.



Więcej informacji o testowaniu czarnoskrzynkowym przedstawiamy w podrozdziale 4.2, gdzie omawiamy czarnoskrzynkowe techniki testowania.

2.2.2.5. Poziomy a typy testów

Poziomy i typy testów są od siebie niezależne. Chociaż w praktyce zdarza się tak, że pewne typy testów występują najczęściej na pewnych określonych poziomach testów, to w ogólności każdy typ testów może być przeprowadzony na dowolnym poziomie testów. Pokazuje to następujący przykład.

Przykład. Testujemy aplikację webową e-sklepu sprzedającego książki. Zarejestrowani użytkownicy po zalogowaniu mają dostęp do katalogu z funkcją wyszukiwania (w tym wyszukiwania po tytule, autorze, kategorii książki, zakresie cenowym). Mogą dodawać wybrane pozycje do koszyka, a następnie przejść do płatności. Poniżej opisano kilka przykładowych testów funkcjonalnych, niefunkcjonalnych, czarnoskrzynkowych i białośkrzynkowych (dla uproszczenia podajemy jedynie krótkie opisy testów, nie opisując dokładnie przypadków testowych).

Przykłady testów funkcjonalnych (i zarazem czarnoskrzynkowych):

- test sprawdzający poprawność rejestracji użytkownika w systemie;
- test sprawdzający, czy możliwe jest zarejestrowanie użytkownika o już istniejącej w systemie nazwie;
- test sprawdzający, czy możliwe jest zarejestrowanie użytkownika o istniejącym już w systemie adresie e-mailowym;
- test sprawdzający standardowy zakup i poprawność płatności;
- test sprawdzający, czy w momencie wyboru książki do koszyka egzemplarz ten jest blokowany w systemie i inny użytkownik nie może go dodać do swojego koszyka;
- test sprawdzający, czy po określonym, ustalonym czasie, jeśli nie nastąpi płatność, sesja wygasza, a wszystkie książki z koszyka trafiają z powrotem do sklepu i stają się dostępne dla pozostałych użytkowników.

Przykłady testów niefunkcjonalnych:

- test sprawdzający czas oczekiwania na odpowiedź po skonstruowaniu wyszukiwania w katalogu dla katalogu liczącego milion pozycji;
- test sprawdzający podatność na atak SQL injection³ przez pole wyszukiwania;
- test sprawdzający czas reakcji systemu w sytuacji jednoczesnego wykonania zapytania przez tysiąc użytkowników.

Przykłady testów białośkrzynkowych:

- test sprawdzający pokrycie instrukcji modułu do rejestracji klienta (intencja: sprawdzenie wszystkich ścieżek oraz możliwych wyjątków i błędów, jakie system powinien obsługiwać);
- test sprawdzający działanie wszystkich elementów menu głównego (pokrycie struktury strony webowej);
- test sprawdzający poprawność działania wszystkich interfejsów związanych z systemem e-płatności.

Każdy z wymienionych powyżej typów testów można wykonywać na dowolnym poziomie testów. Aby zilustrować tę prawidłowość, poniżej przedstawiono przykłady testów funkcjonalnych, niefunkcjonalnych, czarnoskrzynkowych i białośkrzynkowych, które są wykonywane na wszystkich poziomach testów w odniesieniu do aplikacji bankowej. Pierwsza grupa to testy funkcjonalne (i jednocześnie czarnoskrzynkowe):

³ SQL injection to atak na stronę lub aplikację internetową, w którym kod w języku SQL jest dodawany do pola w formularzu w celu zdobycia dostępu do konta lub zmiany danych.

- na potrzeby testowania modułowego projektowane są testy odzwierciedlające sposób, w jaki dany moduł powinien obliczać odsetki składane;
- na potrzeby testowania integracji modułów projektowane są testy odzwierciedlające sposób, w jaki informacje na temat konta pozyskane w interfejsie użytkownika są przekazywane do warstwy logiki biznesowej;
- na potrzeby testowania systemowego projektowane są testy odzwierciedlające sposób, w jaki posiadacze rachunków mogą składać wnioski o przyznanie linii kredytowej na rachunku bieżącym;
- na potrzeby testowania integracji systemów projektowane są testy odzwierciedlające sposób, w jaki dany system sprawdza zdolność kredytową posiadacza rachunku przy użyciu zewnętrznej mikrousługi;
- na potrzeby testowania akceptacyjnego projektowane są testy odzwierciedlające sposób, w jaki pracownik banku zatwierdza lub odrzuca wniosek kredytowy.

Następna grupa zawiera przykłady testów niefunkcjonalnych:

- na potrzeby testowania modułowego projektowane są testy wydajnościowe, które oceniają liczbę cykli procesora niezbędnych do wykonania złożonych obliczeń dotyczących łącznej kwoty odsetek;
- na potrzeby testowania integracji modułów projektowane są testy zabezpieczeń, które mają na celu wykrycie podatności zabezpieczeń związanych z przepełnieniem bufora danymi przekazywanymi z interfejsu użytkownika do warstwy logiki biznesowej;
- na potrzeby testowania systemowego projektowane są testy przenaszalności, które sprawdzają, czy warstwa prezentacji działa we wszystkich obsługiwanych przeglądarkach i urządzeniach przenośnych;
- na potrzeby testowania integracji systemów projektowane są testy niezawodności, które oceniają odporność systemu w przypadku braku odpowiedzi ze strony mikrousługi służącej do sprawdzania zdolności kredytowej;
- na potrzeby testowania akceptacyjnego projektowane są testy użyteczności, które pozwalają ocenić ułatwienia dostępu dla osób niepełnosprawnych zastosowane w interfejsie do przetwarzania kredytów po stronie banku.

Kolejna grupa zawiera przykłady testów białoskrzynkowych:

- na potrzeby testowania modułowego projektowane są testy, których celem jest zapewnienie pełnego pokrycia instrukcji kodu i decyzji we wszystkich modułach wykonujących obliczenia finansowe;
- na potrzeby testowania integracji modułów projektowane są testy, które sprawdzają, w jaki sposób każdy ekran interfejsu wyświetlanego w przeglądarce przekazuje dane do następnego ekranu i do warstwy logiki biznesowej;
- na potrzeby testowania systemowego projektowane są testy, których celem jest zapewnienie pokrycia możliwych sekwencji stron internetowych wyświetlanych podczas składania wniosku o przyznanie linii kredytowej;

- na potrzeby testowania integracji systemów projektowane są testy, które sprawdzają wszystkie możliwe typy zapytań wysyłanych do mikrousługi służącej do sprawdzania zdolności kredytowej;
- na potrzeby testowania akceptacyjnego projektowane są testy, których celem jest pokrycie wszystkich obsługiwanych struktur i zakresów wartości dla plików z danymi finansowymi używanych w przelewach międzybankowych.

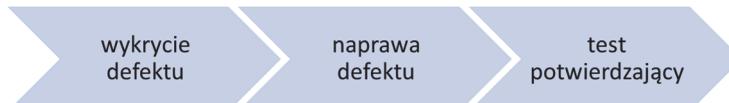
W tym punkcie przedstawiono przykłady wszystkich typów testów wykonywanych na wszystkich poziomach, jednak nie każde oprogramowanie wymaga uwzględnienia każdego typu na każdym poziomie. Ważne jest to, aby na poszczególnych poziomach zostały wykonane odpowiednie testy — dotyczy to zwłaszcza najwcześniejszego poziomu, na jakim występuje dany typ testów.

2.2.3. Testowanie potwierdzające i testowanie regresji

Po wprowadzeniu w systemie zmian mających na celu usunięcie defektów bądź dodanie lub zmodyfikowanie funkcjonalności należy przeprowadzić testy, które potwierdzą, że wprowadzone zmiany faktycznie spowodowały naprawienie defektu lub poprawne zaimplementowanie odpowiednich funkcjonalności, a przy tym nie wywołały żadnych nieprzewidzianych, niekorzystnych konsekwencji. Sylabus poziomu podstawowego rozróżnia dwa rodzaje takich testów: testy potwierdzające (zwane również retestami) oraz testy regresji.

Testowanie potwierdzające

Testowanie potwierdzające (inaczej: retestowanie, ang. *confirmation testing*) wykonywane jest w związku z wystąpieniem awarii oraz naprawą związanego z nią defektu. Rolą testu potwierdzającego jest sprawdzenie, czy defekt został rzeczywiście naprawiony (patrz rysunek 2.17). Retest to ten sam test, który ujawnił awarię, wykonywany ponownie, po jej naprawie, ale czasami testem potwierdzającym może być inny test — na przykład jeśli zgłoszonym defektem był brak funkcjonalności. Absolutnym minimum podczas wykonywania testów potwierdzających jest wykonanie kroków, które wywołały wcześniej awarię. Jeśli test jest zdany, możemy przyjąć, że defekt został naprawiony.

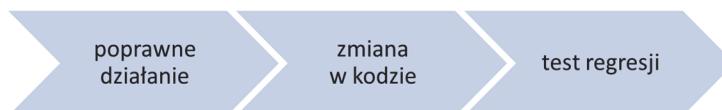


RYSUNEK 2.17. Testowanie potwierdzające

Ponieważ nigdy nie wiadomo, kiedy zdarzy się awaria i kiedy nastąpi jej naprawa, wykonania retestów nie można z góry zaplanować! Należy jednak podczas planowania (patrz rozdział 5.) przydzielić na testy potwierdzające odpowiednio dużo czasu.

Testowanie regresji

Testowanie regresji (ang. *regression testing*) sprawdza, czy zmiana (poprawka lub modyfikacja) wprowadzona w jednej części kodu nie wpłynie przypadkowo na zachowanie innych części kodu w tym samym module, w innych modułach tego samego systemu, a nawet w innych systemach (patrz rysunek 2.18). Ponadto należy wziąć pod uwagę zmiany dotyczące środowiska, takie jak wprowadzenie nowej wersji systemu operacyjnego lub systemu zarządzania bazami danych. Powyższe niezamierzone skutki uboczne są nazywane *regresjami* (od słowa „*regres*”, oznaczającego „pogorszenie się czegoś”), a testowanie regresji polega na ponownym wykonaniu testów w celu ich wykrycia.



RYSUNEK 2.18. Testowanie regresji

Ponieważ testy regresji najczęściej wykonuje się w momencie dodania kolejnej funkcjonalności bądź na koniec każdej iteracji, testy te można zazwyczaj — w przeciwieństwie do testów potwierdzających — doskonale zaplanować. Ponieważ testy regresji są wykonywane często, a ponadto ewoluują raczej wolno, są naturalnymi kandydatami do automatyzacji. Automatyczne wykonywanie testów regresji oszczędza czas i wysiłek, ponieważ testerzy mogą zaoszczędzony czas przeznaczyć na inne działania, na przykład tworzenie i wykonywanie nowych testów. Automatyzację tego rodzaju testów należy rozpocząć na wczesnym etapie projektu.

Częstym, praktycznym problemem związanym z testami regresji jest bardzo szybki wzrost ich liczby. Po pewnym czasie ich wykonanie trwa na tyle długo, że realizuje się w nocy, a wyniki czekają na testerów następnego dnia rano. Czasami jednak zestaw testów regresji nie jest w stanie wykonać się nawet przez całą noc. W takim przypadku należy podjąć decyzję o optymalizacji wykonania testów regresji. Możliwe rozwiązania to np.:

- **priorytetyzacja** — wykonanie najpierw najważniejszych, najistotniejszych testów; te, które nie zdążą się wykonać, są mniej ważne, więc ryzyko niezauważenia poważnej awarii jest niewielkie;
- **redukcja** — usunięcie części testów z zestawu testów regresji; kryterium usunięcia może np. opierać się na zdolności testu do wykrywania awarii — jeśli jakiś test wykrywa awarie dosyć często, zdecydowanie powinien zostać; jeśli zaś jakiś test nie wykrył nigdy żadnej awarii, może być po jakimś czasie usunięty z zestawu testów regresji;
- **strategia mieszana** — najważniejsze testy wykonują się planowo, natomiast ta część, która nie jest w stanie się wykonać ze względów czasowych, wykonywana jest rzadziej, np. co którąś iterację.

Testowanie potwierdzające i testowanie regresji można wykonywać na wszystkich poziomach testów. Zwłaszcza w iteracyjnych i przyrostowych modelach wytwarzania oprogramowania (np. w modelu zwinnym) nowe funkcjonalności, zmiany dotyczących czasowych funkcjonalności oraz refaktoryzacja powodują częste zmiany kodu, które

pociągają za sobą konieczność przeprowadzenia odpowiednich testów. Z uwagi na ciągłą ewolucję systemu testowanie potwierdzające i testowanie regresji są bardzo istotne, szczególnie w przypadku systemów związanych z Internetem rzeczy (IoT), w których poszczególne obiekty (np. urządzenia) są często aktualizowane lub wymieniane.

Przykład. Podczas testowania e-sklepu z książkami (patrz poprzedni punkt) tester doprowadził do awarii: system pozwolił na zarejestrowanie użytkownika o tym samym adresie e-mailowym, który posiada już inny, uprzednio zarejestrowany użytkownik. Nazwijmy test, który to wykrył, testem A. Tester zgłosił raport o defekcie i zajął się innymi czynnościami. W szczególności wykonał test B (próba zapłaty za standardowe zamówienie), który został zdany. W kolejnej iteracji wdrożono nowy moduł do raportowania operacji związanych z płatnościami. Po tych zmianach tester wykonał ponownie test B, aby sprawdzić, czy moduł ten nie wpływał na działanie samej funkcjonalności płatności. Chwilę później tester otrzymał komunikat, że zgłoszony przez niego wcześniej defekt został naprawiony. Tester wykonał ponownie test A, który tym razem został zdany, i zamknął zgłoszenie defektu.

W powyższym przykładzie dwa razy wykonano test A i test B, w kolejności: A, B, B, A.

- pierwsze wykonanie testu A oraz pierwsze wykonanie testu B nie dotyczą ani testowania regresji, ani testowania potwierdzającego, ponieważ testy te wykonano po raz pierwszy;
- drugie wykonanie testu B to przykład testowania regresji, ponieważ intencją testera jest sprawdzenie, czy wdrożenie nowego modułu nie zepsuło czegoś w module płatności;
- drugie wykonanie testu A to przykład testowania potwierdzającego, ponieważ intencją testera jest sprawdzenie, czy test A, który wywołał wcześniej awarię, teraz, po naprawce, zostanie zdany.

Testy związane ze zmianami można wykonywać na wszystkich poziomach testów. Odwołując się do przykładu aplikacji bankowej z punktu 2.2.2.4, możemy wyróżnić następujące przykłady testów związanych ze zmianami:

- na potrzeby testowania modułowego projektowane są automatyczne testy regresji dotyczące poszczególnych modułów (testy te zostaną następnie uwzględnione w strukturze ciągłej integracji);
- na potrzeby testowania integracji modułów projektowane są testy służące do potwierdzania skuteczności poprawek związanych z defektami w interfejsach w miarę umieszczenia takich poprawek w repozytorium kodu;
- na potrzeby testowania systemowego ponownie wykonywane są wszystkie testy dotyczące danego przepływu pracy, jeśli którykolwiek z ekranów objętych tym przepływem pracy uległ zmianie;
- na potrzeby testowania integracji systemów codziennie wykonywane są ponownie testy aplikacji współpracującej z mikrousługą do sprawdzania zdolności kredytowej (w ramach ciągłego wdrażania tej mikrousługi);
- na potrzeby testowania akceptacyjnego wszystkie wcześniej niezaliczone testy są wykonywane ponownie po usunięciu defektu wykrytego w ramach testowania akceptacyjnego.

2.3. Testowanie pielęgnacyjne

FL-2.3.1 (K2) Kandydat podsumowuje testowanie pielęgnacyjne i zdarzenia je wyzwalające.

Moment wydania systemu do klienta to nie koniec, lecz dopiero początek pracy nad produktem. Jakkolwiek wielu menedżerów z zakresu IT nie zdaje sobie z tego sprawy, z licznych badań z zakresu empirycznej inżynierii oprogramowania wynika, że koszt utrzymania systemu jest zazwyczaj wyższy niż koszt jego wytworzenia (sic!). Jest to spowodowane prostym faktem: użytkowanie oprogramowania trwa zwykle o wiele dłużej niż jego wytworzenie, a podczas jego użytkowania może zdarzyć się wiele różnych sytuacji wymagających wprowadzania do oprogramowania zmian lub poprawek. Przykładami takich sytuacji są:

- konieczność wprowadzenia do systemu łatki w związku z odkryciem podatności na atak;
- dodawanie, usuwanie lub modyfikowanie funkcjonalności programu;
- naprawa defektu, który spowodował awarię zgłoszoną przez użytkownika;
- konieczność archiwizacji danych w związku z wycofaniem oprogramowania z użytku;
- konieczność przeniesienia systemu do nowego środowiska, spowodowana np. podniesieniem wersji systemu operacyjnego.

Po wdrożeniu w środowisku produkcyjnym oprogramowanie lub system wymagają zatem dalszej pielęgnacji. Różnego rodzaju zmiany, takie jak opisane powyżej, są praktycznie nieuniknione. Ponadto pielęgnacja jest niezbędna do utrzymania lub poprawy wymaganych niefunkcjonalnych charakterystyk jakościowych oprogramowania lub systemu przez cały cykl jego życia — zwłaszcza w zakresie wydajności, kompatybilności, niezawodności, bezpieczeństwa i przenaszalności.

Dobrze jest pamiętać, że testy pielęgnacyjne wykonywane są *po wydaniu* oprogramowania, w trakcie jego użytkowania. Testy wykonywane przed wydaniem oprogramowania do klienta nie mogą być uznane za testy pielęgnacyjne. Pielęgnacja z definicji dotyczy już użytkowanego produktu.

Po dokonaniu każdej zmiany w fazie pielęgnacji należy wykonać **testowanie pielęgnacyjne** (ang. *maintenance testing*), którego celem jest zarówno sprawdzenie, czy zmiana została wprowadzona pomyślnie, jak i wykrycie ewentualnych skutków ubocznych (np. regresji) w niezmienionych częściach systemu (czyli zwykle w większości jego obszarów). W związku z tym testowanie pielęgnacyjne obejmuje zarówno te części systemu, które zostały zmienione, jak i części niezmienione, na które zmiany mogły mieć wpływ. Pielęgnacja może być wykonywana zarówno planowo (w związku z nowymi wersjami), jak i w sposób niezaplanowany (w związku z poprawkami doraźnymi — ang. *hotfix*).



W świetle powyższych rozważań jest jasne, że najczęstszym typem testowania pielęgnacyjnego będzie testowanie regresji. Nie jest to jednak jedyny typ testów

pielęgnacyjnych. Na przykład jeśli oprogramowanie ma zostać wymienione na inne, należy przeprowadzić testy archiwizacji bądź migracji danych. To mogą być zupełnie nowe testy, które będziemy wykonywać dopiero pierwszy raz w całej, być może wieloletniej, historii użytkowania aplikacji.

Wydanie pielęgnacyjne może wymagać wykonania testów pielęgnacyjnych na wielu poziomach testów i z wykorzystaniem różnych typów testów, zależnie od zakresu wprowadzonych zmian.

Na zakres testowania pielęgnacyjnego wpływają między innymi:

- poziom ryzyka związanego ze zmianą (np. stopień, w jakim zmieniony obszar oprogramowania komunikuje się z innymi modułami lub systemami);
- wielkość dotychczasowego systemu;
- wielkość wprowadzonej zmiany.

Istnieje kilka powodów, dla których wykonuje się pielęgnację oprogramowania, a tym samym testowanie pielęgnacyjne. Dotyczy to zarówno zmian planowanych, jak i nieplanowanych. Zdarzenia wywołujące pielęgnację można podzielić na następujące kategorie:

- **Modyfikacja.** Ta kategoria obejmuje między innymi zaplanowane udoskonalenia (np. w postaci nowych wersji oprogramowania), zmiany korekcyjne i awaryjne, zmiany środowiska operacyjnego (np. planowe uaktualnienia systemu operacyjnego lub bazy danych), uaktualnienia oprogramowania do powszechniej sprzedaży (COTS) oraz poprawki usuwające defekty i podatności zabezpieczeń.
- **Migracja.** Ta kategoria obejmuje między innymi przejście z jednej platformy na inną, co może wiązać się z koniecznością przeprowadzenia testów produkcyjnych nowego środowiska i zmienionego oprogramowania bądź testów konwersji danych (w przypadku migracji danych z innej aplikacji do pielęgnowanego systemu).
- **Wycofanie.** Ta kategoria dotyczy sytuacji, w której aplikacja jest wycofywana z użytku.

Kiedy aplikacja lub system są wycofywane, może to wymagać testowania migracji lub archiwizacji danych, jeśli zachodzi potrzeba ich przechowywania przez dłuższy czas. Testowanie procedur odzyskiwania/pozyskiwania po archiwizacji przez dłuższy czas może także być konieczne. Dodatkowo konieczne może być uwzględnienie testów regresji, aby zapewnić dalszą prawidłową pracę funkcjonalności pozostających w użyciu.

W przypadku systemów związanych z Internetem rzeczy (IoT) testowanie pielęgnacyjne może być konieczne po wprowadzeniu w systemie zupełnie nowych lub zmodyfikowanych elementów, takich jak urządzenia sprzętowe czy usługi programowe. Podczas testowania pielęgnacyjnego takich systemów szczególny nacisk kładzie się na testowanie integracyjne na różnych poziomach (np. na poziomie sieci i aplikacji) oraz na aspekty związane z zabezpieczeniami, szczególnie w zakresie danych osobowych.

Przykład. Firma produkuje system typu CRM (zarządzanie relacjami z klientami, ang. *Customer Relationship Management*). System ten oparty jest na relacyjnej bazie danych, która nie została zbyt dobrze zaprojektowana — zawierała sporo redundanckich danych. Aplikacja odwoływała się do bazy poprzez zestaw ściśle określonych funkcji, a nie bezpośrednio poprzez konstruowanie zapytań. Architekt aplikacji zdecydował, że struktura bazy danych zostanie poprawiona — sprowadzona zostanie do tzw. trzeciej postaci normalnej. Poprawka będzie wymagać instalacji na każdym terminalu użytkownika.

Jest to zdarzenie wywołujące pielęgnację, związane z modyfikacją (zmiany korekcyjne) i w pewnym sensie z migracją (gdyż następuje tu migracja bazy danych do bazy o zmodyfikowanej strukturze). Dlatego też należy przeprowadzić następujące testy pielęgnacyjne związane z instalacją poprawki (przed jej wprowadzeniem):

- testy regresji związane bezpośrednio z wykorzystywaniem funkcji komunikujących się z bazą danych;
- testy migracji i odzyskiwania danych oraz powrotu do starej struktury bazy na wypadek, gdyby okazało się, że z jakichś powodów nowa struktura bazy danych na komputerze klienta nie działa tak, jak powinna.

Analiza wpływu

Analiza wpływu (ang. *impact analysis*) pozwala ocenić zmiany wprowadzone w wersji pielęgnacyjnej pod kątem zarówno skutków zamierzonych, jak i spodziewanych lub potencjalnych skutków ubocznych, a także umożliwia zidentyfikowanie obszarów systemu, na które będą miały wpływ wprowadzone zmiany. Ponadto może pomóc w zidentyfikowaniu wpływu zmiany na dotychczasowe testy. Skutki uboczne zmiany oraz obszary systemu, na które może ona wpływać, należy przetestować pod kątem regresji, przy czym czynność ta może być poprzedzona aktualizacją istniejących testów, na które oddziałuje dana zmiana.

Analizę wpływu można przeprowadzić przed dokonaniem zmiany, aby ustalić, czy zmianę tę należy faktycznie wprowadzić (z uwagi na potencjalne konsekwencje dla innych obszarów systemu).

Przeprowadzenie analizy wpływu może być utrudnione, jeśli:

- specyfikacje (np. wymagania biznesowe, historyjki użytkownika, architektura) są nieaktualne lub niedostępne;
- przypadki testowe nie zostały udokumentowane lub są nieaktualne;
- nie stworzono możliwości dwukierunkowego śledzenia powiązań między testami a podstawą testów;
- wsparcie narzędziowe nie istnieje lub jest niewystarczające;
- zaangażowane osoby nie dysponują wiedzą z danej dziedziny i/lub na temat danego systemu;
- podczas wytwarzania oprogramowania poświęcono zbyt mało uwagi jego charakterystyce jakościowej w zakresie utrzymywalności.

Trudności w przeprowadzeniu analizy wpływu są dosyć powszechnie. Wynika to stąd, że pielęgnację wykonuje się często wiele miesięcy lub lat po wydaniu oprogramowania i zwykle nie ma już w organizacji ludzi, którzy tworzyli pierwotną wersję systemu, a dokumentacja jest nieaktualna i przestarzała.

Przykład. Konieczne jest wdrożenie poprawki do oprogramowania, związanej ze zmianą przepisów prawa dotyczących naliczania podatku. Zmiana dotyczy modułu X. Przy wykorzystaniu dwukierunkowego śledzenia okazuje się, że moduł X powiązany jest z ryzykami R1, R3, R6 i R8, z których wszystkie mają bardzo wysoki poziom (są krytyczne). Pozostałe ryzyka — R2, R4, R5 i R7 mają niski poziom (są pomijalne). Ponadto moduł X komunikuje się z czterema innymi modułami spośród wszystkich ośmiu modułów systemu. Ta prosta analiza wpływu pokazuje, że zmiana jest obarczona dużym ryzykiem, a jej wpływ na system jest znaczący.

Pytania testowe do rozdziału 2.

Pytanie 2.1

(FL-2.1.1, K2)

Jesteś testerem w projekcie polegającym na stworzeniu oprogramowania autopilota do samolotów. Projekt prowadzony jest zgodnie z modelem V.

Które z poniższych zdań NAJLEPIEJ opisuje konsekwencje związane z wyborem tego modelu?

- A. Nacisk w testowaniu położony zostanie na testowanie oparte na doświadczeniu.
- B. Zespół testerski nie będzie przeprowadzał testów statycznych.
- C. Po ukończeniu pierwszej iteracji dostępny będzie działający prototyp systemu.
- D. Testerzy w początkowych fazach uczestniczą w przeglądach wymagań, analizie i projektowaniu testów.

Wybierz jedną odpowiedź.

Pytanie 2.2

(FL-2.1.2, K1)

Jedna z dobrych praktyk testowania mówi, że każdej czynności wytwórczej powinna odpowiadać związana z nią czynność testowa. Który model cyklu wytwarzania WPROST opisuje tę zasadę?

- A. Model iteracyjny.
- B. Model spiralny Boehma.
- C. Scrum.
- D. Model V.

Wybierz jedną odpowiedź.

Pytanie 2.3

(FL-2.1.3, K1)

Które z podejść „najpierw test” wykorzystuje kryteria akceptacji historyjek użytkownika jako podstawę do wyprowadzania przypadków testowych?

- A. Wytwarzanie sterowane testami (TDD — ang. *Test-Driven Development*).
- B. Wytwarzanie sterowane testami akceptacyjnymi (ATDD — ang. *Acceptance Test-Driven Development*).
- C. Wytwarzanie oparte na cechach (FDD — ang. *Feature-Driven Development*).
- D. Wytwarzanie sterowane zachowaniem (BDD — ang. *Behavior-Driven Development*).

Wybierz jedną odpowiedź.

Pytanie 2.4

(FL-2.1.4, K2)

Które z poniższych stwierdzeń opisuje wsparcie TESTOWANIA przez podejście DevOps?

- A. DevOps umożliwia skrócenie cyklu wydania oprogramowania poprzez zastosowanie automatycznej generacji danych testowych dla testów modułowych.
- B. DevOps umożliwia szybką informację zwrotną do programisty o jakości kodu wysłanego przez niego do repozytorium.
- C. DevOps umożliwia automatyczną generację przypadków testowych dla nowego kodu wysłanego przez programistę do repozytorium.
- D. DevOps umożliwia skrócenie czasu potrzebnego na planowanie wydania oraz planowanie iteracji.

Wybierz jedną odpowiedź.

Pytanie 2.5

(FL-2.1.5, K2)

Które z poniższych jest przykładem zastosowania podejścia przesunięcie w lewo (ang. *shift-left*)?

- A. Zastosowanie wytwarzania sterowanego testami akceptacyjnymi (ATDD — ang. *Acceptance Test-Driven Development*).
- B. Oparcie testowania na testach eksploracyjnych.
- C. Tworzenie prototypów graficznego interfejsu użytkownika w fazie pozyskiwania wymagań.
- D. Ciągłe monitorowanie jakości produktu po wydaniu go do klienta.

Wybierz jedną odpowiedź.

Pytanie 2.6

(FL-2.1.6, K2)

Które z poniższych zdań najlepiej opisuje działania testera uczestniczącego w spotkaniu retrospektynym?

- A. Tester podnosi jedynie kwestie dotyczące testowania. Wszystkie inne tematy zostaną poruszone przez któregoś z pozostałych uczestników.
- B. Tester jest obserwatorem, upewniającym się, że spotkanie przebiega zgodnie z zasadami retrospektyny oraz wartościami metodyk zwinnych.
- C. Tester dostarcza informację zwrotną oraz inne informacje dotyczące wszystkich czynności wykonanych przez zespół podczas ukończonej iteracji.
- D. Tester zbiera informacje przekazane przez innych uczestników spotkania, aby na ich podstawie zaprojektować testy dla kolejnej iteracji.

Wybierz jedną odpowiedź.

Pytanie 2.7

(FL-2.2.1, K2)

Testujesz system autopilota w samolocie. Chcesz przeprowadzić testy poprawności komunikacji pomiędzy modułem geolokalizacji i modułem kontrolera silnika.

Które z poniższych będzie NAJLEPSZYM przykładem podstawy testów do zaprojektowania tych testów?

- A. Szczegółowy projekt modułu geolokalizacji.
- B. Projekt architektury.
- C. Raport z analizy ryzyka.
- D. Regulacje prawne z obszaru awioniki/prawa lotniczego.

Wybierz jedną odpowiedź.

Pytanie 2.8

(FL-2.2.2, K2)

Który z poniższych jest przykładem testu niefunkcjonalnego?

- A. Pokrycie określonej kombinacji warunków i obserwowanie wykonanych akcji, aby sprawdzić poprawność implementacji pewnej reguły biznesowej.
- B. Pokrycie wyniku PRAWDA (true) w decyzji „IF ($x > 5$) THEN...” w kodzie.
- C. Sprawdzenie, czy system poprawnie waliduje składnię adresu e-mailowego wpisywanego przez użytkownika w formularzu rejestracji użytkownika.
- D. Sprawdzenie, czy czas logowania do systemu jest mniejszy niż 5 ms w sytuacji, gdy jednocześnie zalogowanych jest już 1000 użytkowników.

Wybierz jedną odpowiedź.

Pytanie 2.9

(FL-2.2.3, K2)

Które z poniższych testów zazwyczaj nie mogą być zaplanowane z góry?

- A. Testy regresji.
- B. Operacyjne testy akceptacyjne (OAT — ang. *Operational Acceptance Testing*).
- C. Testy akceptacyjne użytkownika (UAT — ang. *User Acceptance Testing*).
- D. Testy potwierdzające.

Wybierz jedną odpowiedź.

Pytanie 2.10

(FL-2.3.1, K2)

Podczas procesu testowania oprogramowania typu IoT (Internet rzeczy) odkryto defekt, ale nie został on naprawiony ze względu na bliski czas wydania oprogramowania. Po wydaniu go nie spowodował do tej pory u klienta awarii. Miesiąc po wydaniu zespół zdecydował się na naprawę tego defektu.

Z którym ze zdarzeń wywołujących pielegnację mamy do czynienia w tej sytuacji?

- A. Aktualizacja oprogramowania.
- B. Migracja oprogramowania.
- C. Modyfikacja oprogramowania.
- D. Dodanie nowej funkcjonalności do systemu.

Wybierz jedną odpowiedź.

ROZDZIAŁ 3.

Testowanie statyczne

Słowa kluczowe

analiza statyczna — proces oceny modułu lub systemu bez jego wykonania, na podstawie jego formy, struktury, zawartości lub dokumentacji.

anomalia — sytuacja odbiegająca od oczekiwania.

inspekcja — typ formalnego przeglądu przeprowadzanego w celu identyfikacji problemów w produkcie prac, dostarczającego danych pomiarowych pozwalających na ulepszenie procesu przeglądu oraz procesu wytwarzania oprogramowania.

przegląd — typ testowania statycznego, podczas którego produkt pracy lub proces jest oceniany przez jedną lub więcej osób w celu wykrycia problemów i wprowadzenia ulepszeń.

przegląd formalny — typ przeglądu, który przebiega według zdefiniowanego procesu, z formalnie udokumentowanym wynikiem.

przegląd nieformalny — rodzaj przeglądu bez formalnego procesu i formalnie udokumentowanych wyników.

przegląd techniczny — typ formalnego przeglądu przeprowadzany przez wykwalifikowany zespół technicznych specjalistów, którzy sprawdzają jakość produktu prac oraz identyfikują odchylenia od specyfikacji i standardów.

przejrzenie — typ przeglądu, w którym autor przechodzi z członkami przeglądu przez produkt pracy, a członkowie zadają pytania i zgłaszają uwagi na temat ewentualnych problemów.

testowanie dynamiczne — testowanie, podczas którego wykonywany jest kod modułu lub systemu.

testowanie statyczne — testowanie produktu prac bez uruchamiania kodu.

3.1. Podstawy testowania statycznego

FL-3.1.1 (K1)	Kandydat rozpoznaje typy produktów, które mogą być badane przy użyciu poszczególnych technik testowania statycznego.
FL-3.1.2 (K2)	Kandydat wyjaśnia korzyści wynikające z testowania statycznego.
FL-3.1.3 (K2)	Kandydat porównuje i zestawia ze sobą testowanie statyczne i dynamiczne.

Testowanie statyczne (ang. *static testing*) to zbiór metod i technik testowania, w ramach których nie uruchamia się (nie wykonuje się) testowanego modułu lub systemu. Oczywiście testowanie statyczne może dotyczyć innych artefaktów niż oprogramowanie, np. projektu, dokumentacji, specyfikacji. Wtedy również, z definicji, testowanie takie ma formę statyczną.



Cele testowania statycznego obejmują w szczególności ulepszanie jakości, wykrywanie defektów oraz ocenę takich charakterystyk jak: czytelność, kompletność, poprawność, testowalność czy spójność produktu pracy poddanego przeglądowi. Testowanie statyczne może więc być stosowane zarówno w celu weryfikacji, jak i weryfikacji produktu pracy.

W zwinnych metodach wytwarzania oprogramowania testerzy, przedstawiciele biznesu (klienci, użytkownicy) oraz programiści pracują wspólnie podczas tworzenia wymagań (np. pisania historyjek użytkownika, tworzenia przykładów wytwarzania oprogramowania czy sesji udoskonalania backlogu). Celem tej wspólnej pracy jest upewnienie się, że wymagania użytkownika i związane z nimi produkty pracy spełniają określone kryteria, np. definicję gotowości (patrz punkt 5.1.3). Techniki przeglądu mogą być stosowane w celu zapewnienia, że wymagania są kompletnie, zrozumiałe i testowalne oraz — w przypadku historyjek użytkownika — zawierają testowalne kryteria akceptacji. Zadając właściwe pytania, testerzy badają, kwestionują i pomagają ulepszyć proponowane wymagania.

Testowanie statyczne dzieli się zwyczajowo na dwie zasadnicze podgrupy:

- analizę statyczną (ang. *static analysis*);
- przeglądy (ang. *reviews*).

Analiza statyczna polega na dokonywaniu oceny testowanego produktu pracy (najczęściej kodu, wymagań lub dokumentów projektowych) za pomocą narzędzi. Sylabus poziomu podstawowego nie opisuje szczegółowo metod analizy statycznej — więcej szczegółów Czytelnik znajdzie w sylabusie poziomu zaawansowanego „Techniczny Analytyk Testów” [ISTQB TTA 2021]. Przykładowe techniki analizy statycznej to:



- pomiar złożoności cyklomatycznej kodu (skomplikowania struktury);
- analiza przepływu sterowania;
- analiza przepływu danych;
- kontrola zgodności typów zmiennych, weryfikacja poprawności stosowania standardów pisania kodu (np. nazewnictwo zmiennych).

Analiza statyczna może zidentyfikować problemy przed **testowaniem dynamicznym** (ang. *dynamic testing*) i wymagać przy tym mniejszego wysiłku, ponieważ nie są konieczne przypadki testowe i analiza ta jest zwykle wykonywana za pomocą narzędzi. Analiza statyczna jest często włączana do frameworków ciągłej integracji jako jeden z kroków automatycznego potoku wdrożeń (patrz punkt 2.1.4). Chociaż w dużej mierze używana jest do wykrywania konkretnych defektów kodu, analiza statyczna jest również szeroko stosowana do oceny utrzymywalności i podatności kodu na ataki na zabezpieczenia.



Przeglądy są o wiele częściej wykorzystywanymi technikami testowania statycznego. Dlatego w podrozdziale 3.2 omówimy je dokładniej.

3.1.1. Produkty pracy badane metodą testowania statycznego

Techniki testowania statycznego mogą być stosowane praktycznie wobec dowolnego produktu pracy. Sylabus wymienia wiele takich przykładowych artefaktów poddawanych testom statycznym. Poniżej podajemy nieco rozszerzoną listę:

- wszelkiego rodzaju specyfikacje (biznesowe, wymagań funkcjonalnych, wymagań niefunkcjonalnych itp.);
- opowieści (ang. *epics*), historyjki użytkownika, kryteria akceptacji oraz inne typy dokumentacji wykorzystywanej w projektach zwinnych;
- projekt architektury, specyfikacje projektowe;
- kod źródłowy — być może z wyłączeniem kodu źródłowego napisanego przez osoby trzecie, do którego nie mamy dostępu (np. w przypadku projektów komercyjnych);
- testalia, w tym: plany testów, procedury testowe, przypadki testowe, skrypty testów automatycznych, dane testowe, dokumenty analizy ryzyka, karty opisu testów;
- podręczniki użytkownika, w tym wbudowana pomoc online, podręczniki operatora systemu, instrukcje instalacji, noty wydania itp.;
- strony internetowe (pod kątem ich treści, struktury, użyteczności itp.);
- dokumenty projektowe np.: umowy, plany projektów, harmonogramy, budżety.

Sylabus zaznacza, że o ile przegląd może dotyczyć w zasadzie każdego produktu pracy, to aby miało to sens, uczestnicy tego przeglądu muszą być w stanie przeczytać i przeanalizować ten produkt ze zrozumieniem. Dodatkowo, w przypadku analizy statycznej, produkty pracy sprawdzane tą techniką muszą posiadać formalną strukturę, w oparciu o którą odbywa się testowanie. Przykładami takich formalnych struktur są:

- kod źródłowy (ponieważ każdy program pisany jest w języku programowania zdefiniowanym w oparciu o formalną gramatykę);
- modele (np. diagramy UML posiadające określoną składnię i formalne zasady ich tworzenia);
- dokumenty tekstowe (ponieważ tekst może być sprawdzany np. pod kątem zgodności z zasadami gramatycznymi danego języka).

Analizę statyczną najczęściej stosuje się wobec sformalizowanych produktów pracy, np. formalnych modeli architektury systemu czy wymagań (np. pisanych w językach specyfikacji takich jak Z czy UML). W przypadku dokumentów pisanych w języku naturalnym analiza statyczna może dotyczyć np. sprawdzania czytelności, składni, gramatyki, interpunkcji czy ortografii.

3.1.2. Korzyści wynikające z testowania statycznego

Testowanie statyczne nie jest zwykle tanie (wymaga ręcznego sprawdzania produktu), ale — przeprowadzone poprawnie — jest skuteczne i efektywne. Pozwala bowiem znajdować bardzo wcześnie w cyklu wytwarzania defekty i problemy, które byłyby trudno wykrywalne w fazach późniejszych. Zwykle dotyczy to błędów projektowych. Ten typ defektów jest bardzo „zdradliwy”, gdyż niewykryty we wstępnych fazach defekt projektowy bardzo łatwo propaguje się do kolejnych faz, a na podstawie błędного projektu powstaje błędna implementacja. Problem zwykle uwidacznia się na etapie testów systemowych lub akceptacyjnych. Wtedy usunięcie takiego defektu projektowego może być bardzo drogie.

Dobrze przeprowadzone testy statyczne przed wykonaniem testów dynamicznych powodują, że szybko (i w miarę tanio) usuwa się problemy, które mogłyby być źródłem innych defektów. W rezultacie w testach dynamicznych wykrywanych jest mniej problemów, zatem całkowity koszt testów dynamicznych jest niższy niż w przypadku, gdyby testy statyczne nie były wykonywane. Takie wykorzystanie testowania statycznego jest zgodne z zasadą wczesnego testowania (zob. podrozdział 1.3).

Testowanie statyczne daje możliwość oceny jakości i budowania zaufania do prze-glądanego produktu pracy. Interesariusze mogą ocenić, czy udokumentowane wymagania opisują ich rzeczywiste potrzeby. Ponieważ testowanie statyczne może być wykonane we wczesnym etapie cyklu wytwarzania oprogramowania, tworzy się wspólne zrozumienie produktu i jego wymagań wśród interesariuszy zaangażowanych w testowanie statyczne. To wspólne zrozumienie poprawia również komunikację. Z tego powodu zaleca się zaangażowanie w przeglądy interesariuszy reprezentujących możliwie najszerzsze, najbardziej różnorodne perspektywy spojrzenia na analizowany produkt pracy.

Defekty w kodzie mogą być wykrywane i usuwane przy użyciu testowania statycznego efektywniej niż w przypadku stosowania testów dynamicznych, ponieważ w testach dynamicznych wystąpienie awarii wymaga zwykle żmudnej analizy jej przyczyny oraz poświęcenia dużej ilości czasu na identyfikację defektu ją powodującego. Ta efektywność testowania statycznego zwykle skutkuje zarówno mniejszą liczbą pozostałych w kodzie defektów, jak i niższym ogólnym nakładem pracy.

Efektywność technik statycznych jest potwierdzona empirycznie. Jones i Bonsignour [Jones 2012] podają mnóstwo danych świadczących o tym, że techniki statyczne wydatnie przyczyniają się do zwiększenia ogólnej jakości produktu oraz do obniżenia kosztów jego testowania i utrzymania. W przypadku inspekcji (jeden z rodzajów przeglądów opisany w następnym podrozdziale) efektywność usuwania defektów wzrasta średnio o 85%!

Poniżej przedstawiamy listę korzyści, jakie dają techniki testowania statycznego:

- wczesne, efektywne wykrywanie i usuwanie defektów, jeszcze przed rozpoczęciem testów dynamicznych — możliwość testowania, jeszcze zanim stworzony zostanie działający prototyp oprogramowania;
- identyfikowanie defektów trudnych do wykrycia w późniejszych testach dynamicznych — dotyczy to zwłaszcza defektów projektowych i architektonicznych;
- identyfikowanie defektów niemożliwych do wykrycia w testach dynamicznych, np. lokalizacja nieosiągalnego kodu¹, niepoprawne użycie lub brak użycia wzorców projektowych w kodzie czy też wszelkiego rodzaju defekty w nieuruchamialnych produktach pracy, np. w dokumentacji;
- zapobieganie występowaniu defektów w projekcie i kodzie poprzez wykrywanie niejednoznaczności, sprzeczności, braków, przeoczeń, elementów nadmiarowych czy niespójności w dokumentach takich jak specyfikacja wymagań czy projekt architektury;
- zwiększenie wydajności prac programistycznych; udoskonalenie projektowania i zwiększenie utrzymywalności kodu można wymusić np. narzuceniem jednolitego standardu jego pisania, przez co kod jest łatwiejszy do modyfikacji nie tylko przez twórcę, ale też przez innego programistę, a szansa wprowadzenia defektu podczas modyfikacji kodu zmniejsza się;
- obniżenie kosztów i zmniejszenie czasochłonności wytwarzania oprogramowania, w tym testowania — zwłaszcza testów dynamicznych;
- obniżenie kosztów jakości w całym cyklu wytwarzania oprogramowania poprzez zmniejszenie kosztów w fazie utrzymania, a także zmniejszenie liczby awarii na etapie eksploatacji oprogramowania;
- usprawnienie komunikacji między członkami zespołu poprzez uczestnictwo w przeglądach, w tym w spotkaniach przeglądowych.

Przy ocenie kosztów poniesionych na testowanie wprowadza się pojęcie kosztu jakości. Jest to całkowity koszt poniesiony na działania jakościowe, na który składają się koszty działań prewencyjnych (np. koszty szkoleń), koszty wykrycia (koszt wykonania czynności testowych), koszty awarii wewnętrznych (koszt usunięcia defektów znalezionych przed wydaniem) i koszty awarii zewnętrznych (koszt usunięcia defektów znalezionych po wydaniu).

Mimo że przeglądy mogą być kosztowne w implementacji, całkowite koszty jakości są zwykle znacznie niższe niż w przypadku, gdy przeglądy nie są wykonywane, ponieważ mniej czasu i wysiłku trzeba poświęcić na usuwanie defektów w późniejszym etapie projektu. Uczestnicy procesu przeglądu korzystają również z lepszego wspólnego zrozumienia produktu poddawanego przeglądowi.

¹ Identyfikacja nieosiągalnego kodu jest możliwa tylko w niektórych przypadkach, ponieważ w ogólności problem osiągalności określonego miejsca w kodzie jest tzw. problemem nierostrzygalnym. Oznacza to, że nie istnieje algorytm, który dla dowolnego programu i dowolnego miejsca w jego kodzie odpowiadałby na pytanie, czy istnieją dane wejściowe do programu umożliwiające osiągnięcie tego miejsca w kodzie.

Przykład. Rozważmy prostą, ekonomiczną kalkulację tego, czy wdrożenie testowania statycznego w organizacji się opłaca. W scenariuszu A zespół będzie wykonywać wyłącznie testy dynamiczne. W scenariuszu B testy dynamiczne po przedzone będą testami statycznymi. Poniższy model jest bardzo prosty i ma za zadanie jedynie zilustrować, w jaki sposób testowanie statyczne może zmniejszyć całkowity koszt wytwarzania i utrzymania oprogramowania.

Założenia modelu:

- Zespół testerów liczy 6 osób.
- Miesięczny koszt utrzymania jednego testera wynosi 7000 PLN.
- Testy statyczne i testy dynamiczne trwają po cztery miesiące, bierze w nich udział cały zespół testerski. Testy statyczne (jeśli istnieją) wykonywane są przed dynamicznymi.
- Całkowita liczba defektów w oprogramowaniu wynosi 130.
- W testach statycznych znajdowane jest 50% spośród wszystkich istniejących defektów (dane przemysłowe, patrz np. [Jones 2012]).
- W testach dynamicznych znajdowane jest 80% spośród wszystkich istniejących defektów. Pozostałe 20% to defekty polowe, wykryte przez klienta po wydaniu oprogramowania (wartości te pochodzą z danych historycznych).
- Koszt usunięcia jednego defektu znalezionejego w testach statycznych wynosi 500 PLN (dane przemysłowe).
- Koszt usunięcia jednego defektu znalezionejego w testach dynamicznych to 1800 PLN.
- Koszt usunięcia jednego defektu polowego (znalezionejego przez użytkownika po wydaniu oprogramowania) wynosi 12 600 PLN.

Scenariusz A — bez testów statycznych

Liczba defektów wykrytych w testach dynamicznych: $80\% \cdot 130 = 104$

Liczba defektów polowych: $20\% \cdot 130 = 26$

Koszt testów dynamicznych: $6 \cdot 7000 \text{ PLN} \cdot 4 = 168\,000 \text{ PLN}$

Koszt usuwania defektów przed wydaniem: $104 \cdot 1800 \text{ PLN} = 187\,200 \text{ PLN}$

Koszt usuwania defektów po wydaniu: $26 \cdot 12\,600 \text{ PLN} = 327\,600 \text{ PLN}$

Całkowity koszt jakości: $168\,000 \text{ PLN} + 187\,200 \text{ PLN} + 327\,600 \text{ PLN} = 682\,800 \text{ PLN}$

Scenariusz B — z testami statycznymi

Liczba defektów wykrytych w testach statycznych: $50\% \cdot 130 = 65$

Liczba defektów wykrytych w testach dynamicznych: $80\% \cdot (130 - 65) = 52$

Liczba defektów polowych: $130 - (65 + 52) = 13$

Koszt testów statycznych: $6 \cdot 7000 \text{ PLN} \cdot 4 = 168\,000 \text{ PLN}$

Koszt testów dynamicznych: $6 \cdot 7000 \text{ PLN} \cdot 4 = 168\ 000 \text{ PLN}$

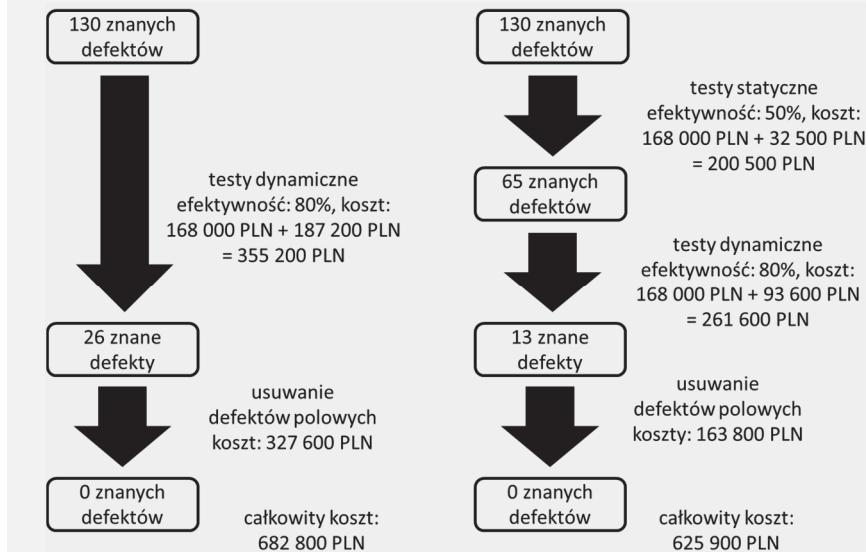
Koszt usuwania defektów znalezionych w testach statycznych: $65 \cdot 500 \text{ PLN} = 32\ 500 \text{ PLN}$

Koszt usuwania defektów przed wydaniem: $52 \cdot 1800 \text{ PLN} = 93\ 600 \text{ PLN}$

Koszt usuwania defektów po wydaniu: $13 \cdot 12\ 600 \text{ PLN} = 163\ 800 \text{ PLN}$

Całkowity koszt jakości: $168\ 000 \text{ PLN} + 168\ 000 \text{ PLN} + 32\ 500 \text{ PLN} + 93\ 600 \text{ PLN} + 163\ 800 \text{ PLN} = 625\ 900 \text{ PLN}$

Schematycznie porównanie scenariuszy pokazane jest na rysunku 3.1.



RYSUNEK 3.1. Porównanie procesów usuwania defektów bez użycia testów statycznych i z ich użyciem

Koszty jakości przed wydaniem są wyższe w scenariuszu B (462 100 PLN w stosunku do 355 200 PLN) ze względu na wysokie koszty inspekcji. Jednak zysk następuje po wydaniu z uwagi na o połowę niższą liczbę defektów polowych w scenariuszu B, dzięki wykorzystaniu przeglądów. Łączny koszt okazuje się więc niższy w scenariuszu B. Dokładna różnica kosztów między scenariuszem A a B wynosi $682\ 800 \text{ PLN} - 625\ 900 \text{ PLN} = 56\ 900 \text{ PLN}$.

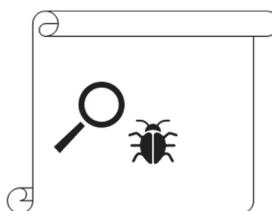
Oczywiście w analizie musimy wziąć pod uwagę błędy oszacowań, a także to, że zespół testerski potrzebował czterech dodatkowych miesięcy na przeprowadzenie testów statycznych. Niemniej analiza wskazuje, że stosowanie testów statycznych może nie tylko znacząco poprawić końcową jakość produktu (26 defektów polowych w stosunku do 13), ale i obniżyć znacząco koszty wytwarzania. W powyższej symulacji zaoszczędziliśmy ok. 57 000 PLN — koszty w scenariuszu B są o ok. 8,3% mniejsze od kosztów w scenariuszu A.

3.1.3. Różnica między testowaniem statycznym a dynamicznym

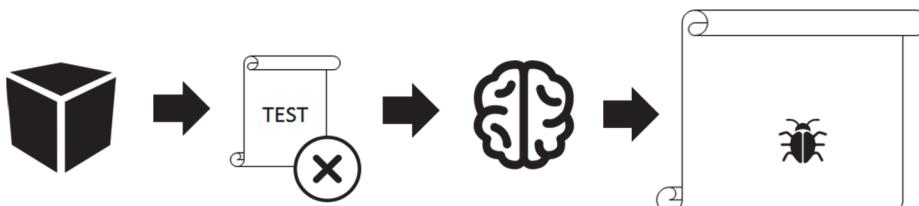
Testowanie statyczne i dynamiczne mają ten sam cel — dokonywanie oceny jakości produktów oraz możliwie jak najwcześniejsze identyfikowanie defektów. Czynności te wzajemnie się uzupełniają, ponieważ umożliwiają wykrycie różnego typu defektów.

Rysunek 3.2 symbolicznie przedstawia zasadniczą różnicę między testowaniem statycznym (górsza część rysunku) a dynamicznym (dolna część rysunku). W przypadku testowania statycznego jesteśmy w stanie bezpośrednio znaleźć defekt w produkcie pracy. Nie jesteśmy natomiast w stanie znaleźć awarii, ponieważ w przypadku technik statycznych z definicji nie uruchamiamy oprogramowania, a awaria może nastąpić wyłącznie w wyniku działania systemu.

W przypadku testów dynamicznych najczęściej pierwszym objawem niepoprawnego działania oprogramowania jest z kolei awaria (skutek niezdania testu). Po zaobserwowaniu awarii uruchamiany jest analityczny proces debugowania, podczas którego analizuje się kod źródłowy i znajduje miejsce, w którym jest defekt odpowiedzialny za wywołanie tej awarii.



Testowanie statyczne: wykrycie defektu



Testowanie dynamiczne: wykrycie awarii, analiza, lokalizacja defektu

RYSUNEK 3.2. Różnica między testowaniem statycznym a dynamicznym

Czasami defekt w produkcie pracy może pozostawać przez bardzo długi czas ukryty, ponieważ nie powoduje awarii. Ponadto ścieżka, na której się znajduje, może być rzadko testowana lub trudno dostępna, przez co nie będzie łatwo zaprojektować i wykonać testu dynamicznego, który go wykryje. Testowanie statyczne może znaleźć defekt przy znacznie mniejszym nakładzie pracy. Z drugiej strony istnieją defekty, które o wiele łatwiej wykryć podczas testów dynamicznych niż statycznych, np. wycieki pamięci.

Inną różnicą jest to, że testowanie statyczne może być wykorzystywane do zwiększenia spójności i jakości wewnętrznej produktów pracy, natomiast testowanie dynamiczne koncentruje się głównie na widocznych z zewnątrz zachowaniach.

Ponadto testowanie statyczne może być stosowane wyłącznie do niewykonywalnych (nieuruchomionych lub niedających się uruchomić) produktów pracy, takich jak kod źródłowy czy dokumentacja. Testowanie dynamiczne z kolei może być wykonywane wyłącznie wobec uruchomionego (wykonanego) produktu pracy, czyli — działającego oprogramowania. Konsekwencją tego faktu jest to, że testowanie dynamiczne może być użyte do pomiaru charakterystyk jakościowych, których pomiar jest niemożliwy w testach statycznych, ponieważ zależą od uruchomionego (działającego) programu. Przykładem mogą być testy wydajnościowe mierzące czas odpowiedzi systemu na określone żądanie użytkownika.

Wspomnieliśmy wcześniej, że testowanie statyczne wykrywa zazwyczaj odmienne defekty niż testowanie dynamiczne. Przykładami typowych defektów, które są łatwiejsze i tańsze do wykrycia oraz usunięcia metodą testowania statycznego niż metodą testowania dynamicznego, są:

- defekty w wymaganiach (takie jak: niespójności, niejednoznaczności, opuszczenia, sprzeczności, nieścisłości, przeoczenia, powtórzenia, elementy nadmiarowe);
- defekty w projekcie (np. nieefektywne algorytmy lub struktury baz danych, wysoki stopień sprzężenia² (ang. *coupling*), mała spójność³ (ang. *cohesion*), zła modularyzacja kodu);
- określone typy defektów w kodzie (np. zmienne z niezdefiniowanymi wartościami, zmienne zadeklarowane, lecz nigdy nieużywane, niedostępny kod, powielony kod, nieefektywnie zaimplementowany algorytm o zbyt dużej złożoności czasowej lub pamięciowej);
- odchylenia od standardów (np. brak zgodności ze standardami tworzenia kodu);
- niepoprawne specyfikacje interfejsów (np. użycie różnych jednostek miary w systemie wywołującym i systemie wywoływanym, niepoprawny typ lub niepoprawna kolejność parametrów przekazywanych w wywołaniu funkcji API);

² Sprzężenie odnosi się do stopnia, w jakim jeden moduł oprogramowania zależy od innego, i wyraża poziom wzajemnej zależności między modułami. Wysokie sprzężenie oznacza, że zmiana w jednym module może wymagać zmiany w innych modułach, co może prowadzić do zwiększenia kosztów utrzymania i rozwijania systemu oraz utrudniać jego modyfikację i rozbudowę. Niskie sprzężenie oznacza, że moduły są bardziej niezależne od siebie, co ułatwia zarządzanie nimi.

³ Spójność odnosi się do stopnia, w jakim elementy wewnętrz modułu są ze sobą powiązane i realizują wspólny cel. Wysoka spójność oznacza, że elementy składowe modułu są skoncentrowane na realizacji jednego celu (zadania lub funkcjonalności), co ułatwia ich zrozumienie, utrzymanie i rozwój. Niska spójność oznacza, że elementy wewnętrz modułu realizują różne cele, co utrudnia ich zrozumienie, wprowadza niepotrzebne zależności między nimi i może prowadzić do zwiększenia kosztów utrzymania i rozwoju systemu oraz do jego słabej czytelności i skalowalności.

- wrażliwości zabezpieczeń (np. podatność na ataki typu przepełnienie bufora, SQL injection, XSS⁴ (ang. *cross-site scripting*), atak DDoS⁵);
- luki lub nieścisłości w zakresie śledzenia powiązań lub pokrycia (np. brak testów odpowiadających kryteriom akceptacji).

Przykład. Zespół testerski chce ocenić łatwość oraz koszt utrzymania oprogramowania po jego wydaniu do klienta. W tym celu zamierza wykorzystać techniki analizy statycznej. Zidentyfikowano trzy metryki, które zostaną wykorzystane do pomiaru parametrów kodu źródłowego:

- LOC — liczba wykonywalnych linii kodu w module;
- KOMENT — procent opatrzonych komentarzami linii kodu w module;
- CYKL — złożoność cyklowatyczna⁶ modułu.

Wyniki przedstawione są w tabeli 3.1.

TABELA 3.1. Wyniki analizy statycznej na użytek utrzymywalności kodu

MODUŁ	LOC	KOMENT	CYKL
main	129	0%	7
split	206	1%	12
to_lower	62	1%	6
to_upper	63	3%	6
merge	70	0%	15
config	42	15%	8
stdio	243	2%	21

Analiza statyczna pozwoliła wyciągnąć następujące wnioski:

- Moduły są raczej niewielkie. Największy z nich, stdio, posiada 243 linie kodu. Rozmiar modułów sam w sobie nie powinien być zatem większym problemem.
- Kod nie jest wystarczająco skomentowany. Poza modułem config, w którym 15% kodu jest skomentowane, we wszystkich pozostałych modułach odsetek skomentowanych linii zwykle jest na poziomie 0 – 3%. Moduł main w ogóle nie zawiera komentarzy. Należy zdecydowanie zwrócić uwagę programistom na ten fakt.

⁴ Cross-Site Scripting (XSS) to forma cyberataku na strony internetowe w celu zhakowania ich użytkowników. Haker umieszcza złośliwy skrypt na pozornie przyjaznej i bezpiecznej stronie internetowej, przez co każda osoba korzystająca z niej może być narażona na atak (wg *home.pl*)

⁵ Atak DDoS polega na przeprowadzeniu ataku równocześnie z wielu miejsc (z wielu komputerów). Atak taki przeprowadzany jest głównie z komputerów, nad którymi przejęta została kontrola przy użyciu specjalnego oprogramowania (np. botów i trojanów) (wg *home.pl*).

⁶ Złożoność cyklowatyczną modułu definiuje się jako liczbę instrukcji decyzyjnych w kodzie tego modułu powiększoną o jeden. Jest to bardzo prosta miara „komplikacji” struktury kodu.

- Dużą (powyżej 10) złożoność cykloomatyczną posiadają trzy moduły: split, merge oraz stdio. Powinny zostać poddane analizie. Dotyczy to zwłaszcza modułu merge, w którym gęstość instrukcji decyzyjnych jest bardzo duża: jedna decyzja na mniej więcej pięć linii kodu, co może sugerować bardzo niską czytelność kodu źródłowego. Brak refaktoryzacji tych modułów może utrudnić utrzymanie oprogramowania w przyszłości.

3.2. Informacje zwrotne i proces przeglądu

FL-3.2.1 (K1)	Kandydat pamięta korzyści wynikające z wczesnego i częstego otrzymywania informacji zwrotnych od interesariuszy.
FL-3.2.2 (K2)	Kandydat podsumowuje czynności wykonywane w ramach procesu przeglądu.
FL-3.2.3 (K1)	Kandydat pamięta, jakie obowiązki są przypisane do najważniejszych ról w trakcie wykonywania przeglądów.
FL-3.2.4 (K2)	Kandydat porównuje i zestawia ze sobą różne typy przeglądów.
FL-3.2.5 (K1)	Kandydat pamięta, jakie czynniki decydują o powodzeniu przeglądu.

W kolejnych punktach omówimy szczegółowo:

1. Zalety wczesnej i częstej informacji zwrotnej od klienta.
2. Proces przeglądu produktów pracy.
3. Role i obowiązki w przeglądach formalnych.
4. Typy przeglądów.
5. Czynniki powodzenia (sukcesu) w przeglądach.

W ostatnim punkcie tego rozdziału (3.2.6) omówimy techniki przeglądania produktów prac. Punkt ten występował w wersji 3.0 sylabusa, ale został usunięty w sylabusie w wersji 4.0. Niemniej postanowiliśmy zostawić go w podręczniku jako nadobowiązkowy, ponieważ omawia ważne zagadnienie, jakim jest praktyczne podejście do samej czynności przeglądania produktu pracy. Etap ten jest centralnym krokiem w procesie przeglądu (patrz punkt 3.2.2), dlatego warto znać podstawowe techniki przeglądania.

3.2.1. Korzyści wynikające z wczesnego i częstego otrzymywania informacji zwrotnych od interesariuszy

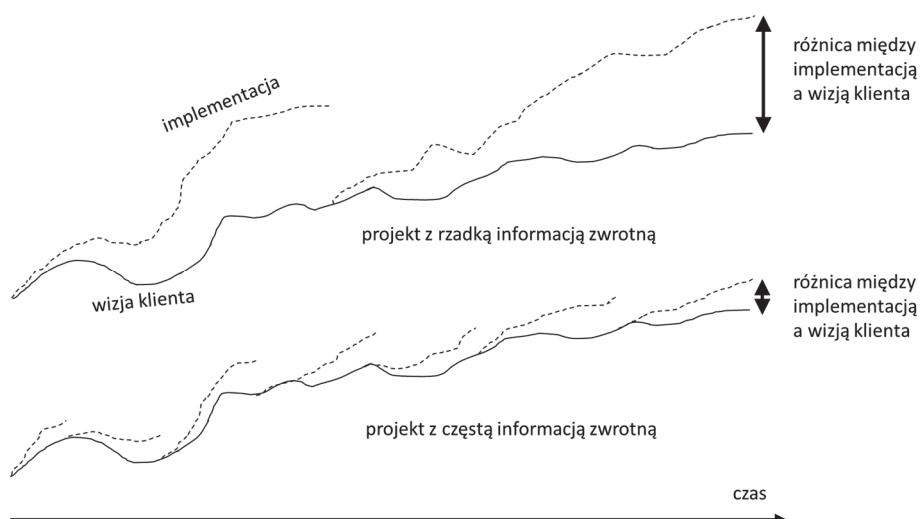
Przeglądy (ang. *reviews*) są formą udzielania wczesnej informacji zwrotnej dla zespołu, ponieważ mogą być wykonywane wcześniej w cyklu tworzenia. Jednak informacja zwrotna nie powinna ograniczać się wyłącznie do przeglądów, lecz stanowić powszechnie stosowaną praktykę w całym cyklu twórczym projektu. Informacja zwrotna może pochodzić od testerów, ale równie istotną dla zespołu twórczego będzie ta pochodząca od samego klienta.



Wczesna i częsta informacja zwrotna pozwala na szybkie informowanie o potencjalnych problemach jakościowych i umożliwia szybką reakcję zespołu na problem. Jeśli zaangażowanie interesariuszy podczas wytwarzania oprogramowania jest niewielkie, rozwijany produkt może nie spełniać ich pierwotnej lub aktualnej wizji. Niedostarczenie tego, czego oczekują interesariusze, może skutkować kosztownymi przeróbkami, niedotrzymaniem terminów, zrzucaniem winy na innych, a nawet doprowadzić do całkowitego niepowodzenia projektu.

Częste przekazywanie informacji zwrotnych przez interesariuszy w trakcie wytwarzania oprogramowania może zapobiec nieporozumieniom dotyczącym wymagań i zapewnić, że zmiany wymagań zostaną zrozumiane i wdrożone wcześniej. Pomaga to zespołowi programistów w lepszym zrozumieniu tego, co budują. Pozwala im skupić się na tych funkcjach, które dostarczają największą wartość dla interesariuszy i które mają najbardziej pozytywny wpływ na uzgodnione ryzyka.

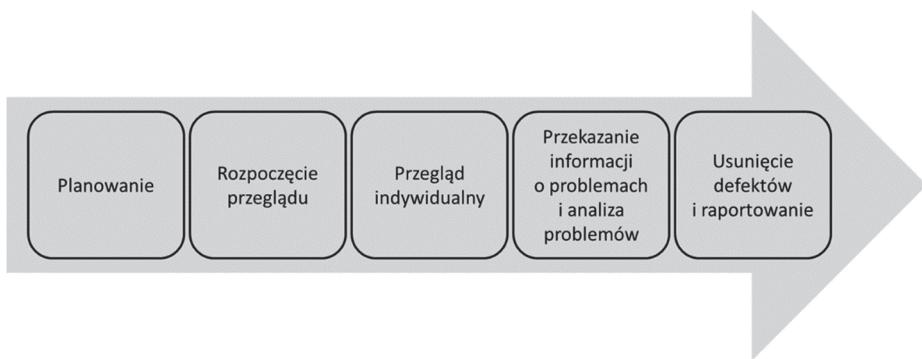
Rysunek 3.3 symbolicznie pokazuje zysk z częstej informacji zwrotnej. Linia ciągła reprezentuje wizję klienta, linia przerywana — to, jak produkt rzeczywiście wygląda w implementacji. Te dwie wizje z czasem coraz bardziej się rozbiegają. W momencach przekazywania informacji zwrotnej implementacja jest dostosowywana do wizji klienta. Górnny wykres prezentuje sytuację, w której kontakt z klientem jest sporadyczny i następuje tylko dwukrotnie. Widać, że efektem jest implementacja, która istotnie odbiega od tego, jaka była wizja klienta. Wykres dolny reprezentuje sytuację, w której klient często udziela informacji zwrotnej dla zespołu twórczego. To skutkuje mniejszymi różnicami między wizją klienta a rzeczywistą implementacją, bo pomiędzy spotkaniami z klientem upływa o wiele mniej czasu niż w przypadku sytuacji z wykresu górnego. W rezultacie końcowy produkt jest o wiele bardziej zbieżny z wizją klienta. Ponadto należy zwrócić uwagę, że koszty „dostosowania” produktu do wizji klienta będą o wiele większe w przypadku rzadkiej informacji zwrotnej ze względu na to, że produkt trzeba będzie zmodyfikować i dostosować w o wiele większym stopniu niż w sytuacji częstej informacji zwrotnej.



RYSUNEK 3.3. Ilustracja zysku z częstej informacji zwrotnej

3.2.2. Czynności wykonywane w procesie przeglądu

Przeglądy różnią się od siebie typem, poziomem sformalizowania czy celami, ale we wszystkich typach przeglądów występują zazwyczaj podobne fazy czy grupy czynności. Norma *Software and systems engineering — Work product reviews* [ISO/IEC 20246] definiuje ogólny proces przeglądu, który zapewnia uporządkowane, ale elastyczne ramy, w ramach których można dostosować konkretny proces przeglądu do danej sytuacji. Jeśli wymagany przegląd jest bardziej formalny, wtedy występować będzie więcej zadań czy elementów tego procesu. Generyczny proces przeglądu opisany w normie ISO/IEC 20246 pokazany jest na rysunku 3.4.



RYSUNEK 3.4. Generyczny proces przeglądu

Rozmiar wielu produktów pracy sprawia, że mogą one być zbyt duże, aby objąć je pojedynczym przeglądem. W takich przypadkach proces przeglądu jest zazwyczaj stosowany wielokrotnie do poszczególnych części składających się na produkt pracy.

Syllabus poziomu podstawowego opisuje pięć generycznych typów czynności pokazanych na rysunku 3.4, jakie mogą wystąpić w procesie przeglądu produktów pracy. Omówimy je poniżej.

Planowanie

W ramach planowania określany jest zakres prac, jakie obejmować będzie przegląd. Planowanie wyznacza granice przeglądu poprzez odpowiedź na pytania typu: kto? co? gdzie? kiedy? i dlaczego?. Określa się cel przeglądu (odpowiedź na pytanie „dla czego?”) oraz to, które dokumenty będą jego przedmiotem, a także definiuje się charakterystyki jakościowe poddawane ocenie (odpowiedź na pytanie „co?”). Szacuje się nakłady pracy potrzebnej do przeprowadzenia przeglądu i definiuje się ramy czasowe oraz miejsce przeprowadzania niektórych faz procesu przeglądu, np. spotkania przeglądowe (odpowiedź na pytania „gdzie?” i „kiedy?”). Na podstawie celu przeglądu określa się jego typ, a co za tym idzie, także i role, czynności oraz stosowane listy kontrolne (jeśli to konieczne). Wybiera się osoby, które mają wziąć udział w przeglądzie, i przydziela się im role (odpowiedź na pytanie „któ?”).

W przypadku **przeglądów formalnych** (ang. *formal review*), takich jak inspekcja, określa się formalne kryteria wejścia i wyjścia, a na zakończenie tej fazy także sprawdza się, czy kryteria wejścia są spełnione i czy można przejść do kolejnej fazy przeglądu.



Rozpoczęcie przeglądu

W ramach rozpoczęcia przeglądu następuje rozesłanie do uczestników przeglądu (wybranych w fazie planowania) produktu pracy poddawanego przeglądowi, wraz ze wszystkimi innymi niezbędnymi materiałami, np. listami kontrolnymi, instrukcjami postępowania, formularzami zgłoszeń problemów. Wszystkie materiały powinny zostać wysłane tak wcześnie, jak tylko to jest możliwe, aby przeglądający mieli wystarczającą ilość czasu na dokonanie przeglądu.

W razie konieczności udziela się uczestnikom wszelkich wyjaśnień dotyczących tego, jak będzie przebiegał przegląd oraz na czym polega ich rola. Jeśli uczestnicy mają jakieś pytania, w fazie rozpoczęcia — zanim nastąpi właściwy przegląd produktu pracy — udziela się im odpowiedzi i wyjaśnień, aby wszyscy dobrze wiedzieli, co mają robić oraz jakie są ramy czasowe dla tych działań.

Istnieje możliwość, aby w fazie rozpoczęcia zorganizować szkolenie z zakresu prowadzenia przeglądów. Ma to sens w sytuacji, gdy uczestnicy nie są doświadczeni w tego typu technikach statycznych, a zależy nam na tym, by proces przeglądu przebiegał sprawnie i efektywnie. W przypadku przeglądów formalnych można również zorganizować wstępne spotkanie, na którym wyjaśni się poszczególnym uczestnikom zakres przeglądu, ich indywidualne role oraz obowiązki jako przeglądających.

Celem fazy rozpoczęcia przeglądu jest upewnienie się, że wszystkie zaangażowane w przegląd osoby są przygotowane i gotowe do jego rozpoczęcia.

Przegląd indywidualny (tj. indywidualne przygotowanie)

To zasadnicza, centralna faza każdego przeglądu. W fazie tej wykonywane są merytoryczne czynności, to znaczy następuje faktyczne przeglądanie produktu pracy przez przeglądających, z wykorzystaniem określonych technik przeglądu (zob. punkt 3.2.6). W ramach tych czynności dokonuje się przeglądu całości lub części produktu pracy (to, jaki fragment ma być poddany przeglądowi, powinno być określone podczas planowania i dokładnie wyjaśnione uczestnikom w fazie rozpoczęcia). Osoby przeglądające zapisują wszelkie uwagi, pytania, zalecenia, wątpliwości oraz istotne obserwacje poczynione podczas przeglądania produktu pracy.

Spośród wszystkich faz przeglądu w fazie przeglądu indywidualnego wykrywany jest największy odsetek problemów. Zidentyfikowane problemy są zazwyczaj dokumentowane w dzienniku problemów, który często jest wspierany przez narzędzie do zarządzania defektami lub narzędzie do wspierania przeglądów.

Według standardu [ISO/IEC 20246] ten krok procesu jest opcjonalny⁷ i może nie być wykonywany dla pewnych typów przeglądu, np. przejrzeń czy przeglądu nieformalnego. Jednakże wielu autorów stoi na stanowisku, że jest to najważniejsza czynność w całym procesie przeglądu.

Przekazanie informacji o problemach i analiza problemów

Jeśli nie planuje się spotkania przeglądowego, znalezione problemy komunikowane są bezpośrednio indywidualnym osobom (np. autorowi przeglądanego produktu pracy) wraz z analizą tych problemów. Jeśli spotkanie przeglądowe jest ujęte w planie przeglądu, problemy raportowane są bezpośrednio uczestnikom spotkania lub, co jest chyba lepszym rozwiązaniem, są one zbiorczo przesyłane do wszystkich uczestników przed spotkaniem, aby zapoznali się z nimi wcześniej, dzięki czemu spotkanie przeglądowe przebiegnie efektywniej.

W ramach spotkania dokonuje się analizy problemów (**anomalii**, ang. *anomaly*) zgłoszonych przez przeglądających. Ponieważ nie każda anomalia musi okazać się defektem, wszystkie zgłoszenia muszą być dokładnie przeanalizowane na spotkaniu, w celu zdecydowania, czy mamy do czynienia z rzeczywistym problemem (defektem), czy też jedynie z problemem pozornym (wynik fałszywie pozytywny). W przypadku, gdy anomalia okaże się defektem, wyznacza się osoby odpowiedzialne za ich naprawę, a także definiuje się takie parametry jak status oraz — w przypadku defektów — priorytet czy dotkliwość. Czynności te dokonywane są na spotkaniu przeglądowym lub (jeśli takiego spotkania nie ma) indywidualnie. Najczęściej stosowane statusy dla zgłoszonych anomalii to:

- problem odrzucony;
- problem zapisany bez podjęcia jakiejkolwiek akcji;
- problem do rozwiązania przez autora produktu pracy;
- problem zaktualizowany w wyniku dalszej analizy;
- problem przypisany do zewnętrznego interesariusza.



W tej fazie dokonuje się również oceny oraz dokumentuje poziom charakterystyk jakościowych, które zdefiniowano w fazie planowania jako te poddawane ocenie. Na koniec dokonuje się oceny wniosków z przeglądu pod kątem kryteriów wyjścia w celu podjęcia decyzji co do tego, co zrobić z przeglądniętym produktem pracy. Przykładowe decyzje to:

- akceptacja produktu pracy (zwykle, gdy nie wykryto żadnych problemów lub wykryto jedynie niewielką liczbę mało znaczących defektów);
- aktualizacja produktu pracy na podstawie zidentyfikowanych problemów (typowa decyzja);
- produkt pracy powinien zostać poddany ponownemu przeglądowi (najczęściej z uwagi na dużą liczbę problemów oraz duży zakres zmian);
- odrzucenie produktu pracy (najrzadszy typ decyzji, ale również może się zdarzyć).

⁷ W takich sytuacjach znajdowanie defektów może odbywać się np. podczas tzw. spotkania przeglądowego omówionego w kolejnym punkcie.

Usunięcie defektów i raportowanie

Jest to końcowy etap przeglądu. Tworzy się w nim raporty o defektach dotyczące wykrytych defektów wymagających wprowadzenia zmian. Oczekuje się, że autor przeglądanej produkcyjnej pracy dokona w tej fazie usunięcia defektów. W tym celu informuje się odpowiednie osoby lub odpowiedni zespół o defektach wykrytych w produkcie pracy związanym z produktem będącym przedmiotem przeglądu. Na koniec, gdy zmiany zostaną potwierdzone, tworzony jest raport z przeglądu.

W przypadku bardziej formalnych typów przeglądu zbiera się ponadto różnego typu miary, które wykorzystuje się do badania efektywności przeglądu. Sprawdza się również, czy zostały spełnione kryteria wyjścia, i akceptuje się produkt pracy po stwierdzeniu ich spełnienia.

Rezultaty przeglądu produktu pracy mogą być różne w zależności od typu przeglądu oraz stopnia jego sformalizowania (patrz punkt 3.2.4).

Przykład. Zespół postanowił wykonać przegląd tzw. drzewka umiejętności (ang. *skill tree*) w produkowanej przez siebie komputerowej grze RPG online. Lider przeglądu wybiera jako uczestników przeglądu pięć osób: dewelopera, testera oraz trzy osoby spoza zespołu, pełniące funkcję zewnętrznych interesariuszy (graczy). Zdecydowano, że typem przeprowadzonego przeglądu będzie indywidualny przegląd nieformalny.

Lider przeglądu rozsyła uczestnikom dokument opisujący drzewko umiejętności, listę kontrolną zawierającą charakterystyki do sprawdzenia oraz formularz dziennika błędów, pokazany w tabeli 3.2.

TABELA 3.2. Formularz dziennika błędów

LP.	STR./LINIA	LISTA KONTROLNA	KATEGORIA DEFEKTU	DOTKLIWOŚĆ DEFEKTU	TYP DEFEKTU	ŹRÓDŁO DEFEKTU	OPIS, UWAGI
.....
.....
.....

Kategorie defektu: Brak, Defekt, Nadmiarowość

Dotkliwość defektu: Duża, Mała

Typ defektu: Interfejs, Funkcjonalny, Dane, Logika, Wydajność, Czynnik ludzki, Standard, Dokumentacja, Składnia, Pielęgnacja, Inny

Każdy z uczestników, w z góry wyznaczonym czasie, dokonuje przeglądu dokumentu i uzupełnia formularz dziennika błędów, a następnie wysyła go do lidera przeglądu. Lider przeglądu scalą formularze w jeden, usuwa redundantne defekty (zaznaczając ewentualnie, że były znalezione przez więcej niż jedną osobę) i tak stworzoną listę problemów przesyła deweloperom, wprowadzając je jednocześnie jako zgłoszenia do systemu zarządzania defektami.

Przykład defektów zgłoszonych przez jednego z przeglądających pokazany jest w tabeli 3.3. Ponieważ przeglądający nie korzystał z listy kontrolnej, nie potrzebował również kolumny Źródło defektu, więc te dwie kolumny zostały pominięte w formularzu.

TABELA 3.3. Wypełniony przez jednego z przeglądających formularz błędów

LP.	STR./LINIA	KATEGORIA DEFEKTU	DOTKLIWOŚĆ DEFEKTU	TYP DEFEKTU	OPIS, UWAGI
1	Rys. 1.	Defekt	Mała	Składnia	„Elektryczna błyskawica” zamiast „Elektryczna błyskawica”
2	Rys. 1.	Defekt	Duża	Logika	Elementem następującym po „Elektryczna błyskawica” powinna być umiejętność „Elektryczna burza”, a nie „Ognista burza”
3	Rys. 1.	Brak	Duża	Logika	Brakuje umiejętności „Oślepienie przeciwnika”, której wprowadzenie do drzewa umiejętności ustaloną na spotkaniu 4 lutego

Autorzy usuwają znalezione defekty i odnotowują to w systemie zarządzania defektami. Lider przeglądu weryfikuje, że wszystkie defekty zostały naprawione. Tworzy wtedy zbiorczy raport z przeglądu, w którym odnotowuje:

- liczbę osób biorących udział w przeglądzie;
- czas trwania przeglądu;
- rozmiar przeglądanego dokumentu (np. liczbę linii kodu, liczbę stron dokumentu);
- liczbę znalezionych defektów w podziale na dotkliwość (duża/mała).

Na tym proces przeglądu się kończy.

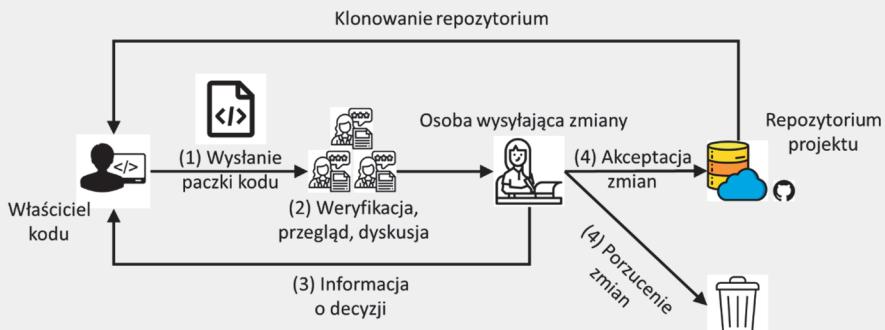
Efektywność spotkań przeglądowych

Syllabus poziomu podstawowego mówi, że spotkanie przeglądowe jest jednym z kroków w procesie przeglądu. Patrząc historycznie na to, jak organizowano przeglądy, można zauważyc, że jeszcze jakiś czas temu tradycyjnie przeglądy wręcz koncentrowały się na spotkaniach przeglądowych, które były główną merytoryczną czynnością w ramach procesu przeglądu. W ostatnich kilkunastu latach badano problem konieczności oraz efektywności tych spotkań. Z badań tych wynika, że generalnie spotkania te nie zwiększą efektywności wykrywania defektów (co zwykle jest najważniejszym celem przeglądów). Spotkania przeglądowe są kosztowne, zatem nie wydaje się, aby przynosiły jakąś szczególną wartość dodaną. Ponadto okazuje się, że jeśli na spotkaniu jest zbyt dużo osób (więcej niż 5), narzuca związek z komunikacją zaczyna mieć negatywny wpływ na proces uczenia się i doskonalenia umiejętności osób biorących udział w spotkaniu.

Z drugiej strony argument za spotkaniami przeglądowymi może być następujący. Spotkania te są znakomitą okazją do rozmowy oraz wymiany doświadczeń, dzieląc się wiedzą czy uzyskania konsensusu. Badania pokazują również, że spotkania przyczyniają się do zmniejszenia liczby wyników fałszywie pozytywnych (zgłoszeń defektów, które okazują się nie być defektami). Zatem wartość dodana może ujawnić się niekoniecznie w liczbie wykrytych defektów, ale w doskonaleniu całego procesu wytwarzanego oraz w podnoszeniu kwalifikacji członków zespołu [Web3].

Modern Code Review (MCR)

Współcześnie wiele firm wykorzystuje proces przeglądu kodu znany pod nazwą „Nowoczesne przeglądy kodu” (ang. *Modern Code Review*, MCR). Schemat tego procesu przedstawiono na rysunku 3.5. MCR to skuteczna technika kontroli jakości, która może weryfikować jakość oprogramowania i zapewnić klientowi zadowolenie poprzez identyfikację defektów, poprawę kodu i przyspieszenie procesu rozwoju. Jest to asynchroniczny i lekki proces przeglądu wspierany przez narzędzia do przeglądu, takie jak Gerrit. Jest to lekka wersja procesu inspekcji Fagana (zob. punkt 3.2.4), która rozwinęła się jako praktyka dla rozwoju oprogramowania open source i przemysłowego [Nazir 2020].



RYSUNEK 3.5. Modern Code Review

3.2.3. Role i obowiązki w przeglądach

W typowym przeglądzie wyróżnia się następujące role (opis ról i lista obowiązków została zaczerpnięta z sylabusa — na końcu podajemy kilka dodatkowych uwag na temat niektórych ról):

Kierownictwo

- odpowiada za zaplanowanie przeglądu;
- podejmuje decyzję o przeprowadzeniu przeglądu;
- wyznacza pracowników oraz określa budżet i ramy czasowe;
- monitoruje na bieżąco opłacalność przeglądu;
- wykonuje decyzje kontrolne w przypadku uzyskania niezadowalających wyników.

Autor

- tworzy produkt pracy będący przedmiotem przeglądu;
- usuwa defekty w produkcie pracy będącym przedmiotem przeglądu (jeśli jest to konieczne).

Autor jest osobą odpowiedzialną za przygotowanie materiałów do przeglądu, choć formalnie materiały te mogą być rozsyłane przez moderatora. Jeśli to konieczne, autor może dostarczać przeglądającym technicznych wyjaśnień dotyczących przeglądanego produktu pracy. Bardzo ważne jest, aby autor rozumiał i akceptował profesjonalny krytycyzm przeglądających pod adresem swojego produktu pracy, a także aby nie bronił się przed uwagami przeglądających ani ich nie negował. Celem przeglądu jest bowiem zazwyczaj wykrycie jak największej liczby problemów, a krytyka nie jest zwrócona w stronę autora (każdy jestomylny), lecz produktu.

Może się jednak zdarzyć, że uwaga przeglądającego jest niezrozumiała dla autora i nie rozumie on, na czym polega problem, a co za tym idzie — nie wie, jak go naprawić. Wtedy może być konieczna komunikacja na linii autor – przeglądający, aby bardziej szczegółowo wyjaśnić wątpliwości przeglądarkiego.

Autor może również ocenić pracę przeglądających w sensie wartości wnioskowanych przez nich uwag. Tego typu informacja zwrotna może być wykorzystana przy planowaniu kolejnych przeglądów w organizacji.

Facilitator (zwany często moderatorem)

- dba o sprawny przebieg spotkań przeglądowych (o ile je prowadzi);
- występuje w roli mediatora, jeśli konieczne jest pogodzenie różnych punktów widzenia;
- dba o stworzenie bezpiecznej atmosfery na spotkaniu przeglądowym, aby przebiegało ono w atmosferze wzajemnego zaufania i szacunku.

Protokolant (zwany też sekretarzem)

- gromadzi potencjalne anomalie wykryte i zgłoszone w ramach przeglądu indywidualnego;
- rejestruje nowe potencjalne defekty znalezione podczas spotkania przeglądowego, a także decyzje podejmowane na tym spotkaniu (gdy ma ono miejsce).

Protokolant powinien pełnić „transparentną” rolę podczas przeglądu, tzn. powinien być „niewidoczny” dla pozostałych uczestników spotkania przeglądowego. Jego zadaniem jest odciążenie pozostałych uczestników od czynności zapisywania wszelkich uwag poczynionych podczas spotkania. Spotkanie przeglądowe wymaga maksymalnego skupienia się przeglądających i autora, a najefektywniej będą oni działać wtedy, gdy nie będą musieli rozpraszać się za każdym razem, gdy zostanie znaleziony nowy problem i zaistnieje konieczność jego zapisania.

Przeglądający

- dokonuje przeglądu, identyfikując potencjalne defekty w produkcie pracy będącym przedmiotem przeglądu;
- może być ekspertem merytorycznym, osobą pracującą przy projekcie, interesariuszem zainteresowanym produktem pracy i/lub osobą mającą określone doświadczenie techniczne lub biznesowe;
- może reprezentować różne punkty widzenia (np. punkt widzenia testera, programisty, użytkownika, operatora, analityka biznesowego, specjalisty ds. użyteczności).

Przeglądający to kluczowa rola w procesie przeglądu, ponieważ osoby pełniące ją znajdują problemy w przeglądany produkcie pracy, a to jest zwykle cel przeglądów. Przeglądający powinni mieć wystarczającą ilość czasu na przygotowanie oraz zgłoszenie znalezionych problemów. Powinni kierować swoje komentarze pod adresem produktu, a nie jego autora.

Lider przeglądu

- ponosi ogólną odpowiedzialność za przegląd;
- decyduje o tym, kto ma wziąć udział w przeglądzie, określa miejsce i termin przeglądu, odpowiada za organizację spotkania przeglądowego.

Role a osoby

W przypadku niektórych typów przeglądów jedna osoba może pełnić kilka ról, a w zależności od typu przeglądu mogą również różnić się czynności związane z poszczególnymi rolami. Ponadto w związku z pojawiением się narzędzi wspomagających proces przeglądu (a zwłaszcza rejestrowania defektów, otwartych punktów i decyzji) często nie ma potrzeby wyznaczania protokolanta. Protokolant nie będzie również potrzebny, jeśli nie planuje się organizacji spotkania przeglądowego.

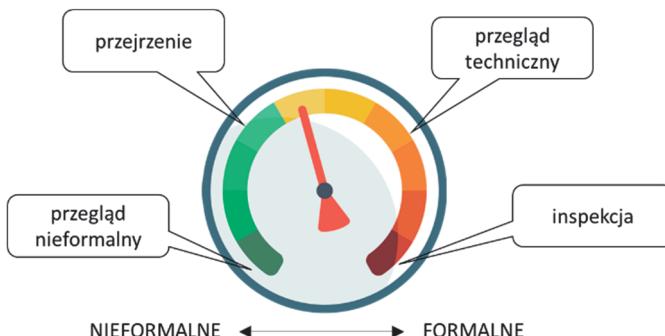
3.2.4. Typy przeglądów

Istnieje wiele rodzajów przeglądów na różnych poziomach formalności, od przeglądów nieformalnych do bardzo sformalizowanych (patrz rysunek 3.6). Wymagany poziom formalności zależy od takich czynników jak stosowany model cyklu tworzenia oprogramowania, dojrzałość procesu rozwoju, krytyczność i złożoność produktu pracy podlegającego przeglądowi, wszelkie wymagania prawne lub regulacyjne oraz potrzeba ścieżki audytu.

Wybór właściwego typu przeglądu jest kluczowy do osiągnięcia wymaganych celów przeglądu. Wybór ten opiera się nie tylko na celach, ale również na takich czynnikach jak potrzeby projektu, dostępne zasoby, rodzaj produktu pracy i ryzyka, domena biznesowa i kultura firmy.

Syłabus poziomu podstawowego opisuje cztery typy przeglądów:

- inspekcję;
- przegląd techniczny;
- przejrzenie;
- przegląd nieformalny.



RYSUNEK 3.6. Poziom formalności różnych typów przeglądu

Inspekcja jest przeglądem formalnym. Przegląd techniczny i przejrzenie mogą przyjmować postać od bardzo sformalizowanych do całkiem nieformalnych. Cechą charakterystyczną przeglądów nieformalnych jest to, że nie przebiegają zgodnie ze zdefiniowanym procesem, a uzyskanych dzięki nim informacji nie trzeba formalnie dokumentować. Z kolei przeglądy formalne są przeprowadzane zgodnie z udokumentowanymi procedurami i z udziałem zespołu o ustalonym wcześniej składzie, a ich rezultaty muszą być obowiązkowo dokumentowane. Stopień sformalizowania przeglądu zależy od takich czynników jak: model cyklu wytwarzania oprogramowania, dojrzałość procesu wytwarzania oprogramowania, złożoność produktu pracy będącego przedmiotem przeglądu, wymogi prawne czy konieczność prowadzenia ścieżki audytu.

Przeglądy mogą być przeprowadzane w różnych celach, ale jednym z głównych celów każdego przeglądu jest ujawnienie defektów. Przeglądy mogą być klasyfikowane według różnych atrybutów. W tabeli 3.4 przedstawiono najczęściej występujące typy przeglądów omówione w syłabusie poziomu podstawowego wraz z odpowiadającymi im atrybutami. Poniżej przedstawimy kilka bardziej szczegółowych informacji na temat tych typów przeglądów.

Kolejność przeglądów

Produkt pracy może być przedmiotem więcej niż jednego typu przeglądu, a w przypadku przeprowadzania kilku przeglądów różnych typów ich kolejność może być zróżnicowana — na przykład przed przeglądem technicznym może zostać przeprowadzony przegląd nieformalny, by upewnić się, że produkt pracy jest gotowy do przeglądu technicznego.

Przeglądy koleżeńskie

Wszystkie typy przeglądów mogą być realizowane jako przeglądy koleżeńskie (ang. *peer review*), czyli przeprowadzane przez współpracowników na zbliżonym szczeblu organizacyjnym bądź o podobnym zakresie odpowiedzialności czy podobnym doświadczeniu.

Typy defektów znajdowanych podczas przeglądów

Typy defektów znalezionych w przeglądach są różne, zależą w szczególności od przeglądanego produktu pracy. Przykłady defektów, które można znaleźć w przeglądach różnych produktów pracy, znajdują się w punkcie 3.1.3, a informacje na temat inspekcji formalnych — w [Gilb 1993].

Przechodzimy teraz do szczegółowego omówienia czterech typów przeglądów opisanych w syllabusie. Zbiorcze porównanie typów omówionych poniżej przeglądów zebrano w tabeli 3.4.

Przegląd nieformalny

Przegląd nieformalny (ang. *informal review*) nie przebiega według żadnego ustalonego czy uprzednio zdefiniowanego procesu. Nie dokumentuje się również w żaden formalny sposób rezultatu przeglądu nieformalnego. Głównym celem tego przeglądu jest wykrycie potencjalnych anomalii.



Przykładem przeglądu nieformalnego może być rozmowa dwóch deweloperów, kiedy jeden z nich prosi drugiego o pomoc w znalezieniu defektu powodującego błąd komplikacji. Innym przykładem może być rozmowa dwóch inżynierów w kuchni podczas lunchu na temat jakiegoś technicznego problemu. Po przeglądzie nieformalnym często nie zostaje żaden ślad, nic nie jest dokumentowane. W metodykach zwinnych to najczęściej spotykany typ przeglądu.

Przykładowe rodzaje przeglądów nieformalnych to:

- sprawdzenie koleżeńskie (ang. *buddy check*);
- przegląd w parach (ang. *pair review*).

TABELA 3.4. Porównanie typów przeglądów

TYP PRZEGLĄDU	PRZEGLĄD NIEFORMALNY	PRZEJRZENIE	PRZEGLĄD TECHNICZNY	INSPEKCJA
Główne cele	Wykrycie potencjalnych defektów	Wykrycie potencjalnych defektów, podniesienie jakości, rozważenie alternatyw, dokonanie oceny zgodności ze standardami	Uzyskanie konsensusu, wykrycie potencjalnych defektów	Wykrycie potencjalnych defektów, dokonanie oceny jakości produktu pracy, zwiększenie zaufania do niego, zapobieganie wystąpieniu podobnych defektów w przyszłości
Potencjalne cele dodatkowe	Generowanie nowych pomysłów, szybkie rozwiązywanie prostych problemów	Wymiana informacji na temat technik lub odmian stylów, przeszkolenie uczestników, osiągnięcie konsensusu	Ocena jakości produktu pracy, zwiększenie zaufania do niego, generowanie nowych pomysłów, motywacja autorów do doskonalenia przyszłych produktów pracy, ocena alternatyw	Motywacja autorów do doskonalenia przyszłych produktów pracy oraz procesu wytwarzania oprogramowania, stworzenie warunków do tego, osiąganie konsensusu
Formalny proces	Brak	Opcjonalne indywidualne przygotowanie przed spotkaniem	Obowiązkowe przygotowanie indywidualne przed spotkaniem	Formalny proces oparty na regułach i listach kontrolnych; obowiązkowe indywidualne przygotowanie przed spotkaniem
Role	Może być prowadzony przez współpracownika autora lub przez grupę osób	Spotkanie prowadzi zwykle autor, wymagana jest obecność protokolanta	Spotkanie powinno być prowadzone przez moderatora, nie przez autora, wymagana obecność protokolanta	Ścisłe zdefiniowane; spotkanie prowadzi facilitator, nie autor; obowiązkowo protokolant; opcjonalnie rola czytającego
Dokumentacja wyników	Opcjonalna	Opcjonalnie tworzone dzienniki defektów i raporty z przeglądu	Na ogół tworzone są dzienniki defektów i raporty z przeglądu	Na ogół tworzone są dzienniki defektów i raporty z przeglądu
Formalizm	Nieformalny	Od nieformalnego po formalny; może mieć formę scenariuszy, przebiegów próbnych (na sucho) lub symulacji	Od nieformalnego po formalny; spotkanie związane z przeglądem opcjonalne	Formalny; obowiązują kryteria wejścia i wyjścia; zbierane są miary wykorzystywane do doskonalenia całego procesu wytwarzczego, w tym procesu inspekcji
Użycie list kontrolnych	Opcjonalne	Opcjonalne	Opcjonalne	Zwykle są używane

Przejrzenie

Przejrzenie (ang. *walkthrough*) polega na sekwencyjnym przeglądaniu produktu pracy. Prowadzone jest przez autora produktu pracy i może obejmować kilka różnych celów, takich jak: ocena jakości produktu pracy, zwiększenie zaufania do produktu pracy, lepsze zrozumienie produktu pracy przez przeglądających, osiągnięcie konsensusu, generowanie nowych pomysłów, motywowanie autorów do tworzenia lepszych produktów i do efektywniejszego znajdowania defektów. Przeglądający mogą dokonać indywidualnego przygotowania przed spotkaniem, na którym odbywa się przejrzenie, ale nie jest to obowiązkowe.



Ten typ przeglądu jest także często stosowany wtedy, gdy zespół nie jest w stanie wykryć przyczyny awarii oprogramowania. Może przyjąć wówczas formę tzw. przebiegów próbnych lub tzw. przebiegów na sucho, zwanych często „zabawą w komputer”. Przebieg próbnego może polegać na ręcznej symulacji wykonywanego kodu. Zespół analizuje wtedy kod dokładnie, linia po linii, aby dobrze zrozumieć, jak on działa i w którym miejscu występuje defekt powodujący awarię.

Przejrzenie prowadzi autor kodu, gdyż jest on zwykle w stanie na bieżąco tłumaczyć, co kod robi (a przynajmniej jakie były jego intencje w tym względzie, gdy ten kod implementował). Jeśli przeglądy kodu (ang. *code review*) przeprowadzane są na spotkaniach przeglądowych, to najczęściej przyjmują formę przejrzenia.

Przegląd techniczny



Celami **przeglądu technicznego** (ang. *technical review*), wykonywanego przez przeglądających o kwalifikacjach technicznych, są uzyskanie konsensusu i podjęcie decyzji dotyczących problemu technicznego, a także wykrycie potencjalnych defektów, ocena jakości i budowanie zaufania do produktu pracy, generowanie nowych pomysłów, motywowanie i umożliwienie autorom doskonalenia ich produktów prac.

Przegląd techniczny ma zazwyczaj formę „panelu ekspertów”, czyli spotkania, na którym kompetentne osoby mają za zadanie podjąć jakąś — zazwyczaj ważną i istotną — decyzję projektową bądź technologiczną, która ma istotny wpływ na dalszy rozwój projektu. Z tego względu obowiązkowe jest indywidualne przygotowanie przed spotkaniem, aby w samym spotkaniu wszyscy mogli uczestniczyć świadomie i w sposób kompetentny.

W ogólności spotkanie przeglądowe w przeglądzie technicznym jest opcjonalne. Jeśli jednak jest organizowane, powinno być prowadzone przez moderatora. Zazwyczaj przegląd techniczny kończy się raportem, gdyż powinien po nim zostać ślad — zwłaszcza jeśli podczas przeglądu podjęto ważne decyzje projektowe.

Inspekcja

Inspekcje (ang. *inspections*) są jednym z najbardziej formalnych typów przeglądów. Ich powstanie datuje się na lata 70. XX wieku, kiedy to Michael Fagan wprowadził je w IBM [Fagan 1976]. Przeprowadzanie inspekcji oprogramowania uważane jest za tzw. dobrą praktykę w inżynierii jakości oprogramowania ze względu na niewątpliwe korzyści, jakie ze sobą niesie. Kierownictwo wyższego szczebla systematycznie podkreśla istotną rolę przeglądów kolejzeńskich (w tym inspekcji) jako kluczowego elementu pozwalającego zapewnić wysoką jakość końcową produktu [Johnson 1996].



Ponieważ inspekcje są najbardziej formalnym rodzajem przeglądu, przebiegają zgodnie z pełnym procesem przeglądu opisany w punkcie 3.2.2. Głównym celem jest uzyskanie maksymalnej wydajności w znajdowaniu defektów. Inne cele to: ocena jakości, budowanie zaufania do produktu pracy oraz motywowanie i umożliwienie autorom doskonalenia ich produktów prac. Cechą szczególną inspekcji jest zbieranie metryk i wykorzystywanie ich do poprawy całego procesu tworzenia oprogramowania, w tym samego procesu inspekcji. W inspekcjach autor nie może pełnić roli lidera przeglądu ani protokolanta.

Inspekcje odbywają się zgodnie ze ścisłe określonym procesem opartym na regulach i listach kontrolnych. Rezultaty inspekcji muszą być udokumentowane. Przykładowa lista kontrolna wykorzystywana podczas inspekcji może mieć taką postać:

Lista kontrolna dla wymagań

Zupełność

- 1.** Czy wyspecyfikowane wymagania odnoszą się do misji projektu w spójny sposób?
- 2.** Czy wymagania zawierają kluczowe i krytyczne potrzeby użytkownika, operatora oraz zespołu wsparcia?
- 3.** Czy każde wymaganie stanowi osobną, niezależną od innych jednostkę specyfikacji bądź ma jasno określone zależności?
- 4.** Czy nie ma braków w wymaganiach?
- 5.** Czy wymagania konieczne są odróżnione od opcjonalnych (ang. nice to have)?

Poprawność

- 1.** Czy wymagania nie są wzajemnie sprzeczne?
- 2.** Czy wymagania są śledzone w przód do projektu i kodu?
- 3.** Czy każde wymaganie posiada unikatowy numer?

Styl

- 1.** Czy każde wymaganie jest zrozumiałe i napisane prostym językiem?
- 2.** Czy w wymaganiach odróżnia się wyraźnie zmiany edytorskie od funkcjonalnych?
- 3.** Czy nazewnictwo oraz definicje pojęć są używane w sposób spójny i jednolity?

Pomiary inspekcji

Choć wiele organizacji wykorzystuje inspekcje, niewiele jest publicznie dostępnych danych o ich efektywności. Jednak na podstawie istniejących informacji (w szczególności na bazie eksperymentu National Software Quality Experiment, przeprowadzonego w 1992 roku) można wyciągnąć następujące wnioski [O'Neill 1998]:

- Kod źródłowy nie jest śledzony wstecz do wymagań, co powoduje utratę „intelektualnej” kontroli, nieprecyzyjność procedur weryfikacyjnych oraz trudność w zarządzaniu zmianą.
- Dobre praktyki pisania kodu są stosowane w sposób nierygorystyczny i niesystematyczny, co powoduje wysoki odsetek defektów w logice, danych, interfejsach i funkcjonalności.
- Projekty architektoniczne aplikacji są często tworzone ad hoc, co drastycznie obniża zrozumiałość, adaptowalność i pielęgnowalność produktu.
- Nie wykorzystuje się w znaczącym stopniu istniejących wzorców projektowych i wzorców programistycznych.
- Rozkład typów defektów wykrytych podczas inspekcji przedstawia się następująco:
 - defekty dokumentacji — 40,51%,
 - niezgodność ze standardami — 23,20%,
 - defekty w logice — 7,22%,
 - defekty funkcjonalności — 6,57%,
 - defekty składni — 4,79%,
 - defekty danych — 4,62%,
 - defekty dotyczące pielęgnacji — 4,09%.
- Średnio zespół musi poświęcić od 8 do 16 minut na wykrycie jednego defektu i ok. 80 minut na wykrycie defektu o wysokiej dotkliwości (tzw. poważnego defektu).
- W kodzie poddawanym inspekcji znajdowano ok. 3,2 poważnego defektu oraz 17 mniej istotnych defektów na każde 1000 linii kodu.
- W godzinę zespół średnio był w stanie przeanalizować 625 linii kodu.
- W jednej sesji zespół średnio znajdował 4,6 defektu.
- Stosunek wysiłku związanego z przygotowaniem do inspekcji do wysiłku związanego z przeprowadzeniem inspekcji wynosił 0,58.
- Współczynnik wykrywalności defektów (ang. *Defect Detection Rate*) wahał się między 80% a 90% — oznacza to, że ok. 80 – 90% wszystkich wykrytych defektów zostało wykrytych w trakcie inspekcji.
- Zwrot z inwestycji (ROI liczone jako iloraz zysków do kosztów) średnio wyniósł 4,1.

Metryki, które można zbierać w ramach procesu inspekcji, to w szczególności:

- czas na przygotowanie się per defekt;
- czas na przygotowanie się per poważny defekt;
- liczba krytycznych defektów na 1000 linii kodu;
- liczba niekrytycznych defektów na 1000 linii kodu;
- liczba linii kodu sprawdzonych w czasie 1 godziny;
- liczba defektów na sesję;
- wysiłek na przygotowanie w stosunku do wysiłku na przeprowadzenie inspekcji;
- liczba sprawdzonych linii w jednej sesji.

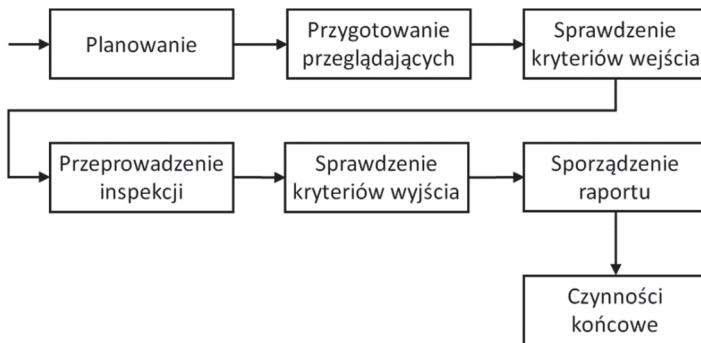
Analiza metryk, zwłaszcza z wielu przeprowadzonych inspekcji, pozwala oszacować koszt, wysiłek i efektywność tej formy testowania statycznego i obliczyć zwrot z inwestycji. Takie analizy mogą być bardzo przydatne do przekonania kadry kierowniczej do wykorzystywania przeglądów w codziennej pracy zespołów poprzez wykazanie ich skuteczności oraz tego, że skutkują obniżeniem całkowitych kosztów produkcji i pielęgnacji oprogramowania.

Przykład. W ramach statycznych technik testowania organizacja ABC stosuje standardowo dwa typy przeglądów: przejrzenia oraz inspekcje. Przejrzenia wykorzystywane są w celach weryfikacji. Przeprowadza się je, aby uzyskać wiedzę na temat różnorakich aspektów produktu pracy. Celem ubocznym przejrzeń jest:

- uzyskanie w organizacji spójnej, jednolitej wizji produktu;
- uzyskanie konsensusu wśród interesariuszy co do poszczególnych cech produktu;
- uzyskanie zgody co do stosowanych technik inżynierskich;
- upewnienie się co do zupełności i poprawności możliwości i cech tworzonego oprogramowania.

Przeglądy prowadzone są przez autorów poszczególnych produktów prac. W ramach każdej aktywności cyklu wytwarzania może zostać przeprowadzonych kilka przeglądów tego samego produktu pracy. Jedyną mierzoną wielkością jest liczba przeprowadzonych przejrzeń.

Inspekcje przeprowadza się, aby spełnić wymagania zespołu odpowiedzialnego za zarządzanie jakością. Zespół ten sprawdza, czy produkt pracy jest zgodny z obowiązującymi standardami, które organizacja musi stosować. Inspekcje formalne przeprowadza się pod koniec każdej czynności cyklu wytwarzania. Stanowią one tzw. bramki jakości (ang. *quality gate*), będące swoistymi kryteriami wyjścia z poszczególnych faz wytwórczych. Ponieważ inspekcja jest procesem formalnym, przebiega według ściśle zdefiniowanego procesu pokazanego na rysunku 3.7.



RYSUNEK 3.7. Proces inspekcji w organizacji ABC

W ramach inspekcji w sposób ścisły i formalny sprawdza się produkt pracy pod kątem:

- kompletności;
- poprawności;
- spójności;
- stylu;
- reguł konstrukcji.

Inspekcja prowadzona jest przez moderatora, a oprócz niego w procesie udział biorą: protokolant, od dwóch do pięciu przeglądających i autor. Przeprowadzenie inspekcji traktowane jest jak formalne sprawdzenie kryteriów wyjścia z danej fazy. W trakcie inspekcji dokonuje się pomiarów produktu i procesu, a cała sesja jest raportowana w specjalnie w tym celu zdefiniowanych szablonach raportów. Znalezione w trakcie inspekcji defekty są śledzone aż do momentu ich zamknięcia.

Tabela 3.5 zawiera podsumowanie różnic pomiędzy przejrzeniami a inspekcjami.

TABELA 3.5. Porównanie przejrzeń i inspekcji stosowanych w organizacji ABC

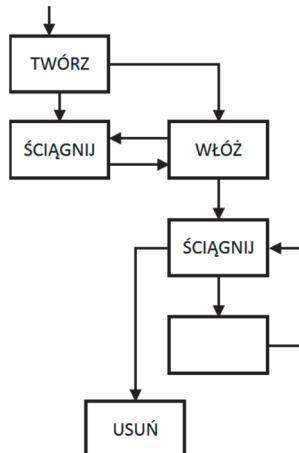
KATEGORIA	PRZEJRZENIE	INSPEKCJA
Cel ogólny	Wykonać właściwą pracę	Wykonać pracę właściwie
Cel szczegółowy	Nauka, zrozumienie, konsensus	Wykrycie defektów, zgodność
Wyzwalacz	Żądanie autora	Kryteria wyjścia z fazy
Pomiar	Instancje przejrzeń	Pomiary produktu i procesu

Więcej informacji na temat przeglądów można znaleźć w [Gilb 1993, Wiegers 2001, Veenendaal 2004].

Na zakończenie tego rozdziału przedstawimy dwa przykłady tego, jak może wyglądać przegląd architektury systemu.

Przykład. Rysunek 3.8 przedstawia projekt architektury przepływu informacji na pewnej strukturze danych używanej w systemie, zwanej stosem. Możliwe operacje na stosie to:

- TWÓRZ — utworzenie stosu;
- WŁÓŻ — włożenie elementu na szczyt stosu;
- ŚCIĄGNIJ — ściągnięcie elementu ze szczytu stosu;
- USUŃ — usunięcie struktury stosu z systemu.



RYSUNEK 3.8. Przepływ w systemie operacji dotyczących stosu

Tester przeprowadza przegląd oparty na liście kontrolnej i jest obecnie w fazie indywidualnego przygotowania. Wykorzystywana przez testera lista kontrolna składa się z następujących elementów:

1. Czy każda operacja WŁÓŻ i ŚCIĄGNIJ jest wykonywana dopiero wtedy, gdy stos został utworzony (TWÓRZ)?
2. Czy system nie dopuszcza do wykonania instrukcji ŚCIĄGNIJ na pustym stosie?
3. Czy system zabrania skasować stos (USUŃ) w sytuacji, gdy jest on niepusty?
4. Czy liczba elementów stosu nigdy nie przekroczy 10?

Tester wykorzystuje technikę przeglądu opartą na liście kontrolnej i sprawdza, które własności z listy są spełnione, a które nie.

Punkt 1. jest spełniony — każda instrukcja WŁÓŻ i ŚCIĄGNIJ jest tworzona po wykonaniu operacji TWÓRZ, a po operacji USUŃ nie można wykonać żadnej instrukcji WŁÓŻ ani ŚCIĄGNIJ. Dociekliwy tester zauważa jednak, że w specyfikacji nie jest powiedziane, czy w momencie tworzenia stosu jest on pusty, czy też np. zawiera jakieś domyślne, początkowe elementy. Warto zwrócić uwagę autorowi dokumentu na to, aby jasno to w specyfikacji określił.

Punkt 2. nie jest spełniony. Po operacji TWÓRZ można od razu wykonać operację ŚCIĄGNIJ (na pustym stosie).

Punkt 3. jest spełniony. Po każdej operacji WŁÓŻ następną operacją musi być operacja ŚCIĄGNIJ. Zatem na stosie można przechowywać tylko jeden element, który w kolejnym kroku będzie ściągnięty. W tym miejscu tester może zwrócić uwagę, że

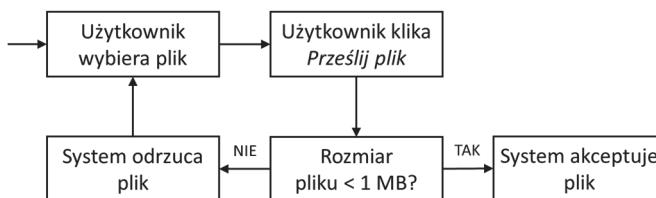
jeśli na stosie może być co najwyżej jeden element, to po co używać skomplikowanej struktury stosu, skoro można by używać struktury pozwalającej na zapamiętanie tylko jednego elementu.

Punkt 4. jest spełniony — wynika to z analizy dla punktu 3.

Jak widać z powyższego przykładu, tester nie ograniczył się do „odhaczenia” poszczególnych elementów listy, ale zauważył też pewne niedociągnięcia w specyfikacji, które są wartościową informacją zwrotną z przeglądu. Mogą służyć udoskonaleniu produktu pracy (specyfikacji) przez jego autora.

Przykład. Architekt poprosił programistę o przeprowadzenie nieformalnego przeglądu koleżeńskiego. Przegląd dotyczy stworzonego przez architekta modelu procesu realizującego następujące wymaganie biznesowe: „Użytkownik może przesłać jakikolwiek plik o rozmiarze mniejszym niż 1 GB przez formularz webowy. Jeśli rozmiar pliku nie przekracza tej wartości, system akceptuje go, w przeciwnym razie odrzuca plik”.

Model poddany przeglądowi przedstawiony jest na rysunku 3.9.



RYSUNEK 3.9. Model procesu biznesowego

Programista otrzymał od architekta zarówno wymaganie, jak i diagram przedstawiający proces. Programista sprawdza, czy diagram odpowiada wymaganiu, i zauważa, że w momencie podjęcia decyzji, czy system ma zaakceptować, czy odrzucić plik, jest literówka: w warunku napisano „Rozmiar pliku < 1 MB”, a powinno być „Rozmiar pliku < 1 GB”. Programista informuje architekta o zauważonej literówce, a architekt poprawia projekt.

3.2.5. Czynniki sukcesu przeglądów

Kluczem do pomyślnego przeprowadzenia przeglądu jest staranny dobór typu przeglądu i stosowanych technik. Ponadto należy uwzględnić szereg innych czynników, które mogą mieć wpływ na uzyskany wynik. Do czynników sukcesu zaliczają się między innymi następujące kryteria:

- każdy przegląd ma jednoznaczne cele zdefiniowane podczas planowania, które mogą służyć jako mierzalne kryteria wyjścia;
- stosowane typy przeglądów sprzyjają osiągnięciu założonych celów, a także są adekwatne do typu i poziomu produktów pracy związanych z oprogramowaniem oraz do uczestników;

- stosowane są różne techniki przeglądu (takie jak przegląd oparty na listach kontrolnych lub oparty na rolach), które pozwalają skutecznie identyfikować defekty występujące w danym produkcie pracy;
- stosowane listy kontrolne są aktualne i uwzględniają główne ryzyka;
- obszerniejsze dokumenty są pisane i przeglądane partiami, dzięki czemu autorzy szybko i często otrzymują informacje zwrotne na temat defektów (co jest elementem kontroli jakości);
- uczestnicy mają wystarczająco dużo czasu na przygotowanie się do przeglądu; przeglądy są planowane z należytym wyprzedzeniem;
- autorom przekazywana jest informacja zwrotna od przeglądających, dzięki czemu mogą oni doskonalić produkt pracy oraz polepszyć jakość swoich działań;
- kierownictwo wspiera przebieg przeglądów (np. poprzez wyznaczenie wystarczającego czasu na wykonanie czynności związanych z przeglądem w harmonogramie projektu);
- przeglądy są traktowane jako naturalna część kultury organizacyjnej, służącej do promowania uczenia się oraz doskonalenia produktów i procesów;
- w przegląd są zaangażowane osoby, których udział sprzyja osiągnięciu jego celów — na przykład osoby o różnych umiejętnościach lub punktach widzenia, które będą potencjalnie korzystać z dokumentu w ramach wykonywanej pracy;
- testerzy są uznawani za ważnych uczestników przeglądu, a zdobywana przez nich wiedza na temat produktu pracy umożliwia wcześniejsze przygotowanie bardziej efektywnych testów;
- uczestnicy przeznaczają wystarczającą ilość czasu na wzięcie udziału w przeglądzie i wykazują się należytą dbałością o szczegóły;
- recenzje są przeprowadzane na małych fragmentach, dzięki czemu recenzenci nie tracą koncentracji podczas indywidualnej recenzji i/lub spotkania przeglądowego (o ile się odbywa);
- wykryte defekty są przyjmowane do wiadomości, potwierdzane i rozpatrywane w sposób obiektywny;
- spotkania przeglądowe są moderowane we właściwy sposób, tak aby uczestnicy nie tracili czasu na zbędne czynności;
- przegląd odbywa się w atmosferze wzajemnego zaufania, a jego wyniki nie są wykorzystywane do oceny uczestników;
- uczestnicy unikają gestów i zachowań, które mogłyby wskazywać na znudzenie, irytację bądź wrogie nastawienie wobec innych uczestników;
- uczestnikom zapewniono należyté szkolenie, zwłaszcza w przypadku bardziej formalnych typów przeglądów (takich jak inspekcje);
- stworzono atmosferę sprzyjającą poszerzaniu wiedzy i doskonaleniu procesów.

Przykład. Lider przeglądu został poproszony o zorganizowanie przeglądu kodu komponentu X (w postaci przejrzenia) dla grupy deweloperów. Przeprowadził inspekcję, zapraszając do niej autora komponentu X oraz zespół testerów.

W tym scenariuszu zabrakło zarówno czynników powodzenia dotyczących kwestii organizacyjnych, jak i związanych z ludźmi:

- Lider przeglądu został poproszony o zorganizowanie przejrzenia, a zorganizował inspekcję, która nie jest najlepszym typem przeglądu dla przeglądu kodu. Nie został zatem spełniony organizacyjny czynnik powodzenia: „stosowane typy przeglądów sprzyjają osiągnięciu założonych celów, a także są adekwatne do typu i poziomu produktów pracy związanych z oprogramowaniem oraz do uczestników”.
- Lider przeglądu został poproszony o zorganizowanie przeglądu kodu dla deweloperów, ale — poza autorem — zaprosił do niego jedynie testerów. Taki przegląd kodu będzie nieefektywny. Nie został zatem spełniony czynnik sukcesu o charakterze kadrowym: „w przegląd są zaangażowane osoby, których udział sprzyja osiągnięciu jego celów”.

3.2.6. (*) Techniki przeglądu

Syllabus poziomu podstawowego w poprzedniej wersji (3.0) opisywał pięć różnych technik, jakie osoba przeglądająca może stosować w ramach czynności przeglądu indywidualnego, to znaczy indywidualnego przygotowania do przeglądu. Czynność ta może być oczywiście wykonywana w ramach wszystkich typów przeglądów, w szczególności wszystkich czterech typów przeglądów opisanych w punkcie 3.2.4. Celem tych technik jest wykrycie defektów w przeglądany produkcie pracy. Pięć wspomnianych technik to:

- przegląd ad hoc;
- przegląd oparty na liście kontrolnej;
- scenariusze i przebiegi próbne;
- przegląd oparty na rolach;
- czytanie oparte na perspektywie.

Przegląd ad hoc

Jest to tradycyjne podejście do wykrywania defektów przez przeglądających. Proces ten jest zupełnie nieustrukturyzowany. Każdy z przeglądających ma za zadanie wykryć jak najwięcej defektów wszelkiego możliwego typu. Efektywność takiego przeglądu jest silnie zależna od umiejętności poszczególnych przeglądających. Zazwyczaj prowadzi do wykrycia tych samych (najczęściej oczywistych) problemów przez wielu różnych przeglądających. Podczas przeglądu ad hoc przeglądający otrzymują niewiele wskazówek dotyczących sposobu wykonywania zadania (lub nie otrzymują ich wcale). Uczestnicy często czytają produkt pracy sekwencyjnie, na bieżąco identyfikując i dokumentując stwierdzone problemy.

Przegląd oparty na liście kontrolnej

Przegląd oparty na liście kontrolnej jest bardziej usystematyzowaną techniką niż przegląd ad hoc. Dobrą praktyką stosowaną w tym typie przeglądu jest, aby różni przeglądający otrzymali różne listy kontrolne. Spowoduje to zwiększenie potencjalnego

pokrycia produktu i wykrycie większej liczby defektów. Zmniejsza to również ryzyko wielokrotnego wykrywania tego samego problemu przez większą liczbę osób, co jest stratą czasu, zasobów i pieniędzy. Najważniejszą zaletą techniki opartej na listach kontrolnych jest systematyczne pokrycie najczęściej występujących typów defektów.

Istnieje niebezpieczeństwo związane z tym podejściem, polegające na tym, że przeglądający mogą ścisłe ograniczyć się jedynie do kwestii zawartych w liście kontrolnej i zignorować inne potencjalne problemy w przeglądany produkcie pracy. Przeglądający powinni zatem mieć świadomość, że mają większą odpowiedzialność niż tylko ślepe podążanie za elementami listy kontrolnej.

Listy kontrolne powinny zostać dobrane do typu produktu pracy oraz do celu, jaki przyświeca zespołowi. Zatem lista kontrolna na użytek przeglądu wymagań będzie zupełnie inna niż na przykład lista kontrolna na użytek testów bezpieczeństwa czy testów użyteczności interfejsu. Lista kontrolna może być nawet specyficzna dla konkretnej metody użytej do wytworzenia produktu pracy. Na przykład lista kontrolna używana do testów produktów związanych z bankowością może zawierać ustawaowe regulacje prawne nałożone na banki, a lista stosowana w przemyśle samochodowym (ang. *automotive*) może z kolei być oparta na standardzie [ISO 26262].

Częstym problemem, jaki pojawia się przy stosowaniu tego podejścia, jest to, że listy kontrolne z czasem stają się coraz dłuższe i w związku z tym coraz mniej poręczne w użyciu. Typowa lista kontrolna nie powinna zawierać więcej niż ok. 10 elementów i powinna być regularnie przeglądana i aktualizowana. Aktualizacje powinny dotyczyć zwłaszcza tych problemów, awarii i defektów, jakie sami wykryliśmy w naszym produkcie. Taka lista kontrolna, oparta na naszych doświadczeniach, będzie zdecydowanie lepsza (w sensie efektywności wykrywania defektów) niż np. standarowa lista kontrolna znaleziona w internecie.

Często stosowanym podejściem jest wykorzystanie analizy ryzyka i rozszerzenie listy kontrolnej o zidentyfikowane ryzyka o największym poziomie (dotkliwości). To pozwala bezpośrednio sprawdzać największe ryzyka podczas stosowania podejścia opartego na listach kontrolnych.

Przykład przeglądu opartego na liście kontrolnej przedstawiony został w punkcie 3.2.4 (przykład ze stosem).

Scenariusze i przebiegi próbne

Podejście oparte na scenariuszach sprawdza się dobrze w sytuacji, gdy wymagania, projekt bądź same testy są udokumentowane w odpowiednim „scenariuszowym” formacie, na przykład w formie przypadków użycia. W podejściu tym przeglądający przeprowadzają tzw. suche przebiegi (ang. *dry runs*) na produkcie pracy, sprawdzając, czy funkcjonalność produktu jest opisana poprawnie i czy typowe wyjątki spowodowane błędnym zachowaniem produktu są odpowiednio obsługiwane.

Istnieje ryzyko, że przeglądający będą zbyt ścisłe podążali za zdefiniowanymi scenariuszami. W takiej sytuacji mogą łatwo przeoczyć niektóre defekty, np. brakującą funkcjonalność w produkcie.

Tak jak w przypadku podejścia opartego na listach kontrolnych, tak i w tym podejściu można rozszerzać scenariusze o aspekty wynikłe z przeprowadzonej analizy ryzyka, aby upewnić się, że najważniejsze i najczęściej wykorzystywane scenariusze są zbadane najdokładniej.

Przegląd oparty na rolach

Przegląd oparty na rolach (ang. *role-based review*) to technika, w której recenzenci oceniają produkt pracy z perspektywy poszczególnych ról interesariuszy. Typowe role obejmują konkretne typy użytkowników końcowych (doświadczonych, niedoświadczonych, starszych, dzieci itp.) lub określone role w organizacji (użytkownik, administrator, administrator systemu, tester wydajności itp.).

Często role modeluje się za pomocą tzw. person. Persona to fikcyjna, ale konkretna postać, która symbolizuje pewien określony typ użytkownika. Dzięki skonkretyzowaniu tej postaci (określeniu np. jej płci, wieku, zainteresowań) zespołowi łatwiej „wzucić się” w dany typ użytkownika. Przykład persony pokazano na rysunku 3.10.



Michał Kaczmarek

Stanowisko: Tester aplikacji mobilnych

Wiek: 28 lat

Miasto: Warszawa

Hobby: film, gry FPS, podróże, nowoczesne technologie

Cechy charakteru: żywiołowy, niecierpliwy, ciekawski

RYSUNEK 3.10. Przykład persony

Czytanie oparte na perspektywie

Technika czytania opartego na perspektywie (ang. *perspective-based reading*) jest opisywana w literaturze przedmiotu jako jedna z najefektywniejszych metod przeglądu indywidualnego [Sauer 2000]. Opiera się ona na tym, że różni przeglądający przyjmują podczas przeglądania produktu pracy różne punkty widzenia interesariuszy i przeglądają produkt pod tym właśnie kątem. Dzięki rozważeniu wielorakich spojrzeń na produkt przeglądający są w stanie odkryć większą liczbę potencjalnych problemów. Jednocześnie, dzięki przypisaniu przeglądającym konkretnych perspektyw, każdy z nich jest w stanie przeglądać produkt dokładnie i szczegółowo, a ponieważ każdemu odpowiada inna perspektywa, ryzyko wykrywania dublujących się defektów jest niewielkie.

Czytanie oparte na perspektywie nie ogranicza się jedynie do przyjmowania różnych punktów widzenia. Wymaga również od przeglądających, aby spróbowali wykorzystać produkt pracy podlegający przeglądowi do wygenerowania pierwszego prototypu produktu w celu sprawdzenia, czy jest to możliwe na podstawie informacji dostępnych w przeglądany produkcie pracy. Na przykład tester będzie próbował wygenerować wersje robocze testów akceptacji, jeśli przeprowadzi przegląd

specyfikacji wymagań oparty na perspektywie, aby sprawdzić, czy specyfikacja zawiera wszystkie niezbędne informacje. Ponadto w czytaniu opartym na perspektywie często używa się list kontrolnych.

Typowe perspektywy, jakie może przyjąć przeglądający, to:

- użytkownik;
- analityk biznesowy;
- projektant;
- tester;
- operator systemu;
- pracownik działu marketingu;
- pracownik działu wsparcia technicznego.

Jeśli technika ogranicza się do punktu widzenia użytkownika końcowego, można także zastosować technikę czytania opartego na perspektywie, rozróżniając typy użytkowników systemu, np.:

- użytkownik końcowy;
- administrator systemu;
- pracownik działu wsparcia technicznego.

Kluczowym czynnikiem sukcesu czytania opartego na perspektywie jest uwzględnienie i wyważenie punktów widzenia poszczególnych interesariuszy w sposób adekwatny do ryzyka. To, jakie punkty widzenia przyjmiemy, powinno zależeć od kontekstu. Na przykład:

- jeśli przeglądany dokumentem są wymagania, typowymi perspektywami będą: użytkownik, projektant i tester;
- jeśli system dotyczy ściśle regulowanego obszaru (np. lotnictwa, bankowości, systemów krytycznych), zdecydowanie należy włączyć w zakres przeglądu punkt widzenia regulatora;
- jeśli system będzie używany przez długi czas, powinno się przyjąć w szczególności punkt widzenia zespołu utrzymującego system.

Przegląd oparty na rolach a czytanie oparte na perspektywie

Czytanie oparte na perspektywie jest podobne do wcześniej omówionej techniki przeglądu opartego na rolach. Czasami w literaturze utożsamia się te dwa podejścia, ale sylabus poziomu podstawowego rozróżnia je. Różnicę tę można chyba najprościej wyjaśnić w następujący sposób:

- w przeglądzie opartym na rolach zmieniają się „rodzaje” użytkowników, ale zadania do wykonania pozostają te same (np. różne typy klientów banku);
- w czytaniu opartym na perspektywie zmieniają się „perspektywy” interesariuszy lub — w przypadku samych użytkowników — rodzaje zadań do wykonania (np. użytkownik końcowy, administrator, analityk biznesowy, tester).

Przykład. Zespół stosuje technikę czytania opartego na perspektywie. Osoba przeglądająca dokument — specyfikację wymagań — przyjmuje perspektywę testera i wykorzystuje następującą procedurę czytania tej specyfikacji.

PROCEDURA CZYTANIA. Dla każdego wymagania stwórz test lub zbiór testów, który pozwoli upewnić się, że implementacja spełnia wymaganie. Wykorzystaj standardeowe podejście do testów oraz kryteria i techniki testowania w celu zbudowania zbioru testów. Podczas tworzenia testów dla każdego wymagania odpowiedz na następujące pytania:

1. Czy posiadasz wystarczającą ilość informacji do tego, by zidentyfikować testowany obiekt oraz kryteria testowania? Czy jesteś w stanie stworzyć rozsądne przypadki testowe dla każdego elementu na podstawie tych kryteriów?
2. Czy istnieje inne wymaganie, dla którego mógłbyś wygenerować podobny przypadek testowy, ale z odmiennym wynikiem oczekiwany?
3. Czy jesteś pewien, że wygenerowany przez ciebie test dostarczy właściwych wartości we właściwych jednostkach?
4. Czy istnieją inne interpretacje tego wymagania, które deweloper mógłby przyjąć, opierając się na sposobie definicji wymagania? Czy miałoby to wpływ na Twoje testy?
5. Czy wymaganie jest sensowne i racjonalne z punktu widzenia Twojej wiedzy o aplikacji i tego, co jest zawarte w ogólnym opisie systemu?

Pytania testowe do rozdziału 3.

Pytanie 3.1

(FL-3.1.1, K1)

Twoja organizacja przygotowała właśnie oficjalny dokument opisujący, jak należy wykonywać przeglądy w organizacji.

Czy ten dokument może podlegać przeglądowi?

- A. Tak, ponieważ każdy dokument zrozumiałы dla człowieka może być testowany statycznie.
- B. Nie, ponieważ musielibyśmy podczas przeglądu zastosować reguły opisane w tym dokumencie do niego samego.
- C. Nie, ponieważ dokument ten nie jest produktem pracy ani procesu testowego, ani procesu wytwarzczego.
- D. Nie, ponieważ przeglądy mogą być przeprowadzone tylko dla specyfikacji i kodu źródłowego.

Wybierz jedną odpowiedź.

Pytanie 3.2

(FL-3.1.2, K2)

Tester, analizując kod, zauważał, że złożoność cyklowatyczna jednego z jego modułów jest bardzo wysoka. Przekazał tę informację programistom, którzy poddali kod refaktoryzacji, czyniąc go bardziej czytelnym i testowalnym. Ten scenariusz pokazuje zysk z:

- A. Testowania dynamicznego.
- B. Testowania statycznego.
- C. Zarządzania testowaniem.
- D. Użycia formalnej techniki testowania.

Wybierz jedną odpowiedź.

Pytanie 3.3

(FL-3.1.3, K2)

Wskaż różnicę pomiędzy technikami statycznymi i dynamicznymi ze względu na cele tych technik.

- A. Techniki statyczne bezpośrednio wykrywają defekty, podczas gdy techniki dynamiczne bezpośrednio wykrywają awarie.
- B. Techniki statyczne są zwykle używane wcześnie w cyklu wytwarzania, podczas gdy techniki dynamiczne zwykle używane są w późniejszych fazach cyklu wytwarzania.
- C. Nie ma żadnej różnicy, ponieważ oba typy technik mają na celu wykrycie defektów tak wcześnie, jak to możliwe.
- D. Techniki statyczne zazwyczaj wymagają umiejętności programistycznych, podczas gdy techniki dynamiczne zazwyczaj ich nie wymagają.

Wybierz jedną odpowiedź.

Pytanie 3.4

(FL-3.2.1, K1)

Rozważ następujące stwierdzenia dotyczące wczesnej informacji zwrotnej.

- i. Wczesna informacja zwrotna daje programistom więcej czasu na wytwarzanie nowych cech systemu, ponieważ spędzają mniej czasu nad modyfikacją cech zaplanowanych w danej iteracji.
- ii. Wczesna informacja zwrotna pozwala zespołom zwinnym dostarczyć cechy o największej wartości biznesowej jako pierwsze, ponieważ uwaga klienta pozostaje skupiona na cechach najważniejszych z jego punktu widzenia.
- iii. Wczesna informacja zwrotna redukuje całociowe koszty testowania, ponieważ zmniejsza ilość czasu, jaki testerzy potrzebują na przetestowanie systemu.
- iv. Wczesna informacja zwrotna zwiększa prawdopodobieństwo, że wytworzony system będzie bliski oczekiwaniom klientów, ponieważ mają oni możliwość wprowadzania zmian podczas trwania poszczególnych iteracji.

Które stwierdzenia są prawdziwe?

- A. Prawdziwe są (i) oraz (iv); nieprawdziwe są (ii) oraz (iii).
- B. Prawdziwe są (ii) oraz (iii); nieprawdziwe są (i) oraz (iv).
- C. Prawdziwe są (ii) oraz (iv); nieprawdziwe są (i) oraz (iii).
- D. Prawdziwe są (i) oraz (iii); nieprawdziwe są (ii) oraz (iv).

Wybierz jedną odpowiedź.

Pytanie 3.5

(FL-3.2.2, K2)

Która z poniższych czynności jest częścią fazy rozpoczęcia przeglądu?

- A. Wybór osób, które będą brały udział w przeglądzie.
- B. Zbieranie metryk.
- C. Odpowiedź na pytania uczestników dotyczące zakresu, celów, procesu, ról i produktów pracy.
- D. Określenie zakresu prac, w tym typu przeglądu i dokumentów (lub ich części) będących jego przedmiotem oraz ocenianych charakterystyk jakościowych.

Wybierz jedną odpowiedź.

Pytanie 3.6

(FL-3.2.3, K1)

Która z ról jest odpowiedzialna za sprawny przebieg spotkania przeglądowego?

- A. Lider przeglądu.
- B. Moderator.
- C. Autor.
- D. Przeglądający.

Wybierz jedną odpowiedź.

Pytanie 3.7

(FL-3.2.4, K2)

W ciągu ostatnich kilku dni zespół usiłował znaleźć przyczynę dziwnej awarii systemu. Ponieważ nie udało się dociec przyczyny, zespół zdecydował, że będzie „symulował komputer” poprzez ręczne wykonanie oprogramowania linia po linii, aby zrozumieć, co dokładnie program robi, i odkryć w ten sposób przyczynę dziwnego zachowania oprogramowania.

Jaki typ przeglądu będzie najodpowiedniejszy do tej czynności?

- A. Inspekcja.
- B. Przegląd nieformalny.
- C. Przegląd techniczny.
- D. Przejrzenie.

Wybierz jedną odpowiedź.

Pytanie 3.8

(FL-3.2.5, K1)

Inspekcja będzie lepiej przeprowadzona, jeśli:

- A. Jej główny cel zostanie zdefiniowany jako „ocena alternatyw”.
- B. Kierownik będzie uczęszczał na spotkania przeglądowe.
- C. Przeglądający będą przeszkoleni w przeprowadzaniu przeglądów.
- D. Metryki nie będą zbierane podczas całego procesu przeglądu.

Wybierz jedną odpowiedź.

ROZDZIAŁ 4.

Analiza i projektowanie testów

Słowa kluczowe

analiza wartości brzegowych — czarnoskrzynkowa technika testowania, w której przypadki testowe są projektowane w oparciu o wartości brzegowe.

białośkrzynkowa technika testowania — technika testowania oparta wyłącznie na wewnętrznej strukturze modułu lub systemu.

czarnoskrzynkowa technika testowania — technika testowania oparta na analizie specyfikacji modułu lub systemu.

element pokrycia — atrybut lub kombinacja atrybutów uzyskana z jednego lub więcej warunków testowych za pomocą techniki testowej.

kryteria akceptacji — kryteria, które moduł lub system musi spełniać, aby został zaakceptowany przez użytkownika, klienta lub inny uprawniony podmiot.

podejście do testów oparte na współpracy — podejście do testowania, które skupia się na unikaniu defektów poprzez współpracę między interesariuszami.

podział na klasy równoważności — czarnoskrzynkowa technika testowania, w której przypadki testowe są projektowane na podstawie klas równoważności poprzez wybór do testów po jednym reprezentancie z każdej klasy.

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

pokrycie gałęzi — pokrycie gałęzi na grafie przepływu sterowania.

pokrycie instrukcji kodu — pokrycie wykonywalnych instrukcji kodu.

technika testowania — procedura używana do definiowania warunków testowych, projektowania przypadków testowych i określania danych testowych.

technika testowania oparta na doświadczeniu — technika testowania bazująca tylko na doświadczeniu, wiedzy i intuicji testera.

testowanie eksploracyjne — podejście do testowania, w którym testerzy dynamicznie projektują i przeprowadzają testy na podstawie swojej wiedzy, badania elementu testowego i wyników z poprzednich testów.

testowanie przejść pomiędzy stanami — czarnoskrzynkowa technika testowania, w której przypadki testowe są zaprojektowane tak, by wykonywać elementy modelu przejścia stanów. *Synonim:* testowanie stanów.

testowanie w oparciu o listę kontrolną — technika projektowania testów oparta na doświadczeniu, w której doświadczony tester używa listy ogólnych zagadnień, które powinny być odnotowywane, sprawdzone, zapamiętane, lub zbioru reguł bądź kryteriów, względem których produkt ma być sprawdzany.

testowanie w oparciu o tablicę decyzyjną — czarnoskrzynkowa technika testowania, w której przypadki testowe są projektowane w celu wykonania kombinacji wejść i wynikających akcji, przedstawione w postaci tablicy decyzyjnej.

wytwarzanie sterowane testami akceptacyjnymi (ATDD, ang. *Acceptance Test-Driven Development*) — wspólne podejście do tworzenia oprogramowania, w którym zespół i klienci używają języka dziedzinowego klienta, aby zrozumieć jego wymagania, co tworzy podstawę do testowania modułu lub systemu.

zagadywanie błędów — technika testowania, w której testy są wyprowadzane na bazie wiedzy testera na temat przeszłych awarii lub generalnej wiedzy o trybach możliwych awarii.

4.1. Ogólna charakterystyka technik testowania

FL-4.1.1 (K2) Kandydat rozróżnia czarnoskrzynkowe i białośkrzynkowe techniki testowania oraz techniki testowania oparte na doświadczeniu.

W niniejszym podrozdziale opiszemy:

- w jaki sposób wybierać odpowiednią technikę testowania;
- jakie rodzaje technik testowania opisuje sylabus poziomu podstawowego;
- jakie są cechy wspólne poszczególnych grup technik testowania.

Zgodnie z definicją z Encyklopedii PWN technika (z gr. *technē* (τέχνη) — sztuka, rzemiosło, kunszt, umiejętność) to „dziedzina ludzkiej działalności, której celem jest oparte na wiedzy (na podstawach naukowych) produkowanie rzeczy i wywoływanie zjawisk niewystępujących w przyrodzie oraz przekształcanie wytworów przyrody; główny czynnik rozwoju cywilizacji i, wraz z nauką, istotny składnik kultury; techniką jest nazywany także sposób i biegłość wykonywania określonych czynności w jakiejś dziedzinie, np. technika gry na instrumencie muzycznym, technika piłkarska, technika pracy umysłowej”¹.

W kontekście testowania oprogramowania czynność ta będzie polegać na wykorzystywaniu pewnych metod do wyprowadzania (produkcji) warunków, przypadków oraz danych testowych na bazie naukowych podstawa wiedzy o oprogramowaniu.

¹ <https://encyklopedia.pwn.pl/haslo/technika;3985955.html>

Od razu powiedzmy, że każda **technika testowania** (ang. *test technique*) jest ściśle zdefiniowana, tzn. jest formalna w sensie zasad, które ją stanowią (choć sama w sobie, jako taka, formalną być nie musi — przykładami może tu być zgadywanie błędów czy testowanie eksploracyjne). Za każdą techniką testowania stoi bowiem coś, co Glenford Myers nazywa „hipotezą błędu” [Myers 2011]. W kontekście słownictwa ISTQB [ISTQB S] być może właściwszą nazwą byłaby „hipoteza defektu”, ale z racji uwarunkowań historycznych pozostały przy nazwie oryginalnej. Zresztą techniki testowania pozwalają również wykrywać defekty pochodzące od konkretnych błędów programistycznych.



Owa hipoteza błędu czy — jak to się czasem mówi — teoria błędu jest podstawą naukową konstrukcji poszczególnych technik. Podczas omawiania każdej z nich w kolejnych podrozdziałach będziemy zwracać szczególną uwagę na to, jakiego typu problemy dana technika jest w stanie wykrywać. Techniki testowania są zatem niezależne od konkretnych technologii, wykorzystywanych narzędzi czy typu testowanego oprogramowania. Każda z nich zbudowana jest wokół abstrakcyjnego problemu związanego z określona hipotezą błędu i stworzona w celu wykrywania tego konkretnego typu błędów czy defektów.

4.1.1. Kategorie technik testowania i ich cechy charakterystyczne

Sylabus poziomu podstawowego omawia dziewięć technik testowania, grupując je w trzy kategorie:

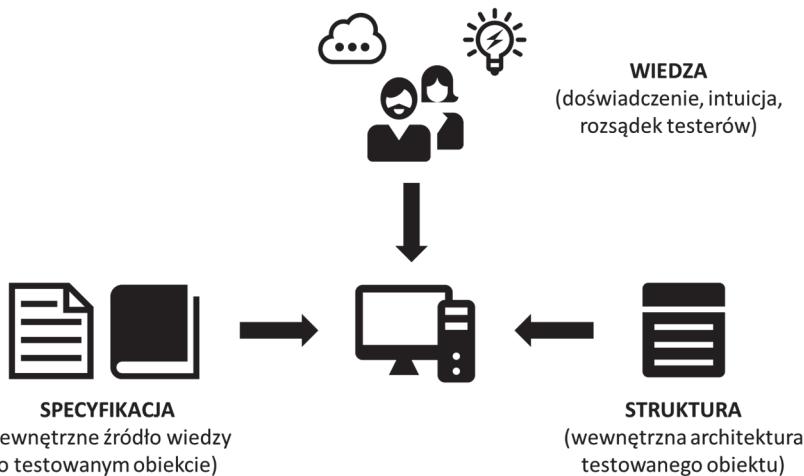
- czarnoskrzynkowe (cztery techniki);
- białośkrzynkowe (dwie techniki);
- oparte na doświadczeniu (trzy techniki).

Podział ten jest przeprowadzony ze względu na *źródło wiedzy* o testowanym obiekcie. Rysunek 4.1 uzasadnia schematycznie powód takiego właśnie podziału technik. Zanim oprogramowanie zostanie stworzone, należy je *zaprojektować*. Rolę dokumentacji projektowej mogą spełniać: specyfikacja wymagań, przypadki użycia czy opis procesów biznesowych. Na ich podstawie tworzone jest oprogramowanie, które ma określoną *strukturę*. Jest nią np. kod, struktura menu, struktura przepływu procesów biznesowych lub też inna forma opisu architektury. Poza tymi formalnymi źródłami wiedzy istnieją również mniej formalne, związane bezpośrednio z ludźmi pracującymi przy tworzeniu oprogramowania. Ludzie ci posiadają *wiedzę, doświadczenie* i *intuicję*, na podstawie których również można wyprowadzić wartościowe testy.

Omówimy teraz dokładniej poszczególne kategorie technik testowania.

Czarnoskrzynkowe techniki testowania

Czarnoskrzynkowe techniki testowania (ang. *black-box test technique*) nazywane są również technikami behawioralnymi bądź technikami opartymi na specyfikacji. Wykorzystują zewnętrzną wobec testowanego obiektu wiedzę o tym, jak obiekt ten powinien się zachowywać. Tą wiedzą może być na przykład: specyfikacja wymagań, opis przypadków użycia, historynki użytkownika (w projektach zwinnych), procesy biznesowe. Wszystkie te modele opisują pożądane zachowania systemu — zarówno funkcjonalne, jak i niefunkcjonalne, przy czym nie odwołują się do jego wewnętrznej budowy.



RYSUNEK 4.1. Źródła wiedzy o oprogramowaniu jako kategorie technik testowania

Zaletą stosowania technik czarnoskrzynkowych jest to, że wspomniane wyżej dokumenty na ogół istnieją na długo przed rozpoczęciem implementacji modułu lub systemu. A to oznacza, że czynności testowe (np. analiza i projektowanie testów) mogą się rozpocząć na długo przed tworzeniem kodu. Dzięki temu możemy wykorzystywać techniki czarnoskrzynkowe nie tylko podczas testów dynamicznych, ale także na etapie projektowania testów, jako swoją metodę testowania statycznego. Techniki czarnoskrzynkowe można stosować zarówno w testach funkcjonalnych, jak i niefunkcjonalnych.

Białośkrzynkowe techniki testowania

Białośkrzynkowe techniki testowania (ang. *white-box test technique*) nazywane są również technikami strukturalnymi bądź opartymi na strukturze. Podstawą do projektowania testów białośkrzynkowych jest *wewnętrzna struktura* obiektu testów. Najczęściej strukturą tą jest kod źródłowy, ale może nią być również inny, bardziej wysokopoziomowy model architektury systemu lub modułu. Na przykład na poziomie testów integracji takim modelem może być tzw. graf wywołań (ang. *call graph*), a na poziomie testów systemowych — struktura przepływu informacji w procesie biznesowym.



Uważny czytelnik może zauważyc, że skoro testując program, odwołujemy się do niego samego (np. do kodu źródłowego), to dochodzimy do paradoksalnej sytuacji, w której program staje się swoją własną wyrocznią, tzn. orzeka sam o sobie, jak ma się zachowywać. Tak naprawdę jest to problem pozorny. Wiedza o wewnętrznej strukturze programu służy jedynie do zaprojektowania testów, które pokryją określone elementy tego modelu (np. instrukcje kodu, decyzje w kodzie, ścieżki w programie). Natomiast dla każdego testu jego oczekiwane wyjście należy ustalić na podstawie wiedzy zewnętrznej wobec testowanego systemu, na przykład na podstawie specyfikacji lub zdrowego rozsądku. Kod nigdy nie może być wyrocznią dla siebie samego.

Techniki testowania oparte na doświadczeniu

Grupa **technik testowania opartych na doświadczeniu** (ang. *experience-based test technique*) różni się od pozostałych dwóch grup tym, że nie bazuje na żadnym formalnym dokumencie: projekcie, wymaganiach, kodzie itp. Techniki testowania oparte na doświadczeniu wykorzystują bowiem nieco bardziej „miękkie” źródła wiedzy o testowanym systemie. Tymi źródłami są: wiedza, intuicja, doświadczenie, znajomość defektów znalezionych w poprzednich wersjach systemu lub podobnych aplikacjach itp. Odwołują się więc bezpośrednio do cech i umiejętności samych testerów, a nie „obiektywnej” wiedzy o testowanym systemie.



Techniki mogą wykorzystywać nie tylko doświadczenie testerów, ale także wszystkich pozostałych interesariuszy projektu. Przykładem może być korzystanie z wiedzy: programistów, użytkowników końcowych, klienta, projektantów, analityków biznesowych, kierownika projektu. Techniki testowania oparte na doświadczeniu często wykorzystuje się w połączeniu z technikami czarnoskrzynkowymi i białośkrzynkowymi. Dzięki temu testerzy mają szansę wykryć problemy, które łatwo przeoczyć, stosując bardziej formalne techniki testowania, które nie są w stanie uwzględnić wszelkich niuansów projektu czy kontekstu, w jakim wytwarzane jest oprogramowanie.

Tabela 4.1 ukazuje podstawowe różnice między trzema wyżej wymienionymi grupami technik testowania.

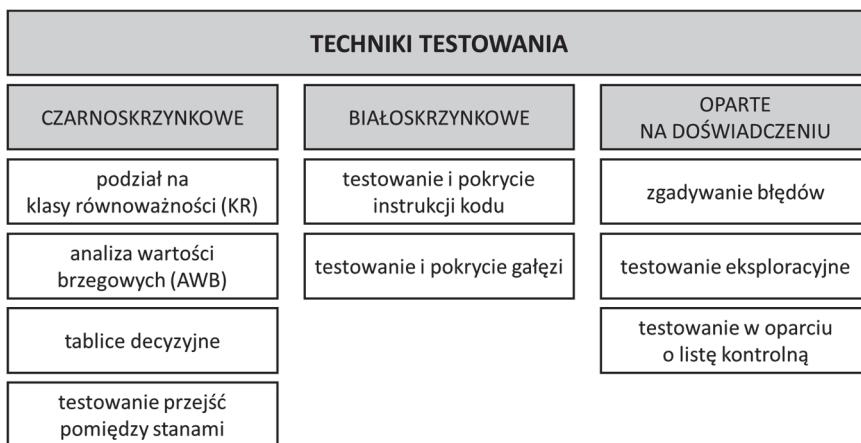
TABELA 4.1. Porównanie kategorii technik testowania

KRYTERIUM PORÓWNANIA	TECHNIKI CZARNOSKRZYNKOWE	TECHNIKI BIAŁOSKRZYNKOWE	TECHNIKI OPARTE NA DOŚWIADCZENIU
Źródło wyprowadzania warunków, przypadków i danych testowych	Podstawa testów zewnętrzna wobec obiektu testów: wymagania, specyfikacje, przypadki użycia, historyki użytkownika	Podstawa testów opisująca wewnętrzną strukturę obiektu testów: kod, architektura, projekt szczegółowy; do opisania wyników oczekiwanych używa się często specyfikacji	Podstawa testów: wiedza, doświadczenie, intuicja testerów, programistów, użytkowników i innych interesariuszy

TABELA 4.1. Porównanie kategorii technik testowania – ciąg dalszy

KRYTERIUM PORÓWNANIA	TECHNIKI CZARNOSKRZYNKOWE	TECHNIKI BIAŁOSKRZYNKOWE	TECHNIKI OPARTE NA DOŚWIADCZENIU
Rodzaj wykrywanych problemów	Rozbieżności między wymaganiami (deklarowanym zachowaniem) a ich implementacją	Problemy związane z błędnym przepływem sterowania bądź danych	Zależy od osoby wykonującej testy lub np. zastosowanej w testach listy kontrolnej
Pokrycie	Mierzone na bazie przetestowanych elementów podstawy testów oraz zastosowanej techniki do podstawy testów	Mierzone na bazie przetestowanych elementów struktury, np. kodu lub interfejsów	Nie definiuje się

Jak wcześniej wspomniano, syllabus poziomu podstawowego wprowadza dziewięć technik testowania, których znajomość obowiązuje na egzaminie. Są to cztery techniki czarnoskrzynkowe, dwie białośkrzynkowe i trzy oparte na doświadczeniu. Zestawienie tych technik pokazane jest na rysunku 4.2. Syllabus nie opisuje dokładnie tych technik ani nie podaje przykładów ich praktycznego użycia, choć umiejętność poprawnego ich stosowania jest jedną z najważniejszych cech dobrego testera. Dlatego w kolejnych podrozdziałach omówimy je *bardzo szczegółowo*, zwracając uwagę na istotne aspekty każdej techniki. Aby wyłożony materiał był bardziej przystępny, zilustrujemy wykorzystanie poszczególnych technik na przykładach.

**RYSUNEK 4.2.** Techniki testowania opisane w syllabusie poziomu podstawowego

Istnieje oczywiście wiele innych technik testowania. Syllabus omawia jedynie te najpopularniejsze i najpowszechniej stosowane. Techniki testowania i odpowiadające im miary **pokrycia** (ang. *coverage*) opisuje międzynarodowy standard ISO/IEC/IEEE 29119-4. Więcej informacji o technikach testowania zawierają także publikacje [Beizer 1990, Craig 2002, Copeland 2004, Koomen 2006, Jorgensen 2014, Roman 2015, Ammann 2016, Forgács 2019].



Wybór techniki testowania

Na wybór technik testowania wpływa wiele czynników. Można je podzielić na trzy zasadnicze grupy:

- czynniki formalne (dokumenty, obowiązujące prawo, zapisy umowy z klientem, procesy funkcjonujące w organizacji, cele testów, obowiązujący model cyklu wytwarzania oprogramowania, krytyczność);
- czynniki produktowe (oprogramowanie, jego złożoność, typ i cechy jakościowe, ryzyko, spodziewane typy defektów, przewidywany sposób korzystania z oprogramowania);
- czynniki projektowe (dostępny czas, budżet, zasoby, narzędzia, umiejętności, wiedza i doświadczenie testerów).

Przykład. Pracujesz w projekcie polegającym na stworzeniu aplikacji do obsługi biblioteki uniwersyteckiej. W Twojej organizacji istnieje obowiązek przeprowadzania testów tworzonych aplikacji, natomiast strategia testów (patrz punkt 5.1.1) obowiązująca w Twoim projekcie wymusza — ze względu na umowy podpisane z klientem — konieczność przeprowadzania testowania z użyciem formalnych technik testowania, tak aby można było udokumentować zaprojektowanie, implementację, wykonanie oraz rezultaty testów. Projekt prowadzony jest w modelu V.

Wymagania zostały zebrane i przedstawione w postaci specyfikacji wymagań. Obecnie architekci systemu pracują nad projektem logiki biznesowej modułu odpowiedzialnego za weryfikację tego, ile książek może wypożyczyć dany użytkownik w zależności od jego typu (student, pracownik) oraz zaległości w płacieniu kar umownych. Jest to moduł kluczowy do zapewnienia właściwej realizacji przepisów regulaminu biblioteki, zatem poziom ryzyk związanych z jego nieprawidłowym działaniem jest bardzo wysoki.

Masz trzy dni na analizę i zaprojektowanie testów manualnych. Nie przewiduje się automatyzacji testów ze względu na to, że testy regresji nie będą odgrywać w tym projekcie wielkiej roli. Ponadto organizacja nie dysponuje wyspecjalizowanymi narzędziami ani umiejętnościami technicznymi testerów w zakresie tworzenia automatycznych skryptów testowych.

Z powyższego opisu wynika, że w kontekście modułu odpowiedzialnego za kontrolę wypożyczeń:

- musimy przeprowadzić testy (wynika to z obowiązujących dokumentów i umów);
- musimy te testy zaprojektować i udokumentować (jw.);
- testy powinny skupiać się na logice biznesowej, zatem sensownym wyborem będzie wykorzystanie odpowiednich technik testowania dla tego aspektu oprogramowania;
- ryzyko związane z tym modułem jest wysokie, dlatego sensowne wydaje się wykorzystanie metod sprawdzających logikę biznesową bardzo dokładnie;
- testy będą wykonywane manualnie (brak narzędzi i doświadczenia, niska rola testów regresji, które zwykle są naturalnym kandydatem do automatyzacji).

Jak widać z powyższego przykładu, na decyzję o wyborze techniki, stopnia szczegółowości analizy, wyborze projektu, implementacji oraz wykonania testów ma wpływ wiele czynników, zarówno formalnych, jak i projektowych oraz produktowych.

Nie istnieje idealna, uniwersalna technika testowania. Każda z technik ma swoje wady i zalety. Każda z nich sprawdzi się lepiej w jednej sytuacji, a gorzej w innej. Na przykład w przypadku wspomnianej wyżej sytuacji z modelem weryfikującym reguły wypożyczania książek w bibliotece jedna z czarnoskrzynkowych technik testowania — testowanie w oparciu o tablicę decyzyjną (patrz 4.2.3) wydaje się dobrym wyborem, ponieważ testuje logikę biznesową, a to jest ta funkcjonalność, na której przetestowaniu nam najbardziej zależy. Natomiast nie ma sensu stosować np. białośkrzynkowych technik testowania, które są oparte na wewnętrznej strukturze programu (kod źródłowy) ani technik testowania opartych na doświadczeniu, takich jak zgadywanie błędów, gdyż problemem testerskim jest tu testowanie logiki biznesowej (wspomniane techniki omówione są dokładnie w dalszej części niniejszego rozdziału).

Dobrą praktyką, często stosowaną przez doświadczonych testerów, jest łączenie technik. Na przykład wykorzystując tablice decyzyjne w powyższym problemie, tester może przy okazji zastosować technikę analizy wartości brzegowych (patrz punkt 4.2.2) i sprawdzać reguły biznesowe dla takich wartości (np. maksymalnej możliwej liczby książek, które może wypożyczyć student).

Stosowanie technik testowych można również rozważyć w kontekście poziomów testów. Niektóre techniki są bardziej uniwersalne, stąd są obecne w naturalny sposób na wszystkich poziomach testów — od testowania modułowego po akceptacyjne. Z kolei niektóre inne techniki mają nieco bardziej ograniczony zakres stosowania; na przykład techniki białośkrzynkowe najczęściej stosuje się na poziomie testów modułowych (np. testy jednostkowe wykonywane przez dewelopera). Oczywiście można wyobrazić sobie przykłady zastosowania tej techniki na poziomie testów akceptacyjnych, jednak w praktyce takie sytuacje zdarzają się raczej rzadko.

Stopień formalizacji procesu tworzenia testów przy użyciu technik testowania może mieć zróżnicowany charakter — od bardzo nieformalnego po bardzo formalny. W tabeli 4.2 podajemy kilka przykładów różnego stopnia formalizacji użycia technik testowania.

TABELA 4.2. Przykłady technik testowania o różnym poziomie formalizacji

STOPIEŃ FORMALIZACJI	PRZYKŁAD
Bardzo niski	Nieplanowane, nieudokumentowane przeprowadzenie zgadywania błędów, bez zapisywania wyników testów
Niski	Przeprowadzenie testów eksploracyjnych z użyciem karty opisu testu
Średni	Wykorzystanie tablic decyzyjnych do przetestowania logiki biznesowej; przypadki testowe są udokumentowane w specyfikacji przypadków testowych, a wyniki testów — logowane
Duży	Wykorzystanie modelu maszyny stanowej i techniki testowania przejść między stanami do przetestowania zachowania programu; udokumentowane są: projekt testów (maszyna stanowa), przypadki testowe (w postaci skryptów testowych) oraz wyniki testów (logi)

4.2. Czarnoskrzynkowe techniki testowania

FL-4.2.1 (K3)	Kandydat używa techniki podziału na klasy równoważności, aby zaprojektować przypadki testowe.
FL-4.2.2 (K3)	Kandydat używa techniki analizy wartości brzegowych, aby zaprojektować przypadki testowe.
FL-4.2.3 (K3)	Kandydat używa techniki testowania w oparciu o tablicę decyzyjną, aby zaprojektować przypadki testowe.
FL-4.2.4 (K3)	Kandydat używa techniki testowania przejść pomiędzy stanami, aby zaprojektować przypadki testowe.

W niniejszym podrozdziale, poza omówieniem czterech technik czarnoskrzynkowych opisanych w sylabusie, omawiamy dodatkowo piątą — testowanie oparte na przypadkach użycia (punkt 4.2.5). Jest to punkt nadobowiązkowy, niewchodzący w zakres egzaminu. Technika ta występowała w starszej wersji sylabusa (3.0). Uważamy, że jest ona na tyle ważna i powszechnie stosowana, że warto ją omówić, nawet jako treść nadobowiązkową w stosunku do sylabusa.

4.2.1. Podział na klasy równoważności (KR)

Jedna z siedmiu zasad testowania przedstawionych w podrozdziale 1.3 mówi, że „testowanie gruntowne jest niemożliwe”. Jest to dosyć oczywiste: liczba możliwych kombinacji danych wejściowych jest praktycznie nieskończona, podczas gdy tester ma możliwość wykonania jedynie skończonej, i to bardzo niewielkiej liczby testów.

Technika **podziału na klasy równoważności** (ang. *equivalence partitioning*) próbuje przezwyciężyć zasadę niemożności testowania gruntowego. Możliwych danych wejściowych jest zazwyczaj nieskończonie wiele, ale oczekiwanych zachowań programu na te dane jest już skończona liczba, zwłaszcza jeśli rozważamy konkretne zachowania, związane ze ścisłe określonym aspektem funkcjonowania aplikacji. Popatrzmy na kilka przykładów.

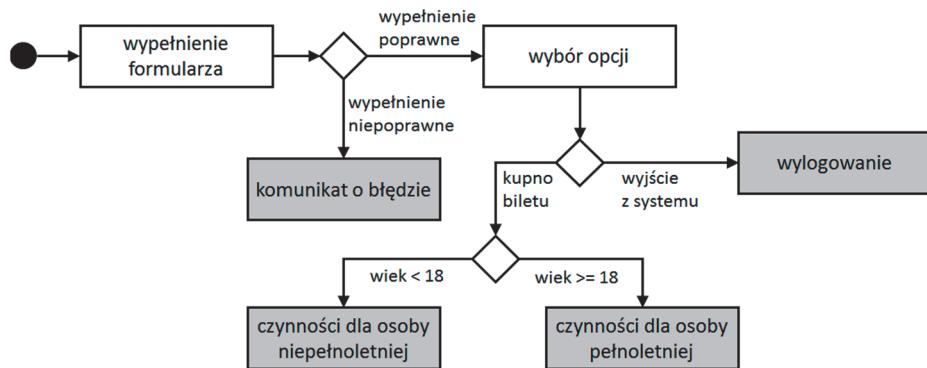


Przykład. System ustala wielkość podatku w zależności od dochodu. Podatek może wynieść 0%, 19%, 33% lub 45%. Istnieje potencjalnie nieskończonie wiele możliwych wartości dochodu, ale tylko cztery możliwe rodzaje decyzji podejmowanej przez system.

Przykład. Użytkownik wypełnia formularz webowy, a następnie chce go wydrukować. Wydruk może być jedno- lub obustronny. Sposobów wypełnienia formularza jest potencjalnie nieskończonie wiele, zwłaszcza jeśli jest on skomplikowany. Jednak to, co chcemy w tym przypadku przetestować, to tylko jedno z dwóch możliwych zachowań programu: poprawny druk jednostronny i poprawny druk obustronny formularza.

Przykład. Użytkownik wypełnia formularz, w tym pole „Wiek”. Następnie może kupić bilet, przy czym procedura jest odmienna w zależności od wieku (pełnoletniości) użytkownika. Dokładny schemat procesu przedstawia rysunek 4.3. Istnieje

potencjalnie nieskończenie wiele możliwości wypełnienia formularza, ale tylko cztery możliwe akcje: komunikat o błędzie, kupno biletu przez osobę pełnoletnią, kupno biletu przez osobę niepełnoletnią, wylogowanie.



RYSUNEK 4.3. Przykład procesu opisującego możliwe scenariusze użycia systemu

Jak więc widać, zwykle da się sprowadzić testowane zachowanie programu do skończonej liczby wariantów. W metodzie podziału na klasy równoważności dzieli się określona dziedzinę na podzbiory w taki sposób, że dla każdych dwóch elementów z jednego podzbioru mamy identyczne zachowanie się programu. Na przykład jeśli system przyznaje zniżkę dla osób poniżej 18 lat, to wszystkie liczby mniejsze niż 18 będą stanowiły jedną klasę równoważności, odpowiadającą za przydzielenie zniżki. W ten sposób, z punktu widzenia testera, utożsamia się ze sobą wartości należące do tej samej klasy. Każdy jej element jest równie dobrym wyborem do przetestowania.

Zastosowanie

Technika KR jest uniwersalna. Można ją stosować praktycznie w każdej sytuacji, na każdym poziomie testów i w każdym ich typie. Sprowadza się ona bowiem do podziału możliwych danych na grupy. Warto wspomnieć, że technikę tę można stosować nie tylko do dziedzin wejściowych, ale również do dziedzin wyjściowych oraz dziedzin wewnętrznych (tzn. związanych ze zmiennymi, które nie są bezpośrednio podawane na wejściu ani zwracane na wyjściu).

Dziedzina, którą dzielimy na klasy równoważności, nie musi być dziedziną liczbową. Może to być tak naprawdę dowolny zbiór. Oto kilka przykładów dziedzin, do których można zastosować technikę podziału na klasy równoważności:

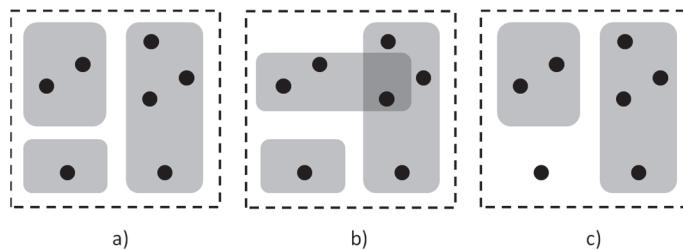
- zbiór liczb naturalnych (np. podział na liczby parzyste i nieparzyste);
- zbiór słów (np. podział ze względu na długość słowa: jednoliterowe, dwuliterowe itd.);
- zbiory odnoszące się do czasu (np. podział na rok urodzenia, podział na miesiące w danym roku);
- zbiór typów systemów operacyjnych: {Windows, Linux, macOS} (np. podział na jednoelementowe klasy: {Windows}, {Linux}, {macOS}).

Poprawność podziału dziedziny

Bardzo ważne jest, aby dokonany przez nas podział dziedziny był poprawny. Poprawny podział to taki, w którym:

- każdy element dziedziny należy do *dokładnie jednej* klasy równoważności;
- żadna klasa równoważności nie jest pusta.

Rysunek 4.4 ilustruje zagadnienie poprawności podziału. Czarne kropki oznaczają elementy dziedziny, a szare prostokąty — podział, czyli klasy równoważności. Na rysunku 4.4 a) przedstawiony jest poprawny podział na klasy równoważności, który spełnia oba wyżej wymienione warunki. Rysunek 4.4 b) przedstawia podział niepoprawny — jeden element należy do dwóch klas równoważności, co narusza poprawność podziału. Rysunek 4.4 c) również pokazuje podział niepoprawny — jeden z elementów dziedziny nie został przyporządkowany do żadnej klasy równoważności.



RYSUNEK 4.4. Przykłady poprawnego (a) i niepoprawnego (b, c) podziału dziedziny

Kwestia poprawności podziału wydaje się dosyć oczywista, ale w praktyce nie jest to problem trywialny. Często zdarza się, że w wyniku dokonanego przez nas podziału jakiś warunek poprawności podziału zostaje naruszony.

Przykład. Rozważmy klasyczny problem testerski opisany przez Myersa [Myers 2011]. Program na wejściu przyjmuje trzy liczby całkowite nieujemne, a na wyjściu podaje typ trójkąta, jaki można zbudować z odcinków o podanych długościach. Możliwe odpowiedzi programu to: trójkąt równoboczny, trójkąt równoramienny, trójkąt różnoboczny, nie można zbudować trójkąta. Założymy, że chcemy zastosować technikę KR ze względu na dziedzinę wyjścia (tzn. typ trójkąta). Naturalny wydaje się podział na te cztery wspomniane wcześniej kategorie. Jednak po bliższej analizie okaże się, że każdy trójkąt równoboczny jest jednocześnie trójkątem równoramennym, zatem mamy do czynienia z klasą równoważności, która jest właściwym podzbiorem innej klasy. Musimy ten podział poprawić, np. w poniższy sposób:

- trójkąt równoboczny;
- trójkąt równoramienny, ale nie równoboczny;
- trójkąt różnoboczny;
- nie trójkąt.

Ten podział jest poprawny: każda trójkąt liczb podanych na wejście odpowiada dokładnie jednemu z powyższych czterech możliwych wyjść programu.

Przykład. Chcemy sklasyfikować zbiór wszystkich możliwych skończonych ciągów liczb ze względu na ich posortowanie. Naturalny podział ciągów na klasy równoważności mógłby wyglądać tak: ciągi rosnące, ciągi niemalejące, ciągi nieuporządkowane. Ale zauważmy, że ciąg jednoelementowy jest jednocześnie ciągiem rosnącym i malejącym. Ponadto powstaje pytanie, gdzie zaklasyfikować ciąg pusty (zawierający 0 elementów). W związku z tym poprawny podział musi uwzględnić te skrajne przypadki i może wyglądać tak:

- ciąg pusty;
- ciągi jednoelementowe;
- ciągi rosnące o więcej niż jednym elemencie;
- ciągi malejące o więcej niż jednym elemencie;
- ciągi nieposortowane o więcej niż dwóch elementach.

Klasy, które zawierają wartości „normalne”, „poprawne”, tzn. oczekiwane (akceptowane) przez system, nazywane są klasami poprawnymi. Klasy zawierające wartości, które moduł lub system powinien odrzucić (np. dane o niepoprawnej składni, przekraczające dopuszczalne zakresy), nazywane są klasami niepoprawnymi.

W naszym ostatnim przykładzie pięć wyróżnionych przez nas klas to klasy poprawne, ponieważ każda z nich zawiera dane poprawne, oczekiwane przez system (być może przypadek ciągu pustego jest kontrowersyjny — jeśli specyfikacja wyraźnie mówi o *niepustych* ciągach, to oczywiście klasa ta staje się klasą niepoprawną). Poza tymi zidentyfikowanymi klasami możemy wyróżnić np. klasę niepoprawną złożoną z elementów, które nie są ciągami liczbowymi (np. zawierającymi znaki alfanumeryczne).

Definicje klas poprawnych i niepoprawnych, a co za tym idzie — wartości poprawnych i niepoprawnych — mogą być różnie rozumiane, dlatego jeśli będziemy używać tych pojęć, warto je precyzyjnie zdefiniować. Na przykład wartości poprawne mogą być rozumiane na co najmniej dwa sposoby:

- jako te, które powinny być przetwarzane przez system;
- jako te, dla których specyfikacja definiuje ich sposób przetwarzania.

Analogicznie wartości niepoprawne mogą być rozumiane:

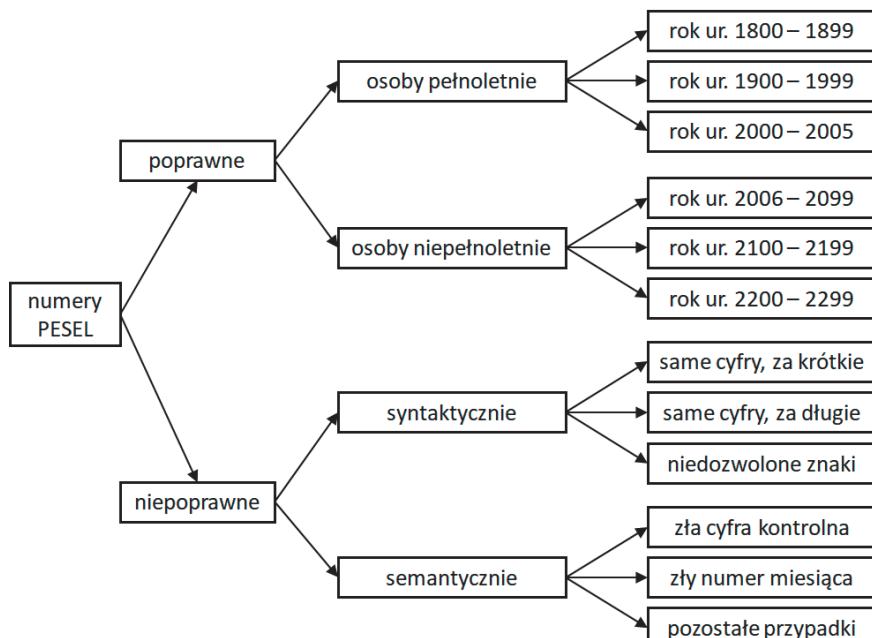
- jako te, które powinny być ignorowane lub odrzucane przez system;
- jako te, dla których specyfikacja nie definiuje ich sposobu przetwarzania.

Przykład. System w formularzu rejestracji użytkownika oczekuje, że w pole *e-mail* wpisany zostanie poprawny adres e-mailowy. Gdy użytkownik wpisze tam łańcuch znakowy *abc@def@ghi*, system odrzuci to wejście jako niepoprawne, informując o tym użytkownika komunikatem „Niepoprawny e-mail”. W tej sytuacji wejście *abc@def@ghi* niektórzy mogą traktować jako wartość niepoprawną (bo system je odrzucił; rejestracja nastąpi tylko, gdy podany zostanie poprawny adres e-mailowy), a inni — jako wartość pochodząca z poprawnej klasy równoważności (bo system jest przygotowany na tego typu sytuację, dowodem na to jest komunikat o błędzie; zatem jest to w pewnym sensie wartość „oczekiwana”, a więc pochodząca z poprawnej klasy).

Dalszy podział klas na podklasy

Ważną zaletą techniki KR jest to, że jeżeli podzielimy którąkolwiek (nawet wszystkie) klasy na podklasy, w dalszym ciągu będziemy mieli podział na klasy równoważności — wtedy testowanie będzie bardziej dokładne. Tester może zdecydować, że z pewnych względów określone klasy powinny podlegać dalszemu podziałowi. Jest to naturalne podejście, które wykorzystuje hierarchiczność klas. Rozważmy następujący przykład.

Przykład. System pobiera od użytkownika numer PESEL i w zależności od tego, czy osoba jest pełnoletnia, czy nie, podejmuje stosowną akcję. Chcemy dokonać podziału wszystkich możliwych numerów PESEL na klasy równoważności. Pierwszym krokiem może być podział na numery poprawne i niepoprawne. Następnie rozważmy każdą z tych klas pod kątem dalszego możliwego podziału. Poprawne numery PESEL niewątpliwie należy podzielić — zgodnie ze specyfikacją — na numery odpowiadające osobom pełnoletnim i niepełnoletnim. Dalszy podział każdej z tych klas mógłby np. uwzględnić odpowiednie kodowanie numeru miesiąca (np. dla osób urodzonych w latach 1900 – 1999 miesiąc kodowany jest normalnie; dla osób urodzonych w latach 2000 – 2099 do liczby miesiąca dodawane jest 20 (czyli np. luty kodowany jest jako 22), w przypadku lat 2100 – 2199 dodawane jest 40, dla lat 2200 – 2299 dodawane jest 60, a dla lat 1800 – 1899 dodawane jest 80). Z kolei PESEL może być niepoprawny z różnych powodów, np. strukturalnych (np. zła długość) lub merytorycznych (np. niezgodność liczby kontrolnej). Przykładowy proces dzielenia klas na podklasy pokazany jest na rysunku 4.5.



RYSUNEK 4.5. Przykład hierarchicznego podziału na klasy równoważności

Zauważmy, że osoby urodzone w latach 2000 – 2099 mogą być zarówno pełnoletnie, jak i niepełnoletnie (przy założeniu, że podziału dokonujemy w 2024 roku), dlatego podzieliliśmy ten przedział czasowy na lata 2000 – 2002 oraz 2003 – 2099. Przymijemy, na potrzeby testowania, że osoby urodzone po 2024 roku (co w 2024 roku jest oczywiście niemożliwe) klasyfikowane są jako osoby niepełnoletnie. W testach chodzić nam będzie jedynie o sprawdzenie, czy akceptowane są numery PESEL z odpowiednim kodowaniem miesiąca, a to chcemy sprawdzić dla wszystkich takich zdefiniowanych w specyfikacji PESEL kodowań (1800 – 1899, 1900 – 1999, 2000 – 2099, 2100 – 2199, 2200 – 2299). Ostatecznie, w wyniku hierarchicznego podziału klas, dziedzinę związaną z numerami PESEL podzieliliśmy na dwanaście klas równoważności:

- trzy klasy poprawne dla osób pełnoletnich (1800 – 1899, 1900 – 1999, 2000 – 2096);
- trzy klasy poprawne dla osób niepełnoletnich (2007 – 2099, 2100 – 2199, 2200 – 2299);
- trzy klasy niepoprawne ze względu na składnię (strukturę) numeru PESEL;
- trzy klasy niepoprawne ze względu na semantykę (znaczenie) numeru PESEL.

Wprowadzanie przypadków testowych. Pokrycie

W przypadku techniki KR **elementami pokrycia** (ang. *coverage item*) są klasy równoważności. Minimalny zbiór przypadków testowych zapewniający stu procentowe pokrycie to taki, który pokrywa każdą klasę równoważności, tzn. dla każdej zidentyfikowanej klasy istnieje przypadek zawierający wartość z tej klasy. W przypadku jednowymiarowym (jedna dziedzina i jeden podział) minimalna liczba przypadków testowych powinna zatem wynosić tyle, ile klas równoważności zidentyfikowaliśmy. W przypadku wielowymiarowym (więcej niż jedna dziedzina) sprawa się nieco komplikuje, gdyż liczba przypadków będzie zależeć np. od liczby kombinacji klas niepoprawnych, a także od ewentualnych zależności pomiędzy wartościami z różnych podziałów. Pokrycie mierzy się jako iloraz liczby klas równoważności przetestowanych przy użyciu co najmniej jednej wartości przez łączną liczbę zdefiniowanych klas równoważności i zazwyczaj wyrażane jest w procentach.



Przykład. Rozważmy ponownie system weryfikacji numeru PESEL i założmy, że zdefiniowaliśmy następujące przypadki testowe:

- PESEL osoby pełnoletniej urodzonej w 1898 roku;
- PESEL osoby pełnoletniej urodzonej w 1999 roku;
- PESEL = „1234” (za krótki).

Ten zbiór trzech testów pokrywa trzy z dwunastu zidentyfikowanych klas równoważności (patrz rysunek 4.5), zatem osiąga pokrycie $3/12 = 1/4 = 25\%$.

Pokrywanie wielu klas równoważności jednocześnie. Maskowanie defektów

Nieco inne podejście jest potrzebne w sytuacji, w której nasze testy mają jednocześnie pokrywać klasy równoważności pochodzące z więcej niż jednego podziału. W takiej sytuacji dobrą praktyką jest, aby nie tworzyć przypadków testowych, które pokrywać będą dwie klasy niepoprawne lub ich większą liczbę. Ma to związek z tzw. *maskowaniem defektów*. Poprawna i zalecana jest następująca strategia:

1. Najpierw stwórz jak najmniejszą liczbę przypadków testowych złożonych wyłącznie z danych z klas poprawnych, które pokryją wszystkie klasy poprawne.
2. Następnie dla każdej niepokrytej klasy niepoprawnej stwórz osobny przypadek testowy, w którym wystąpi dana z tej klasy, a wszystkie pozostałe dane powinny chodzić będą z klas poprawnych.

Rozważmy następujący przykład ilustrujący to zagadnienie:

Przykład. System wystawia ocenę studentowi na podstawie dwóch danych: liczby punktów za ćwiczenia (0 – 50) oraz za egzamin (0 – 50). Student zalicza przedmiot, jeśli suma punktów przekracza 50. Dane wejściowe przypadku testowego będą zatem składały się z dwóch części: punktów za ćwiczenia i punktów za egzamin. Założymy, że wyróżniliśmy następujące dziedziny oraz ich podziały:

- dziedzina zmiennej A = „punkty za ćwiczenia”:
 - (A1) klasa niepoprawna: liczby mniejsze od 0,
 - (A2) klasa poprawna: liczby od 0 do 50,
 - (A3) klasa niepoprawna: liczby większe od 50;
- dziedzina zmiennej B = „punkty za egzamin”:
 - (B1) klasa niepoprawna: liczby mniejsze od 0,
 - (B2) klasa poprawna: liczby od 0 do 50,
 - (B3) klasa niepoprawna: liczby większe od 50.

Chcemy pokryć wszystkie klasy równoważności obu dziedzin: A i B. Klasa poprawna to A2 i B2. Wszystkie pozostałe to klasy niepoprawne. Każdy przypadek testowy zawiera jakąś liczbę punktów za ćwiczenia i jakąś liczbę punktów za egzamin, a zatem pokrywa zawsze jedną klasę z dziedziny A i jedną z dziedziny B. Postępując według opisanej powyżej strategii, najpierw pokrywamy klasy poprawne. Ponieważ mamy tylko po jednej klasie poprawnej w każdej dziedzinie, wystarczy jeden przypadek testowy, np.:

PT1: A = 25, B = 30 (pokrywa poprawną klasę A2 i poprawną klasę B2).

Zostały do pokrycia dwie klasy niepoprawne z A i dwie z B. Zatem potrzebujemy czterech kolejnych przypadków, w których klasy te będziemy testować indywidualnie (druga klasa musi być klasą poprawną):

PT2: A = -8, B = 35 (pokrywa niepoprawną klasę A1; dodatkowo pokrywa B2),

PT3: A = 48, B = -11 (pokrywa niepoprawną klasę B1; dodatkowo pokrywa A2),

- PT4: A = 64, B = 4 (pokrywa niepoprawną klasę A3; dodatkowo pokrywa B2),
PT5: A = 12, B = 154 (pokrywa niepoprawną klasę B3; dodatkowo pokrywa A2).

Gdybyśmy testowali sytuację, w której obie wartości pochodzą z klas niepoprawnych swoich dziedzin, mogłoby wystąpić zjawisko *maskowania defektu*. Założymy, że system weryfikuje, czy student zaliczył przedmiot, poprzez następującą procedurę (opisaną w pseudokodzie):

```
DANE WEJŚCIOWE: PunktyZaĆwiczenia, PunktyZaEgzamin  
JEŻELI (PunktyZaĆwiczenia + PunktyZaEgzamin > 50) TO  
    ZWRÓĆ WYNIK "Przedmiot zdany"  
W PRZECIWNYM RAZIE  
    ZWRÓĆ WYNIK "Przedmiot niezdany"
```

Rozważmy teraz następujący przypadek testowy:

PT6: A = -28, B = 105 (pokrywa niepoprawną klasę A1 i niepoprawną klasę B3).

W takiej sytuacji suma punktów wyniesie $-28 + 105 = 77$, a zatem system zwróci wynik „Przedmiot zdany” pomimo błędnych obu danych wejściowych!

Pokrycie „each choice”

Pokrycie „each choice” stosuje się w przypadku wielowymiarowym, czyli takim, w którym mamy do czynienia z więcej niż jedną dziedziną, a każdy przypadek testowy pokrywa po jednej klasie z podziału każdej dziedziny. Pokrycie to jest jednym z prostszych (i najsłabszych) pokryć stosowanych do przypadku wielowymiarowego. Wymaga ono, by każda klasa każdej dziedziny była przetestowana przynajmniej raz. W praktyce, stosując tę metodę, tester stara się, aby kolejny przypadek testowy pokrywał jak najwięcej niepokrytych dotąd elementów pokrycia.

Przykład. Testujemy produkt komercyjny do powszechnej sprzedaży (COTS — ang. *commercial off the shelf*). Oznacza to, że musimy przetestować go w różnych środowiskach. Program współpracuje z różnymi systemami operacyjnymi i różnymi przeglądarkami. Należy zatem przetestować działanie różnych przeglądarek pod różnymi systemami operacyjnymi. Założymy, że mamy do przetestowania następujące cztery przeglądarki:

- Google Chrome (GC),
- Firefox (F),
- Safari (S),
- Opera (O)

oraz następujące trzy systemy operacyjne:

- Windows (W),
- Linux (L),
- iOS.

Dla uproszczenia nie uwzględniamy konkretnych wersji systemów operacyjnych ani przeglądarek. Każdy typ przeglądarki tworzy jednoelementową klasę równoważności, co daje cztery klasy: {GC}, {F}, {S}, {O}. Każdy system operacyjny tworzy z kolei jednoelementową klasę równoważności, co daje łącznie trzy klasy: {W}, {L}, {iOS}.

Zgodnie z kryterium pokrycia „each choice” dla każdej zidentyfikowanej klasy musi istnieć przypadek testowy pokrywający wartość z tej klasy. W naszym przykładzie przypadki testowe są reprezentowane przez parę danych testowych (typ przeglądarki, system operacyjny).

W przykładzie tym do spełnienia kryterium pokrycia wystarczą cztery przypadki testowe, np.:

PT1: (GC, W)

PT2: (F, L)

PT3: (S, iOS)

PT4: (O, W)

Typy wykrywanych problemów

Technika KR identyfikuje problemy wynikłe z błędnego przetwarzania danych, tzn. wynikłe z błędów w modelu dziedziny.

Określenie dziedziny jest kluczowe

Technika KR stosowana jest zawsze w odniesieniu do *konkretej* dziedziny, która modeluje problem. Czasami wybór dziedziny jest oczywisty, czasami jednak sprawa może być nieco bardziej skomplikowana. Rozważmy następujący, dosyć typowy przykład.

Przykład. Termostat wyłącza ogrzewanie, gdy temperatura (liczona w pełnych stopniach) przekracza 21 stopni, i włącza, gdy temperatura spadnie poniżej 18 stopni. Należy zaprojektować testy za pomocą podziału na klasy równoważności.

Jak zastosować technikę KR do tego problemu? Ile będzie klas równoważności do sprawdzenia? To zależy od tego, *jaką konkretnie własność* systemu chcemy sprawdzić. Do naszego problemu możemy podejść na co najmniej trzy sposoby:

Sposób 1. Analizując jedynie dziedzinę, zauważamy, że dla wartości 22 i większej system wyłącza ogrzewanie, dla wartości 17 i mniejszej — włącza, a dla wartości między 18 a 21 nie podejmuje żadnej akcji. W związku z tym mamy trzy klasy równoważności dla dziedziny „temperatura”: do 17 stopni, od 18 do 21 stopni oraz powyżej 21 stopni (patrz rysunek 4.6 a) i aby je pokryć, potrzebujemy trzech testów, np.:

- temperatura = 15 (oczekiwane wyjście: ogrzewanie włączone);
- temperatura = 20 (oczekiwane wyjście: stan ogrzewania nie zmienia się wobec poprzedniego stanu);
- temperatura = 22 (oczekiwane wyjście: ogrzewanie wyłączone).

Sposób 2. Zauważmy, że w zakresie temperatur 18 – 21 ogrzewanie może być zarówno włączone, jak i wyłączone. Rozważamy zatem dziedzinę złożoną z par (temperatura, stan ogrzewania). W takiej sytuacji mamy cztery możliwości:

- temperatura < 18, ogrzewanie włączone;
- temperatura 18 – 21, ogrzewanie włączone;
- temperatura 18 – 21, ogrzewanie wyłączone;
- temperatura > 21, ogrzewanie wyłączone.

Mamy zatem cztery sytuacje do pokrycia (patrz rysunek 4.6 b). Sytuacja jest teraz nieco bardziej skomplikowana niż poprzednio, bo przed wykonaniem testu trzeba wymusić odpowiedni stan ogrzewania dla temperatur 18 – 21. Cztery odpowiadające powyższym klasom testy mogłyby więc wyglądać tak:

- temperatura = 15 (oczekiwane wyjście: ogrzewanie włączone);
- temperatura = 18 po wzroście z 17 (oczekiwane wyjście: ogrzewanie nadal jest włączone);
- temperatura = 21 po spadku z 22 (oczekiwane wyjście: ogrzewanie nadal jest wyłączone);
- temperatura = 22 (oczekiwane wyjście: ogrzewanie wyłączone).

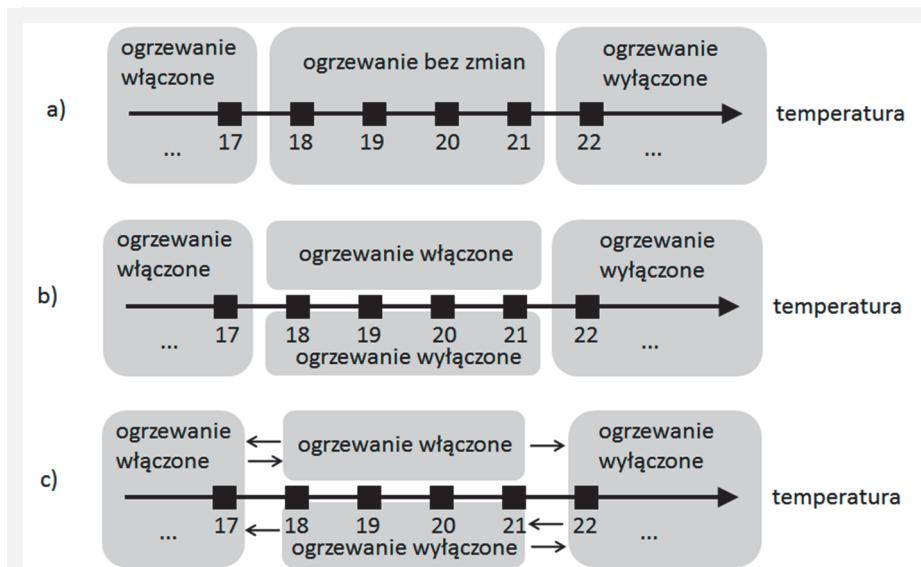
Sposób 3. Możemy rozważyć nie statyczne pary (temperatura, stan ogrzewania), ale przejścia pomiędzy takimi parami. Wtedy nasza dziedzina będzie opisywała możliwe przejścia między parami (temperatura, stan ogrzewania) i będzie się składać z sześciu możliwych elementów (na poniżej liście elementów dziedziny liczby oznaczają temperatury, a WY i WŁ oznaczają, odpowiednio, ogrzewanie wyłączone i włączone):

- (17, WŁ) → (18, WŁ);
- (18, WŁ) → (17, WŁ);
- (18, WY) → (17, WŁ);
- (21, WŁ) → (22, WY);
- (21, WY) → (22, WY);
- (22, WY) → (21, WY).

Sytuację tę przedstawia rysunek 4.6 c). Potrzebujemy więc sześciu testów, które będą symulowały te przejścia, np. dla pierwszego elementu konfiguracja początkowa systemu to 17 stopni i stan ogrzewania włączony. Test polega na zwiększeniu temperatury do 18 stopni i sprawdzeniu, czy ogrzewanie zostanie wyłączone.

Zauważmy, że moglibyśmy do naszej listy dodać jeszcze cztery inne przejścia, polegające na pozostawaniu w tej samej dziedzinie, jeśli nie chcemy się ograniczać wyłącznie do zmian polegających na przejściu pomiędzy różnymi zakresami temperatur tworzącymi klasy równoważności dziedziny „temperatura”:

- (17, WŁ) → (16, WŁ);
- (19, WŁ) → (20, WŁ);
- (19, WY) → (18, WY);
- (22, WY) → (23, WY).



RYSUNEK 4.6. Trzy różne podejścia do wyboru dziedziny dla klas równoważności

Jak więc widać z powyższego przykładu, cel testu wpływa istotnie na postać dziedziny. W naszym przypadku dziedzinę modelowaliśmy na trzy różne sposoby: jako zbiór liczb-temperatur, zbiór par (temperatura, stan ogrzewania) oraz zbiór przejść między takimi parami. Zastosowanie techniki KR do każdej z nich sprawdzało istotnie inny aspekt systemu. Dlatego zawsze przed zastosowaniem tej techniki musimy dokładnie wiedzieć, jaką postać ma dziedzina, na której będziemy pracować.

Zauważmy przy okazji, że czasami konieczność precyzyjnego określenia oczekiwanej wyników testu pozwala wykryć błędy lub niejednoznaczności w specyfikacji. W sposobie 1., dla danych z klasy 18 – 21, nie bardzo wiadomo, jaki ma być wynik oczekiwany: czy termostat ma być włączony, czy nie.

Na koniec analizy tego przykładu zauważmy jeszcze jedną, bardzo istotną rzeczą. Mianowicie w tym konkretnym problemie technika podziału na klasy równoważności nie jest dobrym wyborem (problemy, które napotkaliśmy przy jej stosowaniu, bardzo dobrze to ilustrują). Technika ta bowiem skupia się na wykrywaniu problemów w implementacji dziedziny, a zasadniczą kwestią w rozważanym przez nas problemie testerskim jest weryfikacja *zachowania się termostatu*. Wydaje się, że właściwszą techniką będzie testowanie przejść między stanami (patrz punkt 4.2.4).

4.2.2. Analiza wartości brzegowych (AWB)

AWB jako rozszerzenie techniki podziału na klasy równoważności

Analiza wartości brzegowych (ang. *boundary value analysis*) jest metodą zbudowaną na podstawie metody podziału na klasy równoważności. Zatem jej uniwersalność oraz typ wykrywanych problemów będą podobne jak w tej ostatniej. Różnica w stosunku do podziału na klasy równoważności jest taka, że w AWB do testów będziemy wybierać pewne szczególne elementy klas równoważności, mianowicie te, które leżą na *brzegach* tych klas.



Do jakich dziedzin stosuje się AWB? Co to jest wartość brzegowa?

Mówiąc o „brzegach”, musimy wprowadzić na elementach dziedziny relację porządku, czyli możliwość uszeregowania elementów dziedziny od najmniejszego do największego (bowiem wartość brzegowa danej klasy to wartość skrajna, czyli najmniejsza lub największa wartość tej klasy). Technika analizy wartości brzegowych może być zatem stosowana tylko do klas, których elementy są uporządkowane w sensie jakiejś relacji *porządku* (np. do zbiorów liczb, na których możemy wprowadzić relację „mniejsze niż”). Przykładem takich dziedzin są zbiory liczb naturalnych, całkowitych czy rzeczywistych, także wartości odnoszące się do czasu lub daty itp.

Wartości brzegowe zawsze definiowane są dla konkretnej klasy równoważności. Wartość brzegowa to po prostu element najmniejszy i element największy w danej klasie. Na przykład jeśli klasa równoważności opisuje możliwe oceny, jakie uczeń może dostać w szkole (w dziedzinie liczb naturalnych), czyli ma postać $\{1, 2, 3, 4, 5, 6\}$, to wartościami brzegowymi tej klasy są wartości 1 oraz 6.

Poza wprowadzeniem relacji porządku musimy założyć jeszcze jedną rzeczą: mianowicie rozważane klasy równoważności nie mogą mieć „dziur”, tzn. formalnie rzeczą ujmując, jeśli dwie wartości a i b takie, że $a < b$, należą do tej samej klasy równoważności, to wszystkie elementy pośrednie c , tzn. takie, że $a < c < b$, również muszą do tej klasy należeć. Rozważmy bowiem wcześniej wspomnianą klasę, ale wyrzućmy z niej element 4. Klasa ta ma postać $\{1, 2, 3, 5, 6\}$. Jej wartościami skrajnymi są oczywiście nadal 1 i 6, ale powstaje pytanie: czy wartości 3 i 5 nie są aby również „brzegami” tej klasy? Obie sąsiadują przecież z elementem, który do klasy nie należy! W tym przypadku musielibyśmy zatem rozbić naszą klasę na dwie: $\{1, 2, 3\}$ oraz $\{5, 6\}$, a cała dziedzina podzielona byłaby na trzy klasy: $\{1, 2, 3\}, \{4\}, \{5, 6\}$.

Wyprowadzanie przypadków testowych metodą AWB

Ogólna procedura wyprowadzania przypadków testowych przy użyciu metody AWB ma taką postać:

1. Zidentyfikuj dziedzinę, którą chcesz poddać analizie.
2. Przeprowadź podział tej dziedziny na klasy równoważności.
3. Dla każdej zidentyfikowanej klasy równoważności wyznacz jej wartości brzegowe (uwaga — czasami analiza może się ograniczać jedynie do pewnych,

ściśle określonych klas i nie musi brać pod uwagę wszystkich wyznaczonych klas równoważności).

4. Dla każdej wartości brzegowej wyznacz elementy do przetestowania.

Pierwsze dwa kroki to po prostu zastosowanie techniki podziału na klasy równoważności. W kroku trzecim wyróżniamy wartości brzegowe klas. Natomiast w kroku czwartym, na podstawie zidentyfikowanych wartości brzegowych, wyznaczamy elementy do testów. W technice AWB wartości brzegowe (warunki testowe) niekoniecznie muszą być tożsame z elementami wybieranymi do testów (tzw. elementami pokrycia). Będzie to zależało od tego, które klasy rozważamy, oraz od wybranego wariantu metody AWB. Istnieją bowiem dwie jej odmiany — tzw. wersja dwupunktowa i trójpunktowa.

Wersja dwupunktowa AWB

W wersji dwupunktowej [Craig 2002, Myers 2011] dla każdej zidentyfikowanej wartości brzegowej do testów wybiera się tę wartość oraz najbliższego jej sąsiada *nienależącego* do klasy, do której należy wartość brzegowa. Na przykład jeśli rozważamy wartości brzegowe klasy {1, 2, 3, 4, 5, 6} w dziedzinie liczb całkowitych (czyli wartości 1 i 6), to do testów wybieramy:

- wartość 1 (jako wartość brzegową);
- wartość 0 (jako najbliższego sąsiada 1 nienależącego do jej klasy);
- wartość 6 (jako wartość brzegową);
- wartość 7 (jako najbliższego sąsiada 6 nienależącego do jej klasy).

Wersja trójpunktowa AWB

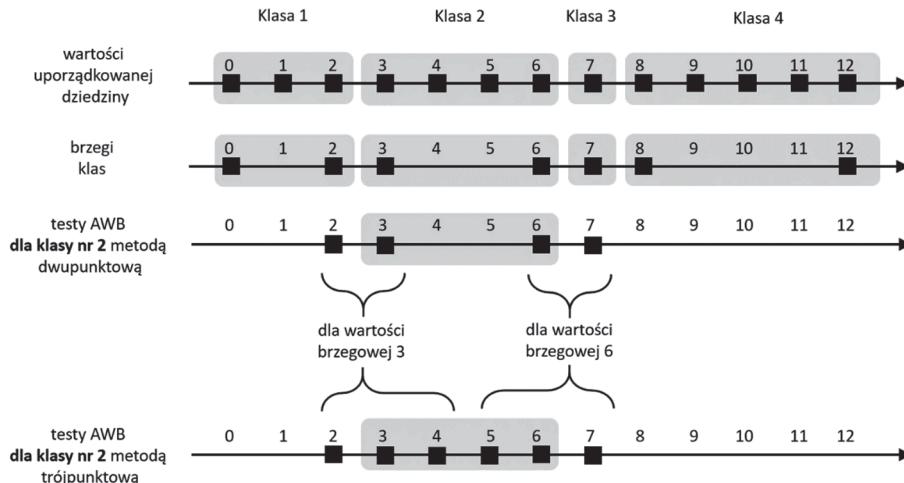
W wersji trójpunktowej [Koomen 2006, O'Regan 2019] dla każdej zidentyfikowanej wartości brzegowej do testów bierzemy tę wartość oraz obu jej sąsiadów, niezależnie od tego, do jakich klas należą. W powyższym przykładzie do testów wybrałyśmy:

- wartość 1 (jako wartość brzegową);
- wartość 0 (lewy sąsiad 1);
- wartość 2 (prawy sąsiad 1);
- wartość 6 (jako wartość brzegową);
- wartość 5 (lewy sąsiad 6);
- wartość 7 (prawy sąsiad 6).

Porównanie obu wersji techniki

Schematycznie zasadę wyznaczania wartości brzegowych (czyli wartości do testów) działania obu technik przedstawiono na rysunku 4.7. Szczególnym przypadkiem może być klasa jednoelementowa, ale reguła działa analogicznie. Jedyny element tej klasy jest zarazem elementem najmniejszym i największym. W przykładzie z rysunku klasa 3 zawiera jedynie wartość 7. W przypadku metody dwupunktowej wartości wzięte do testów byłyby następujące:

- wartość 7 jako minimalna wartość brzegowa i wartość 6 jako sąsiad spoza klasy;
- wartość 7 jako maksymalna wartość brzegowa i wartość 8 jako sąsiad spoza klasy.



RYSUNEK 4.7. Wyznaczanie brzegów w dwu- i trójpunktowej wersji metody AWB

Do testów wzięlibyśmy zatem zestaw wartości 6, 7, 8. Analogicznie w przypadku metody trójpunktowej wartości do testów to 6, 7, 8 wyznaczone przez 7 jako wartość minimalną oraz te same wartości (6, 7, 8) wyznaczone przez 7 jako wartość maksymalną. Do testów wzięlibyśmy zatem zestaw tych samych wartości co w metodzie dwupunktowej: 6, 7, 8. Taka sytuacja ma miejsce jedynie w przypadku klas jednoelementowych. Dla klas zawierających co najmniej dwa elementy metoda trójpunktowa zawsze da nam więcej elementów do przetestowania niż metoda dwupunktowa.

Rozważmy teraz kilka przykładów zastosowania techniki AWB.

Przykład. System przydziela zniżkę na bilet osobom poniżej 18. oraz powyżej 65. roku życia. Pasażerowi w wieku od 18 do 65 lat zniżka nie przysługuje. Chcemy sprawdzić poprawność przydzielania zniżki. Przeprowadźmy czterokroikową procedurę opisaną poniżej.

Krok 1. Identyfikacja dziedziny. Analizowaną zmienną jest wiek pasażera, a więc nieujemna liczba całkowita. Dziedzina ma więc postać $\{0, 1, 2, 3, \dots\}$.

Krok 2. Identyfikacja klas równoważności. Mamy następujące klasy równoważności:

- K1: wiek uprawniający do zniżki dla młodego pasażera $\{0, 1, 2, \dots, 17\}$;
- K2: wiek uprawniający do biletu normalnego $\{18, 19, \dots, 64, 65\}$;
- K3: wiek uprawniający do zniżki dla starszego pasażera $\{66, 67, 68, \dots\}$.

Krok 3. Identyfikacja wartości brzegowych:

- wartości brzegowe K1 to 0 oraz 17;
- wartości brzegowe K2 to 18 oraz 65;
- wartość brzegowa K3 to 66 (klasa nie ma wartości największej).

Krok 4. Identyfikacja wartości do przetestowania. W przypadku gdy chcemy przeprowadzić testy dla wszystkich klas równoważności metodą dwupunktową, tak naprawdę do testów należy wziąć wszystkie zidentyfikowane wartości brzegowe i to wystarczy. Jest tak dlatego, że mamy pewnego rodzaju symetrię między każdymi dwiema sąsiadującymi ze sobą wartościami brzegowymi dwóch klas. Na przykład 65 jako wartość brzegowa klasy K2 ma sąsiada 66 spoza klasy, który jednocześnie jest wartością brzegową K3, a jego sąsiadem z innej klasy jest 65. Dlatego, stosując metodę dwupunktową, do testów wybieramy wartości: 0, 17, 18, 65, 66. Zakładamy tutaj, że wartość -1 jest nieosiągalna.

W przypadku metody trójpunktowej do testów wybieramy wartości: 0, 1, 16, 17, 18, 19, 64, 65, 66, 67, ponieważ:

- dla wartości brzegowej 0 bierzemy ją i jej sąsiada: 0 i 1 (zakładamy, że wartość -1 jest nieosiągalna);
- dla wartości brzegowej 17 bierzemy ją i jej sąsiadów: 16, 17, 18;
- dla wartości brzegowej 18 bierzemy ją i jej sąsiadów: 17, 18, 19;
- dla wartości brzegowej 65 bierzemy ją i jej sąsiadów: 64, 65, 66;
- dla wartości brzegowej 66 bierzemy ją i jej sąsiadów: 65, 66, 67.

Oczywiście niektóre wartości się powtarzają, ale każdą wartość do testów bierzemy tylko raz. Jeśli naszej analizie podlegałaby wyłącznie środkowa klasa (bilet normalny), to ograniczylibyśmy się do zidentyfikowania wartości brzegowych tylko tej klasy, a więc wartości 18 i 65. Zatem w wersji dwupunktowej do testów wybraliśmy wartości 17, 18, 65, 66, a w wersji trójpunktowej — 17, 18, 19, 64, 65, 66.

Przykład. W pewnym miejscu kodu podejmowana jest decyzja, w zależności od wartości zmiennej całkowitej x. Decyzja ta powinna mieć postać:

JEŻELI $x < 16$ **TO**

 Wykonaj AKCJA_1

W PRZECIWNYM RAZIE

 Wykonaj AKCJA_2

W tym przypadku mamy do czynienia z dwiema klasami równoważności: pierwsza zawiera wartości, które powodują wykonanie AKCJA_1, a więc liczby całkowite mniejsze od 16; druga klasa jest jej dopełnieniem, czyli wartościami większymi lub równymi 16, które powodują wykonanie AKCJA_2. Wartością brzegową pierwszej klasy jest 15, a drugiej — 16. Zmienna x nie posiada wartości najmniejszej ani największej, zatem pierwsza klasa nie ma wartości brzegowej minimalnej, a druga — maksymalnej. Tak więc w metodzie dwupunktowej do testów wybierzemy wartości 15 i 16, a w trójpunktowej — 14, 15, 16, 17.

W praktyce zazwyczaj wartości globalnie największe/najmniejsze istnieją — są to np. zakresy akceptowane przez pola odpowiedniego typu (np. pole akceptuje liczbę złożoną z maksymalnie pięciu cyfr) czy też wartości odpowiednich zmiennych. Na przykład zmienna typu `int` (całkowitego) w C++ posiada zakres od -2^{31} do $2^{31} - 1$, czyli od -2147483648 do 2147483647. Dobry tester będzie sprawdzał zatem nie tylko granice klas wyznaczonych specyfikacją, ale również granice wyznaczone architekturą testowanego rozwiązania, np. zakresem wartości zmiennych.

Uważne wyznaczanie wartości brzegowych

W przypadku metody AWB należy bardzo uważać na to, jak zdefiniowane są granice klas. Założymy, że poruszamy się w dziedzinie liczb naturalnych. Wtedy, przykładowo, sformułowanie „do klasy należą elementy nie mniejsze niż 7” oznacza, że najmniejszą liczbą należącą do tej klasy jest 7. Z kolei sformułowanie „do klasy należą wartości większe od 7” oznacza, że najmniejszą wartością z tej klasy jest 8. Analogicznie „co najwyższej 65” oznacza liczby do 65 włącznie, a warunek „ $x < 65$ ” oznacza, że największą wartością spełniającą tę nierówność jest 64. Sformułowanie „do klasy należą wartości od x do y ” oznacza, że do klasy należą wartości od x do y włącznie z wartościami x, y . Osoby mające w tym względzie wątpliwości zachęcamy do rozważenia problemu: w jakich dniach otwarty jest sklep, na którym wisie kartka z informacją: „sklep jest czynny od poniedziałku do piątku”? Czy sklep jest czynny przez pięć dni w tygodniu od poniedziałku do piątku, czy też tylko od wtorku do czwartku?

Zastosowanie

Metoda AWB jest bardzo prosta, a jednocześnie niezwykle efektywna w wykrywaniu defektów. Powodem jest to, że programiści często popełniają tzw. błędy przesunięcia o 1, nieprawidłowo implementując granice klas równoważności. Oto dwie przykładowe, typowe pomyłki programistów skutkujące powstaniem defektów wykrywalnych metodą AWB:

- Programista powinien zaimplementować warunek „ $x < 10$ ” (nierówność ostra), ale błędnie zaimplementował go jako „ $x \leq 10$ ” (nierówność nieostra).
- Programista powinien zainicjalizować zmienną sterującą pętli (iterator) jako `for i=0 to 10`, ale błędnie założył, że elementy tablicy indeksowane są od 1 i zapisał to jako

```
for i=1 to 10.
```

W pierwszym przypadku klasy równoważności, zgodnie ze specyfikacją, powinny wyglądać tak: $\{..., 8, 9\}, \{10, 11, ...\}$. Błędna implementacja podzieliła jednak dziedzinę (ze względu na zachowanie się sterowania) następująco: $\{..., 9, 10\}, \{11, 12, ...\}$. Analizując ten przykład za pomocą AWB, widzimy, że wartościami brzegowymi (poprawnymi, tzn. wynikającymi ze specyfikacji) są wartości 9 i 10. Jeśli stosujemy metodę dwupunktową, bierzemy te właśnie wartości do testów. W przypadku wartości 10 zgodnie ze specyfikacją program powinien pójść ścieżką odpowiadającą fałszywej decyzji „ $x < 10$ ”, ale w błędnej implementacji warunek „ $x \leq 10$ ” jest prawdziwy, zatem sterowanie programu pójdzie nie tą ścieżką, którą powinno — istnieje zatem duża szansa wykrycia tego defektu za pomocą testu $x = 10$.

Metoda trójpunktowa jest silniejsza od dwupunktowej, tzn. istnieją sytuacje, w których metoda dwupunktowa nie ma szans na wykrycie awarii, podczas gdy metoda trójpunktowa może ją wykryć. Rozważmy następujący przykład.

Przykład. Programista implementuje moduł kontrolowania temperatury w chłodni. Temperatura — mierzona w pełnych stopniach — nie może przekraczać 10°C. Po przekroczeniu tego progu włącza się mechanizm chłodzenia. Logika biznesowa jest taka:

JEŻELI temperatura nie przekracza 10°C **TO**

Nic nie rób

W PRZECIWNYM RAZIE

Włącz chłodzenie

Programista zaimplementował poprawny warunek **JEŻELI** $x \leq 10$ **TO...** w sposób błędny, jako **JEŻELI** $x == 10$ **TO...** Zgodnie ze specyfikacją podział dziedziny jest taki: $\{..., 9, 10\}, \{11, 12, ...\}$. Zgodnie z błędnią implementacją podział ten przedstawia się tak: $\{..., 8, 9\}, \{10\}, \{11, 12, ...\}$, przy czym klasa środkowa daje wartość prawdy, a klasy skrajne — wartość fałszu warunku w błędnej implementacji.

Jeśli stosujemy metodę dwupunktową, to jako wartości brzegowe identyfikujemy wartości 10 i 11 i tylko dla nich przeprowadzamy testy. Dla wartości 10 zarówno w oczekiwanej, jak i w błędnej implementacji warunek jest prawdziwy: jest bowiem prawdą, że $10 \leq 10$ i to, że $10 == 10$. Z kolei dla wartości 11 oba warunki są fałszywe: nie jest bowiem prawdą, że $11 \leq 10$, i nie jest prawdą, że $11 == 10$. Zatem żaden z tych dwóch testów nie będzie w stanie wykryć defektu — sterowanie programu (zawierającego błędnią implementację warunku!) w obu przypadkach pojedzie poprawną ścieżką — tą, która wynika z oczekiwanej implementacji! Dla 10°C program bowiem nie wykona żadnej akcji, a dla 11°C włączy chłodzenie. Metoda dwupunktowa nie będzie więc w stanie wykryć defektu w kodzie.

Jeśli jednak użyjemy metody trójpunktowej, to do testów weźmiemy cztery wartości: 9, 10, 11, 12. W szczególności wartość 9 różnicuje ścieżki wykonania według poprawnej i błędnej implementacji: w implementacji poprawnej warunek $9 \leq 10$ jest prawdziwy, a w implementacji błędnej warunek $9 == 10$ jest fałszywy. Zatem istnieje szansa, że dla $x = 9$ wykryjemy niepoprawne działanie programu w związku z wyborem niewłaściwej ścieżki przepływu sterowania. Zgodnie bowiem ze specyfikacją, dla 9°C program ma nie podejmować żadnej akcji, ale włączyć chłodzenie. Powyższe rozważania podsumowuje tabela 4.3.

Wartości spoza dziedziny

Na koniec rozważmy jeszcze jeden problem. W przykładzie z przydzielaniem zniżki na bilet zaznaczyliśmy, że dla wartości brzegowej 0 nie bierzemy do testów jej sąsiada -1, ponieważ założyliśmy, że jest ona nieosiągalna, tzn. użytkownik nie jest w stanie wprowadzić wartości niepoprawnych — ujemnych. W praktyce jednak nie zawsze tak musi być. Jeśli na przykład wartość wieku użytkownika wprowadza w pole formularza z klawiatury, może oczywiście wpisać wartość ujemną. To, czy skrajne wartości brzegowe spoza dziedziny testować, czy nie, zależy od tego, czy interfejs na to pozwala. Jeśli tak, tester oczywiście powinien przetestować działanie aplikacji dla takich wartości.

TABELA 4.3. Różnice między dwu- i trójpunktową AWB dla różnych modeli usterek

	METODA DWUPUNKTOWA		METODA TRÓJPUNKTOWA			
Zidentyfikowane wartości brzegowe (wartości do testów):						
	x = 10	x = 11	x = 9	x = 10	x = 11	x = 12
Wyniki testów						
Specyfikacja: x <= 10	(10 <= 10)	(11 <= 10)	(9 <= 10)	(10 <= 10)	(11 <= 10)	(12 <= 10)
	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
Błędna implementacja: x == 10	(10 == 10)	(11 == 10)	(9 == 10)	(10 == 10)	(11 == 10)	(12 == 10)
	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
Interpretacja:	Test zdany	Test zdany	Test niezdany	Test zdany	Test zdany	Test zdany
	Oba testy zdane, defekt nie został wykryty		Istnieje test, który wykrył defekt			

Pokrycie

Elementami pokrycia w AWB są wartości brzegowe klas równoważności (w wersji dwupunktowej) lub wartości brzegowe wraz z ich wszystkimi sąsiednimi wartościami (w wersji trójpunktowej). Pokrycie AWB w wersji dwupunktowej definiuje się zatem jako iloraz liczby przetestowanych wartości brzegowych oraz łącznej liczby zidentyfikowanych wartości brzegowych. W przypadku odmiany trójpunktowej jest to iloraz liczby przetestowanych wartości brzegowych wraz z ich sąsiadami oraz łącznej liczby zidentyfikowanych wartości brzegowych i ich sąsiadów. W metodzie trójpunktowej do testów bierzemy bowiem nie tylko wartości brzegowe, ale także wartości z wnętrza klas. Na przykład dla klasy {2, 3, 4, 5, 6} wartościami branymi do testów w metodzie trójpunktowej będą w szczególności liczby 3 i 5, które nie są wartościami brzegowymi.

4.2.3. Testowanie w oparciu o tablicę decyzyjną

Obszar zastosowania techniki

Testowanie w oparciu o tablicę decyzyjną (ang. *decision table testing*) to technika, którą wykorzystuje się do weryfikacji poprawności implementacji *reguł biznesowych*. Reguła biznesowa zazwyczaj ma postać logicznej implikacji:



JEŻELI (warunek) **T0** (akcja),

przy czym zarówno warunek, jak i akcja mogą być złożone z więcej niż jednego czynnika. Mamy wtedy do czynienia z *kombinacją* warunków lub akcji. Oto kilka przykładów reguł biznesowych:

- **JEŻELI** ($wiekKlienta < 18$) **TO** (przydziel zniżkę na bilet);
- **JEŻELI** ($a+b>c>0$ **ORAZ** $a+c>b>0$ **ORAZ** $b+c>a>0$) **TO** (można zbudować trójkąt o bokach długości a , b , c);
- **JEŻELI** (zarobki > 10000) **TO** (przydziel kredyt **ORAZ** zaproponuj złotą kartę).

Tablice decyzyjne pozwalają w systematyczny sposób przetestować poprawność implementacji kombinacji warunków. Jest to jedna z tzw. technik kombinatorycznych.Więcej informacji o tych technikach zawiera syllabus poziomu zaawansowanego Analityk Testów [ISTQB TA 2021].

Budowa tablicy decyzyjnej

Budowę tablicy decyzyjnej opiszemy na przykładzie (patrz tabela 4.4).

TABELA 4.4. Przykładowa tablica decyzyjna

Warunki	REGUŁY BIZNESOWE							
	1	2	3	4	5	6	7	8
Ma kartę stałego klienta?	T	T	T	T	N	N	N	N
Sumaryczna kwota > 1000?	T	T	N	N	T	T	N	N
Kupował w ostatnich 30 dniach?	T	N	T	N	T	N	T	N
Akcje								
Przyznana zniżka	10%	5%	5%	0%	0%	0%	0%	0%

Tablica decyzyjna składa się z dwóch części, opisujących, odpowiednio, warunki oraz akcje. W poszczególnych kolumnach opisane są reguły biznesowe. Tabela 4.4 opisuje reguły dotyczące przydzielania zniżki na zakupy w zależności od trzech czynników opisujących danego klienta:

- Czy posiada kartę stałego klienta? (TAK lub NIE)
- Czy sumaryczna kwota zakupów przekracza 1000 zł? (TAK lub NIE)
- Czy klient dokonywał zakupów w okresie ostatnich 30 dni? (TAK lub NIE)

Na podstawie odpowiedzi na te pytania przydzielana jest zniżka: 0%, 5% lub 10%.

Przykład. Klient posiada kartę stałego klienta i dokonał do tej pory zakupów na kwotę 1250 PLN, a ostatnie zakupy miały miejsce 5 dni temu. Sytuacja ta odpowiada regule nr 1 (posiada kartę, kwota > 1000, zakupy w ciągu ostatnich 30 dni). Zatem system powinien przydzielić temu klientowi zniżkę 10%.

Procedura wyprowadzania przypadków testowych z tablicy decyzyjnej

Proces tworzenia konkretnych przypadków testowych z wykorzystaniem tablic decyzyjnych można przedstawić w następujących pięciu krokach.

- Krok 1. Identyfikacja wszystkich możliwych warunków (z podstawy testów, np. na bazie specyfikacji, rozmów z klientem, tzw. zdrowego rozsądku) i wypisanie ich w kolejnych wierszach w górnej części tablicy.
- Krok 2. Identyfikacja wszystkich odpowiadających warunkom akcji, jakie mogą zajść w systemie i jakie są zależne od tych warunków (również z podstawy testów), i wypisanie ich w kolejnych wierszach w dolnej części tablicy.
- Krok 3. Generowanie wszystkich kombinacji warunków i eliminacja kombinacji nieosiągalnych. Dla każdej osiągalnej kombinacji tworzona jest osobna kolumna w tablicy z wypisanyimi wartościami poszczególnych warunków.
- Krok 4. Identyfikacja, dla każdej zidentyfikowanej kombinacji warunków, które akcje i w jaki sposób mają zajść. Uzupełnienie dolnej części kolumny odpowiadającej tablicy decyzyjnej.
- Krok 5. Wyprowadzenie, dla każdej kolumny tablicy decyzyjnej, przypadku testowego, w którym wymuszana jest zadana w kolumnie kombinacja warunków. Test jest zaliczony, jeśli po jego wykonaniu system podejmie akcje w taki sposób, jak są opisane w dolnej części kolumny tablicy odpowiadającej temu przypadkowi testowemu.

Notacja i możliwe wartości w tabeli

Zazwyczaj wartości warunków i akcji przyjmują postać wartości logicznych PRAWDA lub FAŁSZ. Mogą być reprezentowane na różne sposoby, np. jako symbole P i F, ewentualnie T i N (tak/nie), 1 i 0 albo słowa „prawda” i „fałsz”. W języku angielskim mogą być użyte symbole T (*true* — prawda) oraz F (*false* — fałsz). Jednak wartości warunków i akcji mogą być w ogólności dowolnymi obiektami, np. liczbami, zakresami liczb, wartościami kategorii. Na przykład w naszej tablicy (tabela 4.4) wartościami akcji „przyznana zniżka” są kategorie wyrażające różne rodzaje zniżek: 0%, 5%, 10%. W tej samej tablicy mogą występować warunki i akcje różnego typu, np. jednocześnie mogą występować warunki logiczne, liczbowe i kategorialne.

Tablice decyzyjne, w których wszystkie warunki i akcje przyjmują jedynie wartości logiczne (prawda/fałsz), nazywa się ograniczonymi tablicami decyzyjnymi. Jeśli choć jeden warunek lub akcja zawiera inne niż logiczne wartości, takie tablice nazywane są uogólnionymi tablicami decyzyjnymi.

Sposób wyznaczania wszystkich kombinacji warunków

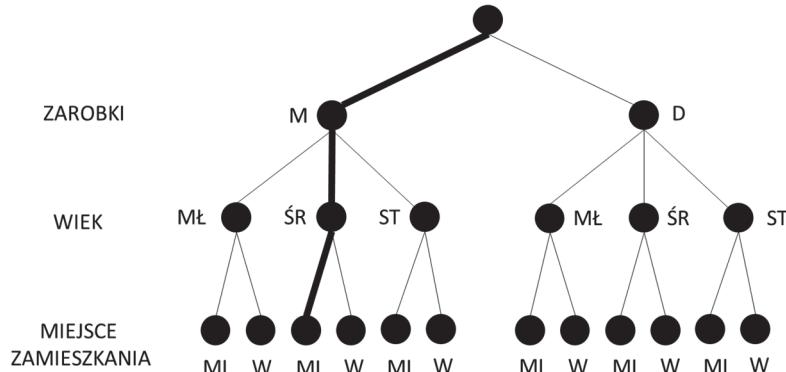
Jeśli musimy ręcznie wyznaczyć kombinacje warunków, a boimy się, że jakąś kombinację pominiemy, możemy zastosować bardzo prostą metodę drzewa do systematycznego wyznaczenia wszystkich kombinacji warunków. Rozważmy następujący przykład:

Przykład. Założmy, że tablica posiada trzy warunki:

- zarobki (dwie możliwe wartości: M — małe, D — duże);
 - wiek (trzy możliwe wartości: MŁ — młody, ŚR — średni, ST — stary);
 - miejsce zamieszkania (dwie możliwe wartości: MI — miasto, W — wieś).

Aby stworzyć wszystkie kombinacje wartości trójkę (wiek, zarobki, miejsce zamieszkania), budujemy drzewo, z którego korzenia wyprowadzamy wszystkie możliwości pierwszego warunku (zarobki). Jest to pierwszy poziom drzewa, Następnie z każdego wierzchołka tego poziomu wyprowadzamy wszystkie możliwości drugiego warunku (wiek). Otrzymujemy drugi poziom drzewa. Na końcu z każdego wierzchołka tego poziomu wyprowadzamy wszystkie możliwe wartości warunku trzeciego (miejsce zamieszkania). Oczywiście, gdyby warunków było więcej, postępowalibyśmy analogicznie. Nasze końcowe drzewo wygląda jak to z rysunku 4.8.

Każda możliwa kombinacja warunków to kombinacja etykiet wierzchołków na ścieżkach prowadzących od korzenia (wierzchołek na górze drzewa) do jakiegokolwiek wierzchołka na samym dole drzewa. Wszystkich kombinacji będzie zatem tyle, ile jest wierzchołków na najniższym poziomie (u nas — 12; wynika to oczywiście z liczby kombinacji: $2 \cdot 3 \cdot 2 = 12$). Na rysunku pogrubionymi liniami zaznaczono ścieżkę odpowiadającą przykładowej kombinacji (M, SR, MI), która oznacza małe zarobki, średni wiek i miasto jako miejsce zamieszkania. Każdą z tych kombinacji możemy teraz wpisać do poszczególnych kolumn naszej tablicy decyzyjnej. Mamy jednocześnie pewność, że żadna kombinacja nie została opuszczona. Warunki tak utworzonej tablicy pokazane są w tabeli 4.5.



RYSUNEK 4.8. Drzewo pomocnicze służące do identyfikacji kombinacji warunków

TABELA 4.5. Kombinacje warunków tablicy decyzyjnej powstałe z drzewa pomocniczego

Kombinacje nieosiągalne

Czasami, po wypisaniu wszystkich możliwych kombinacji warunków, może się okazać, że niektóre z nich są z różnych powodów nieosiągalne (ang. *infeasible*). Na przykład jeśli mamy dwa warunki:

- wiek klienta > 18? (możliwe wartości: TAK, NIE);
- wiek klienta <= 18? (możliwe wartości: TAK, NIE),

to oczywiste jest, iż pomimo że mamy cztery możliwe kombinacje tych warunków: (TAK, TAK), (TAK, NIE), (NIE, TAK) i (NIE, NIE), jedynie dwie z nich są osiągalne: (TAK, NIE) i (NIE, TAK), bowiem nie można mieć jednocześnie więcej niż 18 lat i mniej niż 19 lat. Oczywiście w tym przypadku moglibyśmy zamienić te dwa warunki na jeden: wiek, z możliwymi dwiema wartościami: większy niż 18, mniejszy lub równy 18.

Czasami tablica nie będzie zawierała wszystkich możliwych kombinacji nie ze względów czysto logicznych, ale semantycznych. Na przykład jeśli mamy dwa warunki:

- zdefiniowano cel? (TAK, NIE);
- osiągnięto cel? (TAK, NIE),

to kombinacja (NIE, TAK) jest nieosiągalna (bezsensowna), bo nie można osiągnąć celu, którego się uprzednio nie zdefiniowało.

Minimalizacja tablicy decyzyjnej

Czasami niektóre warunki mogą nie mieć żadnego wpływu na podejmowane przez system akcje. Na przykład jeśli system pozwala wykupić ubezpieczenie jedynie pełnoletnim klientom, a w zależności od tego, czy pali, czy nie, dostają zniżkę na to ubezpieczenie, to jeśli tylko klient jest niepełnoletni, system nie pozwoli na wykup ubezpieczenia niezależnie od tego, czy klient pali, czy nie. Takie wartości nieistotne najczęściej zaznacza się w tablicy decyzyjnej symbolem kreski lub znakiem N/A (z ang. *not applicable*). W naszym przypadku tablica wyglądałaby jak tabela 4.6. Symbol N/A stosuje się, gdy dany warunek nie może wystąpić (np. jeśli jedną z wartości warunku „typ płatności” jest „gotówka”, to warunek „czy PIN poprawny?” w ogóle nie występuje; wystąpi tylko wtedy, gdy wartością warunku „typ płatności” będzie „płatność kartą”).

TABELA 4.6. Tablica decyzyjna z wartościami nieistotnymi

WARUNKI			
Klient pełnoletni?	NIE	TAK	TAK
Klient pali?	—	TAK	NIE
AKCJE			
Przyznać ubezpieczenie?	NIE	TAK	TAK
Przyznać zniżkę?	NIE	NIE	TAK

Dzięki tej minimalizacji tablica jest bardziej kompaktowa i zawiera mniej kolumn, a zatem mniej przypadków testowych do wykonania. Z drugiej strony dla testu z wartością nieistotną w rzeczywistym przypadku testowym musimy wybrać jakąś konkretną wartość tego warunku. Istnieje zatem niebezpieczeństwo, że jeśli defekt pojawia się dla pewnej specyficznej kombinacji wartości oznaczonych w tablicy jako nieistotne, możemy ją łatwo przegapić i nie przetestować jej.

Gdyby w naszym przykładzie chcieć wykonać *fizyczny* przypadek testowy dla pierwszej kolumny, trzeba by się zdecydować, czy klient pali, czy nie (choć z punktu widzenia specyfikacji jest to nieistotne), bo należy podać tę daną wejściową. Możemy zdecydować się na kombinację (pełnoletni = NIE, pali = NIE) i taki test zostanie zdany, ale możemy sobie wyobrazić, że na skutek jakichś defektów w kodzie program nie działa prawidłowo dla kombinacji (pełnoletni = NIE, pali = TAK). Taka kombinacja nie została przez nas przetestowana i awaria nie zostanie wykryta.

Na rzeczywistym egzaminie mogą pojawić się pytania dotyczące minimalizacji tablic, ale od kandydata nie wymaga się umiejętności przeprowadzenia minimalizacji, lecz jedynie umiejętności posługiwania się już zminimalizowanymi tablicami. Umiejętność minimalizacji wymagana jest na egzaminie na certyfikat ISTQB Analityk Testów na poziomie zaawansowanym. Dlatego w niniejszej książce nie przedstawiamy algorytmu minimalizowania tablic decyzyjnych.

Pokrycie

W przypadku tablic decyzyjnych elementami pokrycia są poszczególne kolumny tablicy, zawierające możliwe do spełnienia kombinacje warunków (czyli tzw. kolumny osiągalne). Dla zadanej tablicy decyzyjnej pełne, stuprocentowe pokrycie wymaga, by dla każdej kolumny przygotować i wykonać przynajmniej jeden przypadek testowy odpowiadający danej kombinacji warunków. Test jest zaliczony, jeśli system rzeczywiście wykona zdefiniowane dla tej kolumny akcje.

Ważne jest to, że pokrycie liczy się w stosunku do liczby kolumn tablicy, a nie w stosunku do liczby wszystkich możliwych kombinacji warunków. Zazwyczaj te dwie liczby są równe, ale w przypadku występowania kombinacji nieosiągalnych, o których mówiliśmy wcześniej, tak być nie musi.

Na przykład w celu osiągnięcia stuprocentowego pokrycia dla tablicy z tabeli 4.6 potrzebujemy trzech (a nie czterech, jakby to wynikało z liczby kombinacji) przypadków testowych. Gdybyśmy dysponowali następującymi testami:

- pełnoletni = TAK, pali = TAK;
- pełnoletni = NIE, pali = TAK;
- pełnoletni = NIE, pali = NIE,

to osiągnęlibyśmy pokrycie 2/3 (czyli ok. 66%), ponieważ dwa ostatnie testy pokrywają tę samą kolumnę tablicy (tabela 4.6).

Tablice decyzyjne jako statyczna technika testowania

Technika tablic decyzyjnych znakomicie nadaje się do wykrywania problemów dotyczących wymagań, np. ich braku lub ich sprzeczności. Po utworzeniu tablicy decyzyjnej na podstawie specyfikacji, lub nawet jeszcze w trakcie jej tworzenia, bardzo łatwo odkryć takie problemy ze specyfikacją, jak:

- niepełność — brak zdefiniowanych akcji dla określonego zestawu warunków;
- sprzeczność — zdefiniowanie w dwóch różnych miejscach specyfikacji dwóch różnych zachowań systemu wobec tego samego zestawu warunków;
- redundantność — zdefiniowanie tego samego zachowania systemu w dwóch różnych miejscach specyfikacji (być może opisanych w inny sposób).

4.2.4. Testowanie przejść pomiędzy stanami

Obszar zastosowania techniki

Testowanie przejść między stanami (ang. *state transition testing*) jest techniką używaną w celu sprawdzenia *zachowania się* modułu lub systemu. Sprawdza więc jego aspekt behawioralny — to, jak zachowuje się w czasie i jak zmienia swój stan pod wpływem różnego typu zdarzeń.



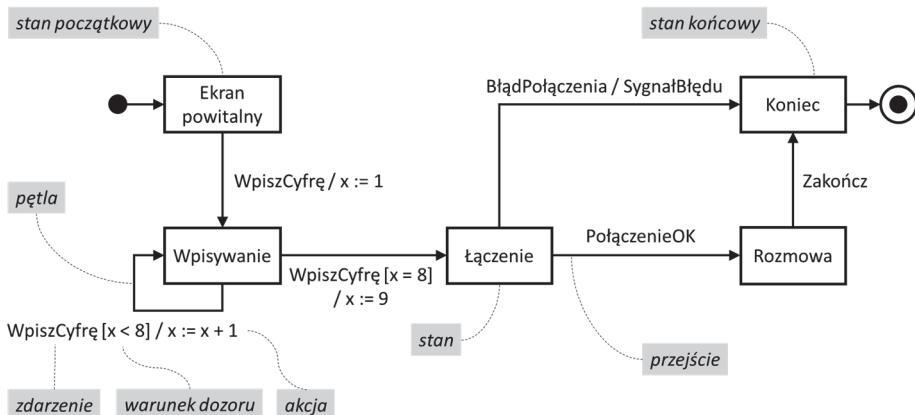
Modelem opisującym ten aspekt behawioralny jest tzw. diagram przejść między stanami. W literaturze model ten najczęściej nazywa się automatem skończonym, automatem skończeniem stanowym bądź maszyną stanów, a w piśmiennictwie angielskim — *Labeled Transition System* (etykietowany system przejść). Sylabus używa nazwy „diagram przejść między stanami” na oznaczenie graficznej postaci maszyny stanów oraz „tabela przejść między stanami” na określenie równoważnej, tabelarycznej postaci maszyny.

Budowa diagramu przejść między stanami

Diagram przejść między stanami jest modelem graficznym. Z teoretycznego punktu widzenia jest to tzw. etykietowany graf skierowany. Diagram przejść między stanami składa się z następujących elementów:

- stanów — reprezentują możliwe sytuacje, w jakich może znajdować się system;
- przejść — reprezentują możliwe (poprawne) zmiany stanów;
- zdarzeń — reprezentują zjawiska, zazwyczaj zewnętrzne wobec systemu, których zajście wzywala odpowiadające im przejścia;
- akcji — czynności, które system może podjąć podczas przejścia między stanami;
- warunków dozoru — warunki logiczne stwarzyszone z przejściami; przejście może zostać wykonane tylko wtedy, gdy stwarzyszony z nim warunek dozoru zachodzi.

Rysunek 4.9 przedstawia przykładowy model diagramu przejść między stanami. Model ten nie jest trywialny, choć w praktyce często korzysta się z jeszcze bardziej skomplikowanych modeli, stosując bogatszą notację niż ta omówiona w sylabusie. Diagram ten pozwala nam jednak pokazać wszystkie istotne elementy maszyny stanowej opisane w sylabusie, zachowując jednocześnie praktyczny charakter przykładu.



RYSUNEK 4.9. Przykładowy diagram przejść między stanami

Diagram reprezentuje model zachowania systemu umożliwiającego przeprowadzenie rozmowy telefonicznej z użytkownikiem telefonu komórkowego o określonym numerze. Użytkownik wybiera 9-cyfrowy numer telefonu, wciskając po kolejnych klawisze odpowiadające kolejnym cyfrom numeru. W momencie wpisania dziewiątej cyfry system automatycznie przeprowadza próbę połączenia.

Diagram przejść między stanami modelujący ten system składa się z pięciu stanów (prostokąty). Możliwe przejścia między nimi wyznaczają strzałki. System rozpoczyna w stanie początkowym „Ekran powitalny” i czeka na zdarzenie (oznaczone na diagramie jako WpiszCyfre) polegające na wyborze przez użytkownika pierwszej cyfry dziewięciocyfrowego numeru telefonu. Po wystąpieniu tego zdarzenia następuje przejście do stanu „Wpisywanie”, a dodatkowo podczas tego przejścia system wykonuje akcję polegającą na ustawieniu wartości zmiennej x na 1. Zmienna ta będzie reprezentowała liczbę dotychczas wprowadzonych przez użytkownika cyfr wybranej numeru telefonu.

W stanie „Wpisywanie” może wystąpić tylko zdarzenie WpiszCyfre, ale zależnie od tego, ile cyfr zostało wpisanych, możliwe są przejścia do dwóch różnych stanów. Dopóki użytkownik nie wpisał dziewiątej cyfry, system pozostaje w stanie „Wpisywanie”, każdorazowo zwiększając zmienną x o jeden. Jest tak dlatego, że w tej sytuacji prawdziwy jest warunek dozoru $x < 8$. Tuż przed wpisaniem przez użytkownika ostatniej, dziewiątej cyfry, zmienna x ma wartość 8. Oznacza to, że warunek dozoru $x < 8$ jest fałszywy, a warunek dozoru $x = 8$ — prawdziwy. W związku z tym wybranie ostatniej, dziewiątej cyfry numeru spowoduje przejście ze stanu „Wpisywanie” pod wpływem zdarzenia WpiszCyfre do stanu „Łączenie”.

Gdy połączenie się uda (wystąpienie zdarzenia PołączenieOK) system przechodzi do stanu „Rozmowa”, w którym pozostaje, dopóki użytkownik nie zakończy połączenia, co zostanie zasygnalizowane zajściem zdarzenia Zakończ. W tym momencie system przechodzi do stanu końcowego i jego działanie kończy się. Jeśli system jest w stanie „Łączenie” i wystąpi zdarzenie BłądPołączenia, system przejdzie do stanu końcowego, ale w odróżnieniu od analogicznego przejścia ze stanu „Rozmowa” wykona dodatkowo akcję SygnałBłędu, sygnalizując użytkownikowi niemożność połączenia z wybranym numerem.

System w każdym momencie znajduje się dokładnie w jednym ze stanów, a zmiana stanów następuje w wyniku zachodzenia odpowiednich zdarzeń. Pojęcie stanu jest abstrakcyjne. Stan może oznaczać bardzo wysokopoziomową sytuację (np. przebywanie przez system w określonym ekranie aplikacji — w naszym przykładzie jest to rozmowa), ale może też opisywać sytuacje niskopoziomowe (np. wykonywanie przez system określonej instrukcji programu). Poziom abstrakcji zależy od przyjętego modelu, tzn. od tego, co tak naprawdę model opisuje i na jakim poziomie ogólności. Zakłada się, że w momencie wystąpienia określonego zdarzenia zmiana stanów jest natychmiastowa (można założyć, że jest to zdarzenie o zerowym czasie trwania). Etykiety przejść pomiędzy stanami (czyli strzałek na diagramie) mają postać:

zdarzenie [warunek dozoru]/akcja

Jeśli w danym przypadku warunku dozoru lub akcji nie ma albo nie jest istotna z punktu widzenia testera, etykiety mogą być pominięte. Tym samym etykiety przejść mogą przyjąć jedną z następujących trzech postaci:

zdarzenie

zdarzenie/akcja

zdarzenie [warunek dozoru]

Warunek dozoru dla danego przejścia pozwala wykonać to przejście tylko wtedy, gdy warunek ten zachodzi. Warunki dozoru pozwalają na zdefiniowanie dwóch różnych przejść pod wpływem tego samego zdarzenia, unikając jednocześnie nie-determinizmu. Przykładowo, na diagramie z rysunku 4.9 ze stanu „Wpisywanie” mamy dwa przejścia pod wpływem zdarzenia WpiszCyfrę, ale w każdej chwili może się wykonać tylko jedno z nich, ponieważ odpowiadające im warunki dozoru są rozłączne (*albo* x jest mniejsze niż 8, *albo* jest równe 8).

Przykładowy test dla maszyny z rysunku 4.9 sprawdzający poprawność wykonania udanego połączenia może wyglądać tak jak w tabeli 4.7.

TABELA 4.7. Przykładowa sekwencja przejść w maszynie stanów z rys. 4.9

KROK	STAN	ZDARZENIE	AKCJA	NASTĘPNY STAN
1	Ekran powitalny	WpiszCyfrę	x := 1	Wpisywanie
2	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
3	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
4	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
5	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
6	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
7	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
8	Wpisywanie	WpiszCyfrę	x := x + 1	Wpisywanie
9	Wpisywanie	WpiszCyfrę		Łączenie
10	Łączenie	PołączenieOK		Rozmowa
11	Rozmowa	Zakończ		Koniec

Scenariusz ten wywołany jest ciągiem jedenastu zdarzeń: WpiszCyfrę (dziewięciokrotnie), PołączenieOK, Zakończ. W każdym kroku wywołujemy odpowiednie zdarzenie i sprawdzamy, czy po jego zajściu system rzeczywiście przechodzi do stanu opisanego w kolumnie *Następny stan*.

Równoważne formy reprezentacji maszyny stanowej

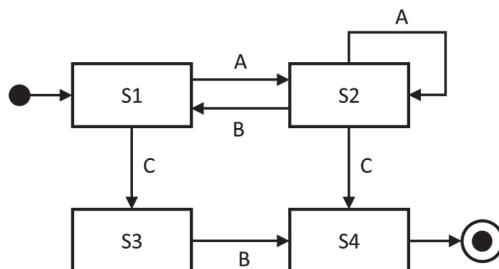
Istnieją co najmniej trzy równoważne formy prezentacji modelu przejść między stanami. Rozważmy prostszą niż w poprzednim przykładzie czterostanową maszynę o stanach S₁, S₂, S₃ i S₄ oraz zdarzeniach A, B, C. *Diagram przejść między stanami* może przybrać, tak jak w poprzednim przykładzie, postać graficzną (rysunek 4.10 a).

Równoważnie jednak można go zaprezentować w postaci *tabeli przejść między stanami* (rysunek 4.10 b), gdzie poszczególne wiersze (odpowiednio: kolumny) tabeli reprezentują kolejne stany (odpowiednio: zdarzenia oraz warunki dozoru, jeśli istnieją), a w komórce na przecięciu kolumny i wiersza odpowiadających określonymu stanowi S i zdarzeniu E wypisany jest stan docelowy, do którego maszyna ma przejść, jeśli będąc w stanie S, nastąpi zdarzenie E. Na przykład jeśli maszyna przebywa obecnie w stanie S₂ i wystąpiło zdarzenie B, to maszyna ma przejść do stanu S₁. Gdyby nasz diagram wykorzystywał akcje, musielibyśmy je dopisać w odpowiednich komórkach tabel z rysunku 4.10 b) i c).

Jeszcze jeden sposób reprezentacji maszyny stanów przedstawia *pełna tabela przejść* pokazana na rysunku 4.10 c). Tutaj tabela reprezentuje *wszystkie możliwe kombinacje stanów oraz wszystkich możliwych zdarzeń*. Ponieważ mamy cztery stany i trzy różne zdarzenia, tabela będzie zawierała $4 \cdot 3 = 12$ wierszy. Ostatnia kolumna zawiera docelowy stan, do którego maszyna ma przejść; jeśli będzie w stanie określonym w pierwszej kolumnie tabeli, nastąpi zdarzenie opisane w drugiej kolumnie tabeli. Jeśli dane przejście jest niezdefiniowane, w kolumnie *Następny stan* oznacza się to np. kreską bądź innym, ustalonym symbolem.

Brak przejścia (tzn. brak kombinacji stan/zdarzenie) na diagramie przejść reprezentowany jest po prostu brakiem odpowiedniej strzałki. Na przykład będąc w stanie S₁, maszyna nie ma zdefiniowanego zachowania na okoliczność wystąpienia zdarzenia B.

Zarówno tabela przejść między stanami, jak i pełna tabela przejść pozwalają wprost ukazać tzw. *przejścia niepoprawne*, które również można, a w niektórych sytuacjach należy testować. Przejścia niepoprawne („brakujące strzałki” na diagramie przejść między stanami) reprezentowane są przez puste komórki tabeli. Innymi słowy, przejściem niepoprawnym jest każda kombinacja (stan, zdarzenie), która nie występuje na diagramie przejść między stanami. Na przykład na diagramie z rysunku 4.10 brakuje przejść: z S₁ pod wpływem zdarzenia B, z S₃ pod wpływem A oraz pod wpływem C, z S₄ pod wpływem A, B oraz C. Łącznie mamy więc sześć przejść niepoprawnych, odpowiadających sześciu pustym komórkom w tabeli z rysunku 4.10 b) czy też sześciu pustym komórkom w kolumnie *Następny stan* w tabeli z rysunku 4.10 c).



a) diagram przejść między stanami

Stan/zdarzenie	A	B	C
S1	S2		S3
S2	S2	S1	S4
S3		S4	
S4			

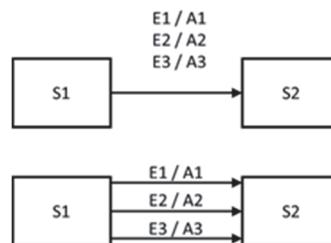
b) tabela przejść między stanami

Stan	Zdarzenie	Następny stan
S1	A	S2
S1	B	---
S1	C	S3
S2	A	S2
S2	B	S1
S2	C	S4
S3	A	---
S3	B	S4
S3	C	---
S4	A	---
S4	B	---
S4	C	---

c) pełna tabela przejść

RYSUNEK 4.10. Różne formy prezentacji maszyny stanowej

Czasami, ze względu na prostotę, na diagramie przejść między stanami rysuje się tylko jedną strzałkę pomiędzy dwoma stanami, nawet jeśli między tymi stanami istnieje więcej równoległych przejść. Należy na to zwracać uwagę, bo czasami kwestią liczby przejść na diagramie jest bardzo istotna. Na rysunku 4.11 pokazano przykładową „uproszczoną” formę rysowania przejść oraz równoważną jej formę „pełną”.



RYSUNEK 4.11. Równoważne formy graficznej reprezentacji przejść równoległych

Projektowanie testów. Pokrycie

W przypadku testowania przejść między stanami istnieje wiele różnych kryteriów pokrycia. Oto trzy najpopularniejsze kryteria, opisane również w sylabusie:

- **Pokrycie wszystkich stanów** (ang. *all states coverage*) — to najsłabsze kryterium pokrycia. Elementami pokrycia są stany. Pokrycie wszystkich stanów wymaga zatem, aby maszyna znalazła się w każdym stanie przynajmniej raz (tzn. przynajmniej podczas wykonywania jednego przypadku testowego).

- **Pokrycie przejść poprawnych** (ang. *valid transitions coverage*, zwane również pokryciem 0-przełączeń, ang. *0-switch coverage*) — to najpopularniejsze kryterium pokrycia. Elementami pokrycia są przejścia między stanami. Pokrycie przejść poprawnych wymaga zatem, aby każde przejście zdefiniowane na diagramie przejść między stanami zostało wykonane przynajmniej raz.
- **Pokrycie wszystkich przejść** (ang. *all transitions coverage*) — elementami pokrycia są wszystkie przejścia wskazane w tablicy stanów. Kryterium to wymaga zatem, aby pokryć wszystkie poprawne przejścia, a ponadto w każdym z możliwych stanów spróbować wywołać każde przejście niepoprawne. Dobrą praktyką jest testowanie jednego takiego zdarzenia w ramach jednego testu (aby uniknąć maskowania błędów).

Można zdefiniować inne kryteria, na przykład pokrycie par przejść (zwane również pokryciem 1-przełączeń, ang. *1-switch coverage*), które wymaga, aby przetestować każdą możliwą sekwencję dwóch następujących po sobie przejść poprawnych. Kryteria tego typu można definiować w nieskończoność, np. pokrycie wszystkich trójek przejść, wszystkich czwórek przejść itd. Ogólnie — można zdefiniować całą rodzinę pokryć N -przełączeń, dla dowolnego nieujemnego N . Można również rozważyć pokrycie pewnych szczególnych ścieżek, pokrycie pętli, itp.

W przypadku testowania przejść między stanami mamy więc do czynienia z potencjalnie nieskończoną liczbą możliwych kryteriów pokrycia. Z praktycznego punktu widzenia najczęściej stosowanym kryterium jest kryterium pokrycia przejść poprawnych oraz kryterium pokrycia wszystkich przejść, ponieważ główną rzeczą, o której nam zazwyczaj chodzi przy testowaniu maszyny stanów, jest weryfikacja poprawnej realizacji przejść między stanami.

Pokrycie definiuje się jako liczba elementów pokrytych przez testy w stosunku do liczby wszystkich zdefiniowanych przez dane kryterium elementów pokrycia. Na przykład dla kryterium pokrycia wszystkich stanów zastosowanego wobec diagramu przejść z rysunku 4.10 mamy cztery elementy do pokrycia: stany S1, S2, S3, S4; dla kryterium pokrycia przejść poprawnych mamy tyle elementów, ile jest przejść między stanami (sześć).

Zazwyczaj wymaga się zaprojektowania *najmniejszej możliwej* liczby testów, które wystarczają do osiągnięcia pełnego pokrycia. Zobaczmy, jak różne typy pokrycia można zrealizować dla diagramu przejść z rysunku 4.10 a).

Dla *kryterium pokrycia stanów* mamy do pokrycia cztery stany: S1, S2, S3, S4. Zauważmy, że można to osiągnąć w ramach jednego testu, na przykład:

S1 (A) S2 (B) S1 (C) S3 (B) S4.

Zastosowana powyżej konwencja opisuje ciąg przejść między stanami pod wpływem zdarzeń (zaznaczonych w nawiasach). Notacja S (E) T oznacza przejście ze stanu S pod wpływem zdarzenia E do stanu T. W powyższym przykładzie w ramach jednego przypadku testowego przeszliśmy przez wszystkie cztery stany, osiągnęliśmy więc 100% pokrycia stanów. W praktyce oczekiwany wynikiem testu jest rzeczywiste

przechodzenie po kolej po stanach tak, jak to opisuje model. Scenariusz testowy odpowiadający przypadkowi testowemu:

S1 (A) S2 (B) S1 (C) S3 (B) S4

mogłby wyglądać tak, jak to opisano w tabeli 4.8.

TABELA 4.8. Scenariusz testowy dla testowania przejść między stanami

LP.	STAN POCZĄTKOWY	ZDARZENIE	OCZEKIWANY WYNIK
1	S1	A	Przejście do stanu S2
2	S2	B	Przejście do stanu S1
3	S1	C	Przejście do stanu S3
4	S3	B	Przejście do stanu S4
5	S4		

Należy pamiętać, że przypadek testowy *nie jest utożsamiany* z warunkiem testowym. W naszym przykładzie warunkami testowymi były poszczególne stany, ale jeden przypadek testowy mógł pokryć je wszystkie. Przypadek testowy to ciąg przejść między stanami, począwszy od stanu początkowego, a skończywszy na stanie końcowym (ewentualnie można przerwać wcześniej tę wędrówkę, jeśli to konieczne). Zatem w ramach jednego przypadku testowego możliwe jest pokrycie *więcej niż jednego* warunku testowego.

Dla kryterium *pokrycia przejść poprawnych* mamy do pokrycia sześć przejść (oznaczmy je jako P1 – P6):

- P1: S1 (A) S2,
- P2: S1 (C) S3,
- P3: S2 (A) S2,
- P4: S2 (B) S1,
- P5: S2 (C) S4,
- P6: S3 (B) S4.

Chcemy zaprojektować możliwie najmniejszą liczbę testów, aby pokryć wszystkie te sześć przejść. Strategia polega na tym, aby w ramach każdego kolejnego testu starać się pokryć jak największej niepokrytych dotąd elementów. Na przykład gdybyśmy rozpoczęli pierwszy test od sekwencji S1 (A) S2, to nie opłaca się w tym momencie wywoływać zdarzenia C i przejść do stanu końcowego S4, skoro możemy jeszcze pokryć kilka innych przejść, np. S2 (A) S2 (B) S1.

Zauważmy, że nie da się jednocześnie, w ramach jednego przypadku testowego, pokryć przejść S2 (C) S4 oraz S3 (B) S4, ponieważ z chwilą osiągnięcia S4 przypadek testowy musi się skończyć. Zatem będziemy potrzebowali co najmniej dwóch przypadeków testowych, aby pokryć wszystkie krawędzie. Istotnie, dwa przypadki wystarczą do osiągnięcia pełnego pokrycia przejść. Przykładowy zestaw takich dwóch testów pokazany jest w tabeli 4.9.

TABELA 4.9. Przypadki testowe pokrywające przejścia między stanami

PRZYPADEK TESTOWY	POKRYTE PRZEJŚCIA
S1 (A) S2 (A) S2 (B) S1 (C) S3 (B) S4	P1, P3, P4, P2, P6
S1 (A) S2 (C) S4	P1, P5

W tabeli tej pierwsza kolumna podaje sekwencję przejść, a druga — odpowiadające im pokryte przejścia. Pogrubioną czcionką zaznaczono przejścia pokryte po raz pierwszy (np. w drugim teście pokrywamy P1, które było już uprzednio pokryte pierwszym testem).

Dla spełnienia *kryterium pokrycia wszystkich przejść* musimy dostarczyć przypadki testowe pokrywające wszystkie przejścia poprawne (patrz wyżej) oraz — dodatkowo — spróbować wywołać każde niezdefiniowane w modelu przejście. Zgodnie z opisaną powyżej dobrą praktyką każde takie przejście będącym testowali w osobnym przypadku testowym. Zatem liczba przypadków testowych będzie równa liczbie testów pokrywających przejścia poprawne powiększonej o liczbę przejść niepoprawnych. W naszej maszynie będzie to sześć następujących niepoprawnych przejść (można je szybko wypisać, analizując pełną tablicę stanów — patrz rysunek 4.10 c):

S1 (B) ?,
 S3 (A) ?,
 S3 (C) ?,
 S4 (A) ?,
 S4 (B) ?,
 S4 (C) ?.

Pamiętajmy, że każdy przypadek testowy rozpoczyna się od stanu początkowego. Dla każdego przejścia niepoprawnego musimy zatem najpierw wywołać sekwencję zdarzeń, która osiągnie zadany stan, a następnie, gdy już w nim będziemy, spróbować wywołać niepoprawne przejście. Dla sześciu opisanych wyżej niepoprawnych przejść odpowiadające im sześć przypadków testowych może wyglądać tak jak w tabeli 4.10.

TABELA 4.10. Przypadki testowe pokrywające niepoprawne przejścia

PRZYPADEK TESTOWY	POKRYTE NIEPOPRAWNE PRZEJŚCIE
S1 (B) ?	S1 (B) ?
S1 (C) S3 (A) ?	S3 (A) ?
S1 (C) S3 (C) ?	S3 (C) ?
S1 (C) S3 (B) S4 (A) ?	S4 (A) ?
S1 (C) S3 (B) S4 (B) ?	S4 (B) ?
S1 (C) S3 (B) S4 (C) ?	S4 (C) ?

Jeśli uda nam się wywołać zdarzenie, które nie jest zdefiniowane w modelu, możemy to interpretować na co najmniej dwa sposoby:

- Jest to bezwzględnie awaria, ponieważ skoro model nie dopuszcza wystąpienia tego zdarzenia w tej sytuacji, to nie powinno być możliwe jego wywołanie.
- Jeśli udało się wywołać nieoczekiwane zdarzenie, ale system pod jego wpływem nie reaguje, to można to uznać za poprawne zachowanie (zignorowanie zdarzenia). Musimy mieć jednak pewność, że semantycznie jest to sytuacja dopuszczalna.

Możliwe rozwiązania takiej sytuacji problematycznej również są co najmniej dwa:

- naprawić system tak, aby nie było możliwe wystąpienie danego zdarzenia;
- dodać do modelu systemu przejście pod wpływem tego zdarzenia tak, by modelowało „ignorowanie” zdarzenia przez system (np. pętla do tego samego stanu).

Wróćmy jeszcze do praktycznego przykładu diagramu przejść między stanami z rysunku 4.9 modelującego system realizacji połączeń telefonicznych. Przykładem przejścia niepoprawnego jest przejście ze stanu „Łączenie” pod wpływem zdarzenia WpiszCyfrę. Taką sytuację jest wywołać bardzo prosto — w momencie nawiązania połączenia naciskamy jeden z klawiszy reprezentujących cyfry. Jeśli system w żaden sposób na to nie zareaguje, uznajemy, że test jest zaliczony.

Omówimy na koniec jedno z kryteriów pokryć nieopisanych w sylabusie — pokrycie par przejść. Jest to materiał nadmiarowy i na egzaminie na certyfikat ISTQB na poziomie podstawowym to kryterium nie obowiązuje.

Przykład ten rozważamy, aby pokazać, że im silniejsze kryterium przyjmiemy, tym trudniej jest je uzyskać i zazwyczaj wymaga to zaprojektowania większej liczby przypadków testowych niż dla kryterium słabszego (w tym przypadku słabszym kryterium jest kryterium pokrycia przejść poprawnych).

Dla spełnienia *kryterium pokrycia par przejść* musimy zdefiniować wszystkie dozwolone pary przejść. Możemy to zrobić w następujący sposób: dla każdego pojedynczego przejścia rozważamy wszystkie jego możliwe kontynuacje w postaci następującego po nim pojedynczego przejścia. Na przykład dla przejścia S1 (A) S2 możliwe kontynuacje to wszystkie przejścia wychodzące ze stanu S2, czyli S2 (A) S2, S2 (B) S1 oraz S2 (C) S4. Wszystkie możliwe pary przejść wyglądają zatem tak:

- PP1: S1 (A) S2 (A) S2,
- PP2: S1 (A) S2 (B) S1,
- PP3: S1 (A) S2 (C) S4,
- PP4: S1 (C) S3 (B) S4,
- PP5: S2 (A) S2 (A) S2,

PP6: S2 (A) S2 (B) S1,
 PP7: S2 (A) S2 (C) S4,
 PP8: S2 (B) S1 (A) S2,
 PP9: S2 (B) S1 (C) S3.

Zauważmy (analogicznie jak w przypadku pokrycia przejść), że pary przejść PP3, PP4 i PP7 kończą się w stanie końcowym, a zatem żadne dwie z nich nie mogą wystąpić w jednym teście, gdyż wystąpienie którejkolwiek z tych par przejść skutkuje zakończeniem przypadku testowego. Zatem do pokrycia par przejść potrzebujemy co najmniej trzech testów i oczywiście trzy testy wystarczą. Przykładowy zestaw testów podany jest w tabeli 4.11 (pogrubioną czcionką w zasadniczej części tabeli zaznaczono pary pokryte po raz pierwszy). Potrzebujemy więc o jeden przypadek testowy więcej niż w przypadku pokrycia przejść poprawnych.

TABELA 4.11. Przypadki testowe pokrywające pary przejść między stanami

PRZYPADEK TESTOWY	POKRYTE PARY PRZEJŚĆ
S1 (A) S2 (A) S2 (A) S2 (B) S1 (A) S2 (B) S1 (C) S3 (B) S4	PP1, PP5, PP6, PP8, PP2, PP9, PP4
S1 (A) S2 (C) S4	PP3
S1 (A) S2 (A) S2 (C) S4	PP1, PP7

4.2.5. (*) Testowanie oparte na przypadkach użycia

Obszar zastosowań techniki

Przypadek użycia (ang. *use case*) jest dokumentem wymagań napisanym w taki sposób, by opisywał interakcję pomiędzy tzw. aktorami, którymi najczęściej są użytkownik oraz system.

Przypadek użycia jest opisem sekwencji kroków, które wykonują użytkownika i system, a które ostatecznie prowadzą do uzyskania przez użytkownika jakiejś korzyści. Każdy przypadek użycia powinien opisywać *jeden*, ściśle określony scenariusz. Na przykład jeśli dokumentujemy wymagania dla bankomatu, to przypadki użycia mogłyby m.in. opisywać:

- scenariusz odrzucenia niepoprawnej karty bankomatowej;
- scenariusz logowania się do systemu poprzez wprowadzenie poprawnego kodu PIN;
- scenariusz poprawnej wypłaty pieniędzy z bankomatu;
- scenariusz próby nieudanej wypłaty pieniędzy z powodu zbyt małej ilości środków na koncie;
- scenariusz zablokowania karty poprzez trzykrotne wpisanie błędnego kodu PIN.

Dla każdego przypadku użycia tester może skonstruować odpowiadający mu przypadek testowy, a także zestaw przypadków testowych sprawdzających wystąpienie nieoczekiwanych zdarzeń podczas przebiegu scenariusza.

Budowa przypadku użycia

Poprawnie zbudowany przypadek użycia, poza „technicznymi” informacjami takimi jak unikatowy numer i nazwa, powinien składać się z:

- warunków wstępnych (w tym danych wejściowych);
- dokładnie jednego, sekwencyjnego tzw. *scenariusza głównego*;
- (opcjonalnie) oznaczenia miejsc wystąpień tzw. *sytuacji wyjątkowych* (z których również *przepływaniami alternatywnymi*) oraz *obsługi błędów*;
- warunków wyjścia (opisujących stan systemu po udanym wykonaniu scenariusza oraz uzyskaną korzyść użytkownika).

Z punktu widzenia testera zasadniczym celem wykorzystania przypadku użycia jest przetestowanie scenariusza głównego, tzn. sprawdzenie, czy rzeczywiście wykonywanie wszystkich kroków tak, jak scenariusz ten opisuje, prowadzi do uzyskania zdefiniowanej korzyści dla użytkownika. Jednak przy okazji należy również przetestować zachowanie systemu na okoliczność wystąpienia sytuacji wyjątkowych (lub inaczej: alternatywnych) oraz obsługę błędów.

Różnica między tymi dwoma typami nieoczekiwanych zdarzeń jest następująca:

- Sytuacja wyjątkowa powoduje wystąpienie nieoczekiwanej zdarzenia, ale pozwala użytkownikowi dokończyć przypadek użycia, tzn. umożliwia powrót do scenariusza głównego i szczęśliwe jego ukończenie.
- Wystąpienie sytuacji niepoprawnej (obsługi błędu) powoduje przerwanie wykonywania scenariusza głównego — w przypadku wystąpienia sytuacji błędnej nie jest możliwe poprawne ukończenie scenariusza głównego. Użytkownik nie osiąga korzyści, a przypadek użycia bądź kończy się komunikatem o błędzie, bądź jest przerywany. Samo wystąpienie sytuacji błędnej może być obsłużone poprzez scenariusz zdefiniowany w odrębnym przypadku użycia.

Przypadek użycia nie powinien zawierać logiki biznesowej, tzn. scenariusz główny powinien być linearny i nie zawierać rozgałęzień. Istnienie takich rozgałęzień sugeruje, że tak naprawdę mamy do czynienia z więcej niż jednym przypadkiem użycia. Każda z takich możliwych ścieżek powinna zostać opisana w osobnym przypadku użycia.

Przykład. Rozważmy przykład scenariusza „Poprawna wypłata pieniędzy (100 PLN) z bankomatu z prowizją 5 PLN”. Przypadek użycia opisujący ten scenariusz mógłby wyglądać tak jak na rysunku 4.12.

Przypadek użycia: PU-003-02

Nazwa: Udana wypłata 100 PLN z bankomatu z prowizją 5 PLN.

Warunki wstępne: Użytkownik jest zalogowany, karta płatnicza rozpoznana jako karta z wypłatą obciążoną prowizją 5 PLN.

Kroki przypadku użycia:

1. Użytkownik wybiera opcję Wypłata pieniędzy.
2. System wyświetla menu z dostępnymi kwotami wypłaty.
3. Użytkownik wybiera opcję 100 PLN.
4. System sprawdza, czy na koncie użytkownika jest co najmniej 105 PLN.
5. System pyta o wydruk potwierdzenia.
6. Użytkownik wybiera opcję NIE.
7. System wyświetla komunikat Zabierz kartę.
8. System zwraca kartę.
9. Użytkownik odbiera kartę.
10. System wypłaca 100 PLN i przelewa 5 PLN prowizji na konto banku.
11. System wyświetla komunikat Zabierz pieniądze. Przypadek użycia kończy się, użytkownik jest wylogowany, system wraca do ekranu początkowego.

Warunki końcowe: Wypłacono 100 PLN użytkownikowi, stan konta użytkownika pomniejszony o 105 PLN, stan kasy bankomatu pomniejszony o 100 PLN.

Sytuacje wyjątkowe i błędy:

3A — użytkownik nie wybiera żadnej opcji przez 30 sekund (system zwraca kartę i wraca do ekranu początkowego).

4A — stan konta wynosi mniej niż 105 PLN (komunikat *Brak środków*, zwrocenie karty, wylogowanie użytkownika, powrót do ekranu początkowego i zakończenie przypadku użycia).

9A — użytkownik nie odbiera karty przez 30 sekund (system „wciąga” kartę, wysyła powiadomienie e-mailowe do klienta i wraca do ekranu początkowego).

9B — użytkownik po odbiorze karty wkłada ją z powrotem (system zwraca kartę).

RYSUNEK 4.12. Przypadek użycia poprawnej wypłaty z prowizją

Ten przypadek użycia składa się z jednego scenariusza głównego (kroki 1 – 11) oraz trzech sytuacji wystąpienia błędu (oznaczonych jako 3A, 4A i 9A) i jednej sytuacji wyjątkowej (9B). Numeracja tych zdarzeń odwołuje się do kroku, w którym mogą wystąpić. Jeśli w jednym kroku możliwe jest wystąpienie więcej niż jednego wyjątku, oznaczane są one kolejnymi literami alfabetu: A, B, C itd.

Wyprowadzanie przypadków testowych z przypadku użycia

Istnieje pewne minimum czynności, które tester powinien wykonać dla przetestowania przypadku użycia. Minimum to (rozumiane jako pełne, 100% pokrycia przypadku użycia) polega na zaprojektowaniu:

- jednego przypadku testowego do realizacji scenariusza głównego, bez wystąpienia żadnych niespodziewanych zdarzeń;
- po jednym przypadku testowym do realizacji każdej sytuacji wyjątkowej oraz błędu.

W naszym przykładzie potrzebowalibyśmy zatem pięciu przypadków testowych:

PT1: realizacja kroków 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,

PT2: realizacja kroków 1, 2, 3A,

PT3: realizacja kroków 1, 2, 3, 4A,

PT4: realizacja kroków 1, 2, 3, 4, 5, 6, 7, 8, 9A,

PT5: realizacja kroków 1, 2, 3, 4, 5, 6, 7, 8, 9B, 10, 11.

Każda sytuacja wyjątkowa oraz każda obsługa błędu powinny być testowane w osobnych przypadkach testowych, aby uniknąć tzw. zjawiska maskowania błędów. Zjawisko to omówiliśmy szczegółowo w punkcie 4.2.1.

Zauważmy, że PT5 (przepływ alternatywny) umożliwia osiągnięcie celu (dochodzimy do kroku 11.). PT2, PT3 i PT4 kończą się wcześniej z powodu wystąpienia sytuacji niepoprawnych podczas realizacji scenariusza. Przypadki testowe możemy rozpisać w formie scenariuszy, opisując przy każdym kroku oczekiwana odpowiedź systemu. Na przykład scenariusz dla przypadku testowego PT3 można opisać tak jak w tabeli 4.12.

Zwróćmy uwagę, że poszczególne kroki (akcje użytkownika i oczekiwane odpowiedzi) odpowiadają kolejnym krokom 1, 2, 3, 4A przypadku użycia. Zauważmy też, że przy okazji tester sprawdził w tym scenariuszu niepoprawną wartość brzegową dla wypłaty: cała operacja miała spowodować pomniejszenie stanu konta użytkownika o 105 PLN, a zadeklarowany stan środków na koncie wyniósł 104,99 PLN. Jest to typowy przykład łączenia technik testowania, który pozwala w małej liczbie przypadków testowych zatrzymać weryfikację wielu różnego rodzaju warunków. Tutaj, przy okazji sprawdzania poprawności działania systemu w sytuacji braku środków na koncie, wykorzystaliśmy technikę analizy wartości brzegowych, aby niewystarczająca kwota była tylko o 1 grosz mniejsza od kwoty, która pozwoliłaby już na poprawną wypłatę pieniędzy.

Pokrycie

Standard ISO/IEC 29119 nie definiuje miary pokrycia dla testowania opartego na przypadkach użycia (ang. *use case testing*). Sylabus (w starej wersji) jednak podaje taką miarę, zakładając, że elementami pokrycia są ścieżki przepływu w przypadku użycia. Pokrycie można zatem zdefiniować jako stosunek liczby zweryfikowanych ścieżek przepływu w przypadku użycia w stosunku do wszystkich możliwych ścieżek przepływu opisanych w przypadku użycia.

TABELA 4.12. Przypadek testowy zbudowany na podstawie przypadku użycia

Przypadek testowy PT3: Wypłata z bankomatu z niewystarczającą ilością środków na koncie (dot. przypadku użycia PU-003-02, wystąpienie błędu 4A).

Warunki wstępne:

- użytkownik jest zalogowany;
- system jest w menu głównym;
- wypłata rozpoznana jako obciążona prowizją 5 PLN (ze względu na typ karty);
- stan środków na koncie: 104,99 PLN.

KROK	ZDARZENIE	OCZEKIWANY REZULTAT
1	Wybór opcji <i>Wypłata pieniędzy</i>	System przechodzi do menu wyboru kwot.
2	Wybór opcji <i>100 PLN</i>	System stwierdza brak środków na koncie, wyświetla komunikat <i>Brak środków</i> , zwraca kartę.

Warunki końcowe:

- stan konta użytkownika nie zmienił się;
- bankomat nie wypłacił pieniędzy;
- bankomat zwrócił użytkownikowi kartę;
- system wrócił do ekranu powitalnego.

Na przykład w przypadku użycia wypłaty 100 PLN opisanego powyżej jest pięć różnych ścieżek przepływu: ścieżka główna, trzy przepływy z obsługą sytuacji nieprawnych oraz jeden przepływ alternatywny. Gdyby zestaw testowy zawierał wyłącznie przypadki testowe PT1 i PT5, pokrycie przypadku użycia tymi testami wyniosłoby $2/5 = 40\%$.

4.3. Białoskrzynkowe techniki testowania

FL-4.3.1 (K2) Kandydat wyjaśnia pojęcie testowanie instrukcji.

FL-4.3.2 (K2) Kandydat wyjaśnia pojęcie testowanie gałęzi.

FL-4.3.3 (K2) Kandydat wyjaśnia korzyści wynikające z testowania białoskrzynkowego.

Testowanie białoskrzynkowe opiera się na *strukturze wewnętrznej* przedmiotu testów. Najczęściej testowanie białoskrzynkowe kojarzone jest z testami modułowymi, w których modelem obiektu testów jest struktura wewnętrzna kodu, przedstawiona np. w postaci tzw. grafu przepływu sterowania. Należy jednak pamiętać, że techniki testowania białoskrzynkowego mogą być stosowane na wszystkich poziomach testów, np.:

- na poziomie testów modułowych (przykład struktury: graf przepływu sterowania);
- na poziomie testów integracyjnych (przykład struktury: graf wywołań, zbiór funkcji API);
- na poziomie testów systemowych (przykład struktury: proces biznesowy zamodelowany w języku BPMN², menu programu);
- na poziomie testów akceptacyjnych (przykład struktury: struktura strony internetowej).

Syabus poziomu podstawowego omawia dwie techniki białośkrzynkowe: testowanie instrukcji oraz testowanie gałęzi. Obie te techniki ze swojej natury skojarzone są z kodem, a więc obszarem ich zastosowania są głównie testy modułowe. Poza nimi istnieje wiele innych (i zwykle silniejszych) technik białośkrzynkowych, takich jak:

- testowanie MC/DC (tzw. zmodyfikowane pokrycie warunkowo-decyzyjne);
- testowanie warunków wielokrotnych;
- testowanie pętli;
- testowanie ścieżek liniowo niezależnych.

Techniki te nie są jednak omawiane w syabusie poziomu podstawowego. Niektóre z nich omówione są w syabusie „ISTQB Techniczny Analityk Testów” na poziomie zaawansowanym [ISTQB TTA 2021]. Tych mocniejszych technik białośkrzynkowych używa się często podczas testowania systemów krytycznych ze względu na bezpieczeństwo (np. oprogramowanie aparatury medycznej lub oprogramowanie lotnicze).

4.3.1. Testowanie instrukcji i pokrycie instrukcji kodu

Testowanie instrukcji jest najprostszą i zarazem najsłabszą ze wszystkich technik białośkrzynkowych. Polega ono na pokrywaniu *instrukcji wykonywalnych* w kodzie źródłowym programu.

Przykład. Rozważmy prosty przykład kodu:

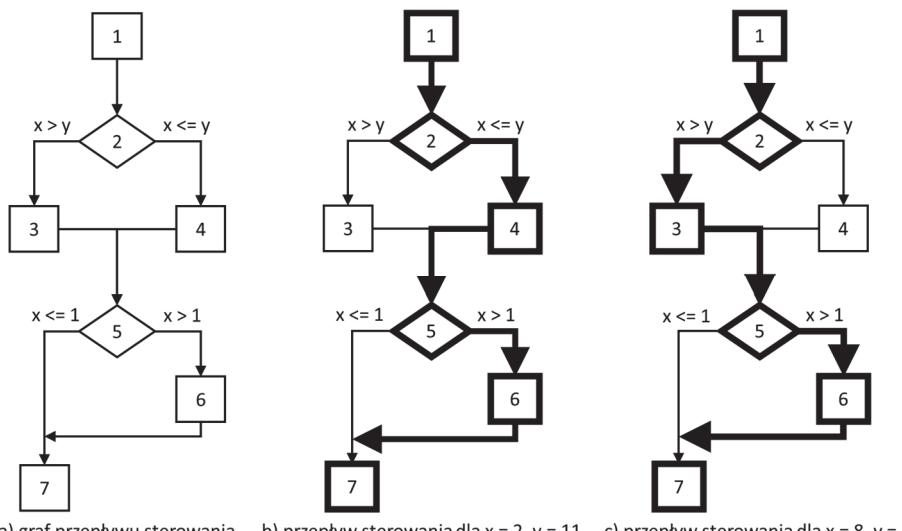
1. **WEJŚCIE** x, y
// pobranie z wejścia dwóch liczb naturalnych
2. **JEŻELI** ($x > y$) **TO**
3. $z := x - y$
W PRZECIWNYM RAZIE
4. $z := y - x$
5. **JEŻELI** ($x > 1$) **TO**
6. $z := z * 2$
7. **ZWRÓĆ** z

² Business Process Model and Notation, BPMN (pol. *Notacja i Model Procesu Biznesowego*) — graficzna notacja służąca do opisywania procesów biznesowych.

Instrukcje wykonywalne zostały w tym kodzie oznaczone cyframi 1 – 7. Linijka rozpoczynająca się od znaków // stanowi komentarz i nie jest wykonywana podczas uruchomienia programu. Słowa kluczowe **W PRZECIWNYM RAZIE** również stanowią jedynie część składni instrukcji **JEŻELI-TO** i same w sobie nie są traktowane jak instrukcja wykonywalna (bo nie ma się co tu wykonać).

Po pobraniu z wejścia dwóch zmiennych x , y w linii 1. program sprawdza w linii 2., czy x jest większe od y . Jeśli tak, wykonywana jest instrukcja 3. (podstawienie do zmiennej z różnicy $x - y$), po której następuje skok do linii 5. (za instrukcję **JEŻELI-TO** zawartą w liniach 2. – 4.). Jeśli jednak x nie jest większe od y , program przekazuje sterowanie z linii 2. do linii 4. (sekcja **W PRZECIWNYM RAZIE** instrukcji **JEŻELI-TO**), w której do zmiennej z podstawiona jest różnica $y - x$, a po wykonaniu tej instrukcji następuje skok do linii 5. W linii 5. sprawdzane jest, czy zmienna x ma wartość większą od 1. Jeśli tak, wykonywane jest ciało instrukcji **JEŻELI-TO** w linii 6. (podwojenie wartości z) i następuje skok za instrukcję **JEŻELI-TO** do linii 7. Jeśli jednak wartość decyzji w linii 5. jest fałszywa, linia 6. zostanie pominięta i sterowanie przejdzie od razu za instrukcję **JEŻELI-TO**, czyli do linii 7., w której zwrotny zostanie wynik — wartość zmiennej z , a program zakończy swoje działanie.

Kod źródłowy można reprezentować jako tzw. graf przepływu sterowania. Graf taki dla wyżej wspomnianego kodu przedstawiony jest na rysunku 4.13 a). Jest to graf skierowany, którego wierzchołki reprezentują instrukcje (decyzje oznaczane są tu rombami, pozostałe instrukcje — kwadratami), a strzałki — możliwy przepływ sterowania między instrukcjami.



a) graf przepływu sterowania b) przepływ sterowania dla $x = 2$, $y = 11$ c) przepływ sterowania dla $x = 8$, $y = 1$

RYSUNEK 4.13. Graf przepływu sterowania oraz przykłady przepływu sterowania

Rozważmy dwa przypadki testowe:

- PT1: wejście: $x = 2$, $y = 11$; oczekiwane wyjście: 18;
- PT2: wejście: $x = 8$, $y = 1$; oczekiwane wyjście: 14.

Na rysunku 4.13 b) pogrubione fragmenty pokazują przepływ sterowania w sytuacji, gdy na wejściu podane zostaną wartości $x = 2$, $y = 11$. Program przejdzie wtedy po instrukcjach:

(sterowanie dla PT1): $1 \rightarrow 2 \text{ (x} \leq \text{ y)} \rightarrow 4 \rightarrow 5 \text{ (x} > \text{ 1)} \rightarrow 6 \rightarrow 7$.

Z kolei dla wejścia $x = 8$, $y = 1$ program przejdzie po instrukcjach (rysunek 4.13 c):

(sterowanie dla PT2): $1 \rightarrow 2 \text{ (x} > \text{ y)} \rightarrow 3 \rightarrow 5 \text{ (x} > \text{ 1)} \rightarrow 6 \rightarrow 7$.

Podkreślimy jedną, bardzo ważną rzeczą w kontekście technik białośkrzynkowych: oczekiwane rezultaty w przypadkach testowych *nie* są wyprowadzane na podstawie kodu (bo kod jest właśnie tym, co testujemy — nie może więc być swoją własną wyrocznią), lecz na podstawie zewnętrznej wobec kodu specyfikacji. W naszym przypadku mogłyby ona brzmieć tak: „Program pobiera dwie liczby naturalne, a następnie od większej odejmuje mniejszą. Jeśli pierwsza liczba jest większa niż 1, wartość ta jest dodatkowo podwajana. Program zwraca tak obliczoną wartość”.

Pokrycie

W technice testowania instrukcji elementami pokrycia są *instrukcje wykonywalne*. Zatem **pokrycie instrukcji kodu** (ang. *statement coverage*) to iloraz instrukcji wykonywalnych pokrytych przez testy przez liczbę wszystkich instrukcji wykonywalnych w analizowanym kodzie, na ogół wyrażany procentowo. Zauważmy, że metryka ta nie uwzględnia instrukcji, które nie są wykonywalne (np. komentarzy, nagłówków funkcji). Rozważmy ponownie powyższy kod źródłowy i przypadki testowe PT1 i PT2. Jeśli nasz zbiór testów zawiera tylko przypadek PT1, to jego wykonanie zapewni nam pokrycie $6/7 = \text{ok. } 86\%$, ponieważ przypadek ten wymusza przejście przez 6 różnych instrukcji wykonywalnych spośród 7, z których składa się nasz kod. PT2 przechodzi również przez 6 z 7 instrukcji, zatem daje takie samo pokrycie ok. 86%. Jeśli jednak nasz zbiór testowy zawiera oba przypadki, PT1 i PT2, pokrycie wyniesie 100%, ponieważ *łącznie* te dwa przypadki testowe wymuszają przejście po wszystkich 7 instrukcjach.



4.3.2. Testowanie gałęzi i pokrycie gałęzi

Gałąź (ang. *branch*) to przepływ sterowania pomiędzy dwoma wierzchołkami grafu przepływu sterowania. Gałąź może być bezwarunkowa lub warunkowa. Gałąź bezwarunkowa między wierzchołkami A i B oznacza, że po zakończeniu wykonywania instrukcji A sterowanie *musi* przejść do instrukcji B. Gałąź warunkowa między A i B oznacza, że po zakończeniu wykonywania instrukcji A sterowanie *może* przejść do instrukcji B, ale niekoniecznie. Gałęzie warunkowe wychodzą z wierzchołków decyzyjnych, czyli z miejsc w kodzie, w których podejmowana jest jakaś decyzja, od której zależy dalszy przebieg sterowania. Przykładami instrukcji decyzyjnej są instrukcje warunkowe IF-THEN, IF-THEN-ELSE oraz SWITCH-CASE, a także instrukcje sprawdzające tzw. warunek pętli w pętlach WHILE, DO-WHILE czy FOR.

Na przykład na rysunku 4.13 a) gałęziami bezwarunkowymi są gałęzie $1 \rightarrow 2$, $3 \rightarrow 5$, $4 \rightarrow 5$ oraz $6 \rightarrow 7$, ponieważ np. jeśli wykonana jest instrukcja 3., następną instrukcją musi być instrukcja 5. Pozostałe gałęzie: $2 \rightarrow 3$, $2 \rightarrow 4$, $5 \rightarrow 6$ oraz $5 \rightarrow 7$ są warunkowe. Na przykład po wykonaniu instrukcji 2. sterowanie może przejść do instrukcji 3. lub do instrukcji 4. To, który z tych przypadków zajdzie, zależy będzie od wartości logicznej decyzji IF w instrukcji 2. Jeżeli $x > y$, to sterowanie przejdzie do 3., jeśli $x \leq y$, to do 4.

W testowaniu gałęzi elementami pokrycia są wszystkie gałęzie, zarówno warunkowe jak i bezwarunkowe. Celem testowania gałęzi jest zaprojektowanie wystarczającej liczby przypadków testowych, które osiągną wymagany (akceptowany) poziom pokrycia gałęzi.

Pokrycie

W technice testowania gałęzi elementami pokrycia są *gałęzie*, zarówno warunkowe jak i bezwarunkowe. **Pokrycie gałęzi** (ang. *branch coverage*) oblicza się zatem jako iloraz liczby gałęzi, które zostały wykonane w trakcie testów, przez całkowitą liczbę gałęzi w kodzie. Tak jak w innych przypadkach, pokrycie to często wyraża się w procentach. Gdy osiągnięte jest pełne, stuprocentowe pokrycie gałęzi, oznacza to, że każda gałąź w kodzie — zarówno warunkowa, jak i bezwarunkowa — została wykonana podczas testów przynajmniej raz. Oznacza to, że przetestowaliśmy wszystkie możliwe bezpośrednie przejścia pomiędzy instrukcjami w kodzie.



Przykład. Rozważmy ponownie kod z przykładu w punkcie 4.3.1:

1. **WEJŚCIE** x , y
// pobranie z wejścia dwóch liczb naturalnych
2. **JEŻELI** ($x > y$) **TO**
3. $z := x - y$
W PRZECIWNYM RAZIE
4. $z := y - x$
5. **JEŻELI** ($x > 1$) **TO**
6. $z := z * 2$
7. **ZWRÓĆ** z

W kodzie tym mamy osiem gałęzi:

$1 \rightarrow 2$, $2 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 5$,

$4 \rightarrow 5$, $5 \rightarrow 6$, $5 \rightarrow 7$, $6 \rightarrow 7$.

Przypadek testowy PT1 ($x = 2$, $y = 11$, oczekiwane wyjście: 18) pokrywa następujące gałęzie:

- $1 \rightarrow 2$ (bezwarunkowa),
- $2 \rightarrow 4$ (warunkowa),
- $4 \rightarrow 5$ (bezwarunkowa),

- $5 \rightarrow 6$ (warunkowa),
- $6 \rightarrow 7$ (bezwarunkowa)

i zapewnia nam pokrycie $5/8 = 67,5\%$.

Przypadek testowy PT2 ($x = 8, y = 1$, oczekiwane wyjście: 14) pokrywa następujące gałęzie:

- $1 \rightarrow 2$ (bezwarunkowa),
- $2 \rightarrow 3$ (warunkowa),
- $3 \rightarrow 5$ (bezwarunkowa),
- $5 \rightarrow 6$ (warunkowa),
- $6 \rightarrow 7$ (bezwarunkowa)

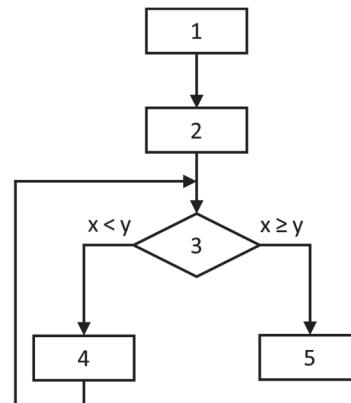
i zapewnia nam pokrycie $5/8 = 67,5\%$.

Oba przypadki razem zapewniają nam pokrycie $7/8 = 87,5\%$, bo pokryte są wszystkie gałęzie poza gałęzią $5 \rightarrow 7$.

Przykład. Rozważmy prosty kod i jego graf przepływu sterowania z rysunku 4.14.

```

1. WEJŚCIE y
2. x=1;
3. WHILE (x < y) DO
4.   x=x+1;
5. DRUKUJ ("Koniec pętli")
END
  
```



RYSUNEK 4.14. Kod źródłowy z pętlą while i jego graf przepływu sterowania

Pojedynczy przypadek testowy z wejściem $y = 3$ zapewnia 100% pokrycia instrukcji. Sterowanie będzie bowiem przechodzić po następujących instrukcjach (w nawiasach podano wartość zmiennej x lub warunek logiczny sprawdzany w linii 3.):

$1 \rightarrow 2 (x := 1) \rightarrow 3 (1 < 3) \rightarrow 4 (x := 2) \rightarrow 3 (2 < 3) \rightarrow 4 (x := 3) \rightarrow 3 (3 < 3) \rightarrow 5$.

Ten sam przypadek testowy zapewnia też 100% pokrycia gałęzi. W kodzie istnieje pięć gałęzi: $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 4$, $3 \rightarrow 5$, $4 \rightarrow 3$. Zgodnie z przepływem sterowania kolejno pokrywane będą gałęzie:

- 1 → 2 (bezwarunkowa),
- 2 → 3 (bezwarunkowa),
- 3 → 4 (warunkowa),

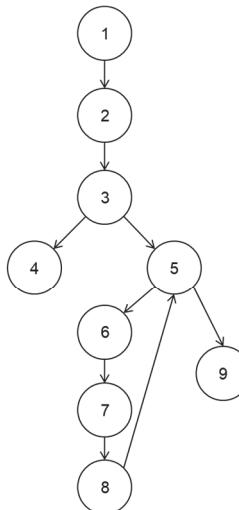
- 4 → 3 (bezwarunkowa),
- 3 → 4 (warunkowa, pokryta wcześniej),
- 4 → 3 (bezwarunkowa, pokryta wcześniej),
- 3 → 5 (warunkowa).

Przykład. Rozważmy kod z rysunku 4.15 a).

```

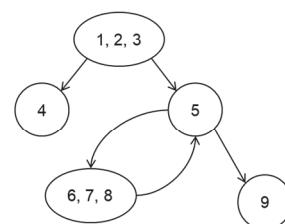
1 WEJŚCIE a, b, c
2 d = a + b + c
3 IF (d>0) THEN
4   RETURN d
ELSE
5   WHILE (d<0) DO
6     a = a + 1
7     b = b - 1
8     d = d + 1
END WHILE
9 RETURN a * b * c
END IF
END

```



a) kod źródłowy

b) graf przepływu sterowania



c) graf przepływu sterowania
z blokami podstawowymi

RYSUNEK 4.15. Dwa grafy przepływu sterowania dla tego samego kodu

Jego graf przepływu sterowania, w którym każdy wierzchołek odpowiada pojedynczej instrukcji, pokazany jest na rysunku 4.15 b). W grafie tym mamy dziewięć gałęzi:

- 1→2 (bezwarunkowa),
- 2→3 (bezwarunkowa),
- 3→4 (warunkowa),
- 3→5 (warunkowa),
- 5→6 (warunkowa),
- 5→9 (warunkowa),
- 6→7 (bezwarunkowa),
- 7→8 (bezwarunkowa),
- 8→5 (bezwarunkowa).

Aby osiągnąć 100% pokrycia gałęzi, potrzebujemy co najmniej dwóch przypadków testowych, na przykład:

PT1: 1→2→3→4,

PT2: 1→2→3→5→6→7→8→5→9.

PT1 pokrywa trzy z dziewięciu gałęzi, osiąga więc $3/9 \approx 33\%$ pokrycia gałęzi. PT2 pokrywa osiem z dziewięciu gałęzi, osiąga więc $8/9 \approx 89\%$ pokrycia gałęzi. Oba testy razem wzięte pokrywają wszystkie dziewięć gałęzi, więc zbiór testów {PT1, PT2} osiąga pełne pokrycie gałęzi, $9/9 = 100\%$.

Czasami graf przepływu sterowania rysuje się tak, aby w jednym wierzchołku ująć kilka instrukcji stanowiących tak zwany blok podstawowy. Blok podstawowy to ciąg instrukcji o takiej własności, że gdy wykona się którakolwiek z nich, muszą się wykonać wszystkie pozostałe. Tak zmodyfikowany graf przepływu sterowania jest mniejszy i czytelniejszy (nie ma w nim „długich ścieżek”), ale jego użycie wpływa na wartość pokrycia, ponieważ w takim grafie będzie mniej krawędzi niż w grafie, w którym każdy wierzchołek reprezentuje pojedynczą instrukcję. Na przykład graf z rysunku 4.15 c) jest równoważny grafowi z rysunku 4.15 b), ale posiada tylko pięć krawędzi. Wspomniany wcześniej przypadek testowy PT1 osiąga w tym przypadku pokrycie $1/5 = 20\%$, a test PT2 — $4/5 = 80\%$.

Jeśli pokrycie mierzy się bezpośrednio na kodzie, to jest ono równoważne pokryciu obliczanemu dla grafu przepływu sterowania, w którym każdy wierzchołek reprezentuje pojedynczą instrukcję, ponieważ gałęzie reprezentują wtedy przepływ sterowania pomiędzy poszczególnymi instrukcjami, a nie ich grupami (blokami podstawowymi).

4.3.3. Korzyści wynikające z testowania białośkrzynkowego

Testowanie instrukcji opiera się na następującej tzw. hipotezie błędu: jeśli w programie jest jakiś defekt, to musi znajdować się w którejś z instrukcji wykonywalnych. A jeśli uda nam się osiągnąć 100% pokrycia instrukcji, to będziemy mieć pewność, że instrukcja z defektem zostanie wykonana. To oczywiście nie zagwarantuje wystąpienia awarii, ale przynajmniej stworzy taką możliwość — rozważmy prosty przykład kodu, który przyjmuje na wejściu dwie liczby i zwraca ich iloczyn (jeśli pierwsza z nich jest dodatnia) lub pierwszą liczbę (jeśli pierwsza liczba nie jest dodatnia). Kod programu wygląda tak:

1. **WEJŚCIE** x, y
2. **JEŻELI** $x > 0$ **TÓMĘ**
3. $x := x + y$
4. **ZWRÓĆ** x

W programie tym jest defekt polegający na błędym użyciu operatora dodawania zamiast mnożenia w linii 3. Dla danych wejściowych $x = 2, y = 2$ program przejdzie po wszystkich instrukcjach, osiągnięte zatem zostanie pełne, stuprocentowe pokrycie instrukcji. W szczególności wykona się błędna instrukcja 3., ale akurat $2 \cdot 2$ jest tym samym co $2 + 2$, zatem zwrócony wynik będzie poprawny pomimo wystąpienia defektu. Ten prosty przykład pokazuje, że pokrycie instrukcji nie jest techniką silną i warto wykorzystywać inne, silniejsze, takie jak testowanie gałęzi.

W testowaniu gałęzi z kolei hipoteza błędu jest następująca: jeśli w programie jest jakiś defekt, to powoduje on błędny przepływ sterowania. W sytuacji, w której osiągnęliśmy stuprocentowe pokrycie gałęzi, musielibyśmy wymusić przynajmniej

raz przejście, które było błędne (np. wartość decyzji powinna być prawdą, a jest fałszem), dzięki czemu program wykona nie te instrukcje, co trzeba, a to skutkować może awarią.

Oczywiście tak jak w przypadku testowania instrukcji pełne pokrycie gałęzi nie gwarantuje, że awaria nastąpi, nawet jeśli program przejdzie błędną ścieżką. Podobnie jak poprzednio, mimo pokrycia gałęzi $2 \rightarrow 3$ defekt nie zostanie wykryty.

Istnieje natomiast ważna zależność między testowaniem instrukcji a testowaniem gałęzi:

Osiągnięcie stuprocentowego pokrycia gałęzi gwarantuje osiągnięcie stuprocentowego pokrycia instrukcji.

Fachowo mówi się, że pokrycie gałęzi subsumuje pokrycie instrukcji. Oznacza to, że dowolny zestaw testów, który osiąga 100% pokrycia gałęzi, z definicji osiąga też 100% pokrycia instrukcji. Nie musimy zatem w takim przypadku osobno sprawdzać pokrycia instrukcji — wiemy, że jest ono spełnione w stu procentach. W drugą stronę relacja subsumpcji nie zachodzi — osiągnięcie 100% pokrycia instrukcji nie gwarantuje osiągnięcia 100% pokrycia gałęzi.

Aby to pokazać, rozważmy podany powyżej fragment kodu. Program ten posiada cztery instrukcje oraz cztery gałęzie: $1 \rightarrow 2$, $2 \rightarrow 3$, $2 \rightarrow 4$ i $3 \rightarrow 4$. Dla danej wejściowej $x = 3$ program przejdzie po wszystkich czterech instrukcjach: 1, 2, 3, 4, a zatem ten przypadek testowy daje pełne, stuprocentowe pokrycie instrukcji. Jednak nie pokryliśmy wszystkich gałęzi — w tym teście po wykonaniu linii 2. program przechodzi do linii 3. (bo $3 > 0$), więc nie pokrywa gałęzi $2 \rightarrow 4$ (która byłaby wykonana w przypadku, w którym decyzja w linii 2. byłaby fałszem). Test dla $x = 3$ osiąga więc 100% pokrycia instrukcji, ale tylko 75% pokrycia gałęzi.

Należy również pamiętać, że każda *instrukcja decyzyjna jest instrukcją* i jako taka wchodzi w skład zbioru instrukcji wykonywalnych. Na przykład w powyższym kodzie linia 2. stanowi instrukcję (i tak jest traktowana, gdy stosujemy technikę pokrycia instrukcji). Zatem możemy powiedzieć, że stuprocentowe pokrycie instrukcji gwarantuje wykonanie każdej decyzji w kodzie, ale nie możemy stwierdzić, że gwarantuje osiągnięcie *każdego wyniku* każdej decyzji (a więc nie możemy powiedzieć, że gwarantuje pokrycie wszystkich gałęzi w kodzie).

Zasadniczą zaletą wszystkich technik białośkrzynkowych jest to, że podczas testowania brana jest pod uwagę cała implementacja oprogramowania, co ułatwia wykrywanie defektów nawet wtedy, gdy specyfikacja oprogramowania jest niejasna lub niekompletna. Projektowanie testów jest niezależne od specyfikacji. Słabością jest to, że jeśli oprogramowanie nie implementuje jednego lub więcej wymagań, testowanie białośkrzynkowe może nie wykryć wynikających z tego defektów polegających na braku funkcjonalności [Watson 1996].

Techniki białośkrzynkowe mogą być stosowane w testowaniu statycznym. Dobrze nadają się do przeglądów zarówno kodu, który nie jest jeszcze gotowy do wykonania [Hetzler 1988], jak również pseudokodu i innej logiki wysokiego poziomu lub takiej, która może być modelowana za pomocą grafu przepływu sterowania.

Jeśli przeprowadza się wyłącznie testy czarnoskrzynkowe, to nie ma możliwości pomiaru rzeczywistego pokrycia kodu. Pomiary pokrycia białośkrzynkowego zapewniają obiektywną miarę pokrycia i dostarczają niezbędnych informacji, aby umożliwić generowanie dodatkowych testów w celu zwiększenia tego pokrycia, a następnie zwiększenia zaufania do kodu.

4.4. Techniki testowania oparte na doświadczeniu

- | | |
|---------------|---|
| FL-4.4.1 (K2) | Kandydat wyjaśnia pojęcie zgadywanie błędów. |
| FL-4.4.2 (K2) | Kandydat wyjaśnia pojęcie testowanie eksploracyjne. |
| FL-4.4.3 (K2) | Kandydat wyjaśnia pojęcie testowanie w oparciu o listę kontrolną. |

Poza technikami opartymi na specyfikacji i białośkrzynkowymi wyróżniamy trzecią rodzinę technik: techniki testowania oparte na doświadczeniu Kategoria ta zawiera techniki, które są mniej formalne od tych omówionych wcześniej, gdzie podstawą działań testera był zawsze jakiś model formalny (model dziedziny, logiki, zachowania, struktury itp.).

Techniki testowania oparte na doświadczeniu wykorzystują przede wszystkim wiedzę, umiejętności, intuicję testerów, a także ich doświadczenie w pracy z podobnymi produktami lub nawet tym samym produktem wcześniej. Dzięki takiemu podejściu testerowi łatwiej jest zidentyfikować awarie bądź defekty, które byłyby trudne do wykrycia przy użyciu bardziej usystematyzowanych technik. Wbrew pozorom techniki testowania oparte na doświadczeniu są jednymi z najbardziej rozpowszechnionych podejść do testowania. Należy jednak pamiętać o tym, że skuteczność tych technik, z samej ich natury, będzie zależeć w dużym stopniu od podejścia i doświadczenia poszczególnych testerów. Sylabus poziomu podstawowego wymienia następujące trzy typy technik testowania opartych na doświadczeniu, które omówimy w kolejnych punktach:

- zgadywanie błędów;
- testowanie eksploracyjne;
- testowanie w oparciu o listę kontrolną.

4.4.1. Zgadywanie błędów

Opis techniki

Zgadywanie błędów (ang. *error guessing*) jest chyba najprostszą koncepcyjnie spośród wszystkich technik testowania opartych na doświadczeniu. Nie jest związana z żadnym uprzednim planowaniem, nie jest również konieczne zarządzanie czynnościami testera związanymi z wykorzystywaniem tej techniki. Tester po prostu przewiduje rodzaje defektów bądź awarii w module lub systemie, odwołując się m.in. do:



- tego, jak do tej pory działał testowany moduł lub system (lub jego poprzednia już działającą produkcyjnie wersja);
- typowych znanych mu błędów popełnianych przez programistów, architektów, analityków i innych członków zespołu twórczego;
- wiedzy o awariach, które wystąpiły wcześniej w podobnych aplikacjach testowanych przez testera lub takich, o których tester ten słyszał.

Błędy, defekty i awarie w ogólności mogą być związane z:

- wejściem (np.: nieakceptowane poprawne wejście, akceptowane niepoprawne wejście, zła wartość parametru, brakujący parametr wejścia);
- wyjściem (np.: zły format wyjścia, niepoprawny wynik, poprawny wynik w złym czasie, wynik niepełny/niekompletny, brakujący wynik, błędy gramatyczne lub interpunkcyjne);
- logiką (np.: brakujące przypadki do rozważenia, zduplikowane przypadki do rozważenia, niepoprawny operator logiczny, brakujący warunek, niepoprawna iteracja pętli);
- obliczeniami (np.: błędny algorytm, brakujące obliczenie, niepoprawny operand, niepoprawny operator, błąd nawiasowania, niewystarczająca precyzja wyniku, niepoprawna wbudowana funkcja);
- interfejsem (np.: niepoprawne przetwarzanie zdarzeń z interfejsu, błędy związane z czasem przy przetwarzaniu wejścia/wyjścia, wywołanie złej lub nieistniejącej funkcji, brak parametru, niekompatybilne typy zmennych);
- danymi (np.: niepoprawna inicjalizacja, definicja lub deklaracja zmiennej, niepoprawny dostęp do zmiennej, zła wartość zmiennej, użycie niepoprawnej zmiennej, niepoprawna referencja do danej, niepoprawne skalowanie lub jednostka danej, zły wymiar danych, niepoprawny indeks, błędny typ zmiennej, niepoprawny zakres zmiennej, „błąd o jeden” (patrz punkt 4.2.2)).

Istnieje bardziej zorganizowana, metodyczna odmiana tej techniki. Podejście to polega na stworzeniu listy potencjalnych pomyłek, defektów i awarii. Tester analizuje listę punkt po punkcie i każdą pomyłkę, defekt bądź awarię stara się odpowiednio uwidocznić w testowanym obiekcie.

Podejście to nazywa się czasami *atakiem usterkowym* bądź *atakiem na oprogramowanie*. Wiele przykładów takich ataków opisanych jest w książkach Whittakera [Whittaker 2002, Whittaker 2003, Andrews 2006]. Technika ta różni się od pozostałych tym, że punktem wyjścia jest w niej negatywne zdarzenie: konkretna usterka czy awaria, a nie pozytywna rzecz do zweryfikowania (np. dziedzina wejścia czy logika biznesowa systemu).

Listy pomyłek, defektów i awarii można tworzyć na podstawie własnego doświadczenia — wtedy będą najefektywniejsze. Można również wykorzystywać różnego typu listy dostępne publicznie np. w internecie.

Przykład. Tester testuje program do obliczenia wskaźnika BMI (indeks masy ciała), który na wejściu pobiera od użytkownika dwie wartości: wagę oraz wzrost, a następnie wylicza wskaźnik BMI. Interfejs aplikacji pokazany jest na rysunku 4.16.

The screenshot shows a mobile application titled "BMI CALCULATOR". At the top, there is a radio button group for "Imperial" and "Metric", with "Metric" selected. Below this are two input fields: "Height" with the value "180" and "cm" unit, and "Weight" with the value "80" and "kg" unit. A green box displays the result: "Your BMI is 24.69 (normal BMI)". At the bottom is a blue "Reset" button and a footer note: "Powered by CalculatorsWorld.com".

RYSUNEK 4.16. Kalkulator BMI (źródło: calculatorsworld.com)

Tester wykorzystuje atak usterkowy z użyciem następującej listy defektów i awarii:

1. Wystąpienie przepełnienia dla pola formularza.
2. Wystąpienie dzielenia przez zero.
3. Wymuszenie wystąpienia nieprawidłowej wartości.

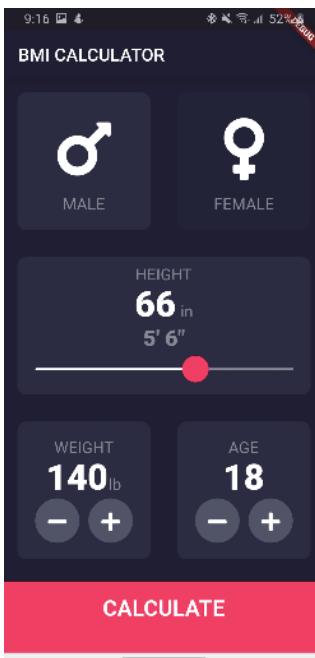
W celu próby wymuszenia awarii z punktu 1. listy tester może spróbować wpisać w pole wejściowe „waga” bardzo długi łańcuch cyfr³.

By wymusić awarię z punktu 2., tester może podać zerowy wzrost, ponieważ BMI obliczane jest jako iloraz masy i kwadratu wzrostu.

W celu próby wymuszenia awarii z punktu 3. tester może podać np. ujemną wartość wagi, aby wymusić ujemną (niepoprawną) wartość BMI.

Zauważmy, że ataki te można przeprowadzić dla powyższej aplikacji, ponieważ jej interfejs umożliwia wpisanie wartości wagi i wzrostu bezpośrednio z klawiatury. Warto definiować interfejsy tak, aby uniemożliwiły wprowadzanie niepoprawnych wartości, na przykład używać pól z wbudowaną kontrolą zakresu wartości, umożliwiać wpisywanie wartości tylko poprzez przyciski pozwalające na zwiększanie lub zmniejszanie wartości lub wykorzystywać inne mechanizmy, takie jak listy rozwijalne czy suwaki. Na rysunku 4.17 przedstawiono taki właśnie interfejs analogicznej aplikacji do obliczania wskaźnika BMI.

³ Wzór na BMI dzieli wagę przez kwadrat wzrostu. Aby uzyskać przepełnienie wartości BMI, licznik ułamka powinien być jak największy. Dlatego nie ma sensu podawać dużych wartości wzrostu, ponieważ im większy mianownik, tym mniejsze BMI.



RYSUNEK 4.17. Kalkulator BMI z innym interfejsem
(źródło: <https://github.com/dseng905/bmi-calculator-mobile>)

4.4.2. Testowanie eksploracyjne

Opis techniki

Testowanie eksploracyjne (ang. *exploratory testing*) to podejście polegające na projektowaniu, wykonywaniu i rejestrowaniu testów nieformalnych (tj. testów, które nie zostały z góry zaprojektowane) oraz ich ocenie w sposób dynamiczny podczas wykonywania. Oznacza to, że tester nie wykonuje testów zaplanowanych z góry, w osobnej fazie planowania, lecz każdy jego kolejny krok w bieżącym scenariuszu testowym jest dynamiczny i zależy od:



- wiedzy i intuicji testera;
- doświadczenia z tą lub podobnymi aplikacjami;
- tego, jak system zachował się w poprzednim kroku scenariusza.

Tak naprawdę ścisły podział na testy eksploracyjne i planowane (ang. *scripted*) nie oddaje do końca prawdy. Jest tak dlatego, że każda czynność testowa zawiera w sobie jakiś element planowania i jakiś element eksploracji. Na przykład nawet w czystych testach eksploracyjnych zazwyczaj planuje się wcześniej sesję eksploracyjną, przydzieliając np. testerowi określony czas na jej wykonanie. Tester eksploracyjny może też przed rozpoczęciem sesji przygotować sobie np. różne dane testowe, które będzie wykorzystywał w trakcie samej sesji.

Testowanie eksploracyjne w sesjach

Właśnie na potrzeby większego ustrukturyzowania aktywności testera oraz możliwości lepszego zarządzania testami eksploracyjnymi wprowadza się pojęcie tzw. *testowania w sesjach* (ang. *session based exploratory testing*). Składa się ono z trzech zasadniczych kroków:

1. Spotkanie testera z kierownikiem, ustalenie zakresu testów, przydział czasu, przekazanie karty opisu testu (ang. *test charter*). Karta opisu testu powinna powstać podczas analizy (patrz punkt 1.4.3)
2. Przeprowadzenie przez testera sesji testowania eksploracyjnego, w trakcie której tester notuje wszelkie istotne uwagi (zaobserwowane problemy, rekomendacje na przyszłość, informacje o tym, czego nie udało się przeprowadzić podczas sesji, itp.).
3. Ponowne spotkanie na zakończenie sesji, na którym następuje omówienie wyników sesji oraz podjęcie decyzji co do ewentualnych kolejnych działań.

Testowanie eksploracyjne w sesjach odbywa się w ścisłe określonym przedziale czasu (zwykle od 1 do 4 godzin). Tester skupia się na zadaniu określonym w *karcie opisu testu*, ale może oczywiście — jeśli jest taka potrzeba — odejść w jakimś stopniu od zasadniczego zadania, w sytuacji gdy zauważy jakiś poważny defekt w innym obszarze aplikacji. Wyniki sesji są przez testera dokumentowane w karcie sesji testowej.

Kiedy stosować testy eksploracyjne?

Testowanie eksploracyjne będzie dobrym, efektywnym i skutecznym rozwiązaniem, jeśli spełniona zostanie jedna lub więcej spośród poniższych przesłanek:

- specyfikacja testowanego produktu jest niepełna, złej jakości lub jest jej brak;
- istnieje presja czasu, testerzy mają mało czasu na przeprowadzenie testów;
- testerzy znają dobrze produkt i są doświadczeni w testach eksploracyjnych.

Testowanie eksploracyjne jest mocno związane z reaktywną strategią testowania (patrz punkt 5.1.1). W ramach testowania eksploracyjnego można korzystać z innych technik czarnoskrzynkowych, białyoskrzynkowych i opartych na doświadczeniu. Nikt nie może narzucić testerowi sposobu przeprowadzenia sesji. Jeśli np. tester uzna, że warto stworzyć model maszyny stanowej, który opisuje działanie systemu, a następnie, w trakcie sesji eksploracyjnej, zaprojektować przypadki testowe i je wykonać, ma do tego prawo — będziemy mieć wtedy do czynienia z wykorzystaniem testowania planowego (ang. *scripted testing*) w ramach testów eksploracyjnych.

Przykład. W organizacji planowane jest przetestowanie podstawowych funkcjonalności strony webowej. Podjęto decyzję o przeprowadzeniu testów eksploracyjnych i przygotowano dla testera kartę opisu testów przedstawioną w tabeli 4.13. Kierownik testów, na podstawie wyników zebranych z wielu kart sesji testowych, może wyprowadzić następujące przykładowe metryki w celu dalszej analizy:

- liczba przeprowadzonych i ukończonych sesji;
- liczba zaraportowanych defektów;

- czas poświęcony na przygotowanie do sesji;
- czas rzeczywistej sesji testowej;
- czas poświęcony na analizę problemów;
- pokryte funkcjonalności.

TABELA 4.13. Arkusz sesji testowania eksploracyjnego

KARTA OPISU TESTÓW — TE 02-001-01	
Cel	Przetestowanie funkcjonalności logowania
Obszary	Logowanie się jako istniejący użytkownik z użyciem loginu i hasła Logowanie się jako istniejący użytkownik poprzez konto Google Logowanie się jako istniejący użytkownik poprzez konto Facebooka Błędne logowanie — brak użytkownika Błędne logowanie — złe hasło Doprowadzenie do blokady konta Wykorzystanie funkcji przypomnienia hasła Atak SQL injection ⁴ i inne ataki na bezpieczeństwo
Środowisko	Strona dostępna przez różne przeglądarki (Chrome, FF, IE)
Czas	2019-06-12, 11:00 – 13:15
Tester	Jan Kowalski
Notatki testera	
Pliki	(zrzuty ekranu, dane testowe itp.)
Znalezione defekty	
Podział czasu	20% przygotowanie do sesji 70% przeprowadzenie sesji 10% analiza problemów

Technika wędrówek

Andrew Whittaker [Whittaker 2009] proponuje zestaw technik testowania eksploracyjnego zainspirowanych zwiedzaniem i turystyką. Ta metaforyka pozwala testerom zwiększyć ich kreatywność podczas przeprowadzania sesji eksploracyjnych. Celem tych sesji jest:

- zrozumienie tego, jak aplikacja działa, jak wygląda jej interfejs, jakie funkcjonalności oferuje użytkownikowi;
- wymuszenie na oprogramowaniu, aby zademonstrowało swoje możliwości;
- znalezienie defektów.

⁴ SQL injection to jeden z typów ataków polegający na tzw. wstrzykiwaniu złośliwego fragmentu kodu. Jest to atak na zabezpieczenia poprzez wstawienie złośliwej instrukcji SQL w pole do wprowadzania danych z intencją jej wykonania.

Metafora turysty pozwala podzielić obszary oprogramowania analogicznie do części zwiedzanego przez turystę miasta:

- dzielnica biznesowa — odpowiada tym częściom oprogramowania, które związane są z jego częścią „biznesową”, czyli z funkcjonalnościami i cechami, jakie oprogramowanie oferuje użytkownikom;
- dzielnica zabytkowa (historyczne centrum) — odpowiada zastanemu kodowi oraz historii wadliwych funkcjonalności;
- dzielnica turystyczna — odpowiada tym częściom oprogramowania, które przyciągają nowych użytkowników (turystów), funkcjom, których zaawansowany użytkownik (mieszkaniec miasta) raczej już nie używa;
- dzielnica rozrywkowa — odpowiada funkcjom wspierającym oraz cechom związanym z użytecznością i interfejsem użytkownika;
- dzielnica hotelowa — miejsce, w którym turysta odpoczywa; odpowiada momentom, w których użytkownik nie używa aktywnie oprogramowania, jednak oprogramowanie to wciąż wykonuje swoją pracę;
- podejrzane dzielnice — miejsca, do których lepiej się nie zapuszczać. W oprogramowaniu odpowiadają miejscom, w których można dokonywać różnego rodzaju ataków na aplikację.

Whittaker dla każdej dzielnicy opisuje szereg typów eksploracji (zwiedzania).

Więcej informacji o testowaniu eksploracyjnym można znaleźć w [Kaner 1999, Hendrickson 2013].

4.4.3. Testowanie w oparciu o listę kontrolną

Opis techniki

Testowanie w oparciu o listę kontrolną (ang. *checklist-based testing*) polega — tak jak poprzednie dwie techniki opisane w tym podrozdziale — na wykorzystywaniu wiedzy i doświadczenia testera, ale podstawą wykonywania testów są elementy zawarte na tzw. *liście kontrolnej*. Lista kontrolna zawiera warunki testowe, które należy zweryfikować. Lista kontrolna nie powinna zawierać elementów, które można sprawdzić automatycznie, elementów lepiej funkcjonujących jako kryteria wejścia/wyjścia czy też elementów zbyt ogólnych [Brykczyński 1999].



Testowanie oparte na liście kontrolnej może wydawać się techniką podobną do ataku usterkowego. Różnica między tymi technikami jest taka, że atak usterkowy wychodzi od defektów i awarii, a działanie testera ma ujawnić tę konkretną, daną awarię czy defekt. W testowaniu opartym na liście kontrolnej tester również działa w sposób systematyczny, ale sprawdza pozytywne cechy oprogramowania. Podstawą testów jest w tej technice sama lista kontrolna.

Elementy listy kontrolnej są często sformułowane w formie pytania. Lista kontrolna powinna umożliwiać sprawdzenie każdej jej pozycji oddzielnie i bezpośrednio. Pozycje na liście mogą odnosić się do wymagań, cech jakościowych lub innych form warunku testowego. Listy kontrolne mogą być tworzone w celu wsparcia różnych typów testów, w tym testów funkcjonalnych i niefunkcjonalnych (np. 10 heurystyk dla testów użyteczności [Nielsen 1994]).

Niektóre pozycje listy kontrolnej mogą stopniowo stawać się mniej efektywne z czasem, ponieważ osoby rozwijające listę nauczą się unikać popełniania tych samych błędów. Może się również okazać, że trzeba będzie dodać nowe pozycje, aby odzwiadczać defekty o dużej wadze, które zostały ostatnio wykryte. Dlatego listy kontrolne powinny być regularnie aktualizowane na podstawie analizy usterek. Należy jednak uważać, aby lista kontrolna nie stała się zbyt długa [Gawande 2009].

W przypadku braku szczegółowych przypadków testowych testowanie oparte na listach kontrolnych może zapewnić pewien stopień spójności dla testów. Dwóch testerów pracujących na podstawie tej samej listy zapewne wykona swoje zadanie nieco inaczej, ale w ogólności przetestują te same rzeczy (te same warunki testowe), więc silną rzeczą ich testy będą do siebie podobne. Jeśli listy kontrolne są wysokopoziomowe, prawdopodobnie wystąpi pewna zmienność w rzeczywistym testowaniu, co może skutkować potencjalnie większym pokryciem, ale mniejszą powtarzalnością testów.

Rodzaje list kontrolnych

Istnieje wiele różnych typów list kontrolnych, dotyczących różnych aspektów oprogramowania. Listy kontrolne mogą mieć ponadto różny stopień ogólności oraz węższe lub szersze pole zastosowania. Na przykład listy kontrolne na użytek testowania kodu (np. w testach jednostkowych) będą zwykle bardzo szczegółowe i będą zazwyczaj zawierały sporo szczegółów technicznych dotyczących aspektów tworzenia kodu w danym języku programowania. Z kolei lista kontrolna na potrzeby testów użyteczności może być bardzo wysokopoziomowa i ogólna. Oczywiście nie jest to reguła — tester zawsze powinien dostosować poziom szczegółowości listy kontrolnej do swoich własnych potrzeb.

Obszar zastosowania

Listy kontrolne można wykorzystywać w zasadzie przy każdym rodzaju testowania. W szczególności mogą one dotyczyć testów funkcjonalnych oraz niefunkcjonalnych.

W testowaniu opartym na liście kontrolnej tester projektuje, implementuje i uruchamia testy, by pokryć warunki testowe znalezione w liście kontrolnej. Tester może wykorzystywać istniejące listy kontrolne (np. dostępne w internecie), może też je modyfikować i dostosowywać do swoich potrzeb. Może również sam tworzyć takie listy, na podstawie doświadczenia własnego oraz swojej organizacji, dotyczącego defektów i awarii, znajomości oczekiwani użytkowników wytwarzanego produktu bądź też wiedzy na temat przyczyn i objawów awarii oprogramowania. Odwołanie się do własnych doświadczeń może zwiększyć efektywność tej techniki, bo zwykle ci sami ludzie, w tej samej organizacji, pracujący nad podobnymi produktami będą popełniać podobne błędy. Zazwyczaj mniej doświadczeni testerzy pracują na już istniejących listach kontrolnych.

Pokrycie

Dla techniki opartej na listach kontrolnych nie definiuje się szczegółowych miar pokrycia. Oczywiście powinno się pokryć testami każdy element listy kontrolnej. Ponieważ jednak ciężko jest stwierdzić, w jakim stopniu taki element został pokryty (ze względu na to, że technika ta odwołuje się do wiedzy, doświadczenia i intuicji indywidualnego testera), ma to zarówno wady, jak i zalety.

Wadą jest oczywiście brak szczegółowej informacji o pokryciu, a także mniejsza niż w przypadku technik formalnych powtarzalność. Zaletą natomiast jest możliwość uzyskania większego pokrycia, jeśli tej samej listy do testów użyje dwóch lub więcej testerów. Każdy z nich bowiem wykona najprawdopodobniej nieco inny zestaw czynności w celu pokrycia tego samego elementu listy kontrolnej. W ten sposób w testach będzie występowała pewna zmienność, która przełoży się na większe pokrycie, ale kosztem mniejszej powtarzalności.

Przykład. Poniżej przedstawiono kilka przykładowych list kontrolnych. Pierwsza z nich zawiera tzw. heurystyki Nielsena dotyczące użyteczności. Przy każdym elemencie listy kontrolnej w nawiasach dodatkowo podano komentarz, który może pomóc testerowi w przeprowadzeniu testów.

Heurystyki Nielsena dotyczące użyteczności systemu

1. Czy w każdym momencie działania systemu pokazywany jest jego status? (użytkownik zawsze powinien wiedzieć, gdzie się znajduje; w systemie powinny być stosowane tzw. okruszki, pokazujące ścieżkę, jaką przeszedł użytkownik, a także wyraźne tytuły poszczególnych ekranów)
2. Czy jest zgodność między systemem a rzeczywistością? (aplikacja nie powinna używać języka technicznego, lecz prostego, stosowanego na co dzień przez użytkowników systemu języka, dotyczącego dziedziny biznesowej, w której program działa)
3. Czy użytkownik ma kontrolę nad systemem? (np. powinna być możliwość cofnięcia akcji, którą użytkownik wykonałomyłkowo, np. wyjęcie z koszyka produktu włożonego tam przez pomyłkę)
4. Czy system zachowuje spójność i standardy? (rozwiązania dotyczące wyglądu powinny być spójne w ramach całej aplikacji, np. taki sam sposób formatowania odsyłaczy, użycie tej samej czcionki; powinno się też wykorzystywać znane podejścia, np. umieszczanie logotypu firmy w lewym górnym rogu ekranu)
5. Czy system odpowiednio zapobiega błędom? (użytkownik nie powinien mieć wrażenia, że coś poszło nie tak; na przykład nie powinniśmy dawać użytkownikowi możliwości wyboru wersji produktu, która jest niedostępna w sklepie, tylko po to, aby później ujrzał błąd „brak towaru”)
6. Czy system pozwala wybierać zamiast zmuszać do zapamiętywania? (np. opisy pól w formularzu nie powinny znikać w momencie rozpoczęcia ich wypełniania — tak się dzieje, gdy opis jest początkowo umieszczony wewnątrz tego pola)

7. Czy system zapewnia elastyczność i efektywność? (np. zaawansowane opcje wyszukiwania powinny być domyślnie ukryte, jeśli większość użytkowników z nich nie korzysta)
8. Czy system jest estetyczny i nieprzeładowany treścią? (aplikacja powinna podobać się użytkownikom, powinna mieć dobry design, odpowiednią kolorystykę, układ elementów na stronie itp.)
9. Czy zapewniona jest skuteczna obsługa błędów? (jeśli już błąd wystąpi, użytkownik nie powinien otrzymywać technicznego komunikatu o tym fakcie ani o tym, że jest to wina użytkownika; powinna również być dostępna informacja o tym, co użytkownik ma zrobić w tej sytuacji)
10. Czy w systemie jest dostępna pomoc? Czy istnieje dokumentacja użytkownika? (część użytkowników będzie potrzebowała funkcji pomocy; taka opcja powinna być dostępna w aplikacji; powinna również zawierać informacje kontaktowe do pomocy technicznej)

Drugi przykład listy kontrolnej dotyczy przeglądu kodu. Lista ta, jako przykład, zawsze jedynie wybrane aspekty techniczne dotyczące dobrych praktyk pisania kodu i jest dalece niepełna.

Lista kontrolna na użytku inspekcji kodu

1. Czy kod pisany jest zgodnie z obowiązującymi standardami?
2. Czy kod jest sformatowany w sposób spójny?
3. Czy istnieją funkcje, które nigdy nie są wywoływane?
4. Czy zaimplementowane algorytmy są efektywne, o odpowiedniej złożoności obliczeniowej?
5. Czy efektywnie wykorzystywana jest pamięć?
6. Czy istnieją zmienne, które są używane bez wcześniejszego zadeklarowania?
7. Czy kod jest odpowiednio udokumentowany?
8. Czy sposób komentowania jest spójny?
9. Czy każda operacja dzielenia jest zabezpieczona przed dzieleniem przez zero?
10. Czy w instrukcjach IF-THEN bloki instrukcji wykonywane najczęściej sprawdzane są jako pierwsze?
11. Czy każda instrukcja CASE ma blok default?
12. Czy każda zaalokowana pamięć jest zwalniana, gdy przestanie być używana?

4.5. Podejścia do testowania oparte na współpracy

FL-4.5.1 (K2)	Kandydat wyjaśnia, w jaki sposób należy pisać historyjki użytkownika we współpracy z programistami i przedstawicielami jednostek biznesowych.
FL-4.5.2 (K2)	Kandydat klasyfikuje różne sposoby pisania kryteriów akceptacji.
FL-4.5.3 (K3)	Kandydat używa metody wytwarzania sterowanego testami akceptacyjnymi (ATDD), aby zaprojektować przypadki testowe.

Popularność metodów zwinnych doprowadziła do rozwoju metod testowania specyficznych dla tego podejścia, uwzględniających artefakty wykorzystywane przez te metodyki i kładących nacisk na współpracę pomiędzy klientami (biznesem), programistami i testerami. W niniejszym podrozdziale omówiono następujące zagadnienia związane z testowaniem w kontekście metodów zwinnych, wykorzystujące **podejście do testowania oparte na współpracy** (ang. *collaboration-based test approach*):



- historyjki użytkownika jako zwinny odpowiednik wymagań użytkownika (punkt 4.5.1);
- kryteria akceptacji jako zwinny odpowiednik warunków testowych, stanowiące podstawę do projektowania testów (punkt 4.5.2);
- wytwarzanie oparte na testach akceptacyjnych (ATDD) jako często stosowana w metodach zwinnych forma testowania wysokiego poziomu (testowanie systemowe i akceptacyjne), oparta na podejściu „najpierw test” (ang. *test-first*) (punkt 4.5.3).

Każda z technik opisanych w poprzednich podrozdziałach (4.2, 4.3, 4.4) ma konkretny cel w odniesieniu do wykrywania defektów określonego typu. Z kolei podejścia oparte na współpracy skupiają się na unikaniu defektów poprzez współpracę i komunikację.

4.5.1. Wspólne pisanie historyjek użytkownika

W zwinnym wytwarzaniu oprogramowania historyjka użytkownika (ang. *user story*) reprezentuje funkcjonalny przyrost, który będzie wartościowy dla użytkownika lub nabywcy systemu czy oprogramowania. Historyjki użytkownika są pisane w celu uchwycenia wymagań z perspektywy programistów, testerów i przedstawicieli biznesu. W sekwencyjnych modelach cyklu wytwarzania ta wspólna wizja określonej cechy czy funkcji oprogramowania jest osiągana poprzez formalne przeglądy po napisaniu wymagań. W podejściach zwinnych wspólna wizja jest z kolei osiągana poprzez częste nieformalne przeglądy podczas pisania wymagań. Historyjki użytkownika składają się z trzech aspektów [Jeffries 2000], znanych pod nazwą „3C”:

- karty (ang. *card*);
- konwersacji (ang. *conversation*);
- potwierdzenia (ang. *confirmation*).

Karta

Karta to medium opisujące historyjkę użytkownika (np. może to być fizyczna karta w postaci karteczki umieszczanej na tablicy scrumowej⁵ lub wpis w elektronicznej tablicy). Karta identyfikuje wymaganie, jego krytyczność, ograniczenia, przewidywany czas rozwoju i testów oraz — co bardzo istotne — kryteria akceptacji dla tej historyjki. Opis musi być dokładny, ponieważ będzie używany w backlogu produktu. Zespoły zwinne mogą dokumentować historyki użytkownika w różny sposób. Jednym z popularnych formatów jest:

Jako (*zamierzony użytkownik*)
chcę (*zamierzone działanie*),
aby (*cel/rezultat działania, osiągnięty zysk*),

po którym następują kryteria akceptacji. Kryteria te również mogą być dokumentowane w różny sposób (patrz punkt 4.5.2). Niezależnie od podejścia przyjętego do dokumentowania historyjek użytkownika dokumentacja powinna być zwięzła i wystarczająca dla zespołu, który będzie ją implementował oraz testował.

Karty raczej *reprezentują* wymagania klienta, niż je *dokumentują*. Podczas gdy karta może zawierać tekst historyjki, szczegóły są opracowywane w rozmowie i zapisywane w potwierdzeniu [Cohn 2004].

Konwersacja

Rozmowa wyjaśnia, jak oprogramowanie będzie używane. Może być ona udokumentowana lub słowna. Testerzy, mając inny punkt widzenia niż programiści i przedstawiciele biznesu, wnoszą cenny wkład w wymianę myśli, opinii i doświadczeń. Rozmowa rozpoczyna się podczas fazy planowania wydania i trwa, gdy historia jest zaplanowana. Jest prowadzona w gronie interesariuszy posiadających trzy zasadnicze perspektywy spojrzenia na produkt: klienta/użytkownika, programisty oraz testera.

Potwierdzenie

Potwierdzenie ma postać kryteriów akceptacji, które reprezentują pozycje pokrycia przekazujące i dokumentujące szczegóły historyjki użytkownika i które mogą być użyte do określenia, kiedy historyjka jest kompletna. Kryteria akceptacji są zwykle wynikiem rozmowy. Na kryteria akceptacji można patrzeć jak na warunki testowe, które tester musi sprawdzić, aby zweryfikować kompletność historyjki.

Historyjki użytkownika powinny dotyczyć zarówno cech funkcjonalnych, jak i niefunkcjonalnych. Zazwyczaj unikatowa perspektywa testera wpłynie na poprawę historyjki użytkownika poprzez zidentyfikowanie brakujących szczegółów lub wymagań niefunkcjonalnych. Tester może wnieść swój wkład, zadając przedstawicielom biznesu otwarte pytania dotyczące historyjki użytkownika, proponując sposoby jej przetestowania i potwierdzając kryteria akceptacji.

⁵ Tablica scrumowa to stosowana opcjonalnie w Scrumie fizyczna tablica prezentująca bieżący stan iteracji. Stanowi wizualną reprezentację harmonogramu pracy, którą należy wykonać w danej iteracji.

W ramach wspólnego autorstwa historyjek użytkownika można wykorzystywać takie techniki jak burza mózgów czy mapy myśli⁶ (ang. *mind mapping*). Dobre historyjki użytkownika spełniają tzw. własności INVEST, to znaczy są [Web2]:

- niezależne (ang. *Independent*) — nie nakładają się na siebie i mogą być rozwijane w dowolnej kolejności;
- negocjalne (ang. *Negotiable*) — nie są jawnymi kontraktami na funkcje; szczegółowo będą raczej współtworzone przez klienta i programistę podczas rozwoju;
- wartościowe (ang. *Valuable*) — po wdrożeniu powinny przynosić wartość dodaną dla klienta;
- szacowalne (ang. *Estimable*) — klient powinien być w stanie nadać im priorytety, a zespół powinien być w stanie oszacować czas ich realizacji, aby łatwiej zarządzać projektem i optymalizować wysiłek zespołu;
- małe (ang. *Small*) — dzięki temu ich zakres jest łatwy do zrozumienia dla zespołu, a tworzenie samych historyjek jest łatwiej zarządzalne;
- testowalne (ang. *Testable*) — jest to ogólna cecha dobrego wymagania, poprawność implementacji historyjki powinno dać się łatwo zweryfikować.

Jeśli klient nie wie, jak coś przetestować, może to wskazywać, że historyjka nie jest wystarczająco jasna lub że nie odzwierciedla czegoś wartościowego dla niego, lub że klient po prostu potrzebuje pomocy w testowaniu [Web2].

Przykład. Poniższy przykład pokazuje historyjkę użytkownika pisana z perspektywy klienta aplikacji webowej do e-bankowości. Zauważmy, że kryteria akceptacji niekoniecznie wynikać muszą bezpośrednio z treści samej historyjki, ale mogą być wynikiem konwersacji między zespołem a klientami. Kryteria te służą testerowi do zaprojektowania testów akceptacyjnych, które będą weryfikować, czy historyjka została w pełni i poprawnie zaimplementowana.

Historyjka użytkownika US-001-03

Jako klient banku

Chcę móc się zalogować do systemu

Aby mieć możliwość korzystania z produktów bankowych

Kryteria akceptacji

- login musi być poprawnym adresem e-mailowym;
- system odrzuca próbę logowania z niepoprawnym hasłem;
- system odrzuca próbę logowania dla nieistniejącego loginu użytkownika;
- system umożliwia wpisanie w pola *Login* i *Hasło* jedynie znaków alfanumerycznych oraz kropki i symbolu @;
- pola *Login* i *Hasło* mogą przyjąć maksymalnie 32 znaki;

⁶ Więcej o mapach myśli można znaleźć np. na stronie <https://www.mindmapping.com/>.

- po kliknięciu linku *Przypomnij hasło* i wpisaniu adresu e-mailowego na adres ten wysyłany jest link do systemu przypominania hasła;
- system rozpoczyna proces logowania po kliknięciu przycisku *Loguj* lub po naciśnięciu przycisku *Enter*; w tym drugim przypadku proces logowania rozpoczyna się, gdy spełnione są jednocześnie trzy warunki: 1) pole *Login* jest niepuste, 2) aktywnym oknem jest okno *Hasło*, 3) pole *Hasło* jest niepuste.

4.5.2. Kryteria akceptacji

Kryteria akceptacji (ang. *acceptance criteria*) to warunki, które produkt (w zakresie opisywanym historyjką użytkownika, której częścią są te kryteria akceptacji) musi spełnić, aby zostać zaakceptowany przez klienta. Z tej perspektywy kryteria akceptacji mogą być postrzegane jako warunki testowe lub elementy pokrycia, które powinny być sprawdzone przez testy akceptacyjne.



Kryteria akceptacji są używane:

- do określenia granic historyjki użytkownika;
- do osiągnięcia konsensusu pomiędzy zespołem programistów a klientem;
- do opisania zarówno pozytywnych, jak i negatywnych scenariuszy testowych;
- jako podstawa dla testów akceptacyjnych historyjki (patrz punkt 4.5.3);
- jako narzędzie umożliwiające dokładne planowanie i szacowanie.

Kryteria akceptacji są omawiane podczas konwersacji (patrz punkt 4.5.1) i definiowane we współpracy przedstawicieli biznesu, deweloperów i testerów. Kryteria akceptacji — jeśli są spełnione — są używane do potwierdzenia, że historyjka została wykonana w pełni i zgodnie ze wspólną wizją wszystkich interesariuszy. Dostarczą one programistom i testerom rozszerzonej wizji funkcji, którą przedstawiciele biznesu będą walidować. Do pokrycia kryteriów powinny być użyte zarówno testy pozytywne, jak i negatywne. Podczas potwierdzania różni uczestnicy odgrywają rolę testera. Mogą to być tak deweloperzy, jak i specjalści skoncentrowani na wydajności, bezpieczeństwie, interoperacyjności i innych cechach jakościowych. Zespół zwinny uznaje zadanie za zakończone, gdy zestaw kryteriów akceptacji zostanie uznany za spełniony.

Nie ma jednego, ustalonego sposobu pisania kryteriów akceptacji dla historyjki użytkownika. Dwa najczęściej spotykane formaty to:

- kryteria akceptacji w postaci scenariusza (ang. *scenario-oriented*);
- kryteria akceptacji w postaci reguł (ang. *rule-oriented*).

Kryteria akceptacji w postaci scenariusza

Ten format zapisu kryteriów zazwyczaj wykorzystuje format Given/When/Then znany z techniki BDD. Jednak w niektórych przypadkach trudno jest zmieścić kryteria akceptacji w takim formacie, np. gdy docelowi odbiorcy nie potrzebują dokładnych szczegółów scenariuszy testowych. W takim przypadku można zastosować format zorientowany na reguły.

Kryteria akceptacji w postaci reguł

Zazwyczaj w tym formacie kryteria akceptacji przyjmują formę listy weryfikacyjnej z wypunktowaniem lub tabelarycznej formy mapowania wejść na wyjścia. Niektóre frameworki i języki do pisania kryteriów w stylu Given/When/Then (np. Gherkin) udostępniają mechanizmy, które pozwalają w ramach scenariusza stworzyć zestaw reguł, z których każda będzie testowana osobno (jest to forma testowania opartego na danych, ang. *data-driven testing*).

Większość kryteriów akceptacji może być udokumentowana w jednym z dwóch wyżej wymienionych formatów. Zespół może jednak użyć innego, niestandardowego formatu, o ile kryteria akceptacji są dobrze zdefiniowane i jednoznaczne.

Przykład. Historyjka użytkownika opisana w poprzednim punkcie (logowanie do systemu e-bankowości) posiada kryteria akceptacji opisane w postaci reguł. Niektóre z nich można uszczegolić i reprezentować w postaci mapowania wejść na wyjścia, definiując konkretne przykłady danych testowych. Na przykład w języku Gherkin pierwsze trzy punkty listy kryteriów akceptacji dla tej historyjki:

- login musi być poprawnym adresem e-mailowym;
- system odrzuca próbę logowania z niepoprawnym hasłem;
- system odrzuca próbę logowania dla nieistniejącego loginu użytkownika

mogą być określone w następujący sposób:

Scenario outline: correct and incorrect logins and passwords

```
Given User enters <login> as login
  And User enters <password> as password
When User clicks "login" button
Then Login result is <result>
```

Examples:

login	password	result
jan.kowalski@gmail.com	hasloJana	OK
jan.kowalski@gmail	abc123	NOT OK
j.kowalski@mail.abc.pl		NOT OK
jan→kowalski@gmail.com	abcDEF123	NOT OK
	hasloJana	NOT OK

W powyższym przykładzie w opisie kryterium akceptacji przedstawionym w formacie Given/When/Then występują odwołania do zmiennych (w nawiasach < oraz >), a konkretne dane testowe znajdują się poniżej, w sekcji Examples. Ten test wykonuje się pięciokrotnie, za każdym razem z innym zestawem danych testowych. Nazwa sekcji, w której umieszczone są dane testowe, sugeruje, że testy mają formę przykładów, które na konkretnych danych pokazują, jak ma się zachować system. Na przykład logowanie ma być poprawne, jeśli login i hasło spełniają zadane warunki (login jest poprawnym adresem e-mailowym, login i hasło są niepuste i zawierają tylko znaki alfanumeryczne). Z kolei w sytuacji, gdy adres e-mailowy jest niepoprawny (wiersz 2.), hasło jest puste (wiersz 3.), użyte są znaki niebędące znakami alfanumerycznymi (wiersz 4.) lub gdy login jest pusty (wiersz 5.), logowanie nie może się udać.

Przykład. Rozważmy jeszcze jeden przykład historyjki użytkownika, tym razem dla systemu CRM pewnego banku. Historyjka dotyczy realizacji pewnych reguł biznesowych związanych z oferowaniem kart kredytowych klientom spełniającym określone wymagania.

Jako instytucja finansowa

Chcę mieć pewność, że tylko klienci z wystarczającym przychodem rocznym otrzymają kartę kredytową.

Aby zapewnić, że karty kredytowe nie są oferowane klientom, którzy nie będą w stanie spłacić debetu na karcie.

Scenariusz: Istnieją dwa typy kart: jedna z limitem debetowym 2500 PLN, druga z limitem debetowym 5000 PLN. Maksymalny limit karty kredytowej zależy od zarobków klienta (w zaokrągleniu do 1 PLN). Klient musi mieć wynagrodzenie w wysokości przekraczającej 10 000 PLN miesięcznie, aby uzyskać niższy limit debetowy. Jeśli wynagrodzenie przekracza 15 000 PLN miesięcznie, klient uzyskuje wyższy limit debetowy.

Jeśli klient ma miesięczne zarobki <wysokość zarobków>

To kiedy klient składa wniosek o kartę kredytową

Wtedy wniosek powinien być <rezultat oczekiwany> i jeśli jest zaakceptowany, to maksymalny limit karty kredytowej powinien wynosić <maksymalny limit>

Przykłady przypadków testowych napisanych na podstawie tej historyjki przedstawione są poniżej. Zauważmy, że w tym przykładzie tester, tworząc przypadki testowe sprawdzające regułę opisaną w historyjce, jednocześnie starał się pokryć wartości brzegowe dziedziny „miesięczne zarobki”.

ZAROBKI	REZULTAT OCZEKIWANY	MAKSYMALNY LIMIT	UWAGI
10 000 PLN	Odrzucony	0 PLN	dochód <= 10 000 PLN
10 001 PLN	Zaakceptowany	2500 PLN	10 000 PLN < dochód <= 15 000 PLN
15 000 PLN	Zaakceptowany	2500 PLN	10 000 PLN < dochód <= 15 000 PLN
15 001 PLN	Zaakceptowany	5000 PLN	15 000 PLN < dochód

4.5.3. Wytwarzanie sterowane testami akceptacyjnymi (ATDD)

Wytwarzanie sterowane testami akceptacyjnymi (ang. *Acceptance Test-Driven Development, ATDD*) jest podejściem typu „najpierw test” (patrz punkt 2.1.3). Przypadki testowe są tworzone *przed* zaimplementowaniem historyjki użytkownika. Przypadki testowe są tworzone przez członków zespołu posiadających różne perspektywy spojrzenia na produkt, np. klientów, programistów i testerów [Adzic 2009]. Przypadki testowe mogą być manualne lub zautomatyzowane.



Pierwszym krokiem jest tzw. warsztat specyfikacji (ang. *specification workshop*), podczas którego członkowie zespołu analizują, omawiają i piszą historyjkę użytkownika oraz jej kryteria akceptacji. Podczas tego procesu naprawiane są wszelkiego rodzaju problemy występujące w historyjce, takie jak niekompletności, niejasności, sprzeczności lub innego rodzaju defekty. Kolejnym krokiem jest stworzenie testów. Mogą być one zrobione wspólnie przez zespół lub indywidualnie przez testera. W każdym przypadku niezależna osoba, np. przedstawiciel biznesu, dokonuje walidacji testów. Testy to przykłady (ang. *examples*), oparte na kryteriach akceptacji, które opisują specyficzne charakterystyki historyjki użytkownika (patrz punkt 4.5.2). Przykłady te pomagają zespołowi prawidłowo zaimplementować historyjkę użytkownika. Ponieważ przykłady i testy są tym samym, terminy te są często używane zamiennie.

Po zaprojektowaniu testów można zastosować techniki testowe opisane w podrozdziałach 4.2, 4.3 i 4.4. Zazwyczaj pierwsze testy są testami pozytywnymi, potwierdzającymi poprawne zachowanie bez wyjątków lub wystąpienie błędów, obejmującymi sekwencję czynności wykonywanych, jeśli wszystko idzie zgodnie z oczekiwaniemi. W języku angielskim mówi się, że tego typu scenariusze realizują tzw. *happy paths*, czyli ścieżki wykonania, na których wszystko idzie zgodne z planem, bez wystąpienia żadnych awarii. Po wykonaniu testów pozytywnych zespół powinien wykonać testy negatywne, a także testy obejmujące atrybuty niefunkcjonalne (np. wydajność, użyteczność). Testy powinny być wyrażone w sposób zrozumiały dla interesariuszy. Zazwyczaj testy składają się ze zdań w języku naturalnym zawierających niezbędne warunki wstępne (jeśli są), wejścia i powiązane z nimi wyjścia.

Przykłady (czyli testy) muszą obejmować wszystkie cechy historyjki użytkownika i nie powinny wykraczać poza nią. Jednakże kryteria akceptacji mogą wyszczególniać niektóre z zagadnień opisanych w historyjce użytkownika. Ponadto żadne dwa przykłady nie powinny opisywać tych samych cech historyjki użytkownika.

Gdy testy są napisane w formacie wspieranym przez framework automatyzacji testów akceptacyjnych, programiści mogą zautomatyzować te testy poprzez napisanie kodu pomocniczego podczas implementacji funkcji opisanej przez historyjkę użytkownika. Testy akceptacyjne stają się wtedy wykonywalnymi wymaganiami. Przykład takiego kodu pomocniczego pokazany jest w punkcie 2.1.3 przy okazji omawiania podejścia BDD.

Przykład. Założymy, że zespół zamierza przetestować następującą historyjkę użytkownika.

Historyjka użytkownika US-002-02

Jako zalogowany klient banku

Chcę móc dokonać przelewu na inne konto

Aby mieć możliwość transferu środków między kontami

Kryteria akceptacji

- (KA1) kwota przelewu nie może przekraczać salda konta;
- (KA2) konto docelowe musi być poprawne;

- (KA3) dla poprawnych danych (kwota, numery kont) konto źródłowe zmniejsza, a docelowe zwiększa swoje saldo o kwotę przelewu;
- (KA4) kwota przelewu musi być dodatnia i reprezentować poprawną ilość pieniędzy (czyli być z dokładnością maksymalnie do dwóch miejsc po przecinku).

Ponieważ testy powinny weryfikować spełnienie kryteriów akceptacji, przykładowymi pozytywnymi testami funkcjonalnymi mogą tu być następujące testy odnoszące się do kryterium akceptacji (KA3) i weryfikujące przy okazji (KA4):

- PT1: dokonanie „typowego” przelewu, np. przelewu na 1500 PLN z konta o saldzie 2735,45 PLN na inne, poprawne konto; oczekiwany wynik: saldo konta = 1235,45 PLN, saldo konta docelowego zwiększa się o 1500 PLN.
- PT2: dokonanie poprawnego przelewu całego salda konta na inne, poprawne konto (np. przelew na 21,37 PLN z konta o saldzie 21,37 PLN na inne, poprawne konto); oczekiwany wynik: saldo konta = 0 PLN, saldo konta docelowego zwiększa się o 21,37 PLN.

Drugi przypadek testowy wykorzystuje analizę wartości brzegowych dla różnicy między saldem konta a kwotą przelewu.

Kolejny krok polega na stworzeniu testów negatywnych, należy przy tym pamiętać, że testy mają weryfikować spełnienie kryteriów akceptacji. Przykładowo, tester może rozważyć następujące sytuacje:

- PT3: próba dokonania przelewu na inne, poprawne konto w sytuacji, gdy kwota przelewu jest większa niż saldo konta źródłowego (weryfikacja kryterium akceptacji (KA1)).
- PT4: próba dokonania przelewu z poprawną kwotą przelewu i saldem, ale na nieistniejące konto (weryfikacja kryterium akceptacji (KA2)).
- PT5: próba dokonania przelewu z poprawną kwotą przelewu i saldem na to samo konto (weryfikacja kryterium akceptacji (KA2)).
- PT6: próba dokonania przelewu na kwotę niepoprawną (KA4).

Oczywiście dla każdego z tych testów tester może zdefiniować szereg danych testowych, które będą weryfikowały określone zachowanie programu. Tworząc te przypadki testowe, może korzystać z technik czarnoskrzynkowych (np. podziału na klasy równoważności, analizy wartości brzegowych). Na przykład dla przypadku testowego PT1 warto rozważyć przelewy na kwotę: minimalną możliwą (np. 0,01 PLN), „typową” (np. 1500 PLN), kwoty o różnych poziomach dokładności (np. 900,2 PLN, 8321,06 PLN), bardzo dużą kwotę (np. 8 438 483 784 PLN).

Z kolei dla przypadku testowego PT3 warto sprawdzić sytuacje (stosujemy tu metodę klas równoważności i analizę wartości brzegowych):

- gdy kwota przelewu jest o wiele większa niż saldo konta;
- gdy kwota przelewu jest większa o 1 grosz (minimalny możliwy przyrost) niż saldo konta.

Dla PT4 niepoprawne konto może być reprezentowane jako:

- pusty łańcuch znaków;
- numer o poprawnej strukturze (26-cyfrowy), ale nieodpowiadający żadnemu fizycznemu kontu;
- numer o niepoprawnej strukturze — zbyt krótki (np. 25-cyfrowy);
- numer o niepoprawnej strukturze — zbyt długi (np. 27-cyfrowy);
- numer o niepoprawnej strukturze — zawierający niedozwolone znaki, np. litery.

A dla przypadku testowego PT6 warto rozważyć w szczególności:

- liczbę ujemną reprezentującą poprawną kwotę (np. -150,25 PLN);
- liczbę 0;
- łańcuch znakowy zawierający litery (np. 15B,20 PLN);
- liczbę reprezentowaną jako wynik operacji matematycznej (np. 15 + 20 PLN);
- liczbę z większą dokładnością niż dwa miejsca po przecinku (np. 0,009 PLN).

Zauważmy, że każdy z podanych powyżej zestawów danych testowych sprawdza istotnie odmienne od pozostałych przypadków potencjalne ryzyko istniejące w systemie. Testy są więc nieredundantne, nie sprawdzają „tego samego”.

Pytania testowe do rozdziału 4.

Pytanie 4.1

(FL-4.1.1, K2)

Projektowanie, implementacja i wykonanie testów w oparciu o analizę dziedziny wejściowej oprogramowania jest przykładem:

- A. Białoskrzynkowej techniki testowania.
- B. Czarnoskrzynkowej techniki testowania.
- C. Techniki testowania opartego na doświadczeniu.
- D. Techniki testowania statycznego.

Wybierz jedną odpowiedź.

Pytanie 4.2

(FL-4.1.1, K2)

Wskaż wspólną cechę technik takich jak podział na klasy równoważności, testowanie przejść między stanami czy tablice decyzyjne.

- A. Używając tych technik, warunki testowe wyprowadza się na podstawie informacji o wewnętrznej strukturze testowanego oprogramowania.
- B. Aby dostarczyć dane testowe do przypadków testowych projektowanych na podstawie tych technik, powinno się dokonać analizy kodu źródłowego.

- C. Pokrycie w tych technikach mierzy się jako stosunek przetestowanych elementów kodu źródłowego (np. linii kodu) do wszystkich takich elementów zidentyfikowanych w kodzie.
- D. Przypadki testowe skonstruowane na podstawie tych technik mogą wykrywać rozbieżności pomiędzy wymaganiami a ich rzeczywistą implementacją.

Wybierz jedną odpowiedź.

Pytanie 4.3

(FL-4.2.1, K3)

System przydziela zniżkę na zakupy w zależności od kwoty zakupów wyrażonej w PLN, która jest dodatnią liczbą z dokładnością do dwóch miejsc po przecinku. Zakupy do kwoty 99,99 PLN nie uprawniają do zniżki. Zakupy od 100 PLN do 299,99 PLN uprawniają do 5% zniżki. Zakupy powyżej 299,99 PLN uprawniają do 10% zniżki.

Wskaż minimalny zbiór wartości (kwoty PLN), które pozwolą osiągnąć 100% pokrycia dla techniki klas równoważności.

- A. 0,01; 100,99; 500.
- B. 99,99; 100; 299,99; 300.
- C. 1; 99; 299.
- D. 0; 5; 10.

Wybierz jedną odpowiedź.

Pytanie 4.4

(FL-4.2.1, K3)

Maszyna sprzedająca kawę akceptuje monety 50 gr, 1 PLN, 2 PLN oraz 5 PLN. Kawa kosztuje 2,50 PLN. Po wrzuceniu pierwszej monety automat czeka, aż kwota wrzuconych przez użytkownika monet będzie równa lub większa od 2,50 PLN. Gdy to nastąpi, wrzutnia monet jest blokowana, wydawana jest kawa oraz (jeśli to konieczne) reszta. Załóż, że w przypadku, gdy maszyna ma wydać resztę, dysponuje wystarczającą liczbą i nominałami monet do przeprowadzenia tej operacji. Rozważ następujące scenariusze testowe:

Scenariusz 1.

Wrzucenie monet w kolejności: 2 PLN, 1 PLN, 1 PLN.

Oczekiwane zachowanie: automat wydaje kawę oraz 1,50 PLN reszty.

Scenariusz 2.

Wrzucenie monet w kolejności: 1 PLN, 50 gr, 1 PLN.

Oczekiwane zachowanie: automat wydaje kawę oraz nie wydaje reszty.

Scenariusz 3.

Wrzucenie monety 5 PLN.

Oczekiwane zachowanie: automat wydaje kawę oraz 2,50 PLN reszty.

Scenariusz 4.

Wrzucenie monet w kolejności: 50 gr, 50 gr, 2 PLN.

Oczekiwane zachowanie: automat wydaje kawę oraz 50 gr reszty.

Chcesz sprawdzić, czy maszyna rzeczywiście wydaje resztę, gdy użytkownik wrzucił monety na kwotę wyższą niż 2,50 PLN, oraz czy nie wydaje żadnej reszty, jeśli wrzucił dokładnie 2,50 PLN.

Które z tych scenariuszy stanowią minimalny zbiór przypadków testowych osiągający ten cel?

- A. Scenariusz 1., scenariusz 2.
- B. Scenariusz 3., scenariusz 4.
- C. Scenariusz 2., scenariusz 3.
- D. Scenariusz 2., scenariusz 3., scenariusz 4.

Wybierz jedną odpowiedź.

Pytanie 4.5

(FL-4.2.1, K3)

Testujesz aplikację do obsługi systemu kart uprawniających do zniżek przy zakupach. Dostępne są cztery rodzaje kart: zwykła, srebrna, złota i diamentowa oraz trzy możliwe zniżki: 5%, 10% i 15%. Wykorzystujesz technikę podziału na klasy równoważności do sprawdzenia, czy system akceptuje wszystkie rodzaje kart i wszystkie zniżki. Masz już przygotowane następujące przypadki testowe:

- PT01: karta zwykła, 10%
- PT02: karta srebrna, 15%
- PT03: karta złota, 15%
- PT04: karta srebrna, 10%

Jaka jest NAJMIEJSZA liczba przypadków testowych, jakie musisz jeszcze przygotować, by osiągnąć 100% pokrycia „each choice” zarówno wszystkich rodzajów kart, jak i wszystkich możliwych zniżek?

- A. 1.
- B. 0.
- C. 2.
- D. 8.

Wybierz jedną odpowiedź.

Pytanie 4.6

(FL-4.2.2, K3)

Użytkownik definiuje swoje hasło w systemie poprzez wpisanie go w pole tekstowe formularza (domyślnie na początku jest ono puste), a następnie kliknięcie przycisku *Zatwierdź*. System uznaje format hasła za poprawny, jeśli hasło posiada co najmniej 6 znaków i nie więcej niż 11 znaków. Istniejące przypadki testowe stanowią minimalny zbiór testów pokrywający 100% wartości brzegowych długości hasła w wersji dwupunktowej. Kierownik testów zdecydował, że ze względu na wysokie ryzyko modułu walidującego poprawność formatu hasła należy dodać testy, które zapewnią 100% pokrycia wartości brzegowych w wersji trójpunktowej. Hasła o jakich długościach należy DODATKOWO przetestować, aby zapewnić wymagane, silniejsze pokrycie?

- A. 5, 12.
- B. 0, 5, 12.
- C. 4, 7, 10, 13.
- D. 1, 4, 7, 10, 13.

Wybierz jedną odpowiedź.

Pytanie 4.7

(FL-4.2.2, K3)

Użytkownicy myjni samochodowej posiadają karty elektroniczne, na których zapisane jest, ile razy do tej pory skorzystali z myjni. Myjnia oferuje promocję: co dziesiąte mycie jest gratis. Testujesz poprawność oferowania promocji z wykorzystaniem techniki analizy wartości brzegowych.

Wskaż najmniejszy zbiór przypadków testowych, który zapewnia 100% pokrycia wartości brzegowych w wersji dwupunktowej. Wartości w odpowiedziach oznaczają numer kolejny danego (bieżącego) mycia.

- A. 9, 10.
- B. 1, 9, 10.
- C. 1, 9, 10, 11.
- D. nie da się osiągnąć 100% pokrycia wartości brzegowych.

Wybierz jedną odpowiedź.

Pytanie 4.8

(FL-4.2.3, K3)

Analityk przygotował zminimalizowaną tablicę decyzyjną (tabela 4.14), która ma opisywać reguły biznesowe przyznawania darmowych przejazdów autobusowych.

Który z poniższych testów, reprezentujących kombinacje warunków, pokazuje, że reguły biznesowe w tej tablicy są SPRZECZNE?

TABELA 4.14. Tablica decyzyjna dla reguł przyznawania darmowych przejazdów

	R1	R2	R3
Warunki			
Członek parlamentu?	T	---	---
Osoba niepełnosprawna?	---	T	---
Student?	---	---	T
Akcja			
Wolny przejazd?	T	T	N

- A. członek parlamentu = T, niepełnosprawny = N, student = T.
 B. członek parlamentu = T, niepełnosprawny = T, student = N.
 C. członek parlamentu = N, niepełnosprawny = N, student = T.
 D. członek parlamentu = N, niepełnosprawny = N, student = N.

Wybierz jedną odpowiedź.

Pytanie 4.9

(FL-4.2.3, K3)

Tworzysz pełną tablicę decyzyjną, która będzie miała następujące warunki i akcje:

- warunek „wiek osoby” — możliwe wartości: (1) do 18; (2) 19 – 40; (3) od 41
- warunek „miejsce zamieszkania” — możliwe wartości: (1) miasto; (2) wieś
- warunek „wysokość zarobków” — możliwe wartości: (1) do 4000/mc; (2) od 4001/mc
- akcja „przypisać kredyt” — możliwe wartości: (1) TAK; (2) NIE
- akcja „zaoferować ubezpieczenie kredytu” — możliwe wartości: (1) TAK; (2) NIE

Ile kolumn będzie miała pełna tablica decyzyjna dla tego problemu?

- A. 6.
 B. 11.
 C. 12.
 D. 48.

Wybierz jedną odpowiedź.

Pytanie 4.10

(FL-4.2.4, K3)

Tabela 4.15 przedstawia w kolejnych wierszach wszystkie poprawne przejścia między stanami w systemie obsługi procesu logowania. System zawiera trzy stany (Początkowy, Logowanie, Zalogowany) oraz cztery możliwe zdarzenia (Loguj, LoginOK, LoginBłędny, Wyloguj).

TABELA 4.15. Poprawne przejścia systemu dla modułu logowania

STAN	ZDARZENIE	KOLEJNY STAN
Początkowy	Loguj	Logowanie
Logowanie	LoginOK	Zalogowany
Logowanie	LoginBłędny	Początkowy
Zalogowany	Wyloguj	Początkowy

Ille w tym systemie występuje przejście niepoprawnych?

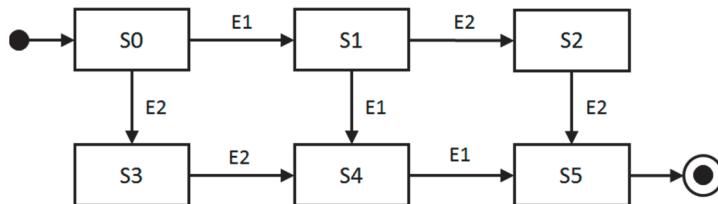
- A. 8.
- B. 0.
- C. 4.
- D. 12.

Wybierz jedną odpowiedź.

Pytanie 4.11

(FL-4.2.4, K3)

Rysunek 4.18 przedstawia diagram przejść między stanami dla pewnego systemu.

**RYSUNEK 4.18.** Diagram przejść sześciostanowej maszyny stanowej

Jaka jest najmniejsza liczba przypadków testowych, które osiągną 100% pokrycia poprawnych przejść?

- A. 2.
- B. 3.
- C. 6.
- D. 7.

Wybierz jedną odpowiedź.

Pytanie 4.12

(FL-4.3.1, K2)

Twoje przypadki testowe osiągają obecnie pełne, stuprocentowe pokrycie instrukcji. Które z poniższych stwierdzeń opisuje poprawną konsekwencję wykonania tych testów?

- A. Osiągnięte zostanie pełne, stuprocentowe pokrycie gałęzi.
- B. Wymuszona zostanie każda wartość logiczna każdej decyzji w kodzie.
- C. Wymuszone zostanie każde możliwe wyjście, jakie może zwrócić testowany program.
- D. Wymuszone zostanie wykonanie każdej instrukcji zawierającej defekt.

Wybierz jedną odpowiedź.

Pytanie 4.13

(FL-4.3.2, K2)

Rozważmy prosty program złożony z trzech linijek (załóż, że w linijkach pierwszej i drugiej pobierane są liczby całkowite dodatnie (0, 1, 2 itd.) i że instrukcje te zawsze wykonają się poprawnie):

1. Pobierz wartość wejściową x.
2. Pobierz wartość wejściową y.
3. Zwróć wartość $x + y$.

Ile przypadków testowych jest potrzebnych do osiągnięcia 100% pokrycia gałęzi w tym kodzie i dlaczego?

- A. Potrzeba dwóch przypadków testowych. Jeden powinien przetestować sytuację, w której zwracana wartość przyjmie wartość 0, a drugi — sytuację, w której zwracana wartość będzie liczbą dodatnią.
- B. Nie trzeba uruchamiać żadnych testów. Pokrycie gałęzi dla tego kodu jest spełnione z definicji, ponieważ program składa się z sekwencyjnego przejścia po trzech instrukcjach i nie ma w nim żadnych rozgałęzień (decyzji).
- C. Wystarczy jeden przypadek testowy z dowolnymi wartościami wejściowymi x, y , ponieważ każdy test spowoduje wykonanie tej samej ścieżki zawierającej wszystkie gałęzie.
- D. Nie da się osiągnąć pokrycia gałęzi skońzoną liczbą przypadków testowych. Aby to osiągnąć, należałoby wymusić zwrócenie wszystkich możliwych wartości sumy $x + y$ w linii trzeciej, co jest niewykonalne.

Wybierz jedną odpowiedź.

Pytanie 4.14

FL-4.3.3 (K2)

Które z poniższych zdań NAJLEPIEJ opisuje korzyść, jaką daje stosowanie biało-skrzynkowych technik testowania?

- A. Możliwość wykrywania defektów w sytuacji, gdy specyfikacja jest niekompletna.
- B. Możliwość wykonywania testów przez programistów, ponieważ techniki te wymagają umiejętności programistycznych.

- C. Upewnienie się, że testy osiągają 100% pokrycia dowolnej czarnoskrzynkowej techniki, ponieważ jest to implikowane przez osiągnięcie 100% pokrycia kodu.
- D. Lepsza kontrola poziomu ryzyka rezydualnego w kodzie, ponieważ jest ono bezpośrednio związane z takimi miarami jak pokrycie instrukcji i pokrycie gałęzi.

Wybierz jedną odpowiedź.

Pytanie 4.15

(FL-4.4.1, K2)

Tester wykorzystuje poniższy dokument w celu zaprojektowania testów:

- Wystąpienie błędu arytmetycznego spowodowanego dzieleniem przez zero
- Wystąpienie błędu zaokrągleń
- Wymuszenie wystąpienia wyniku ujemnego

Jaką technikę stosuje tester?

- A. Analizę wartości brzegowych.
- B. Testowanie oparte na liście kontrolnej.
- C. Testowanie oparte na przypadku użycia.
- D. Zgadywanie błędów.

Wybierz jedną odpowiedź.

Pytanie 4.16

(FL-4.4.2, K2)

Wskaż poprawne zdanie dotyczące używania formalnych (tzn. czarno- i białośkrzynkowych) technik testowania w ramach podejścia testowania eksploracyjnego w sesjach.

- A. Wszystkie formalne techniki testowania są dozwolone, bo testowanie eksploracyjne nie narzuca żadnej konkretnej metody działania.
- B. Wszystkie formalne techniki testowania są zakazane, bo testowanie eksploracyjne jest oparte na wiedzy, umiejętności, intuicji i doświadczeniu.
- C. Wszystkie formalne techniki testowania są zakazane, bo kroki wykonywane w testowaniu eksploracyjnym nie są uprzednio planowane.
- D. Wszystkie formalne techniki testowania są dozwolone, bo tester eksploracyjny potrzebuje podstawy testów, z której może wyprowadzać przypadki testowe.

Wybierz jedną odpowiedź.

Pytanie 4.17

(FL-4.4.3, K2)

Na osiągnięcie jakiego zysku pozwala wykorzystanie testowania opartego na liście kontrolnej?

- A. Docenienie testowania niefunkcjonalnego, które często jest niedoceniane.
- B. Umożliwienie dokładnego określenia pokrycia kodu testami.
- C. Wykorzystanie eksperckiej wiedzy testera.
- D. Zwiększenie spójności testów.

Wybierz jedną odpowiedź.

Pytanie 4.18

(FL-4.5.1, K2)

Podczas spotkania, na którym odbywa się planowanie iteracji, zespół dzieli się ze sobą przemyśleniami na temat historyjki użytkownika. Właściciel produktu chce, by klient wpisywał informacje w formularzu mieszącym się na jednym ekranie. Programista objaśnia, że ta cecha posiada pewne techniczne ograniczenia, związane z ilością informacji możliwych do ujęcia na ekranie.

Która z poniższych sytuacji najlepiej przedstawia kontynuację pisania tej historyjki?

- A. Tester decyduje, że formularz do wprowadzania danych musi mieścić się na jednym ekranie, i opisuje to jako jedno z kryterium akceptacji historyjki, ponieważ to on będzie później przeprowadzał testy akceptacyjne.
- B. Tester wysłuchuje punktów widzenia właściciela produktu oraz programisty i tworzy dwa testy akceptacyjne na potrzeby każdego z dwóch proponowanych rozwiązań.
- C. Tester radzi programistę, by kryteria akceptacyjne wydajności opierały się na standardzie: maksymalnie 1 sekunda na zapis danych. Programista opisuje to jako jedno z kryteriów akceptacji, ponieważ to on odpowiada za wydajność aplikacji.
- D. Tester ustala z programistą i właścicielem produktu ilość niezbędnych informacji wejściowych i wspólnie decydują o redukcji tych informacji do najważniejszych, aby formularz do ich wprowadzania zmieścił się na jednym ekranie.

Wybierz jedną odpowiedź.

Pytanie 4.19

(FL-4.5.2, K2)

Dana jest następująca historyjka użytkownika:

JAKO potencjalny klient e-sklepu

CHCE móc dokonać rejestracji poprzez wypełnienie formularza rejestracyjnego

ABY móc korzystać w pełni z funkcjonalności e-sklepu.

Które DWA z poniższych są przykładami testowalnych kryteriów akceptacyjnych dla tej historyjki?

- A. Proces przeprowadzania rejestracji musi się odbywać odpowiednio szybko.
- B. Po rejestracji użytkownik ma dostęp do funkcji zamawiania zakupów do domu.
- C. System odmawia rejestracji, jeśli użytkownik poda w formularzu e-mail już istniejącego użytkownika.
- D. Operator systemu może przesyłać do realizacji złożone przez użytkownika zamówienie.
- E. Kryteria akceptacji historyjki muszą mieć format Given/When/Then.

Wybierz DWIE odpowiedzi.

Pytanie 4.20

(FL-4.5.3, K3)

Reguła biznesowa dotycząca wypożyczeń książek w uniwersyteckim systemie bibliotecznym stanowi, że czytelnik może wypożyczyć nowe książki, jeśli spełnione są łącznie dwa poniższe warunki:

- czas najdłużej przetrzymywanej książki nie przekracza 30 dni;
- po wypożyczeniu całkowita liczba wypożyczonych książek nie przekroczy 5 książek w przypadku studenta oraz 10 książek w przypadku pracownika uczelni.

Zespół ma zaimplementować historyjkę użytkownika dotyczącą tego wymagania. Historyjka tworzona jest przy użyciu framework'a ATDD i następujących kryteriów akceptacji spisanych jako przykłady (ang. *examples*) w formacie Given/When/Then:

Given <TypUżytkownika> ma już wypożyczonych <Liczba> książek na koncie

And czas najdłużej przetrzymywanej książki wynosi <Dni> dni

When użytkownik chce wypożyczyć <LiczbaW> nowych książek

Then system <Decyzja> na wypożyczenie

Examples:

Nr	TypUżytkownika	Liczba	Dni	LiczbaW	Decyzja
1	Student	3	30	2	nie pozwala
2	Student	4	1	1	pozwala
3	Pracownik	6	32	3	nie pozwala
4	Pracownik	0	0	6	pozwala

Ile z powyższych czterech testów (przykładów) jest BŁĘDNIE zdefiniowanych, tzn. niepoprawnie definiują reguły biznesowe wypożyczania książek?

- A. Żaden — wszystkie są zgodne z regułą biznesową.
- B. Jeden.
- C. Dwa.
- D. Trzy.

Ćwiczenia do rozdziału 4.

Ćwiczenie 4.1

(FL-4.2.1, K3)

Użytkownik wypełnia formularz webowy w celu zakupu biletu na koncert. Dostępne są bilety na trzy koncerty: Iron Maiden, Judas Priest i Black Sabbath. Na każdy z koncertów można zakupić jeden z dwóch typów biletów: na sektor pod sceną i na sektor oddalony od sceny. Użytkownik potwierdza swój wybór poprzez zaznaczenie pól na odpowiednich listach rozwijanych (jedna z nich zawiera nazwy zespołów, druga — typ biletu). W jednej sesji można zakupić tylko jeden bilet na koncert jednego zespołu.

Chcesz sprawdzić poprawność działania systemu dla każdego zespołu oraz — niezależnie — dla typu biletu.

- A) Zidentyfikuj dziedziny oraz ich podziały na klasy równoważności.
- B) Czy w problemie występują klasy niepoprawne? Odpowiedź uzasadnij.
- C) Zaprojektuj najmniejszy możliwy zestaw przypadków testowych, który w 100% pokryje wszystkie klasy równoważności.

Ćwiczenie 4.2

(FL-4.2.1, K3)

Liczba naturalna większa od 1 jest nazywana liczbą pierwszą, jeśli jest podzielna tylko przez dwie liczby: przez 1 oraz przez siebie samą. System pobiera na wejściu liczbę naturalną (wpisywaną przez użytkownika w pole formularza) i zwraca informację, czy jest to liczba pierwsza, czy nie. Pole formularza na daną wejściową posiada mechanizm kontroli poprawności i nie pozwoli na wpisanie żadnego łańcucha znaków, który nie reprezentuje poprawnej liczby naturalnej.

- A) Zidentyfikuj dziedzinę oraz jej podział na klasy równoważności.
- B) Czy w problemie występują klasy niepoprawne? Odpowiedź uzasadnij.
- C) Zaprojektuj najmniejszy możliwy zestaw przypadków testowych, który w 100% pokryje wszystkie klasy równoważności.

Ćwiczenie 4.3

(FL-4.2.2, K3)

Testujesz funkcjonalność płatności w e-sklepie. Moduł na wejściu otrzymuje do datnią kwotę zakupów (w PLN z dokładnością do 1 gr). Kwota ta jest następnie zaokrąglana w górę do najbliższej liczby całkowitej i na podstawie tej zaokrąglonej wartości wyliczany jest rabat według reguł opisanych w tabeli 4.16:

TABELA 4.16. Reguły udzielania rabatu

KWOTA	UDZIELONY RABAT
do 300 PLN	brak
od 301 PLN do 800 PLN	5%
powyżej 800 PLN	10%

Chcesz zastosować analizę wartości brzegowych w wersji dwupunktowej dla sprawdzenia poprawności wyliczania rabatu. Daną wejściową w przypadku testowym jest kwota *przed zaokrągleniem*. Przeprowadź podział na klasy równoważności, wyznacz wartości brzegowe i zaprojektuj przypadki testowe.

Ćwiczenie 4.4

(FL-4.2.2, K3)

System wylicza cenę oprawy obrazu na podstawie podanych parametrów: szerokości i wysokości obrazu (w centymetrach). Poprawna szerokość obrazu mieści się w granicach między 30 a 100 cm włącznie. Poprawna wysokość obrazu mieści się w granicach między 30 a 60 cm włącznie.

System pobiera obie wartości wejściowe poprzez interfejs akceptujący tylko prawne wartości z określonych wyżej przedziałów.

System oblicza pole powierzchni obrazu jako iloczyn szerokości i wysokości. Jeśli pole powierzchni przekracza 1600 cm^2 , cena oprawy wynosi 500 PLN. W przeciwnym razie cena obrazu wynosi 450 PLN.

Zastosuj analizę wartości brzegowych do powyższego problemu w wersji dwupunktowej.

- A) Zidentyfikuj dziedziny, klasy równoważności oraz ich wartości brzegowe.
- B) Wyprowadź możliwie najmniejszą liczbę przypadków testowych pokrywających wartości brzegowe dla wszystkich istotnych parametrów. Czy da się osiągnąć pełne pokrycie wartości brzegowych? Odpowiedź uzasadnij.

Ćwiczenie 4.5

(FL-4.2.3, K3)

Operator systemu wspomagającego przeprowadzanie egzaminów na prawo jazdy wprowadza do systemu, dla kandydata, który przystępuje do egzaminów po raz pierwszy, następujące informacje:

- liczba punktów z egzaminu teoretycznego (liczba całkowita z zakresu 0 – 100),
- liczba błędów popełnionych przez kandydata podczas egzaminu praktycznego (liczba całkowita 0 lub większa).

Kandydat musi przystąpić do obu egzaminów. Kandydat otrzymuje prawo jazdy, jeśli spełnia jednocześnie następujące dwa warunki: uzyskał co najmniej 85 punktów z egzaminu teoretycznego oraz popełnił nie więcej niż dwa błędy na egzaminie praktycznym. Jeśli kandydat nie zda jednego z egzaminów, musi powtórzyć ten egzamin.

Ponadto jeśli kandydat nie zda obu egzaminów, jest zobowiązany do wzięcia dodatkowych godzin nauki jazdy.

Przeprowadź proces tworzenia przypadków testowych na podstawie tablicy decyzyjnej zgodnie z pięciokroikową procedurą opisaną w punkcie 4.3.4, tzn.:

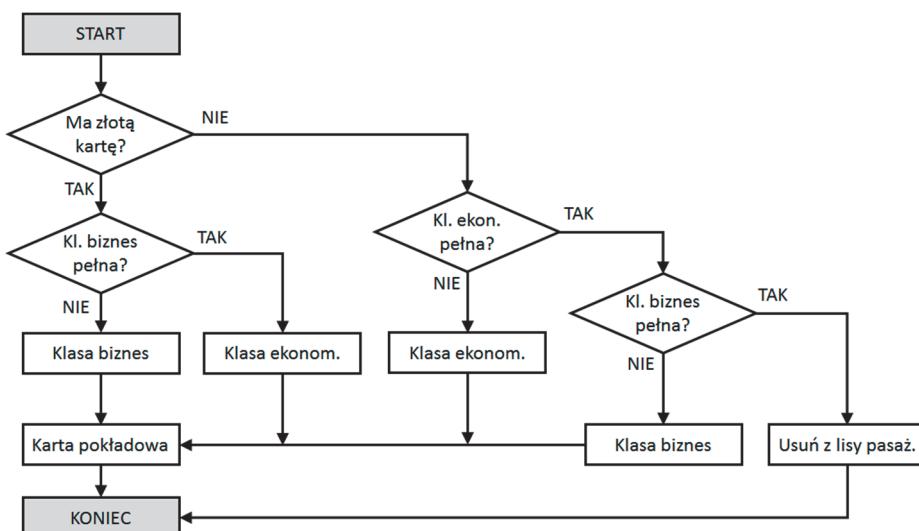
- Zidentyfikuj wszystkie możliwe warunki oraz wypisz je w górnej części tablicy.
- Zidentyfikuj wszystkie możliwe akcje, jakie mogą zajść w systemie, i wypisz je w dolnej części tablicy.
- Wygeneruj wszystkie kombinacje warunków. Wyeliminuj kombinacje nieosiągalne (jeśli takie będą) i wypisz wszystkie pozostałe kombinacje w poszczególnych kolumnach w górnej części tablicy.
- Dla każdej tak zidentyfikowanej kombinacji warunków określ, na podstawie specyfikacji, jakie akcje powinny zajść w systemie, i wpisz je w odpowiedniej kolumnie w dolnej części tablicy.
- Dla każdej kolumny zaprojektuj przypadek testowy zawierający nazwę (opisującą, co dany przypadek testowy sprawdza), warunki wstępne (ang. *pre-conditions*), dane wejściowe, oczekiwane wyjście oraz warunki końcowe (ang. *post-conditions*).

Ćwiczenie 4.6

(FL-4.2.3, K3)

Rysunek 4.19 przedstawia proces przyznawania pasażerom miejsc w samolocie na podstawie tego, czy posiadają złotą kartę, oraz na podstawie tego, czy są wolne miejsca w klasach, odpowiednio, biznes i ekonomicznej.

Wykorzystaj technikę tablic decyzyjnych i opisz za pomocą tablicy decyzyjnej ten sam proces. Czy podczas tworzenia tablicy zauważysz(-łaś) jakiś problem ze specyfikacją? Jeśli tak, zaproponuj rozwiązanie tego problemu.



RYSUNEK 4.19. Proces przyznawania miejsc w samolocie

Ćwiczenie 4.7

(FL-4.2.4, K3)

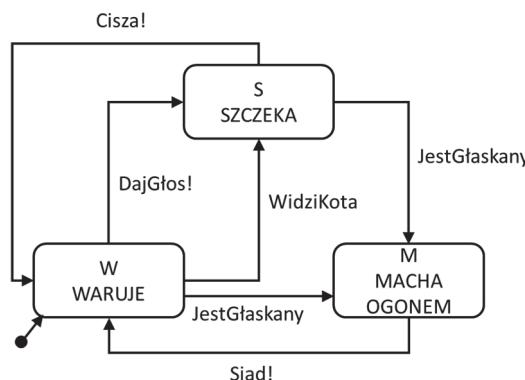
Bankomat początkowo znajduje się w stanie oczekiwania (ekran powitalny). Po włożeniu karty przez użytkownika następuje jej walidacja. Jeśli karta jest niepoprawna, system zwraca ją i kończy działanie z komunikatem „Błąd karty”. W przeciwnym razie system prosi użytkownika o podanie numeru PIN. Jeśli użytkownik poda poprawny PIN, system przechodzi do stanu Zalogowany i kończy działanie. Jeśli poda zły PIN, system prosi o ponowne jego podanie. Jeśli użytkownik poda zły PIN trzykrotnie, karta jest blokowana, system przechodzi do stanu końcowego, a użytkownik otrzymuje komunikat „Blokada karty”.

- Zaprojektuj diagram przejść między stanami dla powyższego scenariusza bez wykorzystywania warunków dozoru. Zidentyfikuj możliwe stany, zdarzenia i akcje, a następnie zbuduj model maszyny stanowej.
- Zaprojektuj model dla tego samego scenariusza, ale z użyciem warunków dozoru. Powinieneś uzyskać model z mniejszą liczbą stanów.
- Dla modelu z punktu A) zaprojektuj możliwie najmniejszą liczbę przypadków testowych, które pokryją:
 - Wszystkie stany.
 - Wszystkie przejścia.

Ćwiczenie 4.8

(FL-4.2.4, K3)

Działanie mechanicznego psa-robota opisane jest przez diagram przejść między stanami przedstawiony na rysunku 4.20.



RYSUNEK 4.20. Diagram stanów psa-robota

- Ille jest przejść niepoprawnych na tym diagramie?
- Zaprojektuj przypadki testowe spełniające kryterium pokrycia wszystkich przejść. Przyjmij zasadę, że w przypadku testowania przejść niepoprawnych jeden przypadek testowy testuje tylko jedno przejście niepoprawne.

Ćwiczenie 4.9

(FL-4.5.3, K3)

Jako tester rozpoczynasz pracę nad projektowaniem testów akceptacyjnych dla następującej historyjki użytkownika:

US-01-002 Rejestracja nowego użytkownika

Pracochłonność: 3

Jako: dowolny użytkownik — potencjalny klient

Chcę: móc wypełnić webowy formularz rejestracji

Żeby: móc korzystać z oferty dostarczyciela usług

Kryteria akceptacji:

KA1 Loginem (nazwą) użytkownika jest poprawny adres e-mailowy.

KA2 Użytkownik nie może zarejestrować się przy użyciu już istniejącego loginu.

KA3 Ze względów bezpieczeństwa użytkownik musi wpisać hasło w dwa niezależne pola i hasła te muszą być identyczne; system nie pozwala wkleić w te pola łańcucha znaków, musi on być wpisany ręcznie.

KA4 Hasło musi zawierać od 6 do 12 znaków, co najmniej jedną cyfrę i co najmniej jedną dużą literę.

KA5 Poprawna rejestracja skutkuje wysłaniem e-maila zawierającego link, którego kliknięcie potwierdza rejestrację i aktywuje konto użytkownika.

Zaproponuj kilka przykładowych testów akceptacyjnych, zawierających konkretne dane testowe oraz oczekiwane zachowanie systemu.

ROZDZIAŁ 5.

Zarządzanie czynnościami testowymi

Słowa kluczowe

analiza ryzyka — całkowity proces identyfikacji ryzyk i ich oceny.

identyfikacja ryzyka — proces znajdowania, rozpoznawania i opisywania ryzyka.

kontrola ryzyka — ogólny proces ograniczania ryzyka i monitorowania ryzyka.

kryterium wejścia — zbiór warunków definiujących, kiedy można oficjalnie rozpoczęć określone zadanie. *Synonim:* definicja gotowości.

kryterium wyjścia — zbiór warunków definiujących, kiedy można oficjalnie uznać określone zadanie za ukończone. *Synonim:* kryteria zupełności, definicja ukończenia, kryteria zakończenia testów.

kwadranty testowe — model klasyfikacji typów/poziomów testów według czterech grup odniesionych do dwóch wymiarów celów testowania: wspieranie zespołu vs. krytyka produktu oraz cel technologiczny vs. cel biznesowy.

łagodzenie ryzyka — proces, w którym podejmowane są decyzje i wdrażane środki ochronne w celu zmniejszenia lub utrzymania ryzyk na określonych poziomach.

monitorowanie ryzyka — czynność polegająca na sprawdzaniu i zgłaszaniu interesariuszom stanu znanych ryzyk (zagrożeń).

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

nadzór nad testami — działalność, która rozwija i stosuje działania naprawcze, aby utrzymać w toku testy projektu, gdy odbiegają od tego, co zostało zaplanowane.

ocena ryzyka — proces oceny zidentyfikowanego ryzyka i oceny poziomu ryzyka.

piramida testów — graficzny model reprezentujący relacje między ilością testów a poziomem testów, z większą ilością testów dla niższych poziomów i mniejszą dla wyższych.

plan testów — dokumentacja opisująca cele testowe do osiągnięcia oraz środki i harmonogram ich realizacji, zorganizowane tak, by koordynować czynności testowe.

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

podejście do testowania — implementacja strategii testowej w określonym projekcie.

poziom ryzyka — jakościowa lub ilościowa miara ryzyka zdefiniowana przez wpływ i prawdopodobieństwo. *Synonim:* eksponowanie ryzyka.

raport o defekcie — dokumentacja wystąpienia, charakteru i statusu usterki.

raport o postępie testów — rodzaj raportu z testów, wykonywany w regularnych odstępach czasu, o postępie w testach na podstawie podstawy testów, ryzyka i alternatyw wymagających decyzji.

ryzyko — czynnik, który w przyszłości może skutkować negatywnymi konsekwencjami.

ryzyko produktowe — ryzyko wpływające na jakość produktu.

ryzyko projektowe — ryzyko wpływające na sukces projektu.

sumaryczny raport z testów — raport z testów, który zawiera ocenę odpowiednich pozycji testowych w odniesieniu do kryteriów wyjścia.

testowanie oparte na ryzyku — testowanie, w którym zarządzanie, wybór, priorytyzacja i wykorzystanie działań testowych i zasobów są oparte na odpowiednich typach i poziomach ryzyka.

zarządzanie defektami — proces rozpoznawania, rejestrowania, klasyfikacji, badania, rozwiązywania i usuwania defektów.

zarządzanie ryzykiem — proces radzenia sobie z ryzykiem.

5.1. Planowanie testów

FL-5.1.1 (K2)	Kandydat omawia na przykładach cel i treść planu testów.
FL-5.1.2 (K1)	Kandydat rozpoznaje, jaki jest wkład testera w planowanie iteracji i wydań.
FL-5.1.3 (K2)	Kandydat porównuje i zestawia ze sobą kryteria wejścia i kryteria wyjścia.
FL-5.1.4 (K3)	Kandydat oblicza pracochłonność testowania przy użyciu technik szacowania.
FL-5.1.5 (K3)	Kandydat stosuje priorytetyzację przypadków testowych.
FL-5.1.6 (K1)	Kandydat pamięta pojęcia związane z piramidą testów.
FL-5.1.7 (K2)	Kandydat podsumowuje kwadranty testowe oraz ich relację do poziomów testów i typów testów.

5.1.1. Cel i treść planu testów

Plan testów (ang. *test plan*) to dokument przedstawiający szczegółowy opis celów projektu testowego, środków niezbędnych do uzyskania tych celów oraz harmonogram czynności testowych. W typowych projektach tworzy się zazwyczaj jeden plan testów, nazywany czasem głównym planem testów (ang. *master test plan*). Jednak w większych projektach można spotkać kilka planów, na przykład główny plan testów oraz plany odpowiadające zdefiniowanym w projekcie poziomom testów. W takiej sytuacji mogą istnieć np.: plan testów integracji modułów, plan testów systemowych, plan testów akceptacyjnych. Szczegółowe plany testów mogą również dotyczyć typów testów, które są planowane do przeprowadzenia w projekcie (np. plan testów wydajnościowych).



Plan testów opisuje **podejście do testowania** (ang. *test approach*) i pomaga zespołowi upewnić się, że czynności testowe mogą zostać rozpoczęte oraz, gdy zostaną ukończone, że zostały przeprowadzone w sposób właściwy. Jest to możliwe dzięki zdefiniowaniu w planie testów określonych kryteriów wejścia i wyjścia (patrz punkt 5.1.3) dla poszczególnych czynności lub aktywności testowych. Plan testów stanowi również potwierdzenie, że — o ile będzie przestrzegany — testowanie będzie przebiegać zgodnie z przyjętą w projekcie strategią testów oraz obowiązującą w organizacji polityką testów.



Bardzo rzadko projekt potoczy się w 100% tak, jak go zaplanowaliśmy. W większości sytuacji konieczne będą mniejsze bądź większe modyfikacje. Pojawia się zatem naturalne pytanie: po co w takim razie tracić czas na planowanie? Dwight Eisenhower, generał armii USA i późniejszy prezydent Stanów Zjednoczonych, powiedział kiedyś słynne zdanie: „Przygotowując się do bitwy, zawsze orientowałem się, że plany są bezużyteczne, ale planowanie jest nieodzowne”¹. Rzeczywiście najbardziej wartościową częścią tworzenia planu testów jest sam proces **planowania testów** (ang. *test planning*). Planowanie w pewnym sensie „wymusza” na testerach ukierunkowanie ich myślenia na przyszłe wyzwania i ryzyka związane z harmonogramem, zasobami, ludźmi, narzędziami, kosztami, wysiłkiem itp.



Podczas planowania testerzy mogą bezpiecznie, „na sucho”, zidentyfikować zagrożenia i przemyśleć podejście do testów, które będzie najefektywniejsze. Bez planowania testerzy byliby nieprzygotowani na wiele różnych niepożądanych zdarzeń, jakie wydarzą się podczas realizacji projektu. To z kolei stwarzałoby bardzo duże ryzyko niepowodzenia projektu, tzn. nieukończenia go lub ukończenia z opóźnieniem, przekroczonym budżetem bądź ze zredukowanym zakresem wykonanych prac.

Na proces planowania wpływa wiele czynników, które testerzy powinni wziąć pod uwagę, aby jak najlepiej zaplanować wszystkie czynności procesu testowego. Wśród tych czynników należy w szczególności rozważyć następujące:

¹ https://pl.wikiquote.org/wiki/Dwight_Eisenhower

- polityka i strategia testów obowiązujące w danej organizacji;
- stosowane cykle i metody wytwarzania oprogramowania;
- zakres testowania;
- cele;
- ryzyka;
- ograniczenia;
- krytyczność;
- testowalność;
- dostępność zasobów.

Typowy plan testów zawiera następujące informacje:

Kontekst testowania. Zakres, cele, ograniczenia, informacja o podstawie testów.

Założenia i ograniczenia projektu testowego.

Interesariusze. Role, odpowiedzialności, wpływ na proces testowy (np. władza vs. zainteresowanie), potrzeby związane z zatrudnieniem i szkoleniem ludzi

Komunikacja. Rodzaje i częstość komunikacji, szablony wykorzystywanych dokumentów

Lista ryzyk. Ryzyka produktowe, ryzyka projektowe (zob. podrozdział 5.2)

Podejście do testowania. Poziomy testów (zob. punkt 2.2.1), typy testów (zob. punkt 2.2.2), techniki testowania (zob. rozdział 4), produkty pracy związane z testowaniem, kryteria wejścia i wyjścia (zob. punkt 5.1.3), poziom niezależności testowania (zob. punkt 1.5.3), definicja wykorzystywanych metryk testowych (zob. punkt 5.3.1), wymagania dotyczące danych testowych, wymagania dotyczące środowiska testowego, odchylenia od dobrych praktyk stosowanych w organizacji (z podaniem powodu)

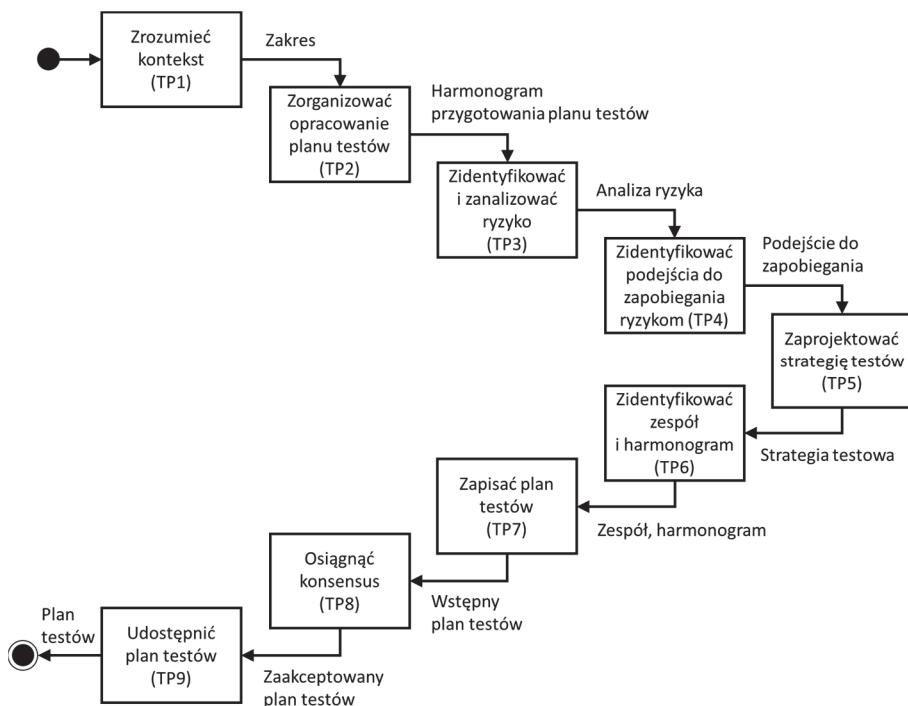
Harmonogram. Zob. punkt 5.1.5.

W miarę realizacji projektu i planu testów pojawiają się dodatkowe informacje, uzupełniające plan. Tym samym planowanie testów to czynność o charakterze ciągłym, wykonywane jest przez cały cykl życia produktu; czasami obejmuje też fazę pielegnacji. Należy zdawać sobie sprawę z faktu, że początkowy plan testów będzie aktualizowany, bo będą wykorzystywane informacje zwrotne z czynności testowych do rozpoznawania zmieniających się ryzyk i odpowiedniego korygowania planów.

Rezultaty procesu planowania mogą być udokumentowane w głównym planie testów oraz w oddzielnych planach dotyczących poszczególnych poziomów testów (np. testowanie systemowe i akceptacyjne) oraz typów testów (np. testowanie użytkownictwa czy testowanie wydajnościowe).

W czasie planowania testów (patrz rysunek 5.1) wykonywane są następujące czynności:

- określanie zakresu i celów testowania oraz związanego z nim ryzyka;
- określanie ogólnego podejścia do testowania;
- integrowanie i koordynowanie czynności testowych w ramach czynności związanych z cyklem wytwarzania oprogramowania;
- podejmowanie decyzji dotyczących tego, co należy przetestować, jakie kadrów i inne zasoby będą niezbędne do wykonania poszczególnych czynności testowych oraz w jaki sposób należy wykonać te czynności;
- planowanie czynności związanych z analizą, projektowaniem, implementacją, wykonywaniem i oceną testów poprzez określenie konkretnych terminów (podejście sekwencyjne) oraz umieszczenie tych czynności w kontekście poszczególnych iteracji (podejście iteracyjne);
- dokonywanie wyboru miar służących do monitorowania i nadzorowania testów;
- określanie budżetu na potrzeby zbierania i oceny miar;
- ustalanie poziomu szczegółowości i struktury dokumentacji (szablony lub dokumenty przykładowe).



RYSUNEK 5.1. Proces planowania testów (zgodnie ze standardem ISO/IEC/IEEE29119-2)

Jedna z części planu testów — podejście do testów — będzie w praktyce realizacją określonej strategii testowej obowiązującej w organizacji bądź w projekcie. Strategia testów stanowi ogólny opis realizowanego procesu testowego, zazwyczaj na poziomie produktu bądź organizacji.

Typowe strategie testów to:

- **Strategia analityczna.** Strategia ta opiera się na analizie określonego czynnika (np. wymagań lub ryzyka). Przykładem podejścia analitycznego jest testowanie oparte na ryzyku, w którym punktem wyjścia do projektowania testów i ustalania ich priorytetów jest poziom ryzyka.
 - **Strategia oparta na modelu.** W przypadku tej strategii testy projektuje się na podstawie modelu konkretnego wymaganego aspektu produktu, np.: funkcji, procesu biznesowego, struktury wewnętrznej bądź charakterystyki niefunkcjonalnej (takiej jak niezawodność). Modele można tworzyć na podstawie następujących informacji: modeli procesów biznesowych, modeli stanów czy modeli wzrostu niezawodności.
 - **Strategia metodyczna.** Podstawą tej strategii jest systematyczne stosowanie z góry określonego zbioru testów lub warunków testowych, na przykład standardowego zbioru testów bądź listy kontrolnej zawierającej typowe lub prawdopodobne typy awarii. Zgodnie z tym podejściem wykonuje się między innymi testowanie na podstawie listy kontrolnej lub ataków usterek oraz testowanie na podstawie charakterystyk jakościowych.
 - **Strategia zgodna z procesem** (lub standardem). Strategia ta polega na tworzeniu przypadków testowych na podstawie zewnętrznych reguł i standardów (wynikających na przykład z norm branżowych), dokumentacji procesów bądź rygorystycznego identyfikowania i wykorzystywania podstawy testów, a także na podstawie wszelkich procesów lub standardów narzuconych przez organizację.
 - **Strategia kierowana** (lub konsultatywna). Podstawą tej strategii są przede wszystkim porady, wskazówki i instrukcje interesariuszy, ekspertów metrycznych lub ekspertów technicznych — również spoza zespołu testowego lub organizacji.
 - **Strategia minimalizująca regresję.** Ta strategia — motywowana chęcią uniknięcia regresji już istniejących funkcjonalności — przewiduje ponowne wykorzystanie dotychczasowych testaliów (zwłaszcza przypadków testowych), szeroką automatyzację testów regresji oraz stosowanie standaryzowanych zestawów testowych.
 - **Strategia reaktywna.** W przypadku tej strategii testowanie jest ukierunkowane bardziej na reagowanie na zdarzenia niż na realizację ustalonego z góry planu (jak w przypadku wyżej opisanych strategii), a testy są projektowane i mogą być natychmiast wykonywane na bazie wiedzy uzyskanej dzięki rezultatom wcześniejszych testów. Przykładem takiego podejścia jest testowanie eksploracyjne, w którym testy są wykonywane i oceniane w odpowiedzi na zachowanie oprogramowania podlegającego testowaniu.
-

W praktyce różne strategie można, a nawet powinno się łączyć. Podstawy wyboru strategii testów obejmują:

- ryzyko niepowodzenia projektu:
 - zagrożenia wobec produktu,
 - zagrożenie dla ludzi, środowiska lub firmy spowodowane awarią produktu,
 - niewystarczające umiejętności i doświadczenie ludzi w projekcie;
- przepisy (zewnętrzne i wewnętrzne) dotyczące procesu wytwarzania;
- cel przedsięwzięcia testowego;
- misję zespołu testowego;
- rodzaj i specyfikę produktu.

5.1.2. Wkład testera w planowanie iteracji i wydań

W iteracyjnych podejściach do tworzenia oprogramowania występują dwa rodzaje planowania: planowanie wydania i planowanie iteracji.

Planowanie wydania

Planowanie wydania wybiera w przyszłość do wydania produktu. Planowanie wydania definiuje i redefiniuje backlog produktu, może też obejmować udoskonalanie większych historyjek użytkownika w zbiór mniejszych historyjek. Planowanie wydania dostarcza podstaw dla podejścia do testów i planu testów obejmującego wszystkie iteracje projektu. Podczas planowania wydania przedstawiciele biznesu (we współpracy z zespołem) ustalają i nadają priorytety historyjkom użytkownika dla danego wydania. W oparciu o te historyjki użytkowników identyfikowane są ryzyka projektowe i produktowe (patrz podrozdział 5.2) oraz przeprowadzane jest wysoko-poziomowe oszacowanie wysiłku (patrz punkt 5.1.4).

Testerzy są zaangażowani w planowanie wydania i wnoszą wartość dodaną szczególnie w następujących działaniach:

- definiowanie testowalnych historyjek użytkownika, w tym kryteriów akceptacji;
- udział w analizie ryzyk projektowych i produktowych (jakościowych);
- szacowanie nakładów na testowanie związane z historyjkami użytkownika;
- definiowanie niezbędnych poziomów testów;
- planowanie testów dla wydania.

Planowanie iteracji

Po zakończeniu planowania wydania rozpoczyna się planowanie iteracji dla pierwszej iteracji i jest powtarzane na początku każdej kolejnej iteracji. Planowanie iteracji wybiera w przyszłość do końca pojedynczej iteracji i dotyczy backlogu iteracji. Podczas planowania iteracji zespół wybiera historyjki użytkownika ze spriorytetyzowanego backlogu produktu, opracowuje je, przeprowadza analizę ryzyka dla historyjek

użytkownika i szacuje pracę potrzebną na implementację każdej z wybranych historyjek (patrz punkt 5.1.4). Jeśli historyjka użytkownika jest niejasna, a próby jej wyjaśnienia nie powiodły się, zespół może odmówić jej przyjęcia i użyć następnej historyjki użytkownika w oparciu o priorytet. Przedstawiciele biznesu muszą odpowiedzieć na pytania zespołu dotyczące każdej historyjki, aby zespół rozumiał, co powinien zaimplementować i jak przetestować każdą historyjkę.

Liczba wybranych historyjek jest oparta na tzw. prędkości zespołu² (ang. *team velocity*) i szacowanym rozmiarze wybranych historyjek użytkownika. Po sfinalizowaniu treści iteracji historyjki użytkownika są dzielone na zadania, które będą realizowane przez odpowiednich członków zespołu.

Testerzy są zaangażowani w planowanie iteracji i wnoszą wartość dodaną szczególnie w następujących działańach:

- uczestniczenie w szczegółowej analizie ryzyka historyjek użytkownika;
- określanie testowalności historyjek użytkownika;
- współtworzenie testów akceptacyjnych dla historyjek użytkownika;
- podział historyjek użytkownika na zadania (w szczególności zadania testowe);
- oszacowanie nakładów na testowanie dla wszystkich zadań testowych;
- identyfikowanie funkcjonalnych i niefunkcjonalnych aspektów testowanego systemu;
- wspieranie i udział w automatyzacji testów na wielu poziomach testowania.

5.1.3. Kryteria wejścia i kryteria wyjścia

Kryteria wejścia

Kryteria wejścia (ang. *entry criteria*, w podejściu zwinnym nazywane **definicją gotowości**, ang. *definition of Ready*) określają warunki wstępne, które muszą zostać spełnione przed rozpoczęciem danej czynności testowej. W przypadku ich niespełnienia wykonanie testowania może być trudniejsze i bardziej czasochłonne, kosztowne i ryzykowne.



Kryteria wejścia obejmują dostępność:

- testowalnych wymagań, historyjek użytkownika i/lub modeli;
- elementów testowych, które spełniły kryteria wyjścia obowiązujące na wcześniejszych poziomach testowania, głównie w podejściu kaskadowym;

² Prędkość zespołu to empirycznie wyznaczona przez zespół wielkość pracy, jaką jest on w stanie wykonać w trakcie pojedynczej iteracji. Zazwyczaj wyrażana jest w tzw. punktach historyjek użytkownika (ang. *user story points*). Rozmiar każdej historyjki jest również szacowany w tych jednostkach, zatem zespół wie, ile historyjek użytkownika może wziąć do backlogu iteracji — suma ich złożoności nie może przekroczyć prędkości zespołu. Dzięki temu zmniejsza się ryzyko, że zespół nie zdąży wykonać całej pracy do wykonania w ramach iteracji oraz że nie zakończy pracy przed końcem iteracji, powodując tzw. „puste przebiegi”.

- środowiska testowego;
- niezbędnych narzędzi testowych;
- danych testowych i innych niezbędnych zasobów,

a także odpowiedni początkowy poziom jakości obiektu testowego (np. wszystkie testy dymne³ zakończyły się pomyślnie).

Kryteria wejścia chronią nas przed rozpoczęciem zadań, do których nie jesteśmy jeszcze w pełni przygotowani.

Kryteria wyjścia

Kryteria wyjścia (ang. *exit criteria*, w podejściu zwinnym nazywane **definicją ukończenia**, ang. *definition of done*) określają warunki, które muszą zostać spełnione, aby można było uznać wykonywanie danego poziomu testów lub zbioru testów za zakończone. Kryteria te powinny być zdefiniowane w odniesieniu do każdego poziomu i typu testów; mogą się różnić w zależności od celów testów.



Typowe kryteria wyjścia to:

- zakończenie wykonywania zaplanowanych testów;
- osiągnięcie należytego poziomu pokrycia (np. wymagań, historyjek użytkownika, kryteriów akceptacji, kodu);
- nieprzekroczenie uzgodnionego limitu nieusuniętych defektów;
- uzyskanie wystarczająco niskiej szacowanej gęstości defektów;
- uzyskanie wystarczająco wysokich wskaźników niezawodności.

Należy pamiętać, że czasami czynności testowe mogą zostać skrócone z powodu:

- wykorzystania całego budżetu;
- upływu zaplanowanego czasu;
- presji związanej z wprowadzeniem produktu na rynek.

W takich sytuacjach interesariusze i właściciele biznesowi projektu powinni zapoznać się z ryzykiem związanym z uruchomieniem systemu bez dalszego testowania i zaakceptować je. Ponieważ te sytuacje często występują w praktyce, testerzy (zwłaszcza osoba w roli kierownika zespołu testowego) powinni dostarczyć informacji opisujących aktualny stan systemu, unaoczniających ryzyko.

Kryteria wyjścia zazwyczaj mają charakter miar staranności bądź kompletności. Wyrażają pożądaną stopień wykonania danej pracy. Kryteria wyjścia pozwalają nam stwierdzić, czy wykonaliśmy określone zadania tak, jak to zostało zaplanowane.

³ Test dymny to szybki, prosty test sprawdzający poprawność realizacji podstawowych funkcjonalności.

Przykład. Zespół przyjął następujące kryteria wyjścia z fazy testów dynamicznych:

- (KW1) osiągnięto 100% pokrycia wymagań testami;
- (KW2) osiągnięto co najmniej 75% pokrycia instrukcji dla każdego z testowanych modułów;
- (KW3) brak otwartych (nienaprawionych) defektów o najwyższym poziomie dotkliwości;
- (KW4) co najwyżej dwa otwarte defekty o średnim bądź niskim poziomie dotkliwości.

Po przeanalizowaniu raportu na zakończenie fazy testów dynamicznych okazało się, że:

- dla każdego wymagania uruchomiony (i zaliczony) został przynajmniej jeden przypadek testowy;
- dla dwóch spośród czterech modułów osiągnięto pełne pokrycie instrukcji, dla trzeciego — 80%, a dla czwartego — 70%;
- jeden z testów wykrył defekt o priorytecie średnim, który nadal nie jest zamknięty.

Z tej analizy wynika, że kryteria (KW1), (KW3) i (KW4) są spełnione, natomiast kryterium (KW2) nie jest spełnione dla jednego z modułów. Zespół postanowił przeanalizować, jakie fragmenty kodu tego modułu nie są pokryte, i dodał przypadek testowy pokrywający dodatkową ścieżkę przepływu sterowania w tym module, co spowodowało zwiększenie pokrycia dla tego modułu z 70% do 78%. W tym momencie kryterium (KW2) zostało spełnione. Ponieważ wszystkie kryteria wyjścia z fazy testów dynamicznych są teraz spełnione, zespół formalnie uznaje fazę testów dynamicznych za zakończoną.

5.1.4. Techniki szacowania

Wysiłek testowy (ang. *test effort*) to ilość pracy związanej z testowaniem, potrzebnej do osiągnięcia celów projektu testowego. Wysiłek testowy jest często głównym lub znaczącym czynnikiem kosztowym podczas rozwoju oprogramowania, który czasami pochłania ponad 50% całkowitych nakładów na rozwój [Jackson 2007]. Jedno z najczęstszych pytań, jakie członkowie zespołu słyszą od swojego menedżera brzmii: „ile czasu wam to zajmie?”. Problemy opisane powyżej sprawiają, że szacowanie testów jest bardzo ważną czynnością z punktu widzenia zarządzania testami. Każdy tester powinien posiadać umiejętność szacowania wysiłku testowego, umieć posługiwać się odpowiednimi technikami szacowania, rozumieć wady i zalety poszczególnych podejść oraz mieć świadomość takich zagadnień jak dokładność szacowania (błąd szacowania).

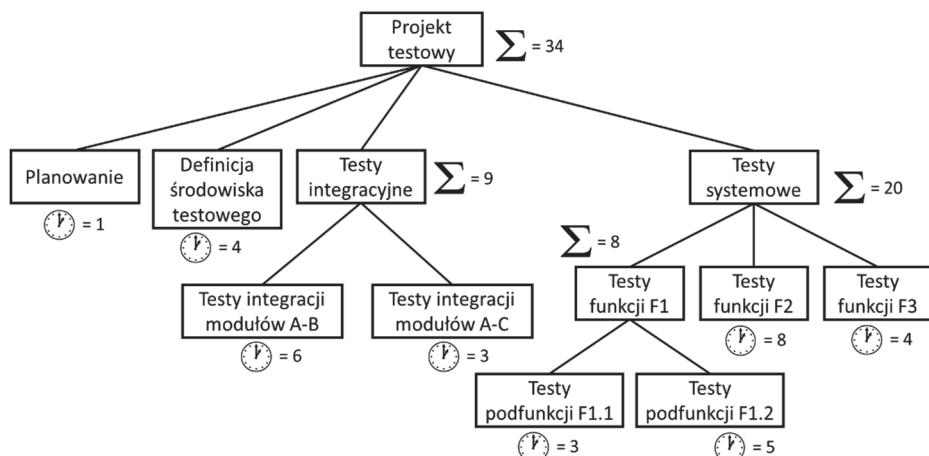
Wysiłek testowy mierzy się jako pracochłonność, czyli miarę iloczynową (czas · zasoby). Na przykład wysiłek 12 osobodni oznacza, że dana praca do wykonania będzie wykonana w 12 dni przez jednego człowieka albo w 6 dni przez 2 ludzi, albo w 4 dni przez 3 ludzi itd. Ogólnie będzie to praca wykonana w x dni przez y ludzi, gdzie $x \cdot y = 12$.

Należy jednak uważać, ponieważ w praktyce miary iloczynowe „nie skalują się”. Jeśli np. zespół liczący 3 osoby wykona jakąś pracę w 4 dni, pracochłonność wyniesie 12 osobodni. Jeśli jednak kierownik doda do zespołu nową osobę, licząc na to, że 4 osoby wykonają tę samą pracę w 3 dni, może się rozczarować — tak powiększony zespół może tę pracę wykonać nadal w 4 dni lub nawet w dłuższym czasie. Jest to spowodowane szeregiem czynników, takich jak wzrost nakładów komunikacyjnych (więcej osób w zespole) czy konieczność poświęcenia czasu przez członków zespołu o dłuższym stażu na przyuczenie nowych członków, co może powodować opóźnienia.

Podczas dokonywania szacowania wysiłku często nie jest dostępna cała wiedza o obiekcie testowym, więc dokładność szacowania może być ograniczona. Ważne jest, aby wyjaśnić interesariuszom, że szacowanie jest oparta na wielu założeniach. Szacunek może być później uszczegółowiony i ewentualnie dostosowany (np. gdy będzie dostępnych więcej danych). Możliwym rozwiązaniem jest użycie przedziału szacowania do przedstawienia wstępnego oszacowania (np. wysiłek wyniesie 10 osobomiesiący z odchyleniem 3 osobomiesiący, co oznacza, że z dużym prawdopodobieństwem rzeczywisty wysiłek wyniesie między 10 – 3 = 7 a 10 + 3 = 13 osobomiesiący).

Szacowanie dla małych zadań jest zwykle bardziej dokładne niż dla dużych. Dlatego przy szacowaniu złożonego zadania można zastosować technikę dekompozycji zwaną Work Breakdown Structure (WBS). W technice tej główne (złożone) zadanie do oszacowania jest hierarchicznie dekomponowane na mniejsze (prostsze) podzadania. Celem jest rozbicie głównego zadania na łatwe do szacowania, ale wykonywalne elementy. Zadanie to powinno być podzielone na tyle dokładnie, na ile jest to możliwe w oparciu o aktualne informacje, ale tylko na tyle, na ile jest to konieczne do zrozumienia i dokładnego oszacowania pojedynczego podzadania. Po dokonaniu szacowania wszystkich podzadań na najniższym poziomie szacowane są zadania nadrzędne (poprzez zsumowanie szacowań odpowiednich podzadań) w sposób oddolny. W ostatnim kroku otrzymuje się szacowanie zadania głównego.

Przykładowe wykorzystanie metody WBS pokazane jest na rysunku 5.2.



RYSUNEK 5.2. Wykorzystanie metody WBS do szacowania wysiłku projektu testowego

Chcemy oszacować wysiłek całego projektu testowego. Jest on jednak na tyle duży, że nie ma sensu estymować od razu całości wysiłku — będzie on bowiem obarczony bardzo dużym błędem. Dzielimy zatem projekt testowy na główne zadania: projektowanie, definiowanie środowiska testowego, testy integracyjne oraz testy systemowe. Założymy, że jesteśmy w stanie w miarę dokładnie oszacować wysiłek dla pierwszych dwóch zadań (odpowiednio: 1 osobodzień oraz 4 osobodni). Testy integracyjne będą jednak wymagać wiele wysiłku, dlatego dokonujemy hierarchicznie podziału tego zadania na mniejsze. Identyfikujemy dwa podzadania w ramach testów integracyjnych: integrację modułów A i B (6 osobodni) oraz integrację modułów A i C (3 osobodni). W ramach testów systemowych identyfikujemy trzy funkcje: F1, F2, F3 jako obiekty testów. Dla F2 i F3 szacujemy wysiłek testowy na, odpowiednio, 8 oraz 4 osobodni. Funkcja F1 jest zbyt duża, więc dzielimy ją na dwa podzadania i szacujemy, odpowiednio, na 3 oraz 5 osobodni. Teraz możemy zebrać wyniki z najwyższych poziomów i szacować zadania na poziomach wyższych poprzez zsumowanie odpowiadających im podzadań. Na przykład wysiłek dla testów integracyjnych (9 osobodni) będzie sumą wysiłku dwóch podzadań o wysiłku, odpowiednio, 6 i 3. Analogicznie wysiłek dla F1 wynosi $3 + 5 = 8$ osobodni, wysiłek dla testów systemowych wynosi $8 + 8 + 4 = 20$ osobodni. Na samym końcu obliczamy wysiłek dla całego projektu testowego jako sumę jego podzadań: $1 + 4 + 9 + 20 = 34$ osobodni.

Techniki szacowania można podzielić na dwie zasadnicze grupy:

- techniki oparte na metrykach — gdzie nakład pracy związany z testowaniem bazuje na metrykach poprzednich podobnych projektów, na danych historycznych z bieżącego projektu lub na wartościach typowych (tzw. wartościach przemysłowych);
- techniki eksperckie — gdzie nakład pracy związany z testami jest oparty na doświadczeniu właścicieli zadań testowych lub ekspertów merytorycznych.

Syllabus opisuje następujące cztery często wykorzystywane w praktyce techniki szacowania.

Szacowanie na podstawie proporcji

W tej technice opartej na metrykach zbiera się jak najwięcej danych historycznych z poprzednich projektów, co pozwala na wyprowadzenie „standardowych” proporcji różnych wskaźników (współczynników) dla podobnych projektów. Własne wskaźniki danej organizacji są zazwyczaj najlepszym źródłem do wykorzystania w procesie szacowania. Te standardowe wskaźniki mogą być następnie użyte do oszacowania wysiłku związanego z testowaniem dla nowego projektu. Na przykład jeżeli w poprzednim, podobnym projekcie stosunek wysiłku związanego z implementacją do wysiłku związanego z testowaniem wynosił 3:2, a w obecnym projekcie nakłady na rozwój mają wynosić 600 osobodni, to nakłady na testowanie mogą być oszacowane jako 400 osobodni, ponieważ najprawdopodobniej wystąpi ten sam lub podobny stosunek tych dwóch wskaźników, co w podobnym projekcie poprzednim.

Ekstrapolacja

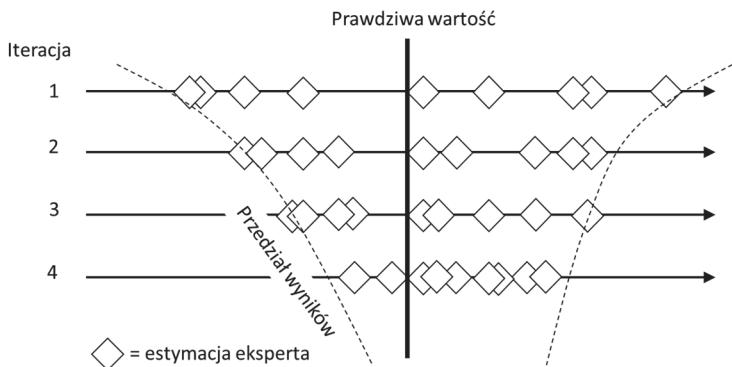
W tej technice opartej na metrykach pomiary są wykonywane jak najwcześniej, aby zebrać prawdziwe, historyczne dane z bieżącego projektu. Mając wystarczająco dużo takich obserwacji (punktów danych), wysiłek wymagany dla pozostałych prac można przybliżyć poprzez ekstrapolację tych danych. Metoda ta jest bardzo przydatna w iteracyjnych technikach wytwarzania oprogramowania. Na przykład zespół może ekstrapolować nakłady na testy w czwartej iteracji jako uśrednione nakłady z ostatnich trzech iteracji. Jeśli wysiłek w ostatnich trzech iteracjach wynosił, odpowiednio, 30, 32 i 25 osobodni, ekstrapolacja wysiłku dla czwartej iteracji wyniesie $(30 + 32 + 25)/3 = 29$ osobodni.

Zauważmy, że postępując w analogiczny sposób, moglibyśmy oszacować wysiłek dla kolejnych iteracji, korzystając z obliczonej ekstrapolacji dla poprzednich iteracji. W naszym przykładzie ekstrapolacja wysiłku dla iteracji piątej będzie średnią z iteracji drugiej, trzeciej i czwartej, a zatem wyniesie $(32 + 25 + 29)/3 = 14,66$. W podobny sposób możemy ekstrapolować wysiłek dla iteracji szóstej: $(25 + 29 + 14,66)/3$ i tak dalej. Należy jednak pamiętać, że im „dalszy” punkt danych ekstrapolujemy, tym większe ryzyko popełnienia coraz większego błędu szacowania, ponieważ już pierwsze szacowanie obarczone jest pewnym błędem. Stosowanie takiego wyniku do kolejnego szacowania może powodować narastanie błędu (zwiększenie dokładności szacowanej wartości).

Szerokopasmowa metoda delficka

W szerokopasmowej metodzie delfickiej (ang. *wideband delphi*) eksperci dokonują szacowania w oparciu o własne doświadczenia. Każdy ekspert, w odosobnieniu, szacuje nakład pracy. Wyniki są zbierane, a eksperci omawiają swoje aktualne szacunki. Następnie każdy ekspert jest proszony o dokonanie nowych przewidywań na bazie tych informacji. Rozmowa pozwala wszystkim ekspertom przemyśleć swój proces szacowania. Może się zdarzyć, że np. niektórzy eksperci nie wzięli pod uwagę niektórych czynników podczas szacowania. Proces ten jest powtarzany aż do osiągnięcia konsensusu lub wystarczająco niewielkiego rozrzutu danych (w takiej sytuacji za wynik końcowy można uznać np. średnią lub medianę z szacunków eksperckich).

Rysunek 5.3 przedstawia ideę stojącą za metodą delficką. Z iteracji na iterację zakres wartości estymowanych przez ekspertów zawęża się. Gdy jest już odpowiednio mały, można wyciągnąć np. średnią lub medianę ze wszystkich szacowań i uznać ją za wynik procesu szacowania. Bardzo często wynik ten nie będzie zbytnio odbiegał od prawdziwej wartości. To zjawisko znane jest potocznie jako „mądrość tłumu”, a wynika z prostego faktu polegającego na tym, że błędy się znoszą: jeden ekspert może niewiele przeszacować, inny — niewiele nie doszacować, jeszcze inny — bardzo nie doszacować, kolejny — bardzo przeszacować. Błąd ma więc zazwyczaj rozkład normalny o średniej w zerze. Rozrzut wyników może być bardzo duży, ale ich uśrednienie będzie często wystarczającym przybliżeniem prawdziwej wartości.



RYSUNEK 5.3. Szerokopasmowa metoda delficka

Odmianą metody delfickiej jest tzw. poker planistyczny (ang. *planning poker*). Jest to podejście powszechnie stosowane w metodach zwinnych. W pokerze planistycznym szacunki są wykonywane przy użyciu ponumerowanych kart. Wartości na kartach reprezentują wielkość wysiłku wyrażonego w określonych, ścisłe zdefiniowanych i zrozumiałych dla wszystkich ekspertów jednostkach.

Karty pokeru planistycznego najczęściej zawierają wartości pochodzące z tzw. ciągu Fibonacciego, przy czym dla większych wartości mogą zachodzić pewne odstępstwa. W ciągu Fibonacciego każdy kolejny wyraz jest sumą dwóch poprzednich. Najczęściej wykorzystywane są następujące wartości:

1, 2, 3, 5, 8, 13, 20, 40, 100.

Jak widać, trzy ostatnie wartości nie są już sumą dwóch poprzednich. Są to bowiem na tyle duże wartości, że przyjmuje się tu zaokrąglone wartości reprezentujące po prostu coś na tyle dużego, że nie ma nawet sensu dokładnie estymować takiej wartości. Jeśli zespół szacuje np. wysiłek potrzebny do implementacji i przetestowania jakiejś historyjki użytkownika i większość ekspertów wyrzuci na stół kartę 40 lub 100, oznacza to, że historyjka jest na tyle duża, że prawdopodobnie jest opowieścią użytkownika i powinna zostać rozbita na kilka mniejszych historyjek, z których każdą będzie już można sensownie estymować.



RYSUNEK 5.4. Karty do pokera planistycznego

Przykładowa talia kart do pokera planistycznego pokazana jest na rysunku 5.4. Wartość „?” oznacza, że szacujący nie posiada wystarczających informacji do tego, aby dokonać szacowania. Wartość „0” oznacza, że szacujący uważa historyjkę za trywialną do implementacji i że czas na to poświęcony będzie zaniedbywalny.

Inne często stosowane w pokerze planistycznym zbiory wartości to zbiór kolejnych potęg dwójki:

1, 2, 4, 8, 16, 32, 64, 128

lub tzw. „rozmiary koszulek”:

XS, S, M, L, XL.

W tym ostatnim przypadku pomiar (szacowanie) nie jest dokonywany na skali liczbowej, ilorazowej, a więc nie reprezentuje wprost fizycznej wielkości estymowanego wysiłku. Rozmiary koszulek są zdefiniowane jedynie na skali porządkowej, która umożliwia wyłącznie porównanie elementów ze sobą. Możemy zatem powiedzieć, że historyjka wyestymowana na „L” jest bardziej pracochłonna od tej wyestymowanej jako „S”, ale nie możemy powiedzieć, ile razy więcej wysiłku zajmie implementacja historyjki o rozmiarze „L” w stosunku do tej o rozmiarze „S”. W tym wariancie metody jesteśmy w stanie jedynie spriorytetyzować historyjki według wysiłku, grupując je w pięć grup o wzrastającej wielkości wysiłku. Nie jest to jednak zalecana praktyka, ponieważ z punktu widzenia teorii pomiarów istotne jest, aby różnice między kolejnymi stopniami skali były stałe. W przypadku gdy skala wyraża punkty historyjek, nie ma problemu: różnica między 4 a 5 punktami jest taka sama jak między 11 a 12 punktami i wynosi 1 punkt. Nie jest to jednak już tak oczywiste w przypadku skali typu „rozmiary koszulek”. Powinno się więc założyć, że np. różnica między S a XS jest taka sama jak między L a M itd.

Dopuszczalne są również inne skale — to, której konkretnie użyjemy, jest wynikiem decyzji zespołu lub kierownictwa. Ważne jest jedynie, aby rozumieć, jak przekłada się jednostka tej skali na jednostkę pracochłonności. Często zespoły jako jednostkę przyjmują tzw. punkt historyjki użytkownika (ang. *user story point*). Wtedy jedna jednostka oznaczać może nakład pracy potrzebny do zaimplementowania/przetestowania jednej, przeciętnej historyjki użytkownika. Jeśli zespół wie na przykład z doświadczenia, że zaimplementowanie jednej historyjki zajmuje mu zwykle 4 osobodni, to element szacowany na 5 punktów historyjek użytkownika powinien zająć ok. $5 \cdot 4 = 20$ osobodni.

Szacowanie trójpunktowe

W tej technice opartej na ekspertach eksperci przeprowadzają trzy rodzaje szacowania: najbardziej optymistyczne (a), najbardziej prawdopodobne (m) i najbardziej pesymistyczne (b). Ostateczny szacunek (E) jest ich ważoną średnią arytmetyczną obliczoną jako

$$E = (a + 4 \cdot m + b)/6.$$

Zaletą tej techniki jest to, że pozwala ona ekspertom obliczyć błąd pomiaru:

$$SD = (b - a)/6.$$

Na przykład jeśli szacunki (w osobogodzinach) wynoszą: $a = 6$, $m = 9$ i $b = 18$, to ostateczny szacunek wynosi 10 ± 2 osobogodziny (tj. między 8 a 12 osobogodzinami), ponieważ $E = (6 + 4 \cdot 9 + 18)/6 = 10$ i $SD = (18 - 6)/6 = 2$. Zauważmy, że wynik szacowania nie zawsze będzie równy wartości najbardziej prawdopodobnej m , ponieważ zależeć będzie od tego, jak bardzo oddalona jest ta wartość od wartości optymistycznej i pesymistycznej. Im wartość m jest bliższa wartości ekstremalnych (optymistycznej a lub pesymistycznej b) tym większa jest różnica między wartością m a estymacją E .

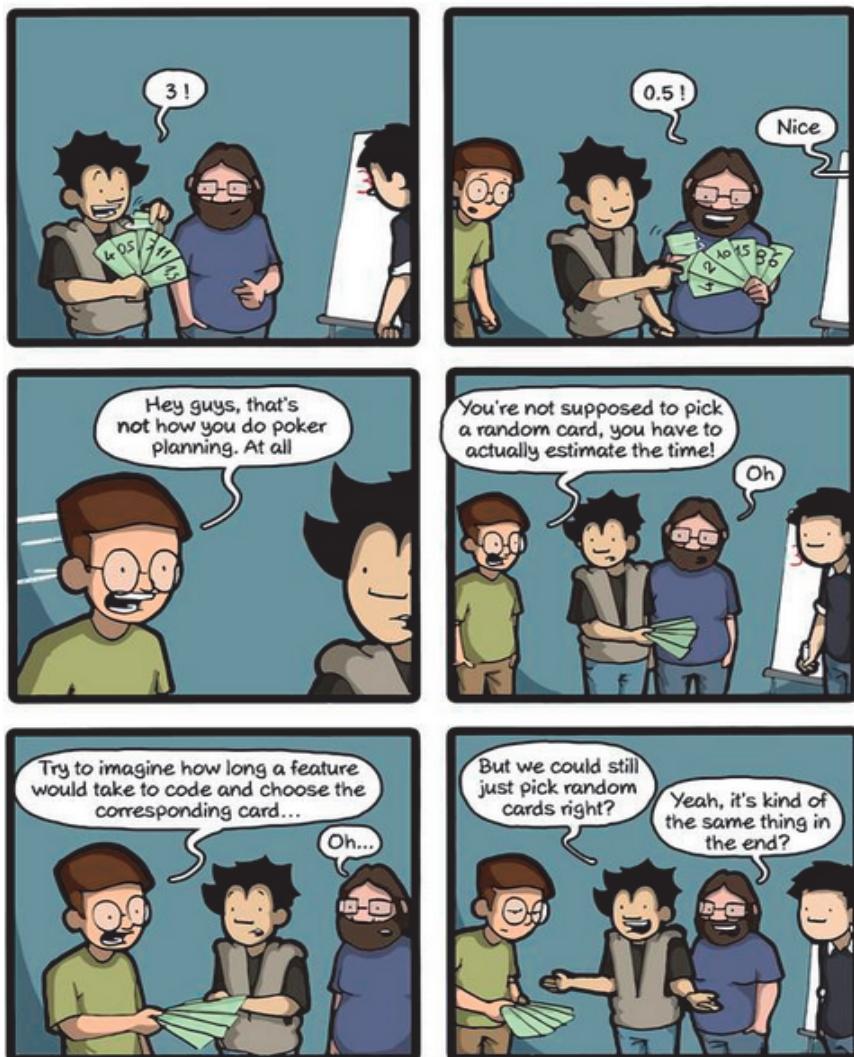
Wzór przedstawiony powyżej jest najczęściej stosowanym w praktyce. Pochodzi z metody PERT (Program Evaluation and Review Technique). Jednak zespoły używają czasem innych wariantów, w odmienny sposób ważąc czynnik m , na przykład z wagą 3 lub 5. Wtedy wzór przyjmuje postać $E = (a + 3 \cdot m + b)/5$ lub $E = (a + 5 \cdot m + b)/7$.

Więcej informacji o tych i o innych technikach szacowania można przeczytać w [Kan 2003, Koomen 2006, Westfall 2009].

Przykład. Zespół stosuje pokeru planistycznego w celu oszacowania wysiłku związanego z implementacją i testowaniem pewnej historyjki użytkownika. Poker planistyczny rozgrywany jest w sesjach według następującej procedury.

1. Każdy z graczy dysponuje pełnym zestawem kart.
2. Moderator (na ogół właściciel produktu) przedstawia historyjkę użytkownika, która ma być szacowana.
3. Następuje krótka dyskusja o zakresie prac, wyjaśnianie niejasności.
4. Każdy z szacujących wybiera jedną kartę.
5. Na znak moderatora wszyscy równocześnie rzucają swoją kartę na środek stołu.
6. Jeżeli wszyscy wybrali tę samą kartę, osiągnięty jest konsensus i spotkanie się kończy — wynikiem jest wartość wybrana przez wszystkich członków zespołu. Jeżeli po ustalonej liczbie iteracji zespół nie osiągnął konsensusu, wynik końcowy określa się według ustalonego, przyjętego wariantu postępowania (patrz niżej).
7. Jeżeli szacunki się różnią, osoba, która wybrała najniższe oszacowanie, i osoba, która wybrała najwyższe, prezentują swój punkt widzenia.
8. Jeśli to konieczne, następuje krótka dyskusja.
9. Powrót do punktu 4.

Poker planistyczny, jak zresztą każda metoda szacowania oparta na ocenie eksperckiej, będzie tak skuteczny, jak dobrymi ekspertami są uczestnicy sesji szacowania (patrz rysunek 5.5).



RYSUNEK 5.5. Žartobliwie o pokerze planistycznym
(<http://www.commitstrip.com/en/2015/01/22/poker-planning>)

Dlatego osoby szacujące powinny być ekspertami z wieloletnim doświadczeniem.

Zespół dysponuje następującymi możliwymi wariantami ustalenia ostatecznego szacowania w przypadku, gdy konsensus nie został osiągnięty po określonej liczbie iteracji pokera. Zakładamy, że zespół dokonujący szacowania pracochłonności pewnej historyjki użytkownika liczy 5 osób, a każdy uczestnik dysponuje pełną talią kart z wartościami 0, 1, 2, 3, 5, 8, 13, 20, 40, 100. Założymy, że w ostatniej turze 5 osób jednocześnie przedstawia wybraną przez siebie kartę. Wyniki to:

3, 8, 13, 13, 20.

Wariant 1. Wynik zostaje uśredniony i ostatecznym szacunkiem jest średnia (11,4) zaokrąglona do najbliższej wartości z talii, czyli 13.

Wariant 2. Wybierana jest najczęściej występująca wartość, czyli 13.

Wariant 3. Wynikiem jest mediana (wartość środkowa) z wartości 3, 8, 13, 13, 20, czyli 13.

Wariant 4. Każda osoba, poza podaniem szacowania, ocenia także stopień swojego przekonania co do tego szacowania, w skali od 1 (duża niepewność) do 5 (duża pewność). Stopień pewności może odzwierciedlać np. poziom doświadczenia i wiedzy eksperckiej w obszarze, w którym dokonujemy szacunków. Założymy, że osoby dokonujące szacowania 3, 8, 13, 13, 20 ocenili pewność swojego wyboru, odpowiednio, na 4, 5, 2, 3, 2. Następnie wynik oblicza się jako średnią ważoną:

$$\frac{3 \cdot 4 + 8 \cdot 5 + 13 \cdot 2 + 13 \cdot 3 + 20 \cdot 2}{4 + 5 + 2 + 3 + 2} = \frac{12 + 40 + 26 + 39 + 40}{16} = \frac{157}{16} = 9,8.$$

Wynik może być następnie zaokrąglony do najbliższej wartości ze skali, czyli 8. Zaletą takiego podejścia jest to, że wyniki osób bardziej przekonanych do poprawności swojego szacowania ważą więcej w uśrednionej ocenie. Oczywiście dokładność metody będzie zależeć w szczególności od trafności oceny pewności indywidualnych oceniających.

Powyższe warianty nie wyczerpują wszystkich możliwych sposobów szacowania. Każdy zespół może przyjąć swoje własne, indywidualne reguły ustalania ostatecznej wartości parametru, który jest poddawany procedurze szacowania. Należy również pamiętać, że wszystkie tego typu warianty wyboru ostatecznej odpowiedzi w przypadku braku konsensusu są zawsze obarczone jakimś błędem. Na przykład wzięcie mediany (13) z wartości 8, 13, 13, 13, 13 będzie obarczone mniejszym błędem niż np. z wartości 1, 8, 13, 40, 100 ze względu na o wiele większą wariancję (rozrzut) wyników w drugim przypadku.

Czynniki wpływające na pracochłonność testowania

Do czynników wpływających na pracochłonność testowania należą:

- charakterystyka produktu (np. ryzyka związane z produktem, jakość specyfikacji (tj. podstawy testów), wielkość produktu, złożoność dziedziny produktu, wymagania dotyczące charakterystyk jakościowych (np. bezpieczeństwa i niezawodności), wymagany poziom szczegółowości dokumentacji testów, wymagania dotyczące zgodności z przepisami);
- charakterystyka procesu tworzenia oprogramowania (stabilność i dojrzałość organizacji, stosowany model tworzenia oprogramowania, podejście do testowania, używane narzędzia, proces testowy, presja czasu);
- czynniki ludzkie (np. umiejętności i doświadczenie testujących, spójność zespołu, umiejętności kierownika);
- rezultaty testów (np. liczba, typ i istotność wykrytych defektów, liczba wymaganych poprawek, liczba testów niezaliczonych).

Informacje dotyczące pierwszych trzech grup czynników są na ogół znane wcześniej, często zanim rozpoczęliśmy projekt. Z kolei rezultaty testów są czynnikiem działającym „od wewnętrz”: nie możemy rezultatów testów uwzględnić w szacowaniu, zanim nie wykonamy testów. Ich wyniki mogą zatem w istotny sposób wpływać na szacowanie czasu pracy w dalszej części projektu.

5.1.5. Ustalanie priorytetów przypadków testowych

Gdy przypadki testowe i procedury testowe są już utworzone i zorganizowane w zestawy testowe, zestawy te mogą być ułożone w harmonogram wykonania testów (ang. *test execution schedule*), który definiuje kolejność, w jakiej mają być one wykonywane. Przy nadawaniu priorytetu przypadkom testowym (a więc kolejności ich wykonywania) powinny być brane pod uwagę różne czynniki.

Harmonogram to inaczej planowanie zadań w czasie. Harmonogram powinien uwzględniać:

- priorytety;
- zależności;
- konieczność wykonania testów potwierdzających i regresji;
- najbardziej efektywną kolejność wykonywania testów.

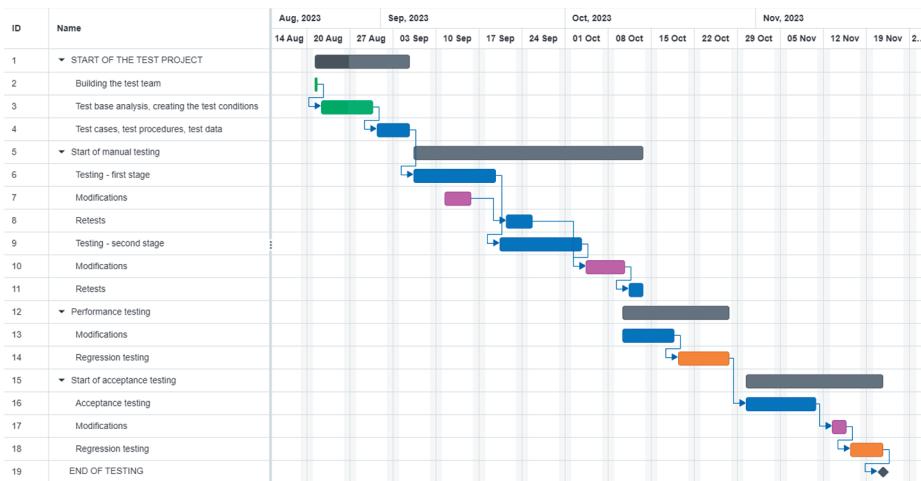
Przy tworzeniu harmonogramu wykonania testów należy brać pod uwagę:

- określenie dat *podstawowych* działań w ramach czasu projektu;
- to, aby harmonogram zawierał się w ramach czasowych harmonogramu projektu;
- *kamienie milowe*: rozpoczęcie i zakończenie etapu.

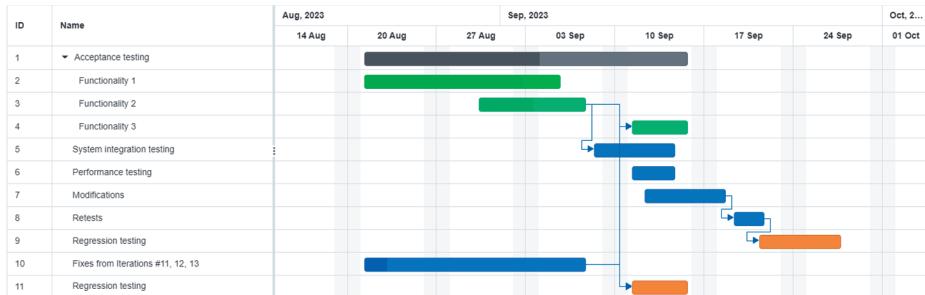
Jednym z najbardziej typowych graficznych sposobów prezentacji harmonogramu jest tzw. wykres Gantta (patrz rysunek 5.6). Na wykresie tym zadania przedstawione są jako prostokąty wyrażające ich czas trwania, a strzałki określają różnego rodzaju relacje między zadaniami. Na przykład jeśli strzałka wychodzi z końca prostokąta X do początku prostokąta Y, oznacza to, że zadanie Y może się rozpocząć dopiero, gdy zadanie X się zakończy.

Dobrze zaplanowane testy powinny mieć tzw. trend liniowy — jeżeli połączy się linią kamienie milowe „start testów” i „koniec testów”, to wykonywane zadania powinny układać się wzdłuż tej linii. Oznacza to, że zadania testowe wykonywane są w taki sposób, by zespół testowy pracował spokojnie, bez niepotrzebnego gonienia czasu.

Praktyka pokazuje, że końcowy okres testów (>95% przypadków) wymaga zaangażowania maksymalnej liczby zasobów w końcowej fazie testów (końcowej fazie projektu) — patrz rysunek 5.7.



RYSUNEK 5.6. Harmonogram w postaci wykresu Gantta



RYSUNEK 5.7. Końcowy okres testów

Najczęściej stosowane strategie priorytetyzacji przypadków testowych są następujące:

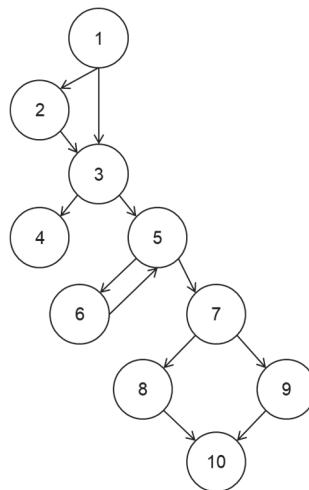
- **Priorytetyzacja oparta na ryzyku**, gdzie kolejność wykonywania testów jest oparta na wynikach analizy ryzyka (patrz punkt 5.2.3). Przypadki testowe obejmujące najważniejsze ryzyka są wykonywane w pierwszej kolejności.
- **Priorytetyzacja oparta na pokryciu**, gdzie kolejność wykonywania testów bazuje na pokryciu (np. kodu, wymagań). Przypadki testowe osiągające najwyższe pokrycie są wykonywane w pierwszej kolejności. W innym podejściu, zwany priorytetyzacją opartą na dodatkowym pokryciu, najpierw wykonywany jest przypadek testowy osiągający najwyższe pokrycie. Każdy kolejny przypadek testowy to ten, który osiąga najwyższe dodatkowe pokrycie.
- **Priorytetyzacja oparta na wymaganiach**, gdzie kolejność wykonywania testów jest oparta na priorytetach wymagań, które są powiązane z odpowiednimi przypadkami testowymi. Priorytety wymagań są określone przez interesariuszy. Przypadki testowe związane z najważniejszymi wymaganiami są wykonywane jako pierwsze.

Idealnie byłoby, gdyby przypadki testowe były zlecanie do wykonania na podstawie ich poziomów priorytetów z użyciem np. jednej z wyżej wymienionych strategii priorytetyzacji. Jednakże ta praktyka może nie działać, jeśli przypadki testowe lub testowane funkcje są zależne. Zachodzą tu następujące reguły:

- jeśli przypadek testowy o wyższym priorytecie jest uzależniony od przypadku o niższym priorytecie, należy najpierw wykonać przypadek o niższym priorytecie;
- jeśli występują wzajemne zależności między kilkoma przypadkami testowymi, kolejność ich wykonywania należy ustalić niezależnie od względnych priorytetów;
- może zajść konieczność priorytetowego wykonania testów potwierdzających i regresji;
- należy odpowiednio wyważyć kwestie efektywności wykonywania testów oraz przestrzegania ustalonych priorytetów.

Kolejność wykonywania testów musi również uwzględniać dostępność zasobów. Na przykład wymagane narzędzia, środowiska lub wymagani ludzie mogą być dostępni tylko w określonym oknie czasowym.

Przykład. Testujemy system, którego graf przepływu sterowania pokazany jest na rysunku 5.8. Przymijmy metodę priorytetyzacji testów opartą na pokryciu. Wykorzystywany przez nas kryterium będzie pokrycie instrukcji. Założymy, że dysponujemy czterema przypadkami testowymi PT1 – PT4 zdefiniowanymi w tabeli 5.1.



RYSUNEK 5.8. Graf przepływu sterowanego systemu

W tabeli tej podane jest również pokrycie, jakie osiągają poszczególne przypadki testowe. Największe pokrycie (70%) osiąga PT4, następnie PT1 i PT2 (po 60%), a najmniejsze PT3 (40%). W związku z tym priorytet wykonania testów będzie następujący: PT4, PT2, PT1, PT3, przy czym PT1 i PT2 można wykonać w dowolnej kolejności, ponieważ osiągają takie samo pokrycie.

TABELA 5.1. Istniejące przypadki testowe oraz osiągane przez nie pokrycie

PRZYPADEK TESTOWY	ŚCIEŻKA WYKONANIA	POKRYCIE	PRIORYTET
PT1	1→3→5→7→8→10	60%	2
PT2	1→3→5→7→9→10	60%	2
PT3	1→2→3→4	40%	3
PT4	1→3→5→6→5→7→8→10	70%	1

Zauważmy jednak, że wykonanie PT1, po wykonaniu PT4 i PT2, nie pokryje żadnych nowych instrukcji. Zastosujmy zatem wariant priorytetyzacji oparty na dodatkowym pokryciu. Najwyższy priorytet będzie miał, tak jak w poprzednim wariantie, przypadek testowy osiągający największe pokrycie, czyli PT4. Zobaczmy teraz, ile dodatkowych, niepokrytych przez PT4 instrukcji pokrywają pozostałe przypadki testowe.

TABELA 5.2. Obliczenie dodatkowego pokrycia osiąganego przez testy

PRZYPADEK TESTOWY	ŚCIEŻKA WYKONANIA	POKRYCIE	DODATKOWE POKRYCIE PO PT4	DODATKOWE POKRYCIE PO PT4 I PT3
PT1	1→3→5→7→8→10	60%	0%	0%
PT2	1→3→5→7→9→10	60%	10% (9)	10% (9)
PT3	1→2→3→4	40%	20% (2, 4)	–
PT4	1→3→5→6→5→7→8→10	70%	–	–

Obliczenia pokazane są w tabeli 5.2 w przedostatniej kolumnie. Przypadek PT1 nie pokrywa żadnych dodatkowych instrukcji, ponieważ pokrywane przez niego instrukcje 1, 3, 5, 7, 8, 10 są już pokryte przez PT4. Przypadek PT2 w stosunku do PT4 pokrywa dodatkowo instrukcję 9, zatem osiąga 10% dodatkowego pokrycia (pokryta jedna nowa instrukcja z 10 wszystkich). Przypadek PT3 w stosunku do PT4 pokrywa dodatkowo dwie instrukcje: 2 i 4, co daje dodatkowe pokrycie 20% (pokryte dwie nowe instrukcje z 10 wszystkich). Zatem kolejnym po PT4 przypadekiem pokrywającym najczęściej nowych, niepokrytych dotąd instrukcji jest PT3. Powtarzamy teraz powyższą analizę, licząc dodatkowe pokrycie w stosunku do pokrycia osiągniętego przez PT4 i PT3. Obliczenia pokazane są w ostatniej kolumnie tabeli 5.2. PT1 nie pokrywa nic nowego. PT2 z kolei pokrywa jedną nową instrukcję, której nie pokryły PT4 i PT3, mianowicie 9. Osiąga więc dodatkowe pokrycie 10%. Tym samym uzyskaliśmy priorytetyzację według dodatkowego pokrycia:

PT4 → PT3 → PT2 → PT1.

Zauważmy, że kolejność ta różni się od kolejności wynikłej z zastosowania poprzedniego wariantu metody. Jej zaletą jest to, że bardzo szybko uzyskujemy pokrycie wszystkich instrukcji (już po pierwszych trzech testach). We wcześniejszym wariantie (z kolejnością PT4→PT2→PT1→PT3) instrukcje 2 i 4 są pokrywane dopiero w ostatnim, czwartym teście.

Kolejny przykład pokazuje, że czasami zależności techniczne lub logiczne między testami mogą zaburzyć pożądaną kolejność wykonywania testów i zmusić nas np. do tego, aby wykonać najpierw test o niższym priorytecie w celu „odblokowania” możliwości wykonania testu o priorytecie wyższym.

Przykład. Założmy, że testujemy funkcjonalności systemu do przeprowadzania elektronicznie egzaminów na prawo jazdy. Do przetestowania zidentyfikowano następujące funkcjonalności, wraz z określeniem ich priorytetów:

1. Dodanie zdającego do bazy danych. Priorytet: niski.
2. Przeprowadzenie egzaminu dla zdającego. Priorytet: wysoki.
3. Przeprowadzenie egzaminu poprawkowego. Priorytet: średni.
4. Podjęcie decyzji i przesłanie wyników egzaminu zdającemu. Priorytet: średni.

Zadaniem o najwyższym priorytecie jest tu zadanie nr 2 (przeprowadzenie egzaminu), ale nie możemy przetestować tej funkcjonalności, zanim nie dodamy zdającego do bazy danych. Zatem mimo że zadanie 2. ma wyższy priorytet niż zadanie 1., występująca zależność logiczna wymaga, abyśmy najpierw wykonali testy dla zadania 1., co „odblokuję” przeprowadzenie testów dla pozostałych zadań. Kolejnym zadaniem do przeprowadzenia będzie zadanie 2., jako zadanie o najwyższym priorytecie. Na samym końcu możemy wykonać zadania 3. i 4. Zauważmy, że aby kandydat mógł zdać egzamin poprawkowy, musi wiedzieć, że nie zdał pierwotnego egzaminu. Dlatego, z perspektywy konkretnego użytkownika, zadanie 3. sensownie jest testować dopiero po wykonaniu zadania 4. Zatem ostateczna kolejność testowania jest taka:

zadanie 1. → zadanie 2. → zadanie 4. → zadanie 3.

W wielu sytuacjach (życiowych) może się zdarzyć, że trzeba będzie określić kolejność wykonania działań, które opisane będą kilkoma czynnikami, np. priorytetem oraz zależnościami logicznymi i technicznymi. Istnieje prosta metoda radzenia sobie z takimi zadaniami, nawet jeśli zależności wydają się skomplikowane. Polega ona na tym, że dążymy do jak najwcześniejszego wykonania zadań o wyższym priorytecie, ale jeśli są blokowane innymi zadaniami, musimy najpierw wykonać te zadania. W ramach tych zadań ich kolejność ustalamy również zgodnie z priorytetami i zależnościami. Rozpatrzmy tę metodę na konkretnym przykładzie.

Przykład. Dana jest lista zadań wraz z ich priorytetami oraz zależnościami:

ZADANIE	PRIORYTET (1 = WYSOKI, 5 = NISKI)	ZALEŻNE OD
1	2	5
2	4	–
3	1	2, 4
4	1	2, 1
5	3	6
6	3	7
7	5	–

Należy ustalić kolejność wykonania zadań. Zaczynamy od ustalenia kolejności zależnej wyłącznie od priorytetów, nie patrząc na zależności, przy czym zadania o równym priorytecie grupujemy:

$$3, 4 \rightarrow 1 \rightarrow 5, 6 \rightarrow 2 \rightarrow 7.$$

W ramach pierwszej grupy zadanie 3. jest zależne od zadania 4. Doprecyzowujemy zatem kolejność zadań 3. i 4.:

$$4 \rightarrow 3 \rightarrow 1 \rightarrow 5, 6 \rightarrow 2 \rightarrow 7.$$

Patrzymy teraz, czy możemy wykonać zadania w tej kolejności. Okazuje się, że nie — zadanie 4. zależy bowiem od zadań 2. i 1., przy czym zadanie 1. ma wyższy priorytet niż zadanie 2. Przenosimy więc oba te zadania przed zadanie 4. z zachowaniem ich priorytetów, a po zadaniu 4. zachowujemy oryginalną kolejność pozostałych zadań:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5, 6 \rightarrow 7.$$

Czy możemy wykonać zadania w tej kolejności? Nadal nie: zadanie 1. jest zależne od zadania 5., to od zadania 6., a to z kolei od zadania 7. Musimy więc zadania 5., 6., 7. przenieść przed zadanie 1. z zachowaniem ich zależności (zauważmy, że tu priorytety zadań 5., 6., 7. nie mają znaczenia — kolejność wymuszona jest przez zależność):

$$7 \rightarrow 6 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3.$$

Zwróćmy uwagę, że w tym momencie możemy już wykonać pełną sekwencję działań: 7, 6, 5, 1, 2, 4, 3, ponieważ każde zadanie w powyższym ciągu zależy wyłącznie od zadań je poprzedzających bądź nie zależy od żadnego innego zadania.

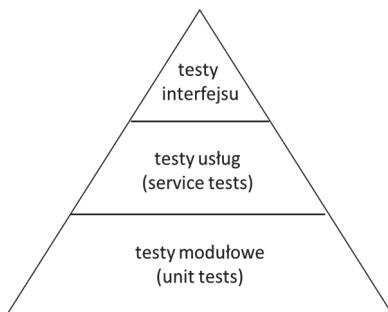
5.1.6. Piramida testów

Piramida testów (ang. *test pyramid*) to model pokazujący, że różne testy mogą mieć różną „ziarnistość”. Model piramidy testów wspiera zespół w automatyzacji testów oraz w alokacji wysiłku testowego. Warstwy piramidy reprezentują grupy testów. Im wyższa warstwa, tym mniejsza ziarnistość testu, izolacja testu i wolniejszy czas jego wykonania.



Testy w dolnej warstwie piramidy są małe, izolowane, szybkie i sprawdzają mały fragment funkcjonalności, więc zwykle potrzeba ich dużo, aby osiągnąć rozsądne pokrycie. Górną warstwą reprezentuje duże, wysokopoziomowe testy typu end-to-end. Te wysokopoziomowe testy są wolniejsze niż testy z niższych warstw i zazwyczaj sprawdzają duży fragment funkcjonalności, więc zwykle wystarczy kilka z nich, aby osiągnąć rozsądne pokrycie.

Liczba i nazewnictwo warstw w piramidzie testów mogą być różne. Na przykład oryginalny model piramidy testów [Cohn 2009] definiuje trzy warstwy: testy modułowe, testy usług i testy interfejsu (patrz rysunek 5.9). Inny popularny model definiuje testy jednostkowe (modułowe), integracyjne i end-to-end.



RYSUNEK 5.9. Piramida testów — przykład

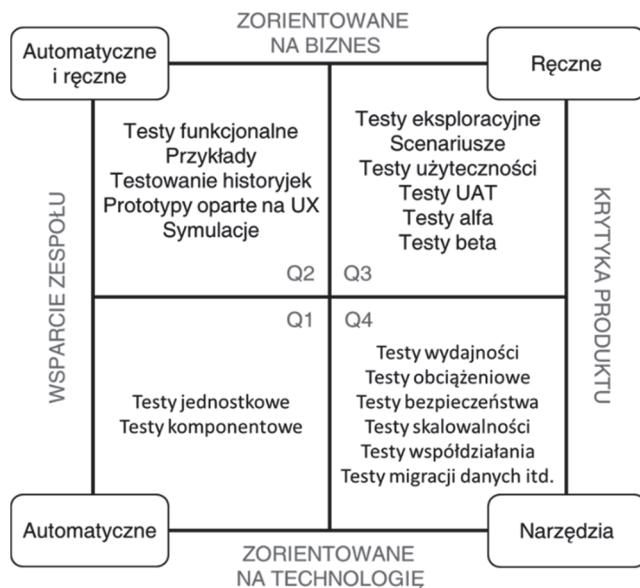
Koszt wykonania testu z niższej warstwy jest zwykle o wiele mniejszy w stosunku do kosztu wykonania testu z warstwy wyższej. Jednak — jak wspomniano wcześniej — pojedynczy test z warstwy niższej zazwyczaj osiąga o wiele mniejsze pokrycie niż pojedynczy test z warstwy wyższej. Dlatego w praktyce, w typowych projektach deweloperskich, wysiłek testowy jest dobrze odzwierciedlony przez model piramidy testów. Zespół zazwyczaj pisze dużo niskopoziomowych testów i mało testów wysokopoziomowych. Należy jednak pamiętać, że liczba testów z różnych poziomów będzie zawsze wynikała z szerokiego kontekstu projektu i produktu. Może się np. zdarzyć tak, że większość testów, jakie zespół będzie wykonywał, to będą testy integracyjne i akceptacyjne (np. w sytuacji integracji dwóch zakupionych od producenta produktów; w tym przypadku nie ma w ogóle sensu przeprowadzać testów modułowych).

5.1.7. Kwadranty testowe

Model **kwadrantów testowych** (ang. *testing quadrants*), zdefiniowany przez Briana Maricka [Marick 2003, Crispin 2008], dopasowuje poziomy testów do odpowiednich typów testów, działań, technik i produktów pracy w metodach zwinnych. Model ten wspiera zarządzanie testami w zapewnieniu, że wszystkie ważne typy testów i poziomy testów są włączone w proces twórco-oprogramowania, oraz w zrozumieniu, że niektóre typy testów są bardziej związane z pewnymi poziomami testów niż inne. Model ten zapewnia również sposób rozróżniania i opisywanie typów testów wszystkim interesariuszom, w tym programistom, testerom i przedstawicielom biznesu. Model kwadrantów testowych pokazany jest na rysunku 5.10.



W kwadrantach testowych testy mogą być nakierowane na potrzeby biznesu (użytkownika) lub technologii (programisty, zespołu twórczego). Niektóre testy walidują zachowanie oprogramowania i wspierają pracę wykonywaną przez zwińny, inne weryfikują (krytykują) produkt. Testy mogą być w pełni manualne, w pełni zautomatyzowane, mogą być kombinacją manualnych i zautomatyzowanych lub być manualne, ale wspierane przez narzędzia.



RYSUNEK 5.10. Kwadranty testowe

Cztery kwadranty przedstawiają się następująco:

Kwadrant Q1

Kwadrant Q1 opisuje poziom testów modułowych, zorientowany na technologię i wspieranie zespołu. Ten kwadrant zawiera testy modułowe. Testy te powinny być zautomatyzowane i włączone w proces ciągłej integracji.

Kwadrant Q2

Kwadrant Q2 opisuje poziom testów systemowych, zorientowany na biznes i wspieranie zespołu. Ten kwadrant zawiera testy funkcjonalne, przykłady (ang. *examples*, zob. punkt 4.5), testy historyjek użytkownika, prototypy i symulacje. Testy te sprawdzają kryteria akceptacji i mogą być manualne lub zautomatyzowane. Często są tworzone podczas tworzenia historyjek użytkownika, dzięki czemu poprawiają ich jakość. Są przydatne podczas tworzenia zautomatyzowanych zestawów testów regresji.

Kwadrant Q3

Kwadrant Q3 opisuje poziom testów akceptacyjnych zorientowany na biznes i zawierający testy krytykujące produkt, wykorzystujące realistyczne scenariusze i dane. Ten kwadrant zawiera testy eksploracyjne, testy oparte na scenariuszach (np. testowanie oparte na przypadkach użycia) bądź przepływach procesów, testy użytkownika, testy akceptacji użytkownika, testy alfa i beta. Testy te są często manualne i zorientowane na użytkownika.

Kwadrant Q4

Kwadrant Q4 opisuje poziom testów akceptacyjnych zorientowany na technologię i zawierający testy krytykujące produkt. Ten kwadrant zawiera większość testów niefunkcjonalnych (z wyjątkiem testów użyteczności). Testy te są często zautomatyzowane.

Podczas każdej iteracji mogą być wymagane testy z każdego lub wszystkich kwadrantów. Kwadranty testowe odnoszą się raczej do testowania dynamicznego niż statycznego.

5.2. Zarządzanie ryzykiem

FL-5.2.1 (K1)	Kandydat określa poziom ryzyka na podstawie prawdopodobieństwa ryzyka i wpływu ryzyka.
FL-5.2.2 (K2)	Kandydat rozróżnia ryzyka projektowe i produktowe.
FL-5.2.3 (K2)	Kandydat wyjaśnia potencjalny wpływ analizy ryzyka produktowego na staranność i zakres testowania.
FL-5.2.4 (K2)	Kandydat wyjaśnia, jakie środki można podjąć w odpowiedzi na przeanalizowane ryzyka produktowe.

Organizacje borykają się z wieloma czynnikami wewnętrznymi i zewnętrznymi, które powodują, że nie ma pewności, czy i kiedy osiągną swoje cele [ISO 31000]. **Zarządzanie ryzykiem** (ang. *risk management*) pozwala organizacjom zwiększyć prawdopodobieństwo osiągnięcia celów, poprawić jakość produktów oraz zwiększyć pewność i zaufanie interesariuszy.



Poziom ryzyka definiuje się zazwyczaj za pomocą prawdopodobieństwa ryzyka i wpływu ryzyka (patrz punkt 5.2.1). Ryzyka, z punktu widzenia testera, można podzielić na dwie zasadnicze grupy: ryzyka projektowe i produktowe (patrz punkt 5.2.2). Do głównych działań związanych z zarządzaniem ryzykiem należą:

- analiza ryzyka (składająca się z identyfikacji ryzyka i oceny ryzyka; patrz punkt 5.2.3);
- kontrola ryzyka (składająca się z łagodzenia ryzyka i monitorowania ryzyka; patrz punkt 5.2.4).

Podejście do testowania, w którym czynności testowe są zarządzane, wybierane i priorytetyzowane na podstawie analizy ryzyka i kontroli ryzyka, jest nazywane **testowaniem opartym na ryzyku** (ang. *risk-based testing*).



Jednym z wielu wyzwań w testowaniu jest właściwy wybór i priorytetyzacja warunków testowych. Ryzyko jest używane do tego, aby właściwie skupić i zaalokować wysiłek wymagany podczas testowania tych warunków testowych. Jest używane do podjęcia decyzji, gdzie i kiedy rozpocząć testowanie oraz do identyfikacji obszarów, które wymagają większej uwagi. Testowanie jest stosowane w celu zmniejszenia

ryzyka, czyli zmniejszenia prawdopodobieństwa wystąpienia niekorzystnego zdarzenia lub zmniejszenia wpływu niekorzystnego zdarzenia. Różne formy testowania są jednym z typowych działań ograniczających ryzyko w projektach rozwoju oprogramowania (patrz punkt 5.2.4). Dostarczają one informacji zwrotnych o zidentyfikowanych i pozostałych (nierożwiązanych) ryzykach. Wczesna analiza ryzyka produktu (patrz punkt 5.2.3) przyczynia się do sukcesu projektu.

Podejście do testowania oparte na ryzyku przyczynia się do zmniejszenia poziomów ryzyka produktu i dostarcza wyczerpujących informacji, które pomagają podjąć decyzję, czy produkt jest gotowy do wydania (jest to jeden z głównych celów testowania, patrz punkt 1.1.1). Zazwyczaj związek pomiędzy ryzykiem produktu a testami jest kontrolowany za pomocą mechanizmu obustronnego śledzenia.

Główne korzyści z testowania opartego na ryzyku to:

- zwiększenie prawdopodobieństwa odkrycia defektów w kolejności ich ważności poprzez wykonywanie testów w kolejności priorytetyzowanych ryzyk;
- minimalizacja ryzyka rezydualnego (tzn. pozostałoego) produktu po wydaniu poprzez alokację wysiłku testowego w zależności od ryzyka;
- raportowanie ryzyka rezydualnego poprzez mierzenie wyników testów w kategoriach poziomów powiązanych ryzyk;
- przeciwdziałanie skutkom presji czasu na testowanie i umożliwienie skrócenia okresu testowania przy najmniejszym możliwym wzrostie ryzyka produktu związanego ze skróceniem czasu przeznaczonego na testy.

Aby zapewnić minimalizację prawdopodobieństwa wystąpienia awarii produktu, działania testowe oparte na ryzyku zapewniają zdyscyplinowane podejście do:

- przeanalizowania (i regularnej ponownej oceny) tego, co może pójść źle (ryzyka);
- określenia, które ryzyka są na tyle ważne, aby się nimi zająć;
- wdrożenia działań mających na celu złagodzenie tych ryzyk;
- zaprojektowania planów awaryjnych, aby poradzić sobie z ryzykiem, jeśli stanie się ono rzeczywiste.

Ponadto testowanie może zidentyfikować nowe ryzyka, pomóc w określaniu, które z nich powinny być ograniczone, oraz obniżyć niepewność dotyczącą ryzyka.

5.2.1. Definicja i atrybuty ryzyka

Zgodnie ze słownikiem terminów testowych ISTQB [ISTQB S] **ryzyko** (ang. *risk*) to czynnik, który w przyszłości może skutkować negatywnymi konsekwencjami; zazwyczaj opisywany jest poprzez *wpływ* oraz *prawdopodobieństwo*. Dlatego **poziom ryzyka** (ang. *risk level*) określają prawdopodobieństwo niekorzystnego zdarzenia oraz wpływ — czyli konsekwencje (szkoda będąca skutkiem tego zdarzenia). Często definiuje się to za pomocą równania:

poziom ryzyka = prawdopodobieństwo ryzyka · wpływ ryzyka.



Równanie to można rozumieć symbolicznie (tak jak to opisano powyżej) lub całkiem dosłownie. Z tym ostatnim przypadkiem mamy do czynienia w tzw. ilościowym podejściu do ryzyka, gdzie prawdopodobieństwo i wpływ wyrażają się liczbowo — prawdopodobieństwo jako liczbę z przedziału (0, 1), natomiast wpływ — w pieniądzu. Wpływ reprezentuje wielkość straty, jaką poniesiemy, gdy ryzyko się zmaterializuje.

Przykład. Organizacja definiuje poziom ryzyka jako iloczyn prawdopodobieństwa jego wystąpienia oraz wpływu. Zidentyfikowano ryzyko polegające na zbyt wolnym generowaniu kluczowego raportu w sytuacji dużej liczby jednocześnie zalogowanych użytkowników.

Ponieważ wymaganie dotyczące wysokiej wydajności jest w tym przypadku krytyczne, wpływ tego ryzyka oceniony został na bardzo duży. Biorąc pod uwagę liczbę użytkowników, którzy mogliby doświadczyć opóźnień w generowaniu raportu, oraz konsekwencje tych opóźnień, wpływ ryzyka został oszacowany na 800 000 PLN.

Prawdopodobieństwo niskiej wydajności programu oszacowano z kolei na bardzo niskie ze względu na dosyć dobrze zdefiniowany proces testowy, zaplanowane przeglądy architektury oraz duże doświadczenie zespołu z testami wydajnościowymi. Prawdopodobieństwo wystąpienia ryzyka oceniono na 5%.

Ostatecznie poziom ryzyka zgodnie z powyższym równaniem został ustalony na:

$$5\% \cdot 800\,000 \text{ PLN} = (5/100) \cdot 800\,000 \text{ PLN} = 40\,000 \text{ PLN}.$$

5.2.2. Ryzyka projektowe i produktowe

W testowaniu oprogramowania mamy do czynienia z dwoma rodzajami ryzyka: ryzykami projektowymi i ryzykami produktowymi.

Ryzyka projektowe

Ryzyka projektowe (ang. *project risk*) związane są z zarządzaniem i kontrolą projektu. Ryzyka projektowe to ryzyka wpływające na sukces projektu (zdolności projektu do osiągnięcia celów). Ryzyka projektowe obejmują:



- kwestie organizacyjne (np. opóźnienia w dostarczaniu produktów pracy, niedokładne szacunki, cięcie kosztów, złe wdrożone, zarządzane i utrzymywane procesy, w tym proces inżynierii wymagań czy proces zapewniania jakości);
- kwestie ludzkie (np. niewystarczające umiejętności, konflikty, problemy z komunikacją, braki kadrowe, brak dostępnych ekspertów merytorycznych);
- kwestie techniczne (np. rozszerzanie zakresu, słabe wsparcie narzędzi, niedokładne szacowanie, zmiany dokonywane w ostatniej chwili, niedostateczne doprecyzowanie wymagań, brak możliwości spełnienia wymagań z uwagi na ograniczenia czasowe, nieudostępnienie na czas środowiska testowego, zbyt późne zaplanowanie konwersji danych, migracji lub udostępnienia potrzebnych do tego narzędzi, wady w procesie wytwórczym, niska jakość produktów prac projektowych, w tym specyfikacji wymagań czy przypadków testowych, kumulacja defektów lub innego rodzaju dług techniczny);

- problemy z dostawcami (np. niepowodzenie w dostawie od strony trzeciej, bankructwo firmy wspierającej, opóźnienie w dostawie, problemy związane z umowami).

Ryzyka projektowe, gdy wystąpią, mogą mieć wpływ na harmonogram, budżet lub zakres projektu, co wpływa na zdolność projektu do osiągnięcia jego celów. Najczęstszą bezpośrednią konsekwencją wystąpienia ryzyka projektowego jest opóźnienie projektu, co pociąga za sobą dalsze problemy, takie jak np. wzrost kosztów spowodowany koniecznością przeznaczenia większej ilości czasu na pewne czynności czy też koniecznością zapłaty kar umownych za opóźnioną dostawę produktu.

Ryzyka produktowe

Ryzyka produktowe (ang. *product risk*) są związane z cechami jakościowymi produktu, takimi jak np. funkcjonalność, niezawodność, wydajność, użyteczność i inne charakterystyki opisywane np. modelem jakości ISO/IEC 25010. Ryzyko produktowe występują wszędzie tam, gdzie produkt pracy (np. specyfikacja, moduł, system lub test) może nie zaspokoić uzasadnionych potrzeb użytkowników i/lub interesariuszy. Ryzyko produktowe określa się poprzez obszary możliwych awarii w testowanym produkcie, gdyż zagrażają one w różny sposób jakości produktu. Przykłady ryzyk produktowych obejmują:



- brakującą lub niewłaściwą funkcjonalność;
- nieprawidłowe obliczenia;
- awarie podczas działania oprogramowania (np. zawieszenie lub wyłączenie się aplikacji);
- złą architekturę;
- nieefektywne algorytmy;
- nieodpowiedni czas odpowiedzi;
- złe doświadczenia użytkownika;
- luki w zabezpieczeniach.

Ryzyko produktowe, gdy wystąpi, może spowodować różne negatywne konsekwencje, w tym:

- niezadowolenie użytkowników końcowych;
- utratę przychodów;
- szkody wyrządzone osobom trzecim;
- wysokie koszty utrzymania;
- przeciążenie działu pomocy technicznej;
- utratę wizerunku;
- utratę zaufania;
- sankcje karne.

W skrajnych przypadkach wystąpienie ryzyka produktowego może spowodować szkody fizyczne, obrażenia, a nawet śmierć.

5.2.3. Analiza ryzyka produktowego

Celem **analizy ryzyka** (ang. *risk analysis*) jest zapewnienie świadomości ryzyka, aby skoncentrować wysiłek testowy w taki sposób, by zminimalizować poziom ryzyka rezydualnego produktu. W idealnym przypadku analiza ryzyka produktowego rozpoczyna się we wczesnym etapie cyklu wytwarzania. Analiza ryzyka składa się z dwóch zasadniczych faz: identyfikacji ryzyka i oceny ryzyka. Uzyskane w fazie analizy ryzyka informacje na temat ryzyka produktowego można wykorzystać do:



- planowania testów;
- specyfikowania, przygotowywania i wykonywania przypadków testowych;
- monitorowania i nadzorowania testów.

Wczesna analiza ryzyka produktowego przyczynia się do powodzenia całego projektu, bowiem analiza ryzyka produktowego umożliwia:

- określenie konkretnych poziomów testów i typów testów, które należy wykonać;
- określenie zakresu wykonywanych testów;
- ustalenie priorytetów testowania (aby jak najwcześniej wykryć defekty krytyczne);
- określenie technik testowania, które w najbardziej efektywny sposób umożliwiają osiągnięcie zadanego pokrycia i wykrycie defektów związanych ze zidentyfikowanymi ryzykami;
- oszacowanie dla każdego zadania nakładu pracy włożonego w testowanie;
- określenie, czy oprócz testowania można zastosować inne działania w celu zmniejszenia (zlagodzenia) ryzyka;
- ustalenie innych czynności niezwiązanych z testowaniem.

Identyfikacja ryzyka



Identyfikacja ryzyka (ang. *risk identification*) polega na wygenerowaniu wyczerpującej listy ryzyk. Interesariusze mogą identyfikować ryzyka za pomocą różnych technik i narzędzi, takich jak:

- burza mózgów (w celu zwiększenia kreatywności w poszukiwaniu ryzyk);
- warsztaty dotyczące ryzyka (w celu wspólnego przeprowadzenia identyfikacji ryzyka przez różnych interesariuszy);
- metoda delficka lub ocena ekspertów (w celu uzyskania zgody lub braku zgody ekspertów na temat ryzyka);
- wywiady (aby zapytać interesariuszy o ryzyko i zebrać ich opinie);
- listy kontrolne (w celu odniesienia się do typowych, znanych, powszechnie występujących ryzyk);
- bazy danych o poprzednich projektach, retrospektyny i wnioski (w celu odniesienia się do doświadczeń z przeszłości);

- diagramy przyczynowo-skutkowe (aby odkryć ryzyka poprzez wykonanie analizy przyczyny podstawowej);
- szablony ryzyka (aby określić, kto może zostać poszkodowany przez ryzyko i w jaki sposób).

Ocena ryzyka

Ocena ryzyka (ang. *risk assessment*) obejmuje kategoryzację zidentyfikowanych ryzyk, określenie ich prawdopodobieństwa, wpływu i poziomu, ustalenie priorytetów i zaproponowanie sposobów postępowania z nimi. Kategoryzacja pomaga w przypisaniu działań łagodzących, ponieważ zazwyczaj ryzyka należące do tej samej kategorii mogą być łagodzone przy użyciu podobnego podejścia.



Ocena ryzyka może wykorzystywać podejście ilościowe lub jakościowe, lub ich mieszankę. W podejściu ilościowym poziom ryzyka jest obliczany jako iloczyn prawdopodobieństwa i wpływu. W podejściu jakościowym poziom ryzyka może być obliczany przy użyciu macierzy ryzyka (ang. *risk matrix*).

Przykład macierzy ryzyka pokazany jest na rysunku 5.11. Macierz ryzyka jest swoistą „tabliczką mnożenia”, w której definiuje się poziom ryzyka jako iloczyn określonych kategorii prawdopodobieństwa i wpływu.

POZIOM RYZYKA		PRAWDOPODOBIĘSTWO			
		niskie	średnie	wysokie	bardzo wysokie
WPŁYW	mały	bardzo niski	niski	średni	średni
	średni	niski	średni	wysoki	wysoki
	duży	średni	wysoki	bardzo wysoki	katastrofalny

RYSUNEK 5.11. Macierz ryzyka

W macierzy ryzyka z rysunku 5.11 mamy zdefiniowane cztery kategorie prawdopodobieństwa: niskie, średnie, wysokie i bardzo wysokie oraz trzy kategorie wpływu: mały, średni i duży. Na przecięciu określonych kategorii prawdopodobieństwa i wpływu zdefiniowany został poziom ryzyka. W tej macierzy ryzyka zdefiniowanych jest sześć kategorii poziomu ryzyka: bardzo niski, niski, średni, wysoki, bardzo wysoki i katastrofalny. Na przykład ryzyko o średnim prawdopodobieństwie i dużym wpływie ma wysoki poziom ryzyka, ponieważ na przecięciu kolumny „prawdopodobieństwo średnie” i wiersza „wpływ duży” zdefiniowano poziom ryzyka jako „wysoki”.

Istnieją inne metody oceny ryzyka, w których — tak jak np. w metodzie FMEA (ang. *Failure Mode and Effect Analysis*) prawdopodobieństwa i wpływ określa się na skali liczbowej (np. od 1 do 5, gdzie 1 oznacza najniższą, a 5 — najwyższą kategorię), a poziom ryzyka oblicza się poprzez wymnożenie tych dwóch liczb. Na przykład dla prawdopodobieństwa 2 i wpływu 4 poziom ryzyka definiuje się jako $2 \cdot 4 = 8$.

W takich przypadkach należy jednak uważać. Mnożenie jest przemienne, zatem poziom ryzyka o prawdopodobieństwie 1 i wpływie 5 wynosi 5 i jest taki sam jak dla ryzyka o prawdopodobieństwie 5 i wpływie 1, ponieważ $1 \cdot 5 = 5 \cdot 1$. W praktyce jednak wpływ jest dla nas o wiele istotniejszym czynnikiem. Ryzyko o prawdopodobieństwie 5 i wpływie 1 być może będzie występowało bardzo często, ale nie spowoduje praktycznie żadnych kłopotów ze względu na znikomy wpływ. Natomiast ryzyko o prawdopodobieństwie 1 i wpływie 5 co prawda ma bardzo małą szansę wystąpienia, ale gdy już wystąpi, jego konsekwencje mogą być katastrofalne.

Poza ilościowym i jakościowym podejściem do oceny ryzyka istnieje jeszcze podejście mieszane. W takim podejściu prawdopodobieństwo i wpływ nie są co prawda definiowane przez konkretne wartości liczbowe, ale są reprezentowane za pomocą przedziałów wartości, które wyrażają pewną niepewność co do szacowania tych parametrów. Na przykład prawdopodobieństwo może być zdefiniowane jako przedział $[0,2, 0,4]$, a wpływ — jako przedział $[1000 \text{ PLN}, 3000 \text{ PLN}]$. Oznacza to, że spodziewamy się, iż prawdopodobieństwo tego ryzyka mieści się gdzieś między 20% a 40%, natomiast jego wpływ — między 1000 a 3000 PLN. Poziom ryzyka w tym podejściu można określić również jako przedział, według następującego wzoru na mnożenie:

$$[a, b] \cdot [c, d] = [a \cdot c, b \cdot d],$$

czyli w naszym przypadku poziom ryzyka wyniesie

$$[0,2, 0,4] \cdot [1000, 3000] = [200, 1200].$$

Poziom ryzyka mieści się zatem w przedziale między 200 a 1200 PLN.

Podejście mieszane jest dobrym rozwiązaniem, jeśli trudno jest nam szacować dokładne wartości prawdopodobieństw i wpływów poszczególnych ryzyk (podejście ilościowe), a jednocześnie chcemy mieć dokładniejsze wyniki oceny ryzyka niż te wyrażone na skali porządkowej (podejście jakościowe).

Przykład. Organizacja definiuje poziom ryzyka jako iloczyn prawdopodobieństwa jego wystąpienia oraz wpływu. Zdefiniowano następujące ryzyka wraz z szacowaniem prawdopodobieństwa i wpływu każdego z nich:

Ryzyko 1. Prawdopodobieństwo 20%, wpływ 40 000 PLN.

Ryzyko 2. Prawdopodobieństwo 10%, wpływ 100 000 PLN.

Ryzyko 3. Prawdopodobieństwo 5%, wpływ 20 000 PLN.

Aby obliczyć całkowity poziom ryzyka, należy zsumować odpowiednie iloczyny:

Całkowity poziom ryzyka = $0,2 \cdot 40\ 000 + 0,1 \cdot 100\ 000 + 0,05 \cdot 20\ 000 = 8\ 000 + 10\ 000 + 1\ 000 = 19\ 000$. Oznacza to, że jeśli nie podejmujemy żadnych działań minimalizujących (fagodzących) te ryzyka, to średnia (oczekiwana) strata, jaką poniesiemy w wyniku ich potencjalnego wystąpienia, wyniesie ok. 19 000 PLN.

Zauważmy, że analiza ilościowa w przypadku pojedynczego ryzyka nie ma zbyt wielkiego sensu, ponieważ ryzyka — z punktu widzenia testera, czyli zewnętrznego obserwatora — są zjawiskami o charakterze losowym. Nie wiemy, kiedy i czy w ogóle

wystąpią, a jeśli tak, to jaką naprawdę szkodę wyrządzają. Analiza pojedynczego zjawiska losowego nie ma sensu. Jednak analiza przeprowadzona dla wielu ryzyk, tak jak w powyższym przykładzie, ma już sens. Suma poziomów ryzyk dla wszystkich zidentyfikowanych ryzyk może bowiem być traktowana jako wartość oczekiwana straty, jaką poniesiemy w związku z wystąpieniem niektórych z tych ryzyk.

5.2.4. Kontrola ryzyka produktowego

Kontrola ryzyka produktowego (ang. *risk control*) odgrywa kluczową rolę w zarządzaniu ryzykiem, ponieważ obejmuje wszystkie środki, które są podejmowane w odpowiedzi na zidentyfikowane i ocenione ryzyko produktowe. Kontrola ryzyka jest definiowana jako środki podejmowane w celu ograniczenia ryzyka i monitorowania tego ryzyka w całym cyklu wytwarzania.



Kontrola ryzyka jest procesem, w skład którego wchodzą dwa główne zadania: łagodzenie ryzyka oraz monitorowanie ryzyka.

Łagodzenie ryzyka

Po przeanalizowaniu ryzyka istnieje kilka opcji **łagodzenia ryzyka** (ang. *risk mitigation*), czyli możliwych reakcji na ryzyko [Veenendaal 2012]:



- akceptacja ryzyka (ang. *risk acceptance*);
- przeniesienie ryzyka (ang. *risk transfer*);
- plan awaryjny (ang. *contingency plan*);
- łagodzenie ryzyka przez testowanie (ang. *risk mitigation by testing*).

Akceptacja (ignorowanie) ryzyka może być dobrym pomysłem w sytuacji ryzyk o niskim poziomie. Takie ryzyka zazwyczaj priorytetyzuje się najniżej i zajmuje się nimi na samym końcu. Często może nam zwyczajnie zabraknąć czasu na zajęcie się nimi, ale to nam niespecjalnie przeszkadza. Nawet bowiem gdy takie ryzyko wystąpi, szkody, jakie przyniesie, są niewielkie lub wręcz zaniedbywalne. Nie ma więc sensu tracić czasu na zajmowanie się nimi w sytuacji, gdy mamy inne, o wiele ważniejsze i poważniejsze kwestie do rozważenia (np. ryzyka o znacznie większym poziomie).

Przykładem przeniesienia ryzyka może być np. wykupienie polisy ubezpieczeniowej. W zamian za opłacanie składek przenosi się ryzyko poniesienia kosztów wystąpienia ryzyka na stronę trzecią, w tym przypadku — na ubezpieczyciela.

Plany awaryjne opracowuje się na wypadek wystąpienia określonych typów ryzyk. Plany te sporządza się po to, aby w momencie wystąpienia niepożądanej sytuacji uniknąć chaosu oraz opóźnienia w reakcji na ryzyko. Dzięki istnieniu takich planów w momencie wystąpienia zagrożenia każdy dokładnie wie, co ma robić. Reakcja na ryzyko jest więc szybka i zawczasu przemyślana, a więc — skuteczna.

W sylabusie spośród powyższych metod reakcji na ryzyko jedynie łagodzenie ryzyka przez testowanie jest przedstawione szczegółowo, ponieważ sylabus dotyczy

testowania. Testy mogą być opracowane w celu sprawdzenia, czy ryzyko występuje w rzeczywistości. Jeśli test wykaże defekt, zespół jest świadomy problemu, co odpowiednio zmniejsza ryzyko, które jest już zidentyfikowane i nie jest nieznane. Jeśli test przejdzie pomyślnie, zespół zyskuje większe zaufanie do jakości systemu. Łagodzenie ryzyka może wpływać na zmniejszenie prawdopodobieństwa wystąpienia ryzyka lub zmniejszenie jego wpływu. Generalnie testowanie przyczynia się do obniżania całkowitego poziomu ryzyka w systemie. Każdy zaliczony test może być interpretowany jako sytuacja, w której pokazaliśmy, że określone ryzyko (z którym związany jest dany test) nie występuje, ponieważ test jest zaliczony. To obniża całkowite ryzyko rezydualne o wartość poziomu tego konkretnego ryzyka.

Działania, które mogą być podjęte przez testerów w celu złagodzenia ryzyka produktowego, są następujące:

- wybór testerów o odpowiednim poziomie doświadczenia, właściwym dla danego typu ryzyka;
- stosowanie odpowiedniego poziomu niezależności testowania;
- przeprowadzenie przeglądów;
- wykonanie analizy statycznej;
- wybór odpowiednich technik projektowania testów i poziomu pokrycia;
- priorytetyzacja testów w oparciu o poziom ryzyka;
- określenie właściwego zakresu testów regresji.

Niektóre działania łagodzące dotyczą wczesnego testowania prewencyjnego poprzez zastosowanie ich przed rozpoczęciem testowania dynamicznego (np. przeglądy). W testowaniu opartym na ryzyku działania łagodzące ryzyko powinny występować w całym cyklu wytwarzania.

Paradygmat zinnego tworzenia oprogramowania wymaga samoorganizacji (patrz punkt 1.4.2 dotyczący ról w testowaniu oraz punkt 1.5.2 dotyczący podejścia „cały zespół”), a jednocześnie dostarcza praktyk zmniejszających ryzyko, które mogą być postrzegane jako całościowy system ograniczania ryzyka. Jedną z jego zalet jest zdolność do rozpoznawania ryzyka i dostarczania dobrych praktyk łagodzących je. Przykładami ryzyka redukowanego przez te praktyki są [Schwaber 2002]:

- ryzyko polegające na niezadowoleniu klienta — w podejściu zinnym klient lub jego przedstawiciel widzi produkt na bieżąco, co przy dobrej realizacji projektu, częstej informacji zwrotnej i intensywnej komunikacji łagodzi to ryzyko;
- ryzyko nieukończenia wszystkich funkcjonalności — częste wydawanie i planowanie produktów oraz priorytetyzacja zmniejszają to ryzyko poprzez zapewnienie, że przyrosty związane z wysokimi priorytetami biznesowymi są dostarczane w pierwszej kolejności;
- ryzyko niewłaściwego oszacowania i planowania — aktualizacje produktów pracy są śledzone codziennie, aby zapewnić kontrolę zarządzania i częste możliwości korekty;

- ryzyko braku szybkiego rozwiązywania problemów — samoorganizujący się zespół, dwukierunkowe zarządzanie i codzienne raporty z pracy dają możliwość codziennego zgłaszania i rozwiązywania problemów;
- ryzyko nieukończenia cyklu rozwojowego — w danym cyklu (im krótszy, tym lepszy) podejście zwinne dostarcza działające oprogramowanie (a dokładniej — jego określony fragment), aby nie było większych problemów z rozwojem;
- ryzyko podjęcia zbyt dużej ilości pracy i zmiany oczekiwanych — zarządzanie backlogiem produktu, planowanie iteracji i działających wydań zapobiegają ryzyku niezauważonych zmian, które negatywnie wpływają na jakość produktu, ponieważ zmusza to zespoły do wczesnej konfrontacji i rozwiązywania problemów.

Monitorowanie ryzyka

Monitorowanie ryzyka (ang. *risk monitoring*) to zadanie z zakresu zarządzania ryzykiem, które dotyczy działań związanych ze sprawdzaniem statusu ryzyka produktowego. Monitorowanie ryzyka pozwala testerom mieć wdrożone działania ograniczające ryzyko (działania łagodzące) w celu zapewnienia, że osiągają one zamierzone efekty, oraz identyfikować zdania lub okoliczności stwarzające nowe lub zwiększone ryzyko. Monitorowanie ryzyka odbywa się zwykle poprzez raporty, które porównują stan rzeczywisty z tym, co było oczekiwane. W idealnym przypadku monitorowanie ryzyka odbywa się przez cały cykl wytwarzania oprogramowania.



Typowe działania związane z monitorowaniem ryzyka polegają na rewidowaniu ryzyk produktowych i sporządzaniu raportów na ich temat. Istnieje kilka korzyści z monitorowania ryzyka, np.:

- znajomość dokładnego i aktualnego statusu każdego ryzyka produktu;
- możliwość raportowania postępów w zakresie redukcji rezydualnego ryzyka produktowego;
- skupienie się na pozytywnych rezultatach ograniczania ryzyka;
- odkrywanie ryzyk, które nie zostały wcześniej zidentyfikowane;
- wychwytywanie nowych ryzyk w miarę ich pojawiania się;
- śledzenie czynników wpływających na koszty zarządzania ryzykiem.

Niektóre metody mają wbudowane metody monitorowania ryzyka. Na przykład we wspomnianej wcześniej metodzie FMEA szacowanie prawdopodobieństwa i wpływu ryzyka odbywa się dwukrotnie — najpierw w fazie oceny ryzyka, a następnie ponownie, po wdrożeniu czynności łagodzących to ryzyko.

5.3. Monitorowanie testów, nadzór nad testami i ukończenie testów

- | | |
|---------------|---|
| FL-5.3.1 (K1) | Kandydat pamięta metryki stosowane w odniesieniu do testowania. |
| FL-5.3.2 (K2) | Kandydat podsumowuje cele i treść raportów z testów oraz wskazuje ich odbiorców. |
| FL-5.3.3 (K2) | Kandydat omawia na przykładach sposób przekazywania informacji o statusie testowania. |

Podstawowe cele **monitorowania testów** (ang. *test monitoring*) to gromadzenie i udostępnianie informacji pozwalających uzyskać wgląd w przebieg czynności testowych oraz unaocznić przebieg testów. Monitorowaną informację można zbierać ręcznie lub automatycznie, wykorzystując narzędzia do zarządzania testami. Sugerujemy raczej podejście automatyczne, wykorzystujące dziennik (log) testów i informacje z narzędzia do zgłaszania defektów, wspomagane podejściem ręcznym (bezpośrednia kontrola pracy testera — umożliwia to bardziej obiektywną ocenę aktualnego stanu testów). Nie należy jednak zapominać o ręcznym zbieraniu informacji, np. podczas codziennych spotkań w Scrumie (patrz punkt 2.1.1).



Uzyskane wyniki stosuje się do:

- pomiaru spełnienia kryteriów wyjścia, np. osiągnięcia zakładanego pokrycia ryzyka produktowego, wymagań, kryteriów akceptacji;
- oceny postępu prac w porównaniu z harmonogramem i budżetem.

Działania wykonywane w ramach **nadzoru nad testami** (ang. *test control*) to głównie:



- podejmowanie decyzji na podstawie informacji uzyskanych w wyniku monitorowania testów;
- ponowne ustalanie priorytetów testów w przypadku zmaterializowania się zidentyfikowanego ryzyka (np. nieterminowego dostarczenia oprogramowania);
- wprowadzanie zmian w harmonogramie wykonania testów;
- ocena dostępności lub niedostępności środowiska testowego lub innych zasobów.

5.3.1. Metryki stosowane w testowaniu

Podczas wykonywania danego poziomu testów i po jego zakończeniu można (należy) zbierać metryki pozwalające oszacować:

- postęp realizacji harmonogramu i budżetu;
- bieżącą jakość przedmiotu testów;
- adekwatność wybranego podejścia do testowania;
- skuteczność czynności testowych z punktu widzenia realizacji celów.

Monitorowanie testów gromadzi różne metryki (ang. *metric*), aby — raportując je — wspomóc działania nadzoru nad testami.

Typowe metryki testowe obejmują:

- **metryki projektowe** (np. ukończenie zadania, wykorzystanie zasobów, wy- siłek testowy, procent wykonania zaplanowanych prac związanych z przygotowaniem środowiska testowego, daty kamieni milowych);
- **metryki przypadków testowych** (np. postęp w implementacji przypadków testowych, postęp w przygotowaniu środowiska testowego, liczba uruchomionych/nieuruchomionych przypadków testowych, zaliczonych/niezaliczonych, czas wykonania testu);
- **metryki jakości produktu** (np. dostępność, czas odpowiedzi, średni czas do awarii — ang. *Mean Time To Failure*, MTTF);
- **metryki defektów** (np. liczba znalezionych/naprawionych defektów, gęstość defektów (liczba defektów na jednostkę objętości, np. kodu), częstość defektów (liczba defektów na jednostkę czasu), procent wykrycia defektów, procent udanych testów potwierdzających);
- **metryki ryzyka** (np. poziom ryzyka rezydualnego, priorytet ryzyka);
- **metryki pokrycia** (np. pokrycie wymagań, historyjek użytkownika, kryteriów akceptacji, warunków testowych, kodu, ryzyk);
- **metryki kosztów** (np. koszt testowania, organizacyjny koszt jakości, średni koszt wykonania testu, średni koszt naprawy defektu).

Należy pamiętać, że różne metryki służą do różnych celów. Na przykład z czysto zarządczego punktu widzenia menedżer będzie zainteresowany tym, ile zaplanowanych testów zostało wykonanych lub czy zaplanowany budżet nie został przekroczony. Z punktu widzenia jakości oprogramowania istotniejsze będą jednak takie kwestie jak np. liczba czy gęstość znalezionych defektów w podziale na ich stopień krytyczności, średni czas między awariami. Umiejętność wybrania odpowiedniego zestawu metryk do mierzenia jest dużą sztuką, ponieważ zazwyczaj nie da się mierzyć wszystkiego. Ponadto im więcej danych w raportach, tym mniej są one czytelne. Powinno się wybrać niewielki zestaw metryk do mierzenia, który jednocześnie pozwoli nam odpowiedzieć na wszystkie pytania dotyczące interesujących nas aspektów projektu. W opracowaniu takiego planu pomiarów może pomóc np. technika Cel Pytanie Metryka (ang. *Goal-Question-Metric*, GQM), jednak nie przedstawiamy jej tutaj, ponieważ treści te wykraczają poza zakres sylabusu.

5.3.2. Cel, treść i odbiorcy raportów z testów

Raporty z testów służą do podsumowywania i przekazywania informacji na temat czynności testowych (np. testów danego poziomu) zarówno w trakcie ich wykonywania, jak i po ich zakończeniu. Raport z testów sporządzany podczas wykonywania czynności testowej może nosić nazwę **raportu o postępie testów** (ang. *test progress report*), a raport sporządzany w momencie zakończenia takiej czynności — **summarycznego raportu z testów** (ang. *test summary report*). Przykłady tych raportów pokazane są na rysunkach 5.12, 5.13 i 5.14.



Summary Status Test Report for: New subscription system (NSS) **Vers.:** Iteration 3

Covers: Complete NSS iteration 3 results.

Progress against Test Plan: Test has been done in the iteration on the 5 user stories for this iteration.

For the one high risk story 92 % statement coverage was achieved, and for the others 68 % statement coverage was achieved on average.

There are no outstanding defects of severity 1 and 2, but the showcase showed that the product has 16 defects of severity 3.

Factors blocking progress: None

Test measures: 6 new test procedures have been developed, and 2 of the other test procedures have been changed.

The testing in the iteration has taken up approx. 30 % of the time. The test took about 2½ hours.

New and changed risks: The risks for the stories have been mitigated satisfactorily. New risks are not identified yet.

Planned testing: As per test plan.

Backlog added: 16 defects (severity 3)

RYSUNEK 5.12. Raport o postępie testów — organizacja zwinna (wg ISO/IEC/IEEE 29119-3)

Test report for: New subscription system (NSS) **Vers.:** Iteration 3361

Covers: NSS final iteration result, including result of previous iterations, in preparation for a major customer delivery (for use).

Risks: The live data risk was retired by creation of a simulated database using historic live data "cleaned" by the test team and customer.

Test Results: Customer accepted this release of the product based on:

16 user stories were successful, including one added after the last status report.

100% statement coverage was achieved in technology-facing testing with the one high risk story, and for the others 72% statement coverage was achieved on average.

Team accepted the backlog on 4 defects of severity 3.

Showcase was accepted by the customer with no added findings. Showcase demo iteration features interfaced with "live" data.

Performance of the iteration features was found to be acceptable by team and customer.

New, changed, and residual risks: Security of the system could become an issue in future releases, assuming a follow work activity is received from the customer.

Notes for future work from retrospective:

Iteration team feels a new member could be needed given possible new risk since no one has knowledge in this area.

Severity 3 defects that move on to backlog should be addressed in next release to reduce technical debt.

The modified live data worked well and should be maintained.

Test automation and exploratory testing is working, but additional test design techniques should be considered, e.g. security and combinatorial testing.

RYSUNEK 5.13. Sumaryczny raport z testów — organizacja zwinna (wg ISO/IEC/IEEE 29119-3)

Project PC-part of the UV/TIT-14 33a product

System Test Completion Report, V 1.0, 08.11.2004

Written by Test Manager Carlo Titlefsen

Approved by Project Manager Benedicte Rytter

Summary of testing performed:

- The test specification was produced; it included 600 test procedures.
- The test environment was established according to the plan.
- Test execution and recording were performed according to the plan.

Deviations from planned testing: The Requirement Specification was updated during the test to V5.6. This entailed rewriting of a few test cases, but this did not have an impact on the schedule.

Test completion evaluation: All test procedures are executed without failures of severity 1 (High). This has not been reached, because one test procedures was not executed. The requirement this test procedure covers is, however, of such low risk exposure that the test has been accepted by the Product Owner.

Factors that blocked progress: None

Test measures:

One (Test procedure 4.7) of the 600 planned test procedures was not executed at all because of lack of time. All the 599 test procedures that were run had passed at the end of the 3 weeks.

During the test 83 incidents were found and 83 were solved. The reported incidents were number 107 to number 189.

Working hours spent:

- 164 working hours were spent on the production of the test specification
- 10 working hours were spent on the establishment of the test environment
- 225 working hours were spent on test execution and recording
- One half hour was spent on this report.

New, changed, and residual risks: All the risks listed in the test plan have been eliminated, except the one with the lowest exposure, Risk no. 19.

Test deliverables: All deliverables specified in the plan have been delivered to the common CM-system according to the procedure.

Reusable test assets: The test specification and the related test data and test environment requirements could be reused for maintenance testing, if and when this is needed.

RYSUNEK 5.14. Sumaryczny raport z testów — organizacja tradycyjna (wg ISO/IEC/IEEE 29119)

Zgodnie ze standardem ISO/IEC/IEEE 29119-3 typowy raport z testów powinien zawierać:

- podsumowanie wykonanych testów;
- opis tego, co się zdarzyło w okresie testowania;
- informacje o odstępstwach od planu;
- informacje o statusie testowania i jakości produktu, w tym informację o spełnieniu kryteriów wyjścia (lub definicji ukończenia);
- informacje o czynnikach blokujących przebieg testów;

- miary związane z defektami, przypadkami testowymi, pokryciem testowym, postępem prac, wykorzystaniem zasobów;
- informacje na temat ryzyka rezydualnego;
- informacje o produktach pracy do ponownego wykorzystania.

Typowy raport o postępie testów zawiera ponadto informacje dotyczące:

- statusu czynności testowych i postępu realizacji planu testów;
- testów zaplanowanych na następny okres raportowania.

Raporty z testów muszą być dostosowane zarówno do kontekstu projektu, jak i do potrzeb docelowych odbiorców. Muszą zawierać:

- szczegółowe informacje na temat typów defektów i związanych z nimi trendów — dla odbiorców technicznych;
- podsumowanie statusu defektów według priorytetów, budżetu i harmonogramu oraz zaliczonych, niezaliczonych i niewykonanych przypadków testowych — dla interesariuszy biznesowych.

Głównymi odbiorcami raportu o postępie testów są ci, którzy są w stanie dokonać zmian w sposobie przeprowadzania testów. Te zmiany mogą obejmować dodanie więcej zasobów testowych lub nawet wprowadzenie zmian do planu testów. Typowymi członkami tej grupy są więc kierownicy projektów i właściciele produktu. Przydatne może być również włączenie w tę grupę adresatów raportów osób odpowiedzialnych za czynniki hamujące postęp testów, aby było jasne, że są one świadome problemu. Wreszcie do grupy tej powinien być również włączony zespół testowy, ponieważ pozwoli mu to zobaczyć, że jego praca jest doceniana, i pomoże zrozumieć, jak jego praca przyczynia się do ogólnego postępu.

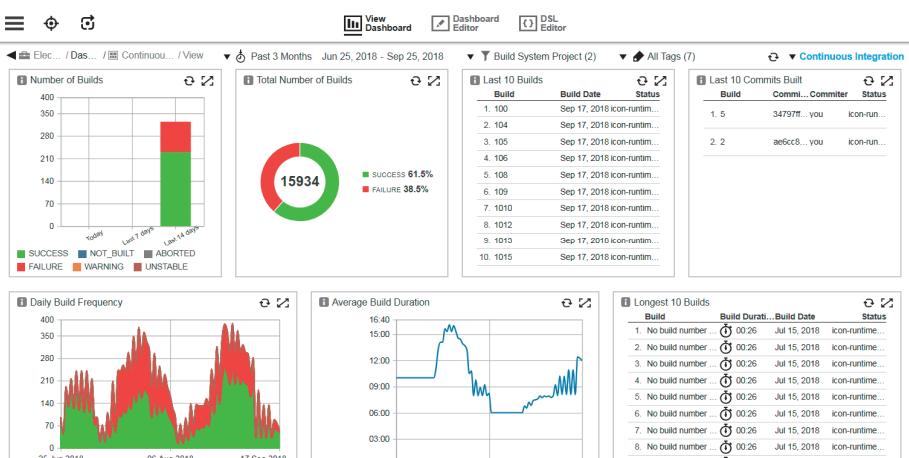
Adresatów raportu z zakończenia testów można podzielić na dwie zasadnicze grupy. Osoby odpowiedzialne za podjęcie decyzji (co robić dalej) i osoby, które będą przeprowadzać testy w przyszłości, korzystając z informacji zawartych w raporcie. Decydenci będą się różnić w zależności od testów, które są raportowane (np. poziom testu, typ testu lub projekt), i mogą decydować o tym, jakie testy powinny być włączone do następnej iteracji lub czy wdrożyć obiekt testowy, biorąc pod uwagę zgłoszone ryzyko szczątkowe. Druga grupa to ci, którzy są odpowiedzialni za przyszłe testowanie obiektu testowego (np. jako część testów pielęgnacyjnych), lub ci, którzy mogą w inny sposób zdecydować o ponownym użyciu produktów z raportowanych testów (np. oczyszczone dane testowe klienta w osobnym projekcie). Dodatkowo wszystkie osoby występujące na oryginalnej liście dystrybucyjnej dla planu testów powinny również otrzymać kopię raportu z zakończenia testów.

5.3.3. Przekazywanie informacji o statusie testowania

Środki komunikowania o statusie testowania są różne, w zależności od obaw, oczekiwani i wizji kadry zarządzającej testowaniem, organizacyjnych strategii testowania, norm regulacyjnych lub, w przypadku zespołów samoorganizujących się (patrz punkt 1.5.2), od samego zespołu. Opcje obejmują w szczególności:

- komunikację słowną z członkami zespołu i innymi interesariuszami;
- tablice wskaźników (ang. dashboards), np. tablice wskaźników CI/CD, tablice zadań (ang. task boards) i wykresy spalania (ang. burn-down charts);
- e-maile w stylu tablicy wskaźników;
- dokumenty online;
- formalne raporty z testów (patrz punkt 5.3.2).

Można używać jednej lub kilku z tych opcji. Bardziej formalna komunikacja może być bardziej odpowiednia dla zespołów rozproszonych, gdzie bezpośrednią komunikacją twarzą w twarz nie zawsze jest możliwa ze względu na różnice geograficzne lub czasowe.



Rysunek 5.15. Przykładowa tablica wskaźników CI/CD (źródło: docs.cloudbees.com)

Na rysunku 5.15 przedstawiono przykładową, typową tablicę wskaźników służącą przekazywaniu informacji dotyczących procesu ciągłej integracji i ciągłego dostarczania. Z tablicy tej można bardzo szybko uzyskać najważniejsze, podstawowe informacje o stanie tego procesu, np.:

- liczbę wydań w podziale na status wydania (udane, niezbudowane, przerwane itp.);
- daty i status ostatnich dziesięciu wydań;
- informację o dziesięciu ostatnich commitach;
- częstość wydań;
- średni czas tworzenia wydania w funkcji czasu.

5.4. Zarządzanie konfiguracją

FL-5.4.1 (K2)

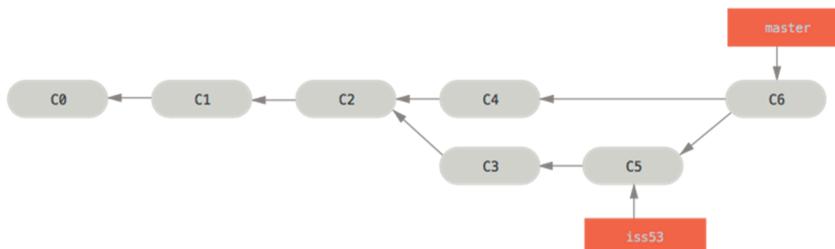
Kandydat podsumowuje, w jaki sposób zarządzanie konfiguracją wspomaga testowanie.

Podstawowym celem zarządzania konfiguracją jest zapewnienie i utrzymanie integralności modułu/systemu i testaliów oraz wzajemnych relacji między nimi przez cały cykl życia projektu i produktu. Zarządzanie konfiguracją ma zagwarantować, że:

- wszystkie przedmioty testów zostały zidentyfikowane, objęte kontrolą wersji i śledzeniem zmian oraz powiązane ze sobą wzajemnie;
- wszystkie testalia zostały zidentyfikowane, objęte kontrolą wersji i śledzeniem zmian oraz powiązane wzajemnie ze sobą i z wersjami przedmiotu testów w sposób pozwalający utrzymać możliwość śledzenia na wszystkich etapach procesu testowego;
- wszystkie zidentyfikowane dokumenty i elementy oprogramowania były przywoływane w sposób jednoznaczny w dokumentacji testów.

Dziś do zarządzania konfiguracją wykorzystuje się odpowiednie narzędzia, należy jednak pamiętać, że procedury zarządzania konfiguracją wraz z niezbędną infrastrukturą (narzędziami) należy zidentyfikować i zaimplementować na etapie planowania testów.

Na rysunku 5.16 przedstawiona jest graficzna reprezentacja pewnego repozytorium przechowywanego w systemie wersjonowania kodu git. Wierzchołki C0, C1, ..., C6 oznaczają tzw. snapshoty (tłumaczone czasem jako „migawki”), czyli kolejne wersje kodu źródłowego. Strzałki określają, na podstawie jakiej wersji powstała kolejna.



RYSUNEK 5.16. Graficzna reprezentacja repozytorium kodu w systemie Git
(źródło: <https://git-scm.com/docs/gittutorial>)

Na przykład commit C1 powstał na bazie commitu C0 (np. programista ściągnął kod C0 na swój lokalny komputer, wprowadził do niego pewne zmiany i przesłał do repozytorium). Repozytorium rozróżnia te dwie wersje kodu. Commity C3 i C4 mogły być tworzone niezależnie przez dwóch innych programistów, na bazie tego samego commita C2. Prostokąt „master” oznacza gałąź główną. Commit C5 z kolei reprezentuje gałąź „iss53”, w której kod powstał w wyniku naprawy błędu wykrytego w wersji C3 kodu. Commit C6 powstał w wyniku połączenia (ang. *merge*) zmian wprowadzonych niezależnie w commitach C5 (naprawa defektu) i C4 (np. dodanie nowej funkcji na bazie wersji C2 kodu).

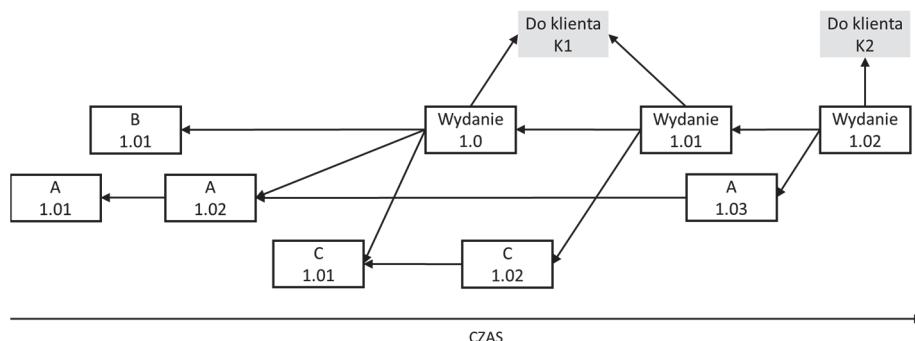
Dzięki stosowaniu takiego systemu kontroli wersji jak Git programiści mają pełną kontrolę nad kodem. Mogą śledzić, zapisywać i — w razie potrzeby — odtwarzać dowolną wersję historyczną kodu. W przypadku popełnienia jakiegoś błędu bądź wystąpienia awarii w jakiejś komplikacji zawsze jest możliwość cofnięcia zmian i powrotu do poprzedniej, działającej wersji.

Obecnie używanie przez zespół (nawet jednoosobowy!) narzędzi do wersjonowania kodu jest *de facto* obowiązującym standardem. Bez tego typu narzędzi w projekcie bardzo szybko zapanowałby chaos.

Przykład. Rozważmy bardzo uproszczony przykład zastosowania zarządzania konfiguracją w praktyce. Założymy, że produkowany przez nas system składa się z trzech modułów: A, B i C. Z systemu korzysta dwóch naszych klientów: K1 i K2. Poniższa sekwencja zdarzeń przedstawia historię zmian w repozytorium kodu oraz historię wydań oprogramowania do klientów. Zakładamy, że stworzenie danej wersji aplikacji zawsze uwzględnia najnowsze wersje wchodzących w jego skład modułów.

1. Wysłanie do repozytorium wersji 1.01 modułu A.
2. Wysłanie do repozytorium wersji 1.01 modułu B.
3. Wysłanie do repozytorium wersji 1.02 modułu A (po usunięciu defektu).
4. Wysłanie do repozytorium wersji 1.01 modułu C.
5. Stworzenie wersji 1.0 aplikacji do wydania i przesłanie jej klientowi K1.
6. Wysłanie do repozytorium wersji 1.02 modułu C (dodanie nowej funkcjonalności).
7. Stworzenie wersji 1.01 aplikacji do wydania i przesłanie jej klientowi K1.
8. Wysłanie do repozytorium wersji 1.03 modułu A (dodanie nowej funkcjonalności).
9. Stworzenie wersji 1.02 aplikacji do wydania i przesłanie jej klientowi K2.

Proces ten graficznie przedstawiony jest na rysunku 5.17.



RYSUNEK 5.17. Historia tworzenia kolejnych wersji modułów i wydań oprogramowania

Założymy, że w tym momencie (po kroku 9.) klient K1 zgłosił awarię oprogramowania. Gdybyśmy nie mieli ustanowionego procesu zarządzania konfiguracją, nie wiedzielibyśmy, w których wersjach modułów A, B, C szukać potencjalnych defektów.

Załóżmy, że klient przesłał nam informację, że problem dotyczy wersji 1.01 oprogramowania. Narzędzie zarządzania konfiguracją na podstawie tej informacji oraz powyższych danych jest w stanie odtworzyć poszczególne części składowe oprogramowania, które weszły w skład wersji 1.01 oprogramowania dostarczonej klientowi K1 w kroku 7. Analizując powyższą sekwencję zdarzeń, widzimy, że na wersję 1.01 oprogramowania składają się:

- moduł A w wersji 1.02;
- moduł B w wersji 1.01;
- moduł C w wersji 1.02.

Oznacza to, że potencjalny defekt znajduje się w jednym (lub kilku) z tych trzech modułów. Zauważmy, że jeśli okaże się, że defekt znajduje się w module B, po jego naprawieniu i stworzeniu wersji 1.02 nową wersję oprogramowania (1.03) należy przesyłać nie tylko klientowi K1, ale również klientowi K2, ponieważ jego bieżąca wersja oprogramowania używa wadliwego modułu B.

5.5. Zarządzanie defektami

FL-5.5.1 (K3)

Kandydat sporządza raport o defekcie.

Jednym z celów testowania jest znajdowanie defektów, w związku z czym wszystkie znalezione defekty należy logować. Sposób zgłoszenia defektu zależy od kontekstu testowania danego modułu lub systemu, poziomu testów oraz wybranego modelu twórczego. Każdy zidentyfikowany defekt powinien zostać zbadany i być śledzony od momentu wykrycia i sklasyfikowania do momentu rozwiązania problemu.

Organizacja powinna wdrożyć proces **zarządzania defektami** (ang. *defect management*), przy czym jego formalizacja może być różna: od podejścia nieformalnego do bardzo formalnego. Może się zdarzyć, że niektóre raporty opisują sytuacje fałszywie pozytywne, a nie rzeczywiste awarie spowodowane defektami. Testerzy powinni starać się ograniczać do minimum liczbę rezultatów fałszywie pozytywnych, które są zgłaszane jako defekty; niemniej jednak w rzeczywistych projektach pewna liczba zgłaszanych defektów to defekty fałszywie pozytywne.



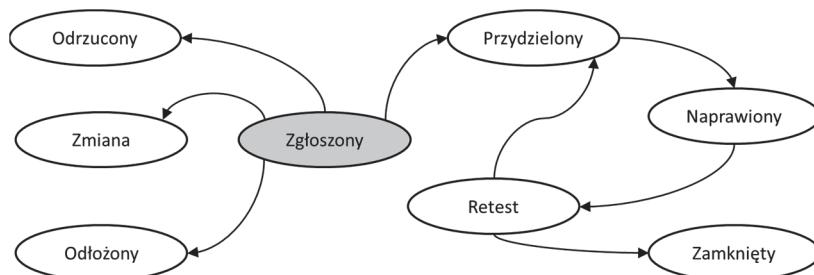
Podstawowe cele **raportu o defekcie** (ang. *defect report*) to:



- przekazywanie informacji:
 - dostarczenie wytwórcom informacji o incydencie, by umożliwić identyfikację defektu, wyizolowanie defektu, w razie potrzeby jego naprawę;
 - dostarczenie kierownictwu testów bieżącej informacji o jakości testowanego systemu i o przebiegu testów;
- umożliwienie zbierania informacji o stanie testowanego produktu:
 - podstawa do podejmowania decyzji;
 - wcześna identyfikacja zagrożeń;
 - zbieranie danych do analizy poprojektowej i udoskonalania procedur twórczych.

W standardzie ISO/IEC/IEEE 29119-3 raporty o defektach nazywane są rapportami o incydentach. Jest to nieco bardziej precyzyjne nazewnictwo, ponieważ nie każda zaobserwowana anomalia od razu musi oznaczać problem do rozwiązania — tester może się pomylić i uznać coś, co jest zupełnie poprawne, za defekt wymagający naprawy. Takie sytuacje zazwyczaj są analizowane w trakcie cyklu wytwarzania defektu i zwykle tego typu zgłoszenia są odrzucane, z nadaniem im statusu „nie defekt” lub podobnego.

Przykładowy cykl życia defektu przedstawiony jest na rysunku 5.18. Z punktu widzenia testera tylko statusy *Odrzucony*, *Zmiana*, *Odłożony*, *Zamknięty* oznaczają, że analiza defektu została zakończona; informacja *Naprawiony* oznacza, że deweloper naprawił defekt, ale zamknąć można go dopiero po reteście.



RYSUNEK 5.18. Statusy raportu o zgłoszeniu defektu

Raport o defektie w testach dynamicznych powinien zawierać:

- unikatowy identyfikator;
- tytuł i krótkie podsumowanie zgłoszanego defektu;
- datę zgłoszenia (= datę odkrycia błędu);
- informację o autorze (o zgłaszającym);
- identyfikację elementu testowego (testowanego elementu, konfiguracji) i środowiska;
- fazę cyklu wytwarzania oprogramowania;
- opis defektu umożliwiający jego odtworzenie i usunięcie, w tym wszelkiego rodzaju dzienniki, zrzuty bazy danych, zrzuty ekranu czy nagrania;
- wynik rzeczywisty i oczekiwany;
- priorytet usunięcia defektu;
- status zgłoszenia;
- opis niezgodności ułatwiający określenie jej przyczyny;
- określenie stopnia pilności rozwiązania;
- określenie elementu konfiguracji oprogramowania lub systemu;
- wnioski i zalecenia;
- historię zmian;
- odwołania do innych elementów, w tym do przypadku testowego, dzięki któremu problem został ujawniony.

Przykładowy raport o defekcie pokazany jest na rysunku 5.19.

Incident Registration Form			
Number	278		
Short Title	Information truncated		
Software product	Project PC-part of the UV/TIT-14 33a product		
Version (n.m)	5.2		
Status = Created			
Registration created by	Heather Small	Date & time	14 th May
Anomaly observed by	Heather Small	Date & time	14 th May
Comprehensive description	<i>The text in field "Summary" is truncated after 54 characters; it should be able to show 75 characters.</i>		
Observed during	Walk-through / Review / Inspection / Code & Build / Test / Use		
Observed in	Requirement / Design / Implementation / Test / Operation		
Symptom	Oper. system crash / Program hang-up / Program crash / Input / Output / Total product failure / System error / Other:		
User impact	High / Medium / Low		
User urgency	Urgent / High / Medium / Low / None		

RYSUNEK 5.19. Przykład raportu o defekcie wg ISO/IEC/IEEE 29119-3

Wśród interesariuszy zainteresowanych zgłoszeniami defektów należy wskazać: testerów, programistów, kierownika testów oraz kierownika projektu. W wielu organizacjach dostęp do systemu zarządzania defektami uzyskuje także klient, zwłaszcza w okresie bezpośrednio po wdrożeniu. Należy jednak brać pod uwagę to, że typowy klient nie zna zasad zgłaszania defektów, co więcej, dla niego prawie każdy defekt jest krytyczny, bo utrudnia mu pracę.

Pytania testowe do rozdziału 5.

Pytanie 5.1

(FL-5.1.1, K2)

Które z poniższych NIE jest częścią planu testów?

- A. Strategia testów.
- B. Ograniczenia budżetowe.
- C. Zakres testowania.
- D. Rejestr ryzyk.

Wybierz jedną odpowiedź.

Pytanie 5.2

(FL-5.1.2, K2)

Która z poniższych czynności wykonuje tester podczas planowania wydania?

- A. Przeprowadza szczegółową analizę ryzyka dla historyjek użytkownika.
- B. Identyfikuje niefunkcjonalne aspekty systemu do przetestowania.
- C. Szacuje wysiłek testowy dla nowych cech planowanych w danej iteracji.
- D. Definiuje historyjki użytkownika oraz ich kryteria akceptacji.

Wybierz jedną odpowiedź.

Pytanie 5.3

(FL-5.1.3, K2)

Dane są następujące kryteria wejścia i wyjścia z testów:

- i. dostępność testerów
- ii. brak otwartych defektów krytycznych
- iii. w testach modułowych osiągnięto 70% pokrycia instrukcji
- iv. wszystkie testy dymne wykonane przed testami systemowymi przechodzą

Które z powyższych są kryteriami wejścia, a które kryteriami wyjścia z testów?

- A. kryteria wejścia to (i), (iii) oraz (iv); kryterium wyjścia to (ii).
- B. kryteria wejścia to (ii) oraz (iii); kryteria wyjścia to (i) oraz (iv).
- C. kryteria wejścia to (i) oraz (ii); kryteria wyjścia to (iii) oraz (iv).
- D. kryteria wejścia to (i) oraz (iv); kryteria wyjścia to (ii) oraz (iii).

Wybierz jedną odpowiedź.

Pytanie 5.4

(FL-5.1.4, K3)

Zespół używa następującego modelu ekstrapolacji do szacowania wyciągu testowego:

$$E(n) = \frac{E(n-1) + E(n-2) + E(n-3)}{3},$$

gdzie $E(n)$ to wyciąg w n -tej iteracji. Wyciąg w n -tej iteracji jest więc średnią wyciągu z trzech ostatnich iteracji.

W trzech pierwszych iteracjach rzeczywisty wyciąg wyniósł, odpowiednio, 12, 15, 18 osobodni. Zespół właśnie zakończył trzecią iterację i chce użyć powyższego modelu do oszacowania wyciągu testowego potrzebnego w PIĄTEJ iteracji. Ile wyniesie szacowanie dokonane przez zespół?

- A. 24 osobodni.
- B. 16 osobodni.
- C. 15 osobodni.
- D. 21 osobodni.

Wybierz jedną odpowiedź.

Pytanie 5.5

(FL-5.1.5, K3)

Chcesz spriorytetyzować przypadki testowe w celu optymalnej kolejności ich wykonania. Wykorzystujesz metodę priorytetyzacji opartą na dodatkowym pokryciu. Jako miarę pokrycia wykorzystujesz pokrycie funkcji. W systemie jest siedem funkcji, oznaczonych jako A, B, C, D, E, F, G. Poniższa tabela zawiera informacje o tym, które funkcje są pokrywane przez poszczególne przypadki testowe:

PRZYPADEK TESTOWY	POKRYTE FUNKCJE
PT1	A, B, C, F
PT2	D
PT3	A, F, G
PT4	E
PT5	D, G

Który przypadek testowy będzie wykonany jako trzeci w kolejności?

- A. PT5.
- B. PT2.
- C. PT4.
- D. PT3.

Wybierz jedną odpowiedź.

Pytanie 5.6

(FL-5.1.6, K1)

Co opisuje piramida testów?

- A. Nakład pracy zespołu poświęcony na testowanie, zwiększający się z iteracji na iterację.
- B. Listę zadań projektowych posortowanych według malejącej liczby wymaganych czynności testowych dla każdego z tych zadań.
- C. Ziarnistość testów na poszczególnych poziomach testów.
- D. Wysiłek testowy na poszczególnych poziomach testów.

Wybierz jedną odpowiedź.

Pytanie 5.7

(FL-5.1.7, K2)

W którym kwadrancie testowym znajdują się testy modułowe?

- A. W kwadrancie zorientowanym na technologię i wspierającym zespół, zawierającym testy automatyczne i będące częścią procesu ciągłej integracji.
- B. W kwadrancie zorientowanym na biznes i wspierającym zespół, zawierającym testy sprawdzające kryteria akceptacji.
- C. W kwadrancie zorientowanym na biznes i krytykującym produkt, będącym zorientowanym na potrzeby użytkownika.
- D. W kwadrancie zorientowanym na technologię i krytykującym produkt, zawierającym automatyczne testy niefunkcjonalne.

Wybierz jedną odpowiedź.

Pytanie 5.8

(FL-5.2.1, K1)

Prawdopodobieństwo ryzyka związanego z wydajnością systemu zostało zdefiniowane jako „bardzo wysokie”.

Co można powiedzieć o wpływie tego ryzyka?

- A. Nic nie wiemy o wpływie; wpływ i prawdopodobieństwo są niezależne.
- B. Wpływ też jest bardzo wysoki; ryzyka o wysokim prawdopodobieństwie też mają wysoki wpływ.
- C. Wpływ jest niski, bo wpływ jest odwrotnie proporcjonalny do ryzyka.
- D. Dopóki nie wystąpi to ryzyko, nie możemy ocenić wpływu.

Wybierz jedną odpowiedź.

Pytanie 5.9

(FL-5.2.2, K2)

Które z poniższych jest przykładem konsekwencji wystąpienia ryzyka projektowego?

- A. Śmierć użytkownika w wyniku awarii oprogramowania.
- B. Niezrealizowanie wszystkich zadań przeznaczonych do wykonania w danej iteracji.
- C. Bardzo wysokie koszty utrzymania oprogramowania.
- D. Niezadowolenie klienta z powodu niewygodnego interfejsu użytkownika.

Wybierz jedną odpowiedź.

Pytanie 5.10

(FL-5.2.3, K2)

Tester pracuje w projekcie przygotowującym nową wersję mobilnej aplikacji bankowej (ang. *home-banking*). Podczas analizy ryzyka zespół zidentyfikował następujące ryzyka:

- zbyt skomplikowany sposób wprowadzania przelewów — zwłaszcza dla seniorów;
- nieprawidłowe działanie opcji „zlecenia stałe” — przelewy są wykonywane z opóźnieniem, gdy płatność wypada w sobotę lub w niedzielę.

Jakie są najbardziej rozsądne działania łagodzące ryzyko, które tester powinien zaproponować w odniesieniu do tych dwóch ryzyk?

- A. Przegląd techniczny dla wprowadzania przelewów, pokrycie gałęzi dla opcji „zlecenie stałe”.
- B. Testy modułowe dla wprowadzania przelewów, testy akceptacyjne dla opcji „zlecenie stałe”.
- C. Testy beta dla wprowadzania przelewów, testy użyteczności dla opcji „zlecenie stałe”.
- D. Testy białoskrzynkowe dla wprowadzania przelewów, testy niefunkcjonalne dla opcji „zlecenie stałe”.

Wybierz jedną odpowiedź.

Pytanie 5.11

(FL-5.2.4, K2)

Po przeprowadzeniu testów systemowych i wydaniu oprogramowania do klienta producent oprogramowania wykupił w firmie ubezpieczeniowej ubezpieczenie. Zrobił to na wypadek, gdyby nieprawidłowe działanie oprogramowania spowodowało utratę zdrowia jego użytkowników.

Z jakim rodzajem działań łagodzących mamy tu do czynienia?

- A. Z planami awaryjnymi.
- B. Z łagodzeniem ryzyka przez testowanie.
- C. Z transferem ryzyka.
- D. Z akceptacją ryzyka.

Wybierz jedną odpowiedź.

Pytanie 5.12

(FL-5.3.1, K1)

Która z poniższych NIE jest miarą stosowaną w odniesieniu do testowania?

- A. Poziom ryzyka rezydualnego.
- B. Pokrycie wymagań przez kod źródłowy.
- C. Liczba znalezionych defektów krytycznych.
- D. Postęp implementacji środowiska testowego.

Wybierz jedną odpowiedź.

Pytanie 5.13

(FL-5.3.2, K2)

Która spośród poniższych informacji NIE będzie zazwyczaj zawarta w sumarycznym raporcie końcowym z testów?

- A. Pozostałe (niezagodzone) ryzyka to ryzyka nr R-001-12 oraz R-002-03.
- B. Odstępstwa od planu testów: opóźnienie testów integracyjnych o 5 dni.
- C. Liczba nienaprawionych defektów krytycznych: 1.
- D. Testy zaplanowane na kolejny okres raportowania: testy modułowe modułu M3.

Wybierz jedną odpowiedzi.

Pytanie 5.14

(FL-5.3.3, K2)

Która z poniższych jest NAJLEPSZĄ formą komunikowania statusu testowania?

- A. Raporty o postępie i raporty sumaryczne, ponieważ są najbardziej formalną formą komunikacji.
- B. To, jaka forma komunikacji będzie najlepsza, zależy będzie od różnych czynników.
- C. E-maile, ponieważ umożliwiają szybką wymianę informacji.
- D. Komunikacja werbalna „twarzą w twarz”, ponieważ jest to najefektywniejsza forma komunikacji między ludźmi.

Wybierz jedną odpowiedź.

Pytanie 5.15

(FL-5.4.1, K2)

Zespół otrzymał od klienta informację o awarii oprogramowania. Na podstawie numeru wersji oprogramowania zespół był w stanie odtworzyć wszystkie wersje modułów oraz testaliów, jakie były wykorzystane do wygenerowania wersji oprogramowania dla tego klienta. To pozwoliło szybciej zlokalizować i naprawić defekt, a także dokonać analizy tego, w jakich innych wersjach wydań oprogramowania należy wprowadzić poprawki związane z tym defektem.

Który proces umożliwił zespołowi wykonanie powyższego scenariusza?

- A. Zarządzanie konfiguracją.
- B. Analiza wpływu.
- C. Ciągłe dostarczanie oprogramowania (ang. *continuous integration*).
- D. Retrospekcja.

Wybierz jedną odpowiedź.

Pytanie 5.16

(FL-5.5.1, K3)

Podczas testowania nowej aplikacji tester stwierdza nieprawidłowe działanie systemu wprowadzania hasła logowania. Zgodnie z dokumentacją hasło ma mieć przy najmniej 10 znaków, w tym minimum jedną dużą i jedną małą literę oraz jedną cyfrę.

Tester sporządza w systemie zarządzania defektami raport o defekcie zawierający następujące informacje:

- Tytuł: Nieprawidłowe logowanie.
- Krótkie podsumowanie: Czasami system dopuszcza hasła o długości 6, 7 i 9 znaków.
- Wersja produktu: Kompilacja 11.02.2020
- Poziom ryzyka: Wysoki.
- Priorytet: Normalny.

Jaką CENNĄ informację pominięto w powyższym zgłoszeniu defektu?

- A. Kroki umożliwiające odtworzenie defektu.
- B. Dane identyfikujące testowany produkt.
- C. Status defektu.
- D. Pomysły dotyczące usprawnienia przypadku testowego.

Wybierz jedną odpowiedź.

Ćwiczenia do rozdziału 5.

Ćwiczenie 5.1

(FL-5.1.4, K3)

Trzech ekspertów szacuje wysiłek testowy dla zadania „przeprowadzenie testów systemowych” metodą będącą połączeniem pokera planistycznego oraz szacowania trójpunktowego. Procedura jest taka:

- Eksperci wyznaczają dla szacowania trójpunktowego wartości: pesymistyczną, średnią i optymistyczną. Każda z nich wyznaczana jest poprzez przeprowadzenie pokera planistycznego. Sesja pokera jest przeprowadzana, dopóki co najmniej dwóch ekspertów nie poda tej samej wartości — w takiej sytuacji poker kończy się, a jego wynikiem jest wartość określona przez większość ekspertów.
- Eksperci stosują szacowanie trójpunktowe z parametrami pesymistycznym, średnim i optymistycznym wyznaczonymi w poprzednim kroku; to szacowanie stanowi wynik końcowy. Obliczane jest także odchylenie standardowe, aby oszacować błąd szacowania.

Wyniki sesji pokera przedstawione są w poniższej tabeli. Wszystkie wartości wyrażone są w osobodniach.

POKER DLA WARTOŚCI:	ITERACJE POKERA
optymistycznej (a)	Wyniki iteracji 1: 3, 5, 8 Wyniki iteracji 2: 3, 3, 5
średniej (m)	Wyniki iteracji 1: 5, 5, 3
pesymistycznej (b)	Wyniki iteracji 1: 13, 8, 21 Wyniki iteracji 2: 8, 13, 40 Wyniki iteracji 3: 13, 13, 13

Jaka jest końcowa wartość szacowania wysiłku testowego oraz błąd szacowania dla analizowanego zadania?

Ćwiczenie 5.2

(FL-5.1.4, K3)

Uwaga. To zadanie jest dosyć trudne i wymaga umiejętności matematycznych, analitycznych oraz dobrego rozumienia miar iloczynowych takich jak wysiłek. Przedstawiamy to zadanie jednak celowo, aby pokazać, jakiego rodzaju problemami w praktyce zajmują się menedżerowie. Często są to problemy nietrywialne, tak jak ten poniższy.

Poniższa tabela przedstawia dane historyczne z zakończonego projektu, w którym fazy projektowania, implementacji i testowania wykonywane były sekwencyjnie, jedna po drugiej:

CZYNNOŚĆ	LICZBA ZAANGAŻOWANYCH OSÓB	CZAS TRWANIA CZYNNOŚCI (DNI)
Projektowanie	4	5
Implementacja	10	18
Testowanie	4	10

Chcesz wykorzystać szacowanie wysiłku na podstawie proporcji, aby oszacować wysiłek potrzebny do przeprowadzenia nowego projektu według tej samej metodyki. Wiadomo, że w nowym projekcie będzie czterech projektantów, sześciu deweloperów i dwóch testerów. Umowa przewiduje, że projekt ma być ukończony w 66 dni.

Ile dni powinniśmy zaplanować na projektowanie, ile na implementację, a ile na testowanie w nowym projekcie?

Wskazówka. Oblicz wysiłek dla każdej z trzech faz w poprzednim projekcie (w osobodniach). Następnie oblicz całkowity wysiłek wymagany dla nowego projektu. Na koniec zastosuj metodę proporcji, aby rozdzielić wysiłek w nowym projekcie na poszczególne fazy.

Ćwiczenie 5.3

(FL-5.1.5, K3)

Testujesz aplikację wspomagającą działanie help-line w firmie ubezpieczeniowej, pozwalającą odszukiwać polisę klienta według jego numeru PESEL. Przypadki testowe opisane są w tabeli 5.3. Priorytet 1 oznacza priorytet najwyższy, 4 — najniższy.

TABELA 5.3. Przypadki testowe dla testowania numeru PESEL wraz z priorytetami i zależnościami

PRZYPADEK TESTOWY	POKRYWANY WARUNEK TESTU	PRIORYTET	ZALEŻNOŚĆ LOGICZNA
001	Odszukanie według numeru PESEL	1	002, 003
002	Wprowadzanie danych osobowych	3	—
003	Modyfikacja numeru PESEL	2	002
004	Usuwanie danych osobowych	4	002

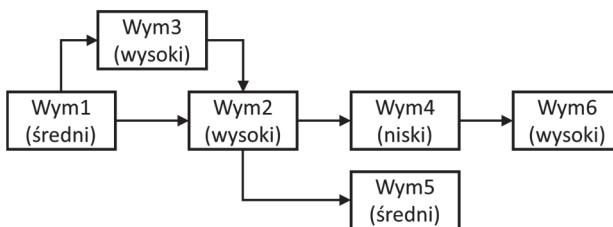
Podaj poprawną kolejność wykonania przypadków testowych.

Ćwiczenie 5.4

(FL-5.1.5, K3)

Zespół chce priorytetyzować testy zgodnie z priorytetyzacją wymagań przedstawioną zespołowi przez klienta. Priorytet każdego z sześciu wymagań Wym1 – Wym6 jest określony przez klienta jako niski, średni lub wysoki. Poza priorytetami pomiędzy wymaganiami zachodzi pewna logiczna kolejność. Niektóre z nich mogą być implementowane (i testowane) dopiero po implementacji innych.

Priorytety oraz zależności pomiędzy wymaganiami pokazane są na rysunku 5.20. Strzałka prowadząca od wymagania A do wymagania B oznacza, że wymaganie B może być implementowane i testowane dopiero po ukończeniu implementacji i przetestowaniu wymagania A.



RYSUNEK 5.20. Priorytety i zależności pomiędzy wymaganiami

Określ ostateczną kolejność implementacji i testowania wymagań.

Ćwiczenie 5.5

(FL-5.5.1, K3)

Testujesz aplikację dla e-sklepu. Wymagania do uwzględniania rabatu podane zostały poniżej:

- Jeżeli klient dokonał zakupu za mniej niż 50 PLN i nie posiada karty stałego klienta, nie otrzymuje rabatu.
- Jeżeli klient dokonał zakupu za mniej niż 50 PLN i posiada kartę stałego klienta, otrzymuje rabat 5%.
- Jeżeli klient dokonał zakupu za co najmniej 50 PLN i za mniej niż 500 PLN i nie posiada karty stałego klienta, otrzymuje rabat 5%.
- Jeżeli klient dokonał zakupu za co najmniej 50 PLN i za mniej niż 500 PLN oraz posiada kartę stałego klienta, otrzymuje rabat 10%.
- Jeżeli klient dokonał zakupu za co najmniej 500 PLN i nie posiada karty stałego klienta, otrzymuje rabat 10%.
- Jeżeli klient dokonał zakupu za co najmniej 500 PLN i posiada kartę stałego klienta, otrzymuje rabat 15%.

Przeprowadziłeś(-łaś) kilka testów — ich wyniki przedstawione są w tabeli 5.4.

TABELA 5.4. Tabela z wynikami przypadków testowych

PRZYPADEK TESTOWY	KWOTA ZAKUPU [PLN]	POSIADA KARTĘ	WARTOŚĆ RABATU [PLN]	DO ZAPŁATY [PLN]	WYNIK POPRAWNY
PT 001	25	T	1	24	N
PT 002	50	T	0	50	N
PT 003	50	N	2,50	48	N
PT 004	500	T	50	450	T
PT 005	600	T	50	550	N

Przygotuj raport o defekcie dla przypadku PT 003.

ROZDZIAŁ 6.

Narzędzia testowe

Słowa kluczowe

automatyzacja testów — użycie oprogramowania do wykonania lub wspierania czynności testowych, np. zarządzania testami, projektowania testów, wykonywania testów i sprawdzania wyników.

6.1. Narzędzia wspomagające testowanie

FL-6.1.1 (K2) Kandydat wyjaśnia, w jaki sposób różnego typu narzędzia testowe wspomagają testowanie.

Znane powiedzenie głosi, że automatyzacja zastępuje to, co działa, czymś, co prawie działa, ale jest szybsze i tańsze [Neeham 1969].

Wiemy już, że podstawowe zadania testowania to:

- analiza i ocena produktu;
- określenie, jakimi metodami będziemy testować;
- zaprojektowanie testów;
- implementacja testów;
- przeprowadzenie testów;
- analiza wyników testów;
- zarządzanie środowiskiem testowym.

Nie wszystkie te czynności można w pełni zautomatyzować. Tak naprawdę to testowanie automatyczne oznacza testowanie wspomagane komputerowo, a w praktyce, mówiąc o **automatyzacji testów** (ang. *test automation*), ma się zazwyczaj na myśli automatyzację ich wykonania, tzn. wykonanie automatycznych skryptów testowych. Należy jednak pamiętać, że automatyzacja może obejmować również pozostałe obszary testowania. Na przykład w podejściu testowania opartym na modelu (ang. *model-based testing*) automatyzacja może obejmować *projektowanie testów* oraz *określanie poprawnego, oczekiwane wyniku* na podstawie analizy modelu dostarczonego przez testera.



Dzisiejsze narzędzia wspomagające testowanie wspierają kilka czynności testowych, takich jak:

- wykonywanie testów;
- przygotowywanie danych testowych;
- zarządzanie testami (wymaganiami, przypadkami testowymi, procedurami testowymi, skryptami testów automatycznych, rezultatami testów, danymi testowymi, defektami);
- raportowanie i monitorowanie wykonywania testów.

Wykorzystanie narzędzi testowych może mieć kilka celów:

- zautomatyzowanie powtarzających się zadań lub zadań, których ręczne wykonywanie wymaga dużych zasobów lub istotnego wysiłku (przede wszystkim testów regresji) — umożliwia to zwiększenie efektywności testów;
- wspieranie czynności manualnych i podniesienie wydajności czynności testowych — zwiększenie niezawodności testowania;
- zwiększenie spójności testowania i odtwarzalności defektów, a więc podniesienie jakości testów;
- zautomatyzowanie czynności, których nie można wykonać lub które bardzo trudno jest wykonać ręcznie (np. testowania wydajnościowego).

Należy jednak pamiętać, że niektóre typy narzędzi testowych mogą mieć charakter inwazyjny — ich użycie może wpływać na rzeczywisty wynik testu. Zjawisko to jest nazywane *efektem próbnika*. Na przykład gdy używamy narzędzi do testów wydajnościowych wydajność testowanego oprogramowania może być nieznacznie gorsza.

Narzędzia testowe wspierają i ułatwiają wiele czynności związanych z testowaniem. Przykłady obejmują następujące grupy narzędzi, choć nie są do nich ograniczone:

- narzędzia do zarządzania — zwiększają efektywność procesu testowego poprzez ułatwienie zarządzania cyklem życia aplikacji, monitorowanie śledzenia podstawy testów do wymagań, zarządzanie testami, defektami, konfiguracją, oferując narzędzia pracy zespołowej (np. tablicę scrumową), a także zapewniają automatyczne raportowanie;
- narzędzia do testowania statycznego — wspierają testera w wykonywaniu przeglądów (głównie w planowaniu przeglądu, wspomaganiu śledzenia, ułatwianiu komunikacji, współpracy przy przeglądach oraz prowadzeniu repozytorium do gromadzenia i raportowania metryk) i analizy statycznej;
- narzędzia do testowania dynamicznego — wspierają testera w przeprowadzaniu analizy dynamicznej, profilowaniu kodu, monitorowaniu zużycia pamięci podczas wykonywania programu itp.;
- narzędzia do projektowania i implementacji testów — ułatwiają generowanie przypadków testowych, danych testowych i procedur testowych, wspierają podejście testowania opartego na modelu, dostarczają frameworki do tworzenia opartego na zachowaniu (BDD) lub testach akceptacyjnych (ATDD);

- narzędzia do wykonywania testów i mierzenia pokrycia — ułatwiają zautomatyzowane wykonywanie testów (skryptów testowych) i automatyczny pomiar pokrycia osiągnięty przez te testy, a także umożliwiają pomiar wyników oczekiwanych i rzeczywistych w testach; w skład tych narzędzi wchodzą narzędzia do automatycznego testowania GUI, API, frameworki do testów modułowych i do wykonywania testów w podejściach takich jak BDD czy ATDD;
- narzędzia do testowania niefunkcjonalnego — umożliwiają testerowi wykonywanie testów niefunkcjonalnych, które są trudne lub niemożliwe do wykonania ręcznie (np. generowanie obciążenia do testów wydajnościowych, skanowanie podatności na ataki na zabezpieczenia, badanie użyteczności interfejsów i stron webowych);
- narzędzia DevOps — wspierają potok wdrożeń, śledzenie przepływu pracy, proces automatyzacji budowy, zautomatyzowanewdrażanieoprogramowania, ciągłą integrację, ciągłe dostarczanieoprogramowania;
- narzędzia do współpracy — ułatwiają komunikację;
- narzędzia wspierające skalowalność i standaryzację wdrożeń (np. maszyny wirtualne, narzędzia do konteneryzacji).

6.2. Korzyści i ryzyka związane z automatyzacją testów

FL-6.2.1 (K1) Kandydat pamięta korzyści i ryzyka związane z automatyzacją testowania.

Samo posiadanie i używanie narzędzia nie gwarantuje jeszcze sukcesu. Znane jest powiedzenie Grad'ego Boocha: *A fool with a tool is still a fool* — głupiec z narzędziem jest nadal głupcem. Osiągnięcie realnych i trwałych korzyści z wdrożenia w organizacji nowego narzędzia zawsze wymaga dodatkowego wysiłku. Narzędzie jest *tylko* narzędziem i nie wykona za testera całej pracy. To tak, jakby oczekwać, że kupiony przez nas młotek sam wbije gwoździe.

Potencjalne korzyści z użycia narzędzi to m.in.:

- oszczędność czasu poprzez redukcję powtarzalnej pracy ręcznej (np. wykonywanie testów regresji, ponowne wprowadzanie tych samych danych testowych, porównywanie oczekiwanych i rzeczywistych wyników, sprawdzanie kodu pod kątem standardów kodowania);
- większa spójność i powtarzalność zapobiegająca prostym ludzkim pomyłkom (np. testy są konsekwentnie wyprowadzane z wymagań, dane testowe są tworzone w sposób systematyczny, a testy są wykonywane przez narzędzie w tej samej kolejności, w taki sam sposób i z tą samą częstotliwością);
- bardziej obiektywna ocena (np. spójny pomiar pokrycia) i możliwość wyliczenia miar, które są zbyt skomplikowane do obliczenia przez człowieka;
- łatwiejszy dostęp do informacji o testach (np. statystyk, wykresów i zagregowanych danych o postępie testów, liczbie defektów i czasie wykonania), aby wspierać zarządzanie testami i raportowanie;

- skrócony czas wykonywania testów, zapewniający wcześniejsze wykrywanie defektów, szybszą informację zwrotną i szybszy czas wprowadzenia produktu na rynek;
- więcej czasu dla testerów na projektowanie nowych, głębszych i bardziej efektywnych testów.

Z użytkowaniem narzędzi do testowania wiążą się również określone (potencjalne) ryzyka:

- nierealistyczne oczekiwania dotyczące korzyści płynących z zastosowania narzędzia (w tym funkcjonalności i łatwości użycia);
- niedokładne/błędne oszacowanie czasu, kosztów, wysiłku potrzebnego do wprowadzenia narzędzia, utrzymania skryptów testowych i zmiany istniejącego procesu testowania manualnego;
- używanie narzędzia testowego, gdy bardziej odpowiednie jest testowanie manualne (np. testy użytkowniczości interfejsu podlegające ludzkiej ocenie);
- poleganie na narzędziu, gdy potrzebne jest ludzkie krytyczne myślenie;
- zależność od dostawcy narzędzia, który może zakończyć działalność, wycofać narzędzie, sprzedać narzędzie innemu dostawcy lub zapewnić słabe wsparcie (np. odpowiedzi na zapytania, aktualizacje i poprawki błędów);
- plan wykorzystania projektu open source może zostać porzucony, co oznacza, że nie będą dostępne dalsze aktualizacje lub jego wewnętrzne komponenty mogą wymagać dość częstych aktualizacji w ramach dalszego rozwoju narzędzia;
- platforma i narzędzie nie są ze sobą kompatybilne;
- nieprzestrzeganie przez narzędzie wymogów prawnych i/lub norm bezpieczeństwa.

Pytania testowe do rozdziału 6.

Pytanie 6.1

(FL-6.1.1, K2)

Która z wymienionych czynności powinna być wspierana przez narzędzie do zarządzania testami?

- A. Projektowanie testów.
- B. Zarządzanie wymaganiami.
- C. Wykonanie testów.
- D. Raportowanie defektów.

Wybierz jedną odpowiedź.

Pytanie 6.2

(FL-6.1.2, K1)

Wskaż DWIE korzyści związane z używaniem narzędzi do testowania.

- A. Uzależnienie od narzędzia.
- B. Zależność od dostawcy narzędzia.
- C. Zwiększenie powtarzalności testów.
- D. Mechaniczna ocena pokrycia.
- E. Koszt utrzymania testaliów większy niż szacowany.

Wybierz dwie odpowiedzi.

CZĘŚĆ III

**ODPOWIEDZI
I ROZWIĄZANIA**

ROZDZIAŁ 7.

Odpowiedzi do pytań testowych

Rozdział 1.

Pytanie 1.1

(FL-1.1.1, K1)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. To jest jeden z celów testowania zgodnie z syllabusem.

Odpowiedź B jest poprawna. Celem testów akceptacyjnych jest raczej potwierdzenie (walidacja), że system działa zgodnie z oczekiwaniami, a nie szukanie awarii (weryfikacja).

Odpowiedź C jest niepoprawna. To jest jeden z celów testowania zgodnie z syllabusem.

Odpowiedź D jest niepoprawna. To jest jeden z celów testowania zgodnie z syllabusem.

Pytanie 1.2

(FL-1.1.2, K2)

Poprawna odpowiedź: A

Ujawnianie awarii (ii) i wykonywanie retestów, czyli testów potwierdzających (iv), to zadania testowania. Debugowanie to proces wyszukiwania, analizowania i usuwania przyczyn awarii w module lub systemie, w związku z tym znajdowanie defektów w kodzie (i) oraz analiza znalezionych defektów (iii) są zadaniami debugowania. Tym samym odpowiedź A jest poprawna.

Pytanie 1.3

(FL-1.2.1, K2)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Taka uwaga może spowodować powstanie konfliktu w zespole, a do tego nie należy dopuszczać, bo konflikt zagraża osiągnięciu celów projektowych.

Odpowiedź B jest poprawna. Zgłoszenie tego braku lub ujęcie kwestii czasu zamiany przedmiotu w złoto jako jednego z kryteriów akceptacji historyjki dobrze obrazuje wkład testowania w procesie wytwarzania produktu. Minimalizuje bowiem ryzyko nieuwzględnienia tego wymagania przez programistów.

Odpowiedź C jest niepoprawna. Brak jest uzasadnienia dla natychmiastowej reakcji właściciela produktu. Ponadto testowanie nikogo do niczego nie może zmuszać. Może jedynie informować o problemach.

Odpowiedź D jest niepoprawna. Wykryty problem dotyczy funkcjonalności, nie jest defektem niefunkcjonalnym.

Pytanie 1.4

(FL-1.2.2, K1)

Poprawna odpowiedź: A

Zapewnienie jakości koncentruje się na zapobieganiu wprowadzania defektów poprzez ustanawianie, wdrażanie i kontrolowanie odpowiednich procesów (i), natomiast testowanie koncentruje się na ocenie oprogramowania oraz powiązanych produktów w celu określenia, czy spełniają one wyspecyfikowane wymagania (iii). Zapewnienie jakości NIE kontroluje jakości tworzonego produktu (ii). Testowanie NIE koncentruje się na usuwaniu defektów z oprogramowania (iv). Zatem poprawne zdania to (i) i (iii). Tym samym poprawna jest odpowiedź A.

Pytanie 1.5

(FL-1.2.3, K1)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Jest to definicja *błędu* zgodnie ze słownikiem terminów testowych.

Odpowiedź B jest poprawna. Zgodnie ze słownikiem terminów testowych usterka (czyli defekt) to niedoskonałość lub wada produktu pracy, polegająca na niespełnieniu wymagań.

Odpowiedź C jest niepoprawna. Jest to definicja *awarii* zgodnie ze słownikiem terminów testowych.

Odpowiedź D jest niepoprawna. Przypadek testowy nie jest usterką.

Pytanie 1.6

(FL-1.3.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Ta zasada mówi o tym, że wcześnie wykonywanie testów i wykrywanie defektów pozwalają usunąć defekty na wcześniejszym etapie cyklu wytwarzania oprogramowania, co ogranicza lub eliminuje kosztowne zmiany, które należałyby wprowadzić, gdyby defekty te były wykryte później, np. po wdrożeniu do klienta.

Odpowiedź B jest niepoprawna. Ta zasada mówi o tym, że testowanie wykonuje się w różny sposób w różnych kontekstach biznesowych.

Odpowiedź C jest niepoprawna. Ta zasada mówi o konieczności modyfikowania dotychczasowych testów i danych testowych, a także pisania nowych testów, aby zestaw testów był nieustannie gotowy na wykrywanie nowych defektów.

Odpowiedź D jest poprawna. Ta zasada właśnie o tym mówi: „Zwykle większość wykrytych defektów lub większość awarii występujących w fazie eksploatacji powstaje lub ma swoje źródło w niewielkiej liczbie modułów systemu, co jest ilustracją tzw. zasad Pareto”.

Pytanie 1.7

(FL-1.4.1, K2)

Poprawna odpowiedź: C

Zgodnie z sylabusem testowalność podstawy testów jest sprawdzana podczas analizy testów. Stąd poprawną odpowiedzią jest C.

Pytanie 1.8

(FL-1.4.2, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Budżet ma istotny wpływ na proces testowy.

Odpowiedź B jest niepoprawna. Normy i standardy mają istotny wpływ na proces testowy, zwłaszcza w projektach audytowanych czy projektach dotyczących systemów krytycznych.

Odpowiedź C jest poprawna. Liczba zatrudnionych certyfikowanych testerów w organizacji nie ma *istotnego* wpływu na proces testowy.

Odpowiedź D jest niepoprawna. Znajomość dziedziny biznesowej przez testerów ma istotny wpływ na proces testowy, ponieważ umożliwia efektywniejszą komunikację z klientem oraz przyczynia się do zwiększenia wydajności testowania.

Pytanie 1.9

(FL-1.4.3, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Raport o postępie testów jest typowym produktem pracy w ramach monitorowania i nadzoru.

Odpowiedź B jest niepoprawna. Informacja o bieżącym poziomie ryzyka w produkcji jest typową informacją raportowaną w ramach monitorowania testów.

Odpowiedź C jest niepoprawna. Jeśli decyzje podjęte w ramach nadzoru są dokumentowane, to ma to miejsce właściwie w tej fazie.

Odpowiedź D jest poprawna. Sumaryczny raport z testów jest produktem pracy powstającym w fazie ukończenia testów.

Pytanie 1.10

(FL-1.4.4, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Jeśli ryzyko jest szacowane ilościowo, wyniki testów mogą być śledzone do przypadków testowych, a te — do pokrywanych przez nie ryzyk. Jeśli wszystkie przypadki testowe śledzone do danego ryzyka są zaliczone, można uznać, że poziom pozostałego w produkcie ryzyka zmniejszył się o wartość tego ryzyka.

Odpowiedź B jest niepoprawna. Definiowanie akceptowalnego poziomu pokrycia kodu to przykład ustanawiania kryterium wyjścia. Mechanizm śledzenia nie ma z tym procesem nic wspólnego.

Odpowiedź C jest niepoprawna. Śledzenie nie pomoże w określaniu oczekiwanej wyników przypadku testowego, bo mechanizm śledzenia nie ma własności wyroczni testowej.

Odpowiedź D jest niepoprawna. Wyprowadzanie tego typu danych testowych może być możliwe dzięki zastosowaniu odpowiedniej techniki testowania, a nie mechanizmu śledzenia.

Pytanie 1.11

(FL 1.4.5, K2)

Poprawna odpowiedź: D

Do zadań osoby pełniącej rolę związaną z zarządzaniem testami należą głównie zadania wykonywane w ramach planowania, monitorowania, nadzoru i ukończenia testów. W szczególności obejmuje to koordynowanie realizacji strategii testów i planu testów (i), tworzenie sumarycznego raportu z testów (iii) oraz decydowanie o implementacji środowisk testowych (v). Osoba pełniąca rolę związaną z testowaniem wykonuje czynności występujące głównie w fazach analizy, projektowania, implementacji i wykonania testów. Odpowiada więc za definiowanie warunków testowych (ii), automatyzowanie testowania (iv) oraz weryfikowanie środowisk testowych (vi). Zatem poprawna jest odpowiedź D.

Pytanie 1.12

(FL-1.5.1, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Analityczne myślenie jest typową, generyczną cechą dobrego testera.

Odpowiedź B jest niepoprawna. Znajomość wiedzy dziedzinowej jest typową, generyczną cechą dobrego testera.

Odpowiedź C jest poprawna. Umiejętność programowania nie jest krytyczną cechą testera. Do dobrego wykonywania testów nie jest ona koniecznie potrzebna (np. w przypadku wykonywania testów manualnych lub testów eksploracyjnych).

Odpowiedź D jest niepoprawna. Umiejętności komunikacyjne to typowa, generyczna cecha dobrego testera.

Pytanie 1.13

(FL-1.5.2, K1)

Poprawne odpowiedzi: B, E

Odpowiedź A jest niepoprawna. Zazwyczaj to programiści, a nie testerzy implementują i wykonują testy modułowe.

Odpowiedź B jest poprawna. To jest jedna z cech podejścia „cały zespół”, polegająca na współpracy wszystkich interesariuszy.

Odpowiedź C jest niepoprawna. Klient nie jest kompetentny w zakresie wyboru narzędzi dla zespołu twórczego — to zespół wybiera narzędzia, jakich chce używać, a formalną decyzję podejmuje kierownictwo lub — w metodach zwinnych — sam zespół.

Odpowiedź D jest niepoprawna. Klient nie jest kompetentny w zakresie projektowania testów niefunkcjonalnych, to zadanie spoczywa na testerach i programistach.

Odpowiedź E jest poprawna. Wspólna odpowiedzialność oraz dbanie o jakość są jedną z podstawowych zasad podejścia „cały zespół”.

Pytanie 1.14

(FL 1.5.3, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Testerzy mają inne spojrzenie na testowany system niż jego twórcy i unikają wielu błędów poznawczych właściwych autorom produktu prac.

Odpowiedź B jest niepoprawna. Programiści mogą (a wręcz powinni) testować tworzony przez siebie kod.

Odpowiedź C jest niepoprawna. Tak powinni działać testerzy, ale awarie w sposób konstruktywny mogą też zgłaszać inni interesariusze.

Odpowiedź D jest niepoprawna. Znajdowanie defektów nie powinno być postrzegane jako krytyka wobec programistów, ale to nie ma związku z niezależnością, tylko z dążeniem do harmonijnej współpracy programistów i testerów.

Rozdział 2.

Pytanie 2.1

(FL-2.1.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. W modelu sekwencyjnym (a takim jest model V) dla oprogramowania krytycznego dla życia (a takim jest autopilot) testowanie aparte na doświadczeniu powinno być uzupełnieniem, a nie podstawą testowania.

Odpowiedź B jest niepoprawna. Wybór modelu cyklu wytwarzania nie wpływa bezpośrednio na to, czy w projekcie obecne będą testy statyczne. Ponadto wczesne wykonywanie testów statycznych jest uznawane za dobrą praktykę.

Odpowiedź C jest niepoprawna. Model V jest sekwencyjnym modelem cyklu wytwarzania, więc nie występują w nim iteracje. Ponadto ze względu na sekwencyjność działające oprogramowanie, nawet w postaci prototypu, zazwyczaj może być dostępne dopiero w późniejszych fazach cyklu wytwarzania.

Odpowiedź D jest poprawna. W początkowych fazach sekwencyjnych modeli cyklu wytwarzania (a takim jest model V) testerzy zazwyczaj uczestniczą w przeglądach wymagań, analizie testów i projektowaniu testów. Kod wykonywalny jest zwykle tworzony w późniejszych fazach, więc testowanie dynamiczne zazwyczaj nie może być wykonane we wczesnych fazach cyklu wytwarzania.

Pytanie 2.2

(FL-2.1.2, K1)

Poprawna odpowiedź: D

W modelu V każdej fazie twórczej (lewe ramię modelu) odpowiada stowarzyszona z nią faza testerska (prawe ramię modelu). Na modelu jest to opisane poprzez przeprowadzenie strzałek pomiędzy poszczególnymi fazami (np. od testów akceptacyjnych do fazy wymagań; od testów systemowych do fazy projektowania itd.). Stąd poprawna odpowiedź to D.

Pytanie 2.3

(FL-2.1.3, K1)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. TDD (wytwarzanie sterowane testami) polega na pisaniu niskopoziomowych testów modułowych, które nie wykorzystują historyjek użytkownika.

Odpowiedź B jest poprawna. ATDD (wytwarzanie sterowane testami akceptacyjnymi) wykorzystuje kryteria akceptacji jako podstawę do projektowania przypadków testowych.

Odpowiedź C jest niepoprawna. W podejściu FDD (wytwarzanie sterowane cechami) podstawą do tworzenia oprogramowania są zdefiniowane cechy (features); podejście to nie ma nic wspólnego z ATDD (patrz odpowiedź poprawna).

Odpowiedź D jest niepoprawna. BDD (wytwarzanie sterowane zachowaniem) wykorzystuje jako podstawę testów opis pożdanego zachowania systemu, zazwyczaj w formacie Given/When/Then (Mając/Kiedy/Wtedy).

Pytanie 2.4

(FL-2.1.4, K2)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. DevOps nie polega na automatycznej generacji danych testowych dla żadnego poziomu testów.

Odpowiedź B jest poprawna. Czynności wykonywane automatycznie po wysłaniu przez programistę kodu do repozytorium, takie jak analiza statyczna, testy modułowe czy testy integracyjne, pozwalają na bardzo szybką informację zwrotną dla programisty na temat poziomu jakości przesłanego przez programistę kodu.

Odpowiedź C jest niepoprawna. Tego typu aktywność jest możliwa np. w ramach testowania opartego na modelu. Podejście DevOps nie umożliwia automatycznej generacji przypadków testowych.

Odpowiedź D jest niepoprawna. Podejście DevOps nie wpływa na czas planowania wydania oraz iteracji; ponadto nawet, gdyby była to prawda, nie jest to korzyść związana z testowaniem, lecz raczej z zarządzaniem projektem.

Pytanie 2.5

(FL-2.1.5, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Jednym z przykładów podejścia „przesunięcie w lewo” (ang. *shift-left*) jest stosowanie podejścia „najpierw test”, którego przykładem jest wytwarzanie sterowane testami akceptacyjnymi (ang. *Acceptance Test-Driven Development*, ATDD).

Odpowiedź B jest niepoprawna. Podejście „przesunięcie w lewo” (ang. *shift-left*) nie wyróżnia w żaden sposób testowania eksploracyjnego. Nacisk na stosowanie konkretnych typów testów zależy raczej od analizy ryzyka.

Odpowiedź C jest niepoprawna. Samo tworzenie prototypów GUI nie jest przykładem stosowania podejścia „przesunięcie w lewo” (ang. *shift-left*), bo nie ma związku z testowaniem.

Odpowiedź D jest niepoprawna. To jest przykład podejścia „przesunięcie w prawo” (ang. *shift-right*), czyli późnego testowania, po wydaniu oprogramowania do klienta, w celu bieżącego monitorowania poziomu jakości produktu w środowisku operacyjnym.

Pytanie 2.6

(FL-2.1.6, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Testerzy powinni brać udział w spotkaniach retrospektwnych, odnosząc się do wszystkich poruszanych na tych spotkaniach kwestii.

Odpowiedź B jest niepoprawna. Testerzy powinni brać udział we wszystkich aspektach spotkania retrospektwnego. Opisana rola bardziej przypomina rolę moderatora.

Odpowiedź C jest poprawna. To jest typowa czynność wykonywana przez testera na spotkaniu retrospektwnym: omawiać to, co wydarzyło się podczas ukończonej iteracji.

Odpowiedź D jest niepoprawna. To nie jest cel spotkania retrospektwnego, tester powinien omawiać to, co zdarzyło się podczas ostatniej iteracji.

Pytanie 2.7

(FL-2.2.1, K2)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Testy integracji modułów, z którymi mamy tu do czynienia, skupiają się na interakcji i komunikacji między modułami, a nie na samych modułach.

Odpowiedź B jest poprawna. Chcemy przeprowadzić testy integracyjne. Projekt architektury jest typową podstawą testów dla tego typu testów, ponieważ opisuje zazwyczaj sposób komunikacji pomiędzy różnymi elementami systemu.

Odpowiedź C jest niepoprawna. Raporty z analizy ryzyka są bardziej przydatne w testowaniu systemowym niż integracyjnym.

Odpowiedź D jest niepoprawna. Regulacje prawne są przydatne zazwyczaj do testowania wysokopoziomowego i walidacji, np. w testowaniu akceptacyjnym.

Pytanie 2.8

(FL-2.2.2, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Ten test sprawdza, „co” system robi, a zatem jest przykładem testu funkcjonalnego.

Odpowiedź B jest niepoprawna. To przykład testu białoskrzynkowego, nie niefunkcjonalnego.

Odpowiedź C jest niepoprawna. Ten test sprawdza, „co” system robi, a zatem jest przykładem testu funkcjonalnego.

Odpowiedź D jest poprawna. To przykład testu niefunkcjonalnego, a dokładniej — testu wydajnościowego. Ten typ testu sprawdza, „jak” system działa, a nie „co” robi.

Pytanie 2.9

(FL-2.2.3, K2)

Poprawna odpowiedź: D

Testowanie potwierdzające przeprowadza się po tym, jak defekt został znaleziony i zgłoszony jako naprawiony. Ponieważ nie wiemy, kiedy test wywoła awarię, ani nie wiemy zazwyczaj, jak długo będzie trwała naprawa, nie jesteśmy w stanie przewidzieć terminu wykonania testu potwierdzającego. Nie da się więc tych testów dokładnie zaplanować z góry. Wszystkie pozostałe typy testów można zaplanować z góry i umieścić harmonogram ich wykonania w planie testów. Tym samym poprawna odpowiedź to D.

Pytanie 2.10

(FL-2.3.1, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Nie aktualizujemy oprogramowania, lecz je naprawiamy.

Odpowiedź B jest niepoprawna. Ze scenariusza nie wynika, abyśmy przeprowadzali jakiekolwiek czynności migracji oprogramowania.

Odpowiedź C jest poprawna. Modyfikacja oprogramowania jest jednym ze zdarzeń wywołujących pielęgnację. Naprawa defektu to modyfikacja oprogramowania.

Odpowiedź D jest niepoprawna. Choć jest to czynnik wyzwalający pielęgnację w przypadku systemów typu IoT, w tym scenariuszu chcemy naprawić defekt — nie wprowadzamy nowych funkcjonalności systemu.

Rozdział 3.

Pytanie 3.1

(FL-3.1.1, K1)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Przeglądom może być poddany także dokument definiujący zasady przeglądów. Nie musi on przy tym być sprawdzany zgodnie z zapisami tego dokumentu; przegląd może być przeprowadzony w oparciu o zdroworozsądakowe kryteria.

Odpowiedź B jest niepoprawna. Dokument mówi o zasadach przeprowadzania przeglądów, ale możemy go przeglądać, nie stosując się do zasad przeżeń omawianych, zachowując po prostu zdrowy rozsądek.

Odpowiedź C jest niepoprawna. To, że dokument nie jest produktem pracy jakiegoś określonego procesu, nie wyklucza możliwości objęcia go przeglądem.

Odpowiedź D jest niepoprawna. Przeglądy mogą być stosowane wobec dowolnego dokumentu.

Pytanie 3.2

(FL-3.1.2, K2)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Nie uruchamialiśmy kodu, lecz tylko analizowaliśmy jego własności.

Odpowiedź B jest poprawna. To klasyczny przykład zysku z użycia techniki statycznej — w tym przypadku analizy statycznej.

Odpowiedź C jest niepoprawna. Pomiar złożoności cyklotomatycznej, analiza tego pomiaru oraz refaktoryzacja kodu nie są czynnościami kierowniczymi, lecz technicznymi.

Odpowiedź D jest niepoprawna. Analiza statyczna nie jest przykładem formalnej techniki testowania; technikami takimi są np. podział na klasy równoważności, analiza wartości brzegowych czy inne techniki czarno- lub białośkrzynkowe. Wynikiem analizy statycznej nie jest projekt przypadków testowych.

Pytanie 3.3

(FL-3.1.3, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Chociaż zdanie to samo w sobie jest prawdziwe, nie rozróżnia tych podejść ze względu na ich cel. Należy zauważać, że testy dynamiczne bezpośrednio znajdują awarie, ale pośrednio — poprzez ich analizę i proces debugowania — znajdują także defekty.

Odpowiedź B jest niepoprawna. Na przykład przeglądy mogą być wykonywane w bardzo późnych fazach (np. mogą dotyczyć dokumentacji użytkownika), a testy dynamiczne mogą rozpocząć się wcześnie w fazie implementacji.

Odpowiedź C jest poprawna. Analiza statyczna i dynamiczna mają te same cele (zobacz sylabus, punkt 1.1.1), np. identyfikację defektów możliwie najwcześniej w cyklu wytwarzania. Zatem, ze względu na cel, nie ma żadnej różnicy między tymi technikami, choć same techniki różnią się w istotny sposób.

Odpowiedź D jest niepoprawna. Po pierwsze, nie wydaje się przekonująca (np. przeglądy nie wymagają żadnych umiejętności programistycznych); po drugie, nie odpowiada na pytanie, w którym chodziło o kryterium celu, a nie wymaganych umiejętności.

Pytanie 3.4

(FL-3.2.1, K1)

Poprawna odpowiedź: C

Zdanie (i) jest nieprawdziwe; programiści implementują te cechy, które są wymagane przez biznes i są częścią iteracji. Gdy zakończą swoje zadania, wspomagają inne zadania związane z iteracją.

Zdanie (ii) jest prawdziwe; częsta informacja zwrotna pozwala skoncentrować uwagę na cechach o największej wartości.

Zdanie (iii) jest nieprawdziwe; wcześniejsza informacja zwrotna może wręcz skutkować koniecznością przeprowadzenia większej liczby testów ze względu na częste lub istotne zmiany.

Zdanie (iv) jest prawdziwe; użytkownicy wskazują, które wymagania są ominięte lub źle zinterpretowane, co powoduje, że końcowy produkt lepiej spełnia ich potrzeby.

Tym samym poprawna jest odpowiedź C.

Pytanie 3.5

(FL-3.2.2, K2)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Te czynności wchodzą w skład fazy planowania.

Odpowiedź B jest niepoprawna. Zbieranie metryk jest częścią czynności „usunięcie defektów i raportowanie”.

Odpowiedź C jest poprawna. Zgodnie z syllabusem jest to czynność wykonywana w fazie rozpoczęcia przeglądu.

Odpowiedź D jest niepoprawna. Te czynności wchodzą w skład fazy planowania.

Pytanie 3.6

(FL-3.2.3, K1)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Zadaniem lidera przeglądu jest ogólna odpowiedzialność za przegląd, a także podejmowanie decyzji o tym, kto ma wziąć udział w przeglądzie oraz gdzie i kiedy przegląd ma się odbyć. Lider pełni więc funkcję organizacyjną, podczas gdy facylitator (patrz poprawna odpowiedź) to rola techniczna, związana bezpośrednio z przeprowadzaniem spotkań przeglądowych.

Odpowiedź B jest poprawna. Moderator (zwany również facylitatorem) jest odpowiedzialny za zapewnienie efektywnego przebiegu spotkań przeglądowych.

Odpowiedź C jest niepoprawna. Odpowiedzialnością autora może być prezentacja bądź komentarz dotyczący jego/jej produktu pracy, ale autor nie wchodzi nigdy w rolę moderatora.

Odpowiedź D jest niepoprawna. Zadaniem przeglądających jest merytoryczny przegląd produktu pracy, a nie moderowanie spotkań. Moderator to rola pozwalająca właśnie odciążyć przeglądających od konieczności notowania spostrzeżeń czynionych przez nich podczas spotkania przeglądowego.

Pytanie 3.7

(FL-3.2.4, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Patrz uzasadnienie odpowiedzi D.

Odpowiedź B jest niepoprawna. Patrz uzasadnienie odpowiedzi D.

Odpowiedź C jest niepoprawna. Patrz uzasadnienie odpowiedzi D.

Odpowiedź D jest poprawna. Zrozumienie produktu pracy (lub nauczenie się czegoś) jest jednym z celów przejrzenia. Przejrzenia mogą przyjąć formę tzw. suchych przebiegów, zatem w wyniku zastosowania tego typu przeglądu zespół może najefektywniej zrozumieć sposób działania oprogramowania, a przez to łatwiej odkryć przyczynę dziwnej awarii.

Pytanie 3.8

(FL-3.2.5, K1)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Głównym celem inspekcji jest znajdowanie defektów. Ocena alternatyw jest celem właściwszym dla przeglądu technicznego.

Odpowiedź B jest niepoprawna. Inspekcje są zwykle przeprowadzane przez osoby zajmujące stanowisko równorzędne z autorem (ang. *peers*). Obecność kierownictwa na spotkaniu przeglądowym niesie ryzyko złego zrozumienia celu przeglądu i np. oceny przez to kierownictwo poszczególnych członków spotkania.

Odpowiedź C jest poprawna. Szkolenie w zakresie technik przeglądu jest jednym z czynników sukcesu dla przeglądów.

Odpowiedź D jest niepoprawna. Pomiar metryk pomaga w doskonaleniu procesu przeglądu. Ponadto w formalnym przeglądzie, jakim jest inspekcja, zbieranie metryk jest czynnością obowiązkową.

Rozdział 4.

Pytanie 4.1

(FL-4.1.1, K2)

Poprawna odpowiedź: B

Analiza dziedziny wejściowej oprogramowania ma miejsce podczas wykorzystywania czarnoskrzynkowych technik testowania takich jak podział na klasy równoważności czy analiza wartości brzegowych.

Pytanie 4.2

(FL-4.1.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna, ponieważ opisuje cechę wspólną technik biało-skrzynkowych.

Odpowiedź B jest niepoprawna, ponieważ jeśli już chcemy projektować dane testowe na podstawie analizy kodu, to musimy mieć do niego dostęp, zatem możemy to robić w ramach testowania białośkrzynkowego.

Odpowiedź C jest niepoprawna, ponieważ opisuje sposób pomiaru pokrycia w technikach białośkrzynkowych.

Odpowiedź D jest poprawna. Wymienione w pytaniu techniki to techniki czarno-skrzynkowe. Zgodnie z sylabusem w przypadku technik czarnoskrzynkowych warunki testowe, dane testowe i przypadki testowe wyprowadza się z podstawy testów zewnętrznej wobec testowanego obiektu (np. z wymagań, specyfikacji, przypadków użycia). Dlatego przypadki testowe będą w stanie wykrywać rozbieżności pomiędzy tymi wymaganiami a ich rzeczywistą implementacją.

Pytanie 4.3

(FL-4.2.1, K3)

Poprawna odpowiedź: A

W pytaniu opisano trzy klasy równoważności:

- klasa bez zniżki {0,01, 0,02, ..., 99,98, 99,99},
- klasa ze zniżką 5% {100,00, 100,01, ..., 299,98, 299,99},
- klasa ze zniżką 10% {300,00, 300,01, 300,02, ...}.

Zauważmy, że najmniejszą możliwą kwotą jest 0,01 jako najmniejsza możliwa liczba dodatnia. Tym samym:

Odpowiedź A jest poprawna. Trzy wartości 0,01, 100,99, 500, z których każda należy do innej klasy, pokrywają wszystkie trzy klasy równoważności.

Odpowiedź B jest niepoprawna, ponieważ mamy tylko trzy klasy, zatem minimalny zbiór wartości branych do testów powinien mieć trzy elementy.

Odpowiedź C jest niepoprawna, ponieważ wartości 1 i 99 należą do jednej klasy. Ten zestaw nie zawiera wartości, dla której system powinien przydzielić 10% zniżki, zatem ostatnia klasa nie jest pokryta.

Odpowiedź D jest niepoprawna, ponieważ wszystkie wartości należą do klasy bez zniżki. Są to wartości reprezentujące możliwe procentowe wartości zniżki (0, 5, 10), ale w zadaniu dziedziną, którą analizujemy, nie jest dziedzina wyjściowa (rodzaj zniżki), lecz dziedzina wejściowa (kwota zakupów).

Pytanie 4.4

(FL-4.2.1, K3)

Poprawna odpowiedź: C

Zgodnie z warunkami opisanyymi w pytaniu mamy sprawdzić dwie sytuacje (dwie klasy równoważności): jedną, w której maszyna nie wydaje reszty, i drugą, w której wydaje resztę.

Odpowiedź A jest niepoprawna. Scenariusz 1. jest niemożliwy do zrealizowania: w momencie wrzucenia dwóch pierwszych monet (2 PLN i 1 PLN) sumaryczna kwota przekroczy 2,50 PLN i maszyna zablokuje możliwość wrzucania dalszych monet. Zatem odpowiedź A możemy wykluczyć.

Odpowiedź B jest niepoprawna, ponieważ oba scenariusze pokrywają jedynie przypadek, w którym automat wydaje resztę. Brakuje testu, w którym automat nie wydawałby reszty.

Odpowiedź C jest poprawna. W scenariuszu 2. automat nie wydaje reszty (kwota wrzucona jest dokładnie równa 2,50 PLN), a w scenariuszu 3. automat wydaje resztę.

Odpowiedź D jest niepoprawna, ponieważ do pokrycia klas równoważności wystarczą dwa scenariusze.

Pytanie 4.5

(FL-4.2.1, K3)

Poprawna odpowiedź: A

Do osiągnięcia 100% pokrycia klas równoważności należy wykorzystać metodę „each choice”. Istniejące przypadki testowe nie pokrywają jedynie karty diamentowej oraz zniżki 5%. Te dwa elementy można pokryć jednym, dodatkowym przypadkiem testowym:

PT05: karta diamentowa, 5%.

Tym samym poprawna odpowiedź to A.

Pytanie 4.6

(FL-4.2.2, K3)

Poprawna odpowiedź: D

Rozważaną zmienną jest długość hasła, a klasy równoważności wyglądają tak:

- hasło zbyt krótkie: {0, 1, 2, 3, 4, 5},
- hasło o poprawnej długości: {6, 7, 8, 9, 10, 11},
- hasło zbyt długie: {12, 13, 14, ...}.

Wartości brzegowe to 0, 5, 6, 11, 12 (i jednocześnie są to wartości, które należy przetestować w metodzie dwupunktowej, aby osiągnąć 100% zadanego pokrycia). W metodzie trójpunktowej musimy pokryć następujące wartości:

- 0, 1 (dla wartości brzegowej 0),
- 4, 5, 6 (dla wartości brzegowej 5),
- 5, 6, 7 (dla wartości brzegowej 6),
- 10, 11, 12 (dla wartości brzegowej 11),
- 11, 12, 13 (dla wartości brzegowej 12).

Zatem w sumie należy przetestować wartości 0, 1, 4, 5, 6, 7, 10, 11, 12, 13. Ponieważ jednak wiemy, że wartości 0, 5, 6, 11 i 12 są już w naszym zestawie testowym (bo jest zapewnione pokrycie AWB dla wersji dwupunktowej), brakującymi wartościami są 1, 4, 7, 10, 13. Stąd odpowiedź D jest poprawna.

Pytanie 4.7

(FL-4.2.2, K3)

Poprawna odpowiedź: D

Osiągnięcie pełnego pokrycia wartości brzegowych jest nieosiągalne. Numery kolejne darmowych myć to wszystkie wielokrotności 10: {10, 20, 30, 40, 50, ...}. Ale pamiętajmy, że aby zastosować metodę AWB, klasy muszą być *spójne*, tzn. nie mogą posiadać „dziur”. Dlatego, gdybyśmy chcieli zastosować AWB do tego problemu, musielibyśmy wyprowadzić nieskończanie wiele klas równoważności:

{1, ..., 9}, {10}, {11, ..., 19}, {20}, {21, ..., 29}, {30}, {31, ..., 39}, {40} itd.,

a to oznaczałoby, że musimy przeprowadzić nieskończanie wiele testów (gdzie mamy nieskończanie wiele wartości brzegowych: 1, 9, 10, 11, 19, 20, 21, 29, 30, 31 itd.).

Uwaga. W takiej sytuacji warto uzgodnić z klientem, jaka jest rozsądna maksymalna liczba myć lub zaproponować np. okres ważności karty.

Zauważmy, że gdybyśmy stosowali metodę podziału na klasy równoważności, problem dałoby się rozwiązać, ponieważ mielibyśmy tylko dwie klasy: liczby podzielne przez 10 i pozostałe liczby. Zatem do osiągnięcia pokrycia klas równoważności wystarczyłyby tylko dwa testy, np. 9 i 10.

Pytanie 4.8

(FL-4.2.3, K3)

Poprawna odpowiedź: A

Wymagania są sprzeczne, jeśli dla zadanej kombinacji warunków możemy wskazać dwa różne zestawy odpowiadających im akcji. W naszym przypadku dwie różne akcje to „wolny przejazd” = T oraz „wolny przejazd” = N. Aby wymusić wartość N, musi być spełniony warunek „student” = T. Aby wymusić wartość T, musi zajść „członek parlamentu” = T lub „osoba niepełnosprawna” = T. Tym samym:

Odpowiedź A jest poprawna. Ta kombinacja pasuje zarazem do reguł R1 i R3, które dają sprzeczne akcje.

Odpowiedź B jest niepoprawna. Ta kombinacja pasuje tylko do reguł R1 i R2, które skutkują taką samą akcją.

Odpowiedź C jest niepoprawna. Ta kombinacja pasuje tylko do kolumny R3, więc nie może być mowy o żadnej sprzeczności w ramach jednego testu.

Odpowiedź D jest niepoprawna. Nie pasuje ani do reguły R1, ani do R2, ani do R3. Jest to więc przykład brakującego wymagania, ale nie wymagań sprzecznych.

Pytanie 4.9

(FL-4.2.3, K3)

Poprawna odpowiedź: C

Liczba kolumn w pełnej tablicy decyzyjnej jest równa liczbie wszystkich możliwych kombinacji warunków. Ponieważ mamy trzy warunki, każdy, odpowiednio, z trzema, dwoma, dwoma możliwymi wyborami, zatem liczba wszystkich możliwych kombinacji tych warunków to $3 \cdot 2 \cdot 2 = 12$. Wypisane są one w tabeli 7.1.

TABELA 7.1. Kombinacje warunków

KOMBINACJA	WIEK	ZAMIESZKANIE	ZAROBKI
1	do 18	miasto	do 4000/mc
2	do 18	miasto	od 4001/mc
3	do 18	wieś	do 4000/mc
4	do 18	wieś	od 4001/mc
5	19 – 40	miasto	do 4000/mc
6	19 – 40	miasto	od 4001/mc
7	19 – 40	wieś	do 4000/mc
8	19 – 40	wieś	od 4001/mc
9	od 41	miasto	do 4000/mc
10	od 41	miasto	od 4001/mc
11	od 41	wieś	do 4000/mc
12	od 41	wieś	od 4001/mc

Pytanie 4.10

(FL-4.2.4, K3)

Poprawna odpowiedź: A

Ponieważ mamy trzy stany i cztery zdarzenia, istnieje $3 \cdot 4 = 12$ możliwych kombinacji (stan, zdarzenie). Tablica przejść zawiera tylko cztery z nich (to znaczy, w maszynee istnieją tylko cztery poprawne przejścia). Zatem przejść niepoprawnych jest $12 - 4 = 8$. Oto one (w nawiasach umieszczone są kolejno stan i zdarzenie):

1. (Początkowy, LoginOK).
2. (Początkowy, LoginBłędny).
3. (Początkowy, Wyloguj).
4. (Logowanie, Loguj).

5. (Logowanie, Wyloguj).
6. (Zalogowany, Loguj).
7. (Zalogowany, LoginOK).
8. (Zalogowany, LoginBłędny).

Żadna z tych kombinacji nie występuje na liście przejść poprawnych podanej w pytaniu.

Pytanie 4.11

(FL-4.2.4, K3)

Poprawna odpowiedź: B

Zauważmy, że żadne dwa z następujących trzech przejść nie mogą wystąpić w ramach jednego przypadku testowego:

- S0 (E2) S3,
- S1 (E1) S4,
- S2 (E2) S5.

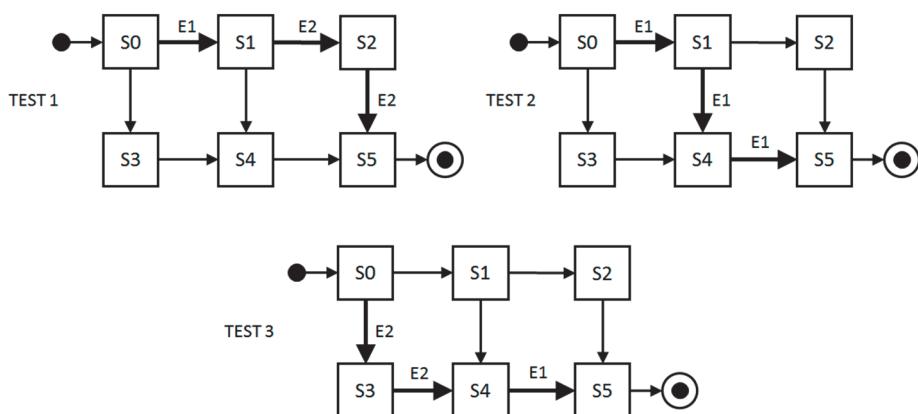
To oznacza, że potrzebujemy co najmniej trzech przypadków testowych, aby pokryć wszystkie przejścia. W istocie trzy testy wystarczą, na przykład:

Test 1. S0 (E1) S1 (E2) S2 (E2) S5.

Test 2. S0 (E1) S1 (E1) S4 (E1) S5.

Test 3. S0 (E2) S3 (E2) S4 (E1) S5.

Ścieżki wyznaczone przez te przypadki testowe pokazane są na rysunku 7.1. Zauważmy, że każde przejście (strzałka) jest pokryte przez przynajmniej jeden przypadek testowy.



RYSUNEK 7.1. Ścieżki wyznaczone przez testy pokrywające wszystkie przejścia (pytanie 4.11).

Pytanie 4.12

(FL-4.3.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Kryterium pokrycia gałęzi subsumuje kryterium pokrycia instrukcji, ale nie na odwrót. Na przykład dla kodu:

```
1. IF (x==0) THEN  
2.     x := x + 1  
3. RETURN x
```

jeden przypadek testowy (dla $x = 0$) spowoduje przejście po ścieżce 1, 2, 3 i w związku z tym osiągnięcie 100% pokrycia instrukcji, ale test ten pokrywa jedynie dwie z trzech gałęzi: (1, 2) oraz (2, 3). Gałąź (1, 3), która wykonana będzie w sytuacji, gdy na wejściu x jest różne od zera, będzie niepokryta.

Odpowiedź B jest niepoprawna. Powyższy przykład kodu to pokazuje. Przypadek testowy ($x = 0$) osiąga 100% pokrycia kodu, ale powoduje, że decyzja w instrukcji 1 przyjmuje tylko wartość prawdy. Nie mamy testu, który wymuszałby wartość fałszu dla tej decyzji.

Odpowiedź C jest niepoprawna. Program może zwrócić dowolną liczbę. Test ($x = 0$) osiąga 100% pokrycia instrukcji, ale wymusza jedynie zwrócenie wartości $x = 1$.

Odpowiedź D jest poprawna. Pokrycie instrukcji wymusza wykonanie każdej instrukcji kodu, więc w szczególności oznacza to wykonanie każdej instrukcji zawierającej defekt. Oczywiście *nie* oznacza to wywołania każdej awarii spowodowanej tymi defektami, ponieważ wykonanie niepoprawnej instrukcji może nie wywołać żadnych negatywnych skutków. Na przykład wykonanie instrukcji $x := a / b$ będzie zupełnie poprawne, o ile mianownik (b) nie będzie przyjmował wartości zera.

Pytanie 4.13

(FL-4.3.2, K2)

Poprawna odpowiedź: C

Pokrycie gałęzi wymaga, aby testy pokryły każdy możliwy przepływ sterowania pomiędzy instrukcjami kodu, tzn. wszystkie możliwe gałęzie w kodzie — zarówno bezwarunkowe, jak i warunkowe. Kod w przykładzie ma strukturę liniową i każde jego uruchomienie spowoduje wykonanie instrukcji w kolejności: 1, 2, 3. Oznacza to, że w każdym uruchomieniu tego programu pokryte zostaną wszystkie dwie występujące w nim gałęzie bezwarunkowe: (1, 2) oraz (2, 3). Stąd poprawną odpowiedzią jest C: wystarczy jeden przypadek testowy z dowolnymi danymi wejściowymi x, y.

Pytanie 4.14

(FL-4.3.3 (K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. To fundamentalna własność i zaleta technik biało-skrzynkowych. W podejściu tym testy projektuje się *bezpośrednio* na podstawie struktury tego, co będzie testowane (np. kodu źródłowego), a więc do samego ich projektowania nie jest konieczna pełna, dokładna specyfikacja.

Odpowiedź B jest niepoprawna. Techniki biało-skrzynkowe nie zawsze wymagają umiejętności programistycznych. Można je stosować np. na poziomie testów systemowych, gdzie pokrywaną strukturą jest np. menu programu.

Odpowiedź C jest niepoprawna. Nie ma związku między metrykami pokrycia technik czarnoskrzynkowych i biało-skrzynkowych.

Odpowiedź D jest niepoprawna. Nie ma bezpośredniego związku między pokryciem biało-skrzynkowym a ryzykiem, ponieważ poziom ryzyka zależy w szczególności od jego wpływu, a nie tylko od tego, ile linii kodu implementuje funkcję związaną z określonym ryzykiem.

Pytanie 4.15

(FL-4.4.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. W dokumencie nie ma informacji o wartościach brzegowych.

Odpowiedź B jest niepoprawna. W testowaniu opartym na liście kontrolnej sprawdzane są pozytywne cechy oprogramowania, a w dokumencie mowa jest o możliwych defektach.

Odpowiedź C jest niepoprawna. To nie są przypadki użycia.

Odpowiedź D jest poprawna. Występujący w pytaniu dokument to lista zawierająca możliwe defekty bądź awarie. Takie listy stosuje się w technice ataków usterkowych, czyli w sformalizowanym podejściu do zgadywania błędów.

Pytanie 4.16

(FL-4.4.2, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Testowanie eksploracyjne wykorzystuje wiedzę, umiejętności, intuicję i doświadczenie testera, ale daje mu pełne pole manewru, jeśli idzie o repertuar technik, jakie może stosować w ramach sesji testowania eksploracyjnego.

Odpowiedź B jest niepoprawna. Patrz uzasadnienie odpowiedzi A.

Odpowiedź C jest niepoprawna. Patrz uzasadnienie odpowiedzi A.

Odpowiedź D jest niepoprawna. Chociaż formalne techniki testowania są dozwolone (patrz uzasadnienie odpowiedzi A), wyjaśnienie w tej odpowiedzi jest niepoprawne. W podejściu eksploracyjnym nie jest konieczne posiadanie podstawy testów, potrzebnej do wyprowadzania przypadków testowych.

Pytanie 4.17

(FL-4.4.3, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Chociaż listy kontrolne mogą zawierać elementy dotyczące aspektów niefunkcjonalnych, to nie jest to główna zaleta stosowania list kontrolnych.

Odpowiedź B jest niepoprawna. W podejściu opartym na doświadczeniu, takim jak listy kontrolne, nie da się precyzyjnie zdefiniować sensownych miar pokrycia, zwłaszcza miar dotyczących pokrycia kodu.

Odpowiedź C jest niepoprawna. Użycie list kontrolnych niekoniecznie wymaga wiedzy eksperckiej (zwłaszcza jeśli lista ma wysoki poziom szczegółowości) — to podejście pasuje bardziej do testowania eksploracyjnego.

Odpowiedź D jest poprawna. W przypadku braku szczegółowych przypadków testowych testowanie oparte na listach kontrolnych może zapewnić pewien stopień spójności dla testów.

Pytanie 4.18

(FL-4.5.1, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Tester nie może sam decydować o tym, jak będzie wyglądało kryterium akceptacji. Historyjki pisane są w oparciu o współpracę właściciela produktu, programisty i testera.

Odpowiedź B jest niepoprawna. To rozwiązywanie nie ma sensu. Po pierwsze, planowanie historyjki to nie jest moment na pisanie testów. Po drugie, testy akceptacyjne muszą być tworzone na podstawie ustalonych i precyzyjnych kryteriów akceptacji, a na razie zespół jest w fazie negocjowania tych kryteriów i nie wiadomo jeszcze, jaką postać przyjmą.

Odpowiedź C jest niepoprawna. Scenariusz nie mówi nic o wydajności, ale nawet gdyby ten temat pojawił się w dyskusji, to nie można z niej wyłączać właściciela produktu.

Odpowiedź D jest poprawna. To wzorcowy przykład negocjowania historyjki użytkownika przez wszystkich członków zespołu w ramach podejścia do testowania opartego na współpracy.

Pytanie 4.19

(FL-4.5.2, K2)

Poprawne odpowiedzi: B, C

Odpowiedź A jest niepoprawna. Kryterium akceptacji może dotyczyć aspektu nie-funkcjonalnego takiego jak wydajność, ale użyte w tym kryterium pojęcie „odpowiednio szybko” jest nieprecyzyjne, a więc nietestowalne.

Odpowiedź B jest poprawna. To jest pożądany mechanizm oferowania funkcjonalności w tym oprogramowaniu: użytkownik może zamawiać zakupy dopiero, gdy jest zarejestrowany. To jest precyzyjne i testowalne kryterium, bezpośrednio związane z treścią historyjki.

Odpowiedź C jest poprawna. Kryterium akceptacji może brać pod uwagę „negatywne” zdarzenia, takie jak np. popełnienie błędu przez użytkownika podczas procesu rejestracji, który może skutkować odmową dalszego procedowania. To jest precyzyjne i testowalne kryterium, bezpośrednio związane z treścią historyjki.

Odpowiedź D jest niepoprawna. Choć jest to sensowne, precyzyjne i testowalne kryterium akceptacji, nie dotyczy bezpośrednio historyjki ze scenariusza. Jest bowiem napisane z punktu widzenia operatora systemu, a nie klienta sklepu.

Odpowiedź E jest niepoprawna. To jest przykład reguły dotyczącej pisania kryteriów akceptacji, a nie konkretne kryterium akceptacji dla historyjki podanej w scenariuszu.

Pytanie 4.20

(FL-4.5.3, K3)

Poprawna odpowiedź: B

Test nr 1 jest niezgodny z regułą biznesową. Pierwszy warunek jest spełniony (czas najdłużej przetrzymywanej książki nie przekracza 30 dni), ale po wypożyczeniu dwóch nowych książek, mając już wypożyczone trzy inne, razem student będzie miał wypożyczonych pięć książek. Nie przekroczy to limitu z drugiego warunku reguły biznesowej. System zatem powinien pozwolić na wypożyczenie, a w teście nr 1 decyzja brzmi: „nie pozwala”.

Test nr 2 jest zgodny z regułą biznesową: student nie przetrzymuje żadnej z czterech wypożyczonych książek i chce wypożyczyć jedną nową, zatem w sumie będzie miał pięć wypożyczonych książek. Nie przekroczy limitu, więc system pozwala na wypożyczenie.

Test nr 3 jest zgodny z regułą biznesową: pracownik ma co najmniej jedną przetrzymaną książkę ($Dni > 30$), zatem system nie może pozwolić na wypożyczenie.

Test nr 4 jest zgodny z regułą biznesową: pracownik ma czyste konto i chce wypożyczyć sześć książek, co mieści się w limicie dziesięciu książek. System powinien pozwolić na wypożyczenie.

Stąd w testach tylko jeden test jest niepoprawny, zatem poprawna odpowiedź to B.

Rozdział 5.

Pytanie 5.1

(FL-5.1.1, K2)

Poprawna odpowiedź: A

Strategia testów jest dokumentem nadrzędnym w stosunku do planu testów, nie jest więc jego częścią. Wszystkie pozostałe elementy: ograniczenia budżetowe, zakres testowania oraz rejestr ryzyk są częściami planu testów. Tym samym poprawna odpowiedź to A.

Pytanie 5.2

(FL-5.1.2, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Zgodnie z syllabusem szczegółowa analiza ryzyka dla historyjek użytkownika jest wykonywana podczas planowania iteracji, a nie podczas planowania wydania.

Odpowiedź B jest niepoprawna. Zgodnie z syllabusem identyfikacja niefunkcjonalnych aspektów systemu do przetestowania jest wykonywana podczas planowania iteracji, a nie podczas planowania wydania.

Odpowiedź C jest niepoprawna. Zgodnie z syllabusem szacowanie wysiłku testowego dla nowych cech planowanych w danej iteracji jest wykonywane podczas planowania iteracji, a nie podczas planowania wydania.

Odpowiedź D jest poprawna. Zgodnie z syllabusem podczas planowania wydania testerzy zaangażowani są w tworzenie testowalnych historyjek użytkownika i ich kryteriów akceptacji.

Pytanie 5.3

(FL-5.1.3, K2)

Poprawna odpowiedź: D

Kryteria wejścia obejmują w szczególności kryteria dostępności oraz początkowy poziom jakości obiektu testów. Dlatego kryteriami wejścia są dostępność testerów (i) oraz zaliczenie wszystkich testów dymnych (iv). Z kolei typowymi kryteriami wyjścia są miary staranności. Dlatego brak defektów krytycznych (ii) oraz osiągnięcie określonego progu pokrycia testowego (iii) to kryteria wyjścia. Tym samym poprawna odpowiedź to D.

Pytanie 5.4

(FL-5.1.4, K3)

Poprawna odpowiedź: B

Zespół chce obliczyć E(5), do czego będzie potrzebował wartości E(4), E(3) i E(2). Ponieważ wartość E(4) jest nieznana, najpierw zespół musi oszacować E(4), a następnie E(5). Zgodnie ze wzorem na ekstrapolację wysiłku mamy:

$$E(4) = (1/3) \cdot (E(3) + E(2) + E(1)) = (1/3) \cdot (12 + 15 + 18) = 15.$$

Zatem $E(4) = 15$. Teraz możemy dokonać ekstrapolacji wysiłku w iteracji piątej:

$$E(5) = (1/3) \cdot (E(4) + E(3) + E(2)) = (1/3) \cdot (15 + 18 + 15) = 16.$$

Zatem $E(5) = 16$. To oznacza, że poprawną odpowiedzią jest B.

Pytanie 5.5

(FL-5.1.5, K3)

Poprawna odpowiedź: C

Pierwszym wykonanym przypadkiem będzie ten osiągający największe pokrycie, czyli PT1, który pokrywa cztery z siedmiu funkcji: A, B, C, F. Drugim w kolejności będzie ten, który pokrywa najwięcej z niepokrytych dotąd funkcji (czyli funkcje D, E, G). Każdy z testów PT2, PT3, PT4 pokrywa tylko jedną z tych dodatkowych funkcji, natomiast PT5 pokrywa dwie dodatkowe, niepokryte dotąd funkcje: D oraz G. Zatem PT5 będzie uruchomiony jako drugi w kolejności. Pierwsze dwa przypadki, PT1 i PT5, pokrywają łącznie sześć z siedmiu funkcji: A, B, C, D, F, G. Trzecim w kolejności testem będzie ten, który pokrywa najwięcej z niepokrytych do tej pory funkcji (czyli funkcję E). Takim testem jest tylko PT4, zatem on zostanie wykonany jako trzeci. Tym samym poprawną odpowiedzią jest C.

Pytanie 5.6

(FL-5.1.6, K1)

Poprawna odpowiedź: C

Odpowiedź A jest niepoprawna. Nakład pracy zespołu nie ma nic wspólnego z pojęciem piramidy testów.

Odpowiedź B jest niepoprawna. Piramida testów nie odnosi się do zadań projektowych, ale bezpośrednio do testów na różnych poziomach testowania.

Odpowiedź C jest poprawna. Piramida testów ilustruje fakt, że mamy bardziej „ziarniste” (szczegółowe) testy na niższych poziomach testowania, a więc zazwyczaj i więcej tych testów, ponieważ każdy z nich osiąga relatywnie niskie pokrycie. Im wyższy poziom, tym niższa ziarnistość testów i zazwyczaj malejąca ich liczba, ponieważ zazwyczaj pojedynczy test na wyższym poziomie osiąga większe pokrycie niż pojedynczy test na niższym poziomie.

Odpowiedź D jest niepoprawna. Piramida testów nie modeluje wysiłku testowego, modeluje ziarnistość i liczbę testów na poszczególnych poziomach testów.

Pytanie 5.7

(FL-5.1.7, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Zgodnie z modelem kwadrantów testowych testy modułowe (jednostkowe) umieszczone są w kwadrancie zorientowanym na technologię i wspierającym zespół, ponieważ kwadrant ten zawiera testy automatyczne i będące częścią procesu ciągłej integracji. Testy modułowe wchodzą w skład tych właśnie testów.

Odpowiedzi B, C, D są niepoprawne — patrz uzasadnienie poprawnej odpowiedzi.

Pytanie 5.8

(FL-5.2.1, K1)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Prawdopodobieństwo ryzyka i wpływ ryzyka są czynnikami niezależnymi.

Odpowiedź B jest niepoprawna. Patrz uzasadnienie odpowiedzi A.

Odpowiedź C jest niepoprawna. Patrz uzasadnienie odpowiedzi A.

Odpowiedź D jest niepoprawna. Wpływ ryzyka powinno się ocenić, zanim ryzyko wystąpi.

Pytanie 5.9

(FL-5.2.2, K2)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. To skrajny przykład efektu materializacji ryzyka związanego z niepoprawnym działaniem produktu, czyli ryzyka produktowego, nie projektowego.

Odpowiedź B jest poprawna. Wystąpienie ryzyk projektowych często skutkuje problemami związanymi z opóźnieniami w realizacji zadań projektowych.

Odpowiedź C jest niepoprawna. Koszty utrzymania oprogramowania są pochodną wad w produkcie, czyli dotyczą ryzyk produktowych, nie projektowych.

Odpowiedź D jest niepoprawna. Niezadowolenie klienta wynika z wady produkto- wej, więc dotyczy ryzyka produktowego, nie projektowego.

Pytanie 5.10

(FL-5.2.3, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Dążymy do ograniczenia ryzyka; dokonanie przeglądu technicznego, by przyjrzeć się ewentualnym problemom, które są związane z wprowadzaniem przelewów, wydaje się dobrym pomysłem, podobnie jak sprawdzenie,

czy istnieje pokrycie gałęzi warunkowych dla zleceń stałych, których termin wyпадa w sobotę lub w niedzielę.

Odpowiedź B jest niepoprawna. Testy modułowe mogą nie wykazać problemów związanych z wprowadzaniem przelewów — nie wiemy, jak wygląda architektura aplikacji.

Odpowiedź C jest niepoprawna. Użyteczność nie ma związku z realizacją logiki biznesowej aplikacji.

Odpowiedź D jest niepoprawna. Testy białośkrzynkowe nie pasują do ryzyka związanego z realizacją zleceń stałych, gdyż problem dotyczy raczej interfejsu i jego użyteczności.

Pytanie 5.11

(FL-5.2.4, K2)

Poprawna odpowiedź: C

Wykupienie ubezpieczenia przenosi ciężar ryzyka na stronę trzecią, w tym przypadku na ubezpieczyciela. Jest to więc przykład transferu ryzyka, zatem poprawną odpowiedzią jest odpowiedź C.

Pytanie 5.12

(FL-5.3.1, K1)

Poprawna odpowiedź: B

Odpowiedź A jest niepoprawna. Poziom ryzyka rezydualnego jest typową miarą używaną w testowaniu, gdyż wyraża bieżący poziom pozostałego w produkcie ryzyka po przeprowadzeniu cyklu testów.

Odpowiedź B jest poprawna. Pokrycie wymagań przez kod źródłowy nie ma nic wspólnego z testowaniem. Ta miara może co najwyżej reprezentować postęp prac programistycznych, nie testerskich.

Odpowiedź C jest niepoprawna. Liczba znalezionych defektów krytycznych jest bezpośrednio związana z testowaniem.

Odpowiedź D jest niepoprawna. Postęp implementacji środowiska testowego dotyczy ważnej czynności wykonywanej w ramach procesu testowego, jest więc miarą używaną w testowaniu.

Pytanie 5.13

(FL-5.3.2, K2)

Poprawna odpowiedź: D

Odpowiedź A jest niepoprawna. Jest to typowa informacja zawarta w sumarycznym raporcie z testów.

Odpowiedź B jest niepoprawna. Jest to typowa informacja zawarta w sumarycznym raporcie z testów.

Odpowiedź C jest niepoprawna. Jest to typowa informacja zawarta w sumarycznym raporcie z testów.

Odpowiedź D jest poprawna. Zgodnie z sylabusem typową informacją zawartą w raporcie o postępie z testów jest w szczególności testowanie zaplanowane na kolejny okres raportowania. Nie jest to typowa informacja zawarta w sumarycznym raporcie z testów, ponieważ raport ten dotyczy zamkniętego, zakończonego zakresu prac, dla których nie będzie już kolejnych okresów raportowania.

Pytanie 5.14

(FL-5.3.3, K2)

Poprawna odpowiedź: B

Nie ma jednej, najlepszej metody komunikacji. Na przykład formalne raporty czy e-maile nie będą przydatne, gdy zespół musi się komunikować szybko, często i na bieżąco. Z kolei komunikacja werbalna „twarzą w twarz” nie sprawdzi się, gdy zespół jest np. rozproszony i pracuje w wielu różnych krajach (rozumienie komunitów, różnice stref czasowych). Formę komunikacji zawsze należy dobrać indywidualnie, do okoliczności, z uwzględnieniem różnych czynników kontekstowych. Dlatego poprawną odpowiedzią jest B.

Pytanie 5.15

(FL-5.4.1, K2)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. Celem zarządzania konfiguracją jest zapewnienie i utrzymanie integralności modułu/systemu i testaliów oraz wzajemnych relacji między nimi przez cały cykl życia projektu i oprogramowania, co pozwala na wykonanie czynności opisanych w scenariuszu.

Odpowiedź B jest niepoprawna. Analiza wpływu może określić wielkość czy ryzyko zmiany, ale nie pozwala odtworzyć źródłowych produktów pracy na podstawie wersji oprogramowania.

Odpowiedź C jest niepoprawna. Ciągłe dostarczanie oprogramowania pozwala automatyzować proces wydawania oprogramowania, nie zapewnia zaś integralności ani wersjonowania produktów prac.

Odpowiedź D jest niepoprawna. Retrospektywy służą doskonaleniu procesu, nie zapewniają integralności ani wersjonowania produktów prac.

Pytanie 5.16

(FL-5.5.1, K3)

Poprawna odpowiedź: A

Odpowiedź A jest poprawna. W raporcie brak informacji na temat kroków umożliwiających odtworzenie testu.

Odpowiedź B jest niepoprawna. Podano wersję produktu (w formie daty ostatniej komplikacji).

Odpowiedź C jest niepoprawna. Podczas tworzenia raportu o defekcie przy użyciu narzędzia do zarządzania defektami najprawdopodobniej automatycznie nadawany jest status „otwarty”. Ponadto nie jest to tak kluczowa informacja jak ta z odpowiedzi A.

Odpowiedź D jest niepoprawna. Informacje te są przydatne dla testera, ale nie muszą być uwzględnione w raporcie o defekcie.

Rozdział 6.

Pytanie 6.1

(FL-6.1.1, K2)

Poprawna odpowiedź: B

Zgodnie z sylabusem (punkt 6.1.) do narzędzi wspomagających zarządzanie testowaniem i testami zaliczyć można między innymi narzędzia do zarządzania wymaganiami (ponieważ pozwalają one np. zarządzać śledzeniem wymagań do przypadków testowych). Tym samym poprawna jest odpowiedź B.

Pytanie 6.2

(FL-6.1.2, K1)

Poprawne odpowiedzi: C, D

Odpowiedź A jest niepoprawna. Nadmierne uzależnienie od narzędzia to ryzyko.

Odpowiedź B jest niepoprawna. Nadmierne uzależnienie od dostawcy to ryzyko.

Odpowiedź C jest poprawna. Zwiększenie powtarzalności testów to korzyść.

Odpowiedź D jest poprawna. Obiektywna ocena poprzez stosowanie ściśle zdefiniowanych definicji operacyjnych pomiaru to korzyść.

Odpowiedź E jest niepoprawna. Błędy w szacowaniu kosztów utrzymania narzędzia to ryzyko.

ROZDZIAŁ 8.

Odpowiedzi do ćwiczeń

Rozdział 4.

Ćwiczenie 4.1

(FL-4.2.1, K3)

- A) Mamy dwie dziedziny: zespół = {Iron Maiden, Judas Priest, Black Sabbath} oraz typ biletu = {przed sceną, oddalony od sceny}. Każdy element z każdej dziedziny będzie stanowił osobną, jednoelementową klasę równoważności. Zatem podział dziedziny „zespół” wygląda tak: {Iron Maiden}, {Judas Priest}, {Black Sabbath}; podział dziedziny „typ biletu” jest następujący: {przed sceną}, {oddalony od sceny}.
- B) W problemie nie występują klasy niepoprawne, ponieważ sposób wyboru zespołu oraz typu biletu to uniemożliwia (użytkownik wybiera te wartości z predefiniowanych list rozwijalnych). Oczywiście można rozważyć sytuację, w której po wysłaniu formularza zapytanie to zostanie przechwycone i spoparowane, np. podmieniona zostanie nazwa zespołu na nieistniejącą. Tu jednak skupiamy się wyłącznie na testach czysto funkcjonalnych i nie rozważamy zaawansowanych zagadnień z zakresu testów bezpieczeństwa aplikacji.
- C) Zestaw testowy musi pokryć każdą z trzech klas dziedziny „zespół” oraz każdą z dwóch klas dziedziny „typ biletu”. Ponieważ każdy przypadek testowy pokrywa po jednej klasie z każdej z tych dziedzin, wystarczą trzy przypadki testowe, np.

PT1 zespół = Iron Maiden, typ biletu = przed sceną,

PT2 zespół = Judas Priest, typ biletu = oddalony od sceny,

PT3 zespół = Black Sabbath, typ biletu = przed sceną.

Oczywiście dla porządku należy dodać, że w każdym z przypadków testowych trzeba również określić oczekiwane wyjście. W naszym przypadku będzie nim przydzielenie biletu określonego typu na koncert określonego zespołu.

Ćwiczenie 4.2

(FL-4.2.1, K3)

- A) Dziedziną wejściową (poprawną) są liczby naturalne większe od 1. Tę dziedzinę możemy podzielić na dwie klasy: liczby pierwsze i liczby złożone, czyli {2, 3, 5, 7, 11, 13, ...} oraz {4, 6, 8, 9, 10, 12, ...}.

- B) Interfejs nie pozwala wpisać wartości niebędącej liczbą naturalną. Dlatego możemy uznać, że w problemie nie występują klasy niepoprawne. Gdybyśmy jednak byli w stanie np. przechwycić komunikat, który przekazuje systemowi wartość wejściową, i odpowiednio go spreparować, moglibyśmy wtedy wymusić wartość niepoprawną. To, czy jest to możliwe i czy testerzy zdecydują się na takie „hakerskie” podejście, zależy oczywiście od wielu czynników, w szczególności wymaganego poziomu zabezpieczeń aplikacji. Jeśli rozważamy problem tylko z perspektywy użytkownika, możemy bezpiecznie założyć, że program będzie operował wyłącznie na poprawnych, oczekiwanych wartościach.
- C) Ponieważ w problemie nie występują klasy niepoprawne, wystarczą dwa testy, aby pokryć wszystkie klasy równoważności: jeden, w którym rozważamy liczbę pierwszą, i jeden, w którym rozważamy liczbę złożoną, np.:
- PT1: wejście jest liczbą pierwszą (np. 7),
 - PT2: wejście jest liczbą złożoną (np. 12).

Ćwiczenie 4.3

(FL-4.2.2, K3)

Analizowana zmienna to całkowita kwota zakupów. Jest to dodatnia liczba wymierna z dokładnością do dwóch miejsc po przecinku (czyli z dokładnością do 1 gr). Musimy znaleźć te spośród tych wartości, które po zaokrągleniu będą wartościami brzegowymi dla zaokrąglonej kwoty (patrz tabela 8.1). Najmniejszą możliwą wartością wejściową spełniającą warunki zadania jest 0,01 PLN.

TABELA 8.1. Wartości brzegowe przed zaokrągleniem i po nim

WARTOŚCI BRZEGOWE DLA KWOTY ZAOKRĄGLONEJ [PLN]			WARTOŚCI BRZEGOWE DLA KWOTY PRZED ZAOKRĄGLENIEM [PLN]	
Rabat	minimalna	maksymalna	minimalna	maksymalna
0%	1	300	0,01	300
5%	301	800	300,01	800
10%	801	–	800,01	–

Wartość 300 jest *największą* kwotą, która po zaokrągleniu (= 300) da wartość brzegową maksymalną dla 0% rabatu. Wartość 300,01 jest *najmniejszą* kwotą, która po zaokrągleniu (= 301) da wartość brzegową minimalną dla rabatu 5% itd.

Zatem mamy następujące przypadki testowe:

- PT1: kwota = 0,01, oczekiwany wynik: 0% rabatu,
- PT2: kwota = 300, oczekiwany wynik: 0% rabatu,
- PT3: kwota = 300,01, oczekiwany wynik: 5% rabatu,
- PT4: kwota = 800, oczekiwany wynik: 5% rabatu,
- PT5: kwota = 800,01, oczekiwany wynik: 10% rabatu.

Ćwiczenie 4.4

(FL-4.2.2, K3)

- A) Dziedziny odpowiadają użytym zmiennym (parametrom): szerokości, wysokości oraz polu powierzchni.

Klasy dla zmiennej „szerokość”:

- klasa poprawna: {30, 31, ..., 99, 100}.

Klasy dla zmiennej „wysokość”:

- klasa poprawna: {30, 31, ..., 59, 60}.

Klasy dla zmiennej „pole powierzchni” (od jej wartości zależeć będzie cena usługi):

- klasa poprawna dla ceny 450 PLN: {900, 901, ..., 1600},
- klasa poprawna dla ceny 500 PLN: {1601, 1602, ..., 6000}.

Wartości dla „pola powierzchni” wynikają stąd, że minimalne wymiary obrazu to 30 cm szerokości i 30 cm wysokości, zatem minimalne pole powierzchni wynosi $30\text{cm} \cdot 30\text{cm} = 900\text{ cm}^2$. Analogicznie maksymalne wymiary to 100 cm szerokości i 60 cm wysokości, zatem maksymalne pole powierzchni wynosi $60\text{cm} \cdot 100\text{ cm} = 6000\text{ cm}^2$.

Wartości brzegowe:

- dla zmiennej „szerokość”: (S1) 30, (S2) 100,
- dla zmiennej „wysokość”: (W1) 30, (W2) 60,
- dla zmiennej „pole powierzchni”: (P1) 900, (P2) 1600, (P3) 1601, (P4) 6000.

- B) Przypadek testowy reprezentować będziemy jako dwójkę (s, w) , gdzie s i w to, odpowiednio, szerokość i wysokość (wejście). Musimy pokryć testami osiem wartości brzegowych: S1, S2, W1, W2, P1, P2, P3, P4. Zauważmy, że niektóre wartości brzegowe dla pola powierzchni można uzyskać z wymnożenia wartości będących wartościami brzegowymi wysokości i szerokości. Na przykład $900 = 30 \cdot 30$ oraz $6000 = 100 \cdot 60$. Możemy to wykorzystać, minimalizując liczbę przypadków testowych.

Przypadki testowe oraz pokryte wartości brzegowe pokazane są w tabeli 8.2.

TABELA 8.2. Pokryte wartości brzegowe

PT	WEJŚCIE		POLE	POKRYTE WARTOŚCI BRZEGOWE DLA:		
	S	W		SZEROKOŚCI	WYSOKOŚCI	POLA POWIERZCHNI
1	30	30	900	S1	W1	P1
2	100	60	6000	S2	W2	P4
3	40	40	1600	–	–	P2

Zauważmy, że nie da się pokryć wartości brzegowej P3. Liczba 1601 jest bowiem liczbą pierwszą, nie da się więc jej wyrazić jako iloczynu dwóch liczb większych lub równych 30.

Ostatecznie zaprojektowaliśmy trzy przypadki testowe pokrywające siedem z ośmiu zidentyfikowanych wartości brzegowych. Najmniejszą liczbą większą od 1600, którą można reprezentować jako iloczyn dwóch liczb spełniających podane w zadaniu warunki jest $1610 = 35 * 46$. Moglibyśmy zatem dodać czwarty przypadek testowy (35, 46), który testowałby tę „osiągalną wartość brzegową” klasy {1601, ..., 6000}.

Ćwiczenie 4.5

(FL-4.2.3, K3)

- A) Występujące w naszym problemie warunki to „punkty ≥ 85 ” (możliwe wartości: TAK, NIE) oraz „liczba błędów ≤ 2 ” (możliwe wartości: TAK, NIE).
- B) System może podjąć następujące akcje:
 - Przyznać prawo jazdy? (TAK, NIE)
 - Powtórzyć egzamin teoretyczny? (TAK, NIE)
 - Powtórzyć egzamin praktyczny? (TAK, NIE)
 - Dodatkowe lekcje nauki jazdy? (TAK, NIE)
- C) Wszystkie kombinacje warunków pokazane są w górnej części tabeli 8.3. Można je wygenerować metodą „drzewka” opisaną wyżej w niniejszym rozdziale.
- D) Wypełniona w sposób pełny tablica decyzyjna pokazana jest w tabeli 8.3.

TABELA 8.3. Tablica decyzyjna dla systemu wspomagania egzaminów na prawo jazdy

WARUNKI	1	2	3	4
punkty ≥ 85 ?	TAK	TAK	NIE	NIE
błędy ≤ 2 ?	TAK	NIE	TAK	NIE
Akcje				
przyznać prawo jazdy?	TAK	NIE	NIE	NIE
powtórzyć egz. teoretyczny?	NIE	NIE	TAK	TAK
powtórzyć egz. praktyczny?	NIE	TAK	NIE	TAK
dodatkowe lekcje nauki jazdy?	NIE	NIE	NIE	TAK

Można łatwo zauważyc, że poszczególne akcje wynikają wprost z zapisów specyfikacji. Na przykład jeśli liczba punktów za egzamin teoretyczny wynosi 85 lub więcej, a kandydat popełnił co najwyżej dwa błędy (kolumna nr 1), oznacza to, że należy przyznać prawo jazdy (akcja „przyznać prawo jazdy” = TAK) i nie należy powtarzać żadnych egzaminów ani brać dodatkowych lekcji (pozostałe akcje = NIE).

Zauważmy tu pewną subtelność: specyfikacja w tym przypadku mówi wprost jedynie o akcji „przyznać prawo jazdy”, natomiast nie mówi *explicite* o pozostałyach akcjach. Można to oczywiście formalnie potraktować jako niepełność specyfikacji (tu ujawnia się zaleta stosowania tablic decyzyjnych jako forma testowania statycznego!). Można jednak również odwołać się do zdrowego rozsądku: jeśli kandydat otrzymuje prawo jazdy, to logiczne jest, że nie musi powtarzać żadnych egzaminów.

W tego typu sytuacjach tester musi zachować ostrożność: czasami sytuacja jest na tyle jasna, że można uznać, iż wiadomo, jakie powinny być wartości akcji, o których specyfikacja nie wspomina. Czasami jednak problem może być bardziej subtelny. Wtedy oczywiście należy zgłosić problem.

- E) Przykładowe przypadki testowe wygenerowane na podstawie tablicy decyzyjnej mogą wyglądać tak jak poniżej.

Przypadek testowy nr 1 (odpowiadający kolumnie nr 1).

Nazwa: przyznanie prawa jazdy.

Warunki wstępne: kandydat podchodził do egzaminów po raz pierwszy.

Wejście: wynik egzaminu teoretycznego = 85 pkt, liczba popełnionych błędów = 2.

Oczekiwane wyjście: przyznanie prawa jazdy, brak konieczności powtarzania egzaminów, brak konieczności brania dodatkowych lekcji nauki jazdy.

Warunki końcowe: kandydat oznaczony jako kandydat, który już zdawał egzaminy na prawo jazdy.

Przypadek testowy nr 2 (odpowiadający kolumnie nr 2).

Nazwa: zaliczenie egzaminu teoretycznego, oblanie egzaminu praktycznego.

Warunki wstępne: kandydat podchodził do egzaminów po raz pierwszy.

Wejście: wynik egzaminu teoretycznego = 93 pkt, liczba popełnionych błędów = 3.

Oczekiwane wyjście: nieprzyznanie prawa jazdy (egzamin niezdany), konieczność powtórzenia egzaminu praktycznego, brak konieczności brania dodatkowych lekcji nauki jazdy.

Warunki końcowe: kandydat oznaczony jako kandydat, który już zdawał egzaminy na prawo jazdy.

Przypadek testowy nr 3 (odpowiadający kolumnie nr 3).

Nazwa: oblanie egzaminu teoretycznego, zaliczenie egzaminu praktycznego.

Warunki wstępne: kandydat podchodził do egzaminów po raz pierwszy.

Wejście: wynik egzaminu teoretycznego = 84 pkt, liczba popełnionych błędów = 0.

Oczekiwane wyjście: nieprzyznanie prawa jazdy, konieczność powtarzania egzaminu teoretycznego, brak konieczności brania dodatkowych lekcji nauki jazdy.

Warunki końcowe: Kandydat oznaczony jako kandydat, który już zdawał egzaminy na prawo jazdy.

Przypadek testowy nr 4 (odpowiadający kolumnie nr 4)

Nazwa: oblanie obu egzaminów.

Warunki wstępne: kandydat podchodził do egzaminów po raz pierwszy.

Wejście: wynik egzaminu teoretycznego = 42 pkt, liczba popełnionych błędów = 3.

Oczekiwane wyjście: nieprzyniesanie prawa jazdy, konieczność powtórzenia obu egzaminów: teoretycznego i praktycznego, konieczność wzięcia dodatkowych lekcji nauki jazdy.

Warunki końcowe: kandydat oznaczony jako kandydat, który już zdawał egzaminy na prawo jazdy.

Zauważmy, że warunki końcowe nie są tu od rzeczy — być może system ma zupełnie inny zestaw zachowań wobec kandydatów, którzy przystępują do egzaminu ponownie. Na przykład system mógłby wtedy sprawdzać, czy kandydat rzeczywiście odbył dodatkowe lekcje nauki jazdy i byłby to element wejścia do systemu.

Ćwiczenie 4.6

(FL-4.2.3, K3)

Szukana tablica decyzyjna przedstawiona jest w tabeli 8.4.

TABELA 8.4. Tablica decyzyjna dla zadania 4.6

WARUNKI	1	2	3	4	5	6	7	8
Ma złotą kartę?	TAK	TAK	TAK	TAK	NIE	NIE	NIE	NIE
Klasa ekonomiczna pełna?	TAK	TAK	NIE	NIE	TAK	TAK	NIE	NIE
Klasa biznes pełna?	TAK	NIE	TAK	NIE	TAK	NIE	TAK	NIE
Akcje								
Karta pokładowa?	TAK	TAK	TAK	TAK	NIE	TAK	TAK	TAK
Przyznanie miejsce	EKO	BIZ	EKO	BIZ	N/D	BIZ	EKO	EKO
Usunięcie z listy pasażerów?	NIE	NIE	NIE	NIE	TAK	NIE	NIE	NIE

Akcje zostały przypisane na podstawie diagramu z rysunku 4.12. Zauważmy, że akcja „przyznanie miejsce” nie jest zmienną logiczną — jej możliwe wartości to EKO, BIZ oraz N/D (nie dotyczy — symbol oznaczający brak przyznanego miejsca ze względu na usunięcie pasażera z lotu).

Według schematu, gdy klient ma złotą kartę, a klasa biznes jest pełna, należy mu przydzielić miejsce w klasie ekonomicznej. Tej sytuacji odpowiadają kolumny nr 1 i 3

(złota karta = TAK, klasa biznes pełna = TAK). Zwróćmy jednak uwagę na kolumnę nr 1. Opisuje ona sytuację, gdy klasa ekonomiczna również jest pełna. Mimo to system nakazuje przyznać pasażerowi miejsce w tej klasie (patrz komórki z szarym tłem)!

Odkryliśmy poważny błąd w specyfikacji! Nie wiemy, jak ten problem powinien być rozwiązany. Oto dwa przykładowe rozwiązania:

- Usunąć innego pasażera bez złotej karty z klasy ekonomicznej, a jego miejsce przydzielić rozważanemu klientowi (pytanie jednak: co wtedy, gdy każdy pasażer w klasie ekonomicznej ma złotą kartę? Zapewne jest to sytuacja bardzo mało prawdopodobna, ale możliwa — specyfikacja powinna to uwzględnić).
- Dopuszczać do schematu dodatkowy warunek — jeśli klient ma złotą kartę, a klasa biznes jest pełna, należy rozważyć, czy klasa ekonomiczna jest pełna. Jeśli nie, to przydzielamy pasażerowi miejsce w tej klasie, tak jak to opisuje kolumna nr 3 w tablicy. Jeśli jednak klasa ekonomiczna jest pełna, to musimy podjąć inną akcję, np. usunąć pasażera z listy lub zastosować rozwiązanie z poprzedniego punktu.

Ćwiczenie 4.7

(FL-4.2.4, K3)

A) W pierwszym kroku przeanalizujmy, w jakich stanach może przebywać system z punktu widzenia operacji wpisywania kodu PIN. Możemy zidentyfikować następujących osiem stanów (wynikają one w zasadzie wprost z analizy scenariusza):

- Ekran powitalny — początkowy stan systemu, oczekiwanie na włożenie karty.
- Walidacja — stan, w którym system sprawdza poprawność włożonej karty.
- Koniec — stan, do którego system przejdzie po walidacji, jeśli karta jest błędna.
- Pytaj o PIN — stan, w którym system prosi o wprowadzenie PIN po raz pierwszy.
- Pytaj o PIN drugi raz — stan, w którym system prosi o wprowadzenie PIN po raz drugi (po błędym wpisaniu kodu PIN za pierwszym razem).
- Pytaj o PIN trzeci raz — stan, w którym system prosi o wprowadzenie PIN po raz trzeci (po błędym wpisaniu kodu PIN za drugim razem).
- Zalogowany — stan, do którego system przechodzi po poprawnym wpisaniu kodu PIN za pierwszym, drugim lub trzecim razem.
- Karta забlokowana — stan, do którego system przechodzi po trzykrotnym błędym wpisaniu kodu PIN.

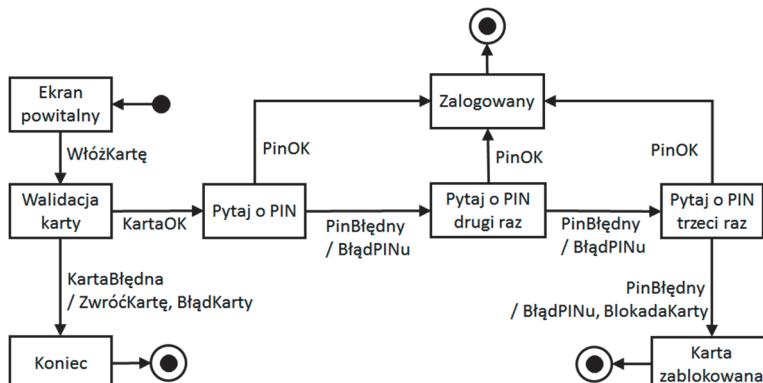
Zauważmy, że w tym modelu maszyny stanowej musimy zdefiniować aż trzy stany związane z oczekiwaniem na wpisanie kodu PIN, ponieważ nasza maszyna stanowa nie ma pamięci — reprezentacją historii dotychczasowych zdarzeń jest jedynie stan, w którym przebywamy. Zatem, aby rozróżnić liczbę niepoprawnie wpisanych kodów PIN, potrzebujemy trzech stanów.

Rozważmy teraz możliwe zdarzenia, jakie mogą zajść w naszym systemie, oraz akcje, jakie system może podjąć w związku z obsługą tych zdarzeń. Jednym z najwygodniejszych sposobów jest analiza poszczególnych stanów i zastanowienie się (na podstawie specyfikacji), co może się zdarzyć, gdy przebywamy w danym stanie. Wyniki naszej analizy prezentuje tabela 8.5.

TABELA 8.5. Możliwe zdarzenia dla poszczególnych stanów maszyny stanowej

STAN	MOŻLIWE ZDARZENIA I AKCJE
Ekran powitalny	WłóżKartę
Walidacja karty	KartaOK (przejście do stanu Pytaj o PIN) KartaBłędna (akcja: komunikat BłądKarty; przejście do stanu Koniec)
Koniec	---
Pytaj o PIN	PinOK (przejście do stanu Zalogowany) PinBłędny (akcja: komunikat BłądPINu; przejście do stanu Pytaj o PIN drugi raz)
Pytaj o PIN drugi raz	PinOK (przejście do stanu Zalogowany) PinBłędny (akcja: komunikat BłądPINu; przejście do stanu Pytaj o PIN trzeci raz)
Pytaj o PIN trzeci raz	PinOK (przejście do stanu Zalogowany) PinBłędny (akcje: komunikat BłądPINu, komunikat BlokadaKarty, przejście do stanu Karta zablokowana)
Zalogowany	---
Karta zablokowana	---

Na podstawie tabeli 8.5 możemy zaprojektować diagram przejść między stanami. Pokazany jest on na rysunku 8.1.

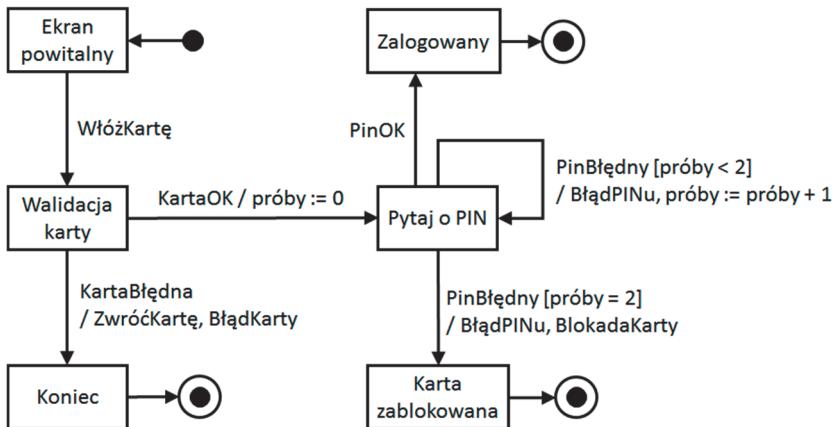


RYSUNEK 8.1. Diagram przejść dla weryfikacji kodu PIN bez użycia warunków dozoru

- B) Diagram przejść z wykorzystaniem warunków dozoru przedstawiony jest na rysunku 8.2. Zauważmy, że dzięki wprowadzeniu warunków dozoru udało nam się zredukować liczbę stanów. Teraz mamy tylko jeden stan związany z pytaniem o kod PIN, a liczba błędnie wpisanych kodów PIN pamiętana jest w zmiennej o nazwie „próby”. To, czy ze stanu „Pytaj o PIN” przejdziemy do stanu „Karta zablokowana” czy „Zalogowany”, zależy od tego, ile razy wpisano błędny PIN. Zauważmy, że pętlą

Pytaj o PIN (PinBłędny) Pytaj o PIN

możemy przejść maksymalnie dwa razy. Za każdym przejściem po tej pętli zwiększa się bowiem o jeden wartość zmiennej „próby”, a warunek dozoru pozwala przejść po tej pętli tylko wtedy, gdy zmienna ta ma wartość mniejszą od 2. Po dwukrotnie błędnie wpisanym kodzie PIN zmienna ta ma wartość 2 i w momencie trzeciej nieudanej próby wpisania kodu PIN warunek dozoru jest fałszywy. Prawdziwy za to staje się warunek dozoru, który stowarzyszony jest z przejściem pod wpływem zdarzenia „BłędnyPIN” do stanu „Karta zablokowana”.



RYSUNEK 8.2. Diagram przejść dla weryfikacji kodu PIN z użyciem warunków dozoru

- C) Zanim przejdziemy do projektowania testów, zastanówmy się, jakie mamy warunki testowe dla obu typów pokryć. W przypadku pokrycia stanów mamy do pokrycia następujące elementy (stany): Ekran powitalny, Walidacja karty, Koniec, Pytaj o PIN, Pytaj o PIN drugi raz, Pytaj o PIN trzeci raz, Zalogowany, Karta zablokowana. Musimy więc pokryć testami osiem elementów pokrycia. W przypadku pokrycia przejść musimy pokryć wszystkie strzałki między stanami. Jest ich dziewięć: WłóżKartę, KartaBłędna, KartaOK, trzy różne przejścia pod wpływem PinOK, trzy różne przejścia pod wpływem PinBłędny. Dla kryterium pokrycia przejść musimy więc pokryć testami dziewięć elementów pokrycia.

Aby pokryć wszystkie stany, zauważmy, że w diagramie z rysunku 8.1 mamy trzy stany końcowe. Potrzebować zatem będziemy co najmniej trzech przypadków testowych, ponieważ osiągnięcie stanu końcowego kończy działanie maszyny. Oto przykładowe przypadki pokrywające wszystkie stany:

PT1: Ekran powitalny (WłóżKartę) Walidacja karty (KartaBłędna) Koniec.

PT2: Ekran powitalny (WłóżKartę) Walidacja karty (KartaOK) Pytaj o PIN (PinOK) Zalogowany.

PT3: Ekran powitalny (WłóżKartę) Walidacja karty (KartaOK) Pytaj o PIN (PinBłędny) Pytaj o PIN drugi raz (PinBłędny) Pytaj o PIN trzeci raz (PinBłędny) Karta zablokowana.

Te trzy przypadki pokrywają wszystkie stany, ale nie pokrywają wszystkich przejść. Dwa nieokryte nimi przejścia to:

Pytaj o PIN drugi raz (PinOK) Zalogowany

Pytaj o PIN trzeci raz (PinOK) Zalogowany

Musimy dodać dwa nowe testy, które pokryją te dwa przejścia, np.:

PT4: Ekran powitalny (WłóżKartę) Walidacja karty (KartaOK) Pytaj o PIN (PinBłędny) Pytaj o PIN drugi raz (PinOK) Zalogowany

PT5: Ekran powitalny (WłóżKartę) Walidacja karty (KartaOK) Pytaj o PIN (PinBłędny) Pytaj o PIN drugi raz (PinBłędny) Pytaj o PIN trzeci raz (PinOK) Zalogowany

Ponownie, zauważmy, że nie jesteśmy w stanie pokryć wszystkich przejść mniej niż pięcioma testami, ponieważ mamy pięć przejść bezpośrednio dochodzących do stanów końcowych. Oznacza to, że wykonanie takiego przejścia kończy przypadek testowy, zatem żadne dwa z tych pięciu przejść nie mogą znajdować się w ramach jednego przypadku testowego.

Ćwiczenie 4.8

(FL-4.2.4, K3)

- A) System posiada trzy stany (W, S, M) i pięć różnych zdarzeń (Cisza!, DajGłos!, Siad!, WidziKota, JestGłaskany). Mamy zatem $3 \cdot 5 = 15$ kombinacji (stan, przejście). Ponieważ z diagramu widać, że jest sześć przejść poprawnych (sześć strzałek pomiędzy stanami), zatem przejścia niepoprawnych jest $15 - 6 = 9$:

- (W, Cisza!)
- (W, Siad!)
- (S, DajGłos!)
- (S, WidziKota)
- (S, Siad!)
- (M, Cisza!)
- (M, DajGłos!)
- (M, JestGłaskany)
- (M, WidziKota)

- B) Przejścia poprawne można przetestować przy użyciu jednego przypadku testowego, np.: W (DajGłos!) S (Cisza!) W (WidziKota) S (JestGłaskany) M (Siad!) W (JestGłaskany) M. Ten przypadek pokrywa wszystkie sześć przejść poprawnych. Ponieważ mamy dziewięć przejść niepoprawnych, musimy dodać dziewięć przypadków testowych, po jednym na każde przejście niepoprawne. Na przykład aby pokryć przejście niepoprawne (M, DajGłos!), możemy zaprojektować przypadek W (JestGłaskany) M (DajGłos!) ?. Po osiągnięciu stanu M próbujemy wywołać (niepoprawne) zdarzenie „DajGłos!”. Jeśli nam się to nie uda lub jeśli system je zignoruje, zakładamy, że test jest zdany. Jeśli jednak system pod wpływem tego zdarzenia przejdzie do innego stanu, uznamy, że test jest niezaliczony, ponieważ system zachował się niezgodnie ze specyfikacją. Przy testowaniu przejść niepoprawnych należy pamiętać, że zawsze rozpoczynamy w stanie początkowym, więc najpierw musimy przejść poprawnymi prześciami, aby osiągnąć żądany stan (w tym przypadku stan M), a następnie podjąć próbę wykonania niepoprawnego zdarzenia (w tym przypadku „DajGłos!”). Analogicznie projektujemy przypadki testowe dla pozostałych przejść niepoprawnych.

Ćwiczenie 4.9

(FL-4.5.3, K3)

Oto kilka przykładowych testów, jakie można zaprojektować.

- Poprawna (udana) rejestracja poprawnym loginem (nieużywanym jeszcze w systemie) i hasłem, np. login *jan.kowalski@poczta.com*, hasło *Abc123Def* wpisane dwukrotnie w oba pola. Oczekiwany wynik: system akceptuje dane i wysyła na adres *jan.kowalski@poczta.com* e-mail z linkiem aktywującym konto (ten test weryfikuje kryteria akceptacji KA1 i KA5).
- Próba rejestracji niepoprawnym syntaktycznie loginem (pokrywa kryterium akceptacji KA1). Zestawy danych testowych:
 - *Jan.kowalski@poczta* (zbyt krótka część po znaku @),
 - *JanKowalski-poczta.com* (brak znaku @),
 - *@poczta.com* (brak tekstu przed znakiem @),
 - *JanKowalski@poczta..com* (dwie następujące po sobie kropki).
- Próba rejestracji poprawnym, ale już istniejącym loginem. Oczekiwany wynik: odmowa rejestracji, brak wysłania e-maila (ten test weryfikuje kryterium akceptacji KA2).
- Próba rejestracji poprawnym loginem, ale złym hasłem. Oczekiwany wynik: odmowa rejestracji, brak wysłania e-maila. Zestawy danych testowych (pokrywających kryteria akceptacji KA3 i KA4):
 - poprawne syntaktycznie hasło, ale inne w obu polach na hasło (np. *Abc123Def* i *aBc123Def*),
 - hasło zbyt krótkie (np. *Ab12*),
 - hasło zbyt długie (np. *ABCD1234abcd1234*),
 - hasło bez cyfr (np. *ABCdef*),
 - hasło bez dużych liter (np. *abc123*).

Rozdział 5.

Ćwiczenie 5.1

(FL-5.1.4, K3)

Ostatnia iteracja pokera dla wartości optymistycznej (a) dała wartości 3, 3, 5, co — zgodnie z opisaną procedurą — oznacza, że eksperci uzyskali konsensus w kwestii wartości optymistycznej. Wynosi ona 3, ponieważ jest to wartość wskazana przez większość ekspertów.

Analogicznie dla wartości średniej eksperci uzyskali konsensus już po pierwszej iteracji. Wynik to 5, ponieważ dwóch z trzech ekspertów wskazało tę wartość.

Dla wartości pesymistycznej eksperci uzyskali konsensus dopiero w trzeciej iteracji. Wynik to 13, ponieważ wskazany został przez wszystkich ekspertów.

Po tych sesjach pokera eksperci ustalili wartości zmiennych, które będą użyte w metodzie trójpunktowej, mianowicie:

- wartość optymistyczna: $a = 3$,
- wartość średnia: $m = 5$,
- wartość pesymistyczna: $b = 13$.

Podstawiając te wartości do wzoru z metody trójpunktowej, otrzymujemy:

$$E = (3 + 4 \cdot 5 + 13) / 6 = 36 / 6 = 6,$$

co oznacza, że estymowany wysiłek wynosi 6 osobodni, z odchyleniem standardowym

$$SD = (b - a) / 6 = (13 - 3) / 6 = 10 / 6 = 1,66.$$

Oznacza to, że końcowy wynik szacowania wynosi $6 \pm 1,66$, czyli mieści się w przedziale od $6 - 1,66$ do $6 + 1,66$, czyli w przedziale między 4,34 a 7,66.

Ćwiczenie 5.2

(FL-5.1.4, K3)

W zakończonym projekcie wysiłek w fazie projektowania wyniósł 20 osobodni (ponieważ 4 osoby wykonały pracę w 5 dni). Analogicznie dla fazy implementacji wysiłek wyniósł $10 \cdot 18 = 180$ osobodni, a dla testowania $4 \cdot 10 = 40$ osobodni. Łącznie wysiłek wyniósł więc 240 osobodni w proporcji projekt : programowanie : testowanie = 1 : 9 : 2.

Niech x = liczba dni pracy projektantów, y = liczba dni pracy programistów. Wtedy $66 - x - y$ jest liczbą dni pracy testerów (bo łącznie projekt ma trwać 66 dni). Gdy weźmiesz się pod uwagę liczbę projektantów, programistów i testerów w nowym projekcie, wysiłek w fazach projektowania, programowania i testowania wynosi więc, odpowiednio, $4x$, $6y$, $2(66 - x - y)$.

Z proporcji mamy $4x : 6y : 2(66 - x - y) = 1 : 9 : 2$. Stąd wynika, że $2 \cdot 4x = 2(66 - x - y)$ oraz że $9 \cdot 4x = 6y$, ponieważ testowanie wymaga dwukrotnie więcej wysiłku niż

projektowanie, a programowanie — dziewięciokrotnie więcej wysiłku niż planowanie. Musimy teraz rozwiązać układ równań:

$$8x = 132 - 2x - 2y$$

$$36x = 6y$$

Z drugiego równania mamy $y = 6x$. Podstawiając do pierwszego równania, otrzymujemy:

$$8x = 132 - 2x - 12x, \text{ czyli } 22x = 132, \text{ skąd obliczamy } x = 132/22 = 6.$$

Podstawiając $x = 6$ do zależności $y = 6x$, otrzymujemy, że $y = 6 \cdot 6 = 36$.

Ponieważ x , y , $66 - x - y$ oznaczają, odpowiednio, liczbę dni przeznaczonych na projektowanie, programowanie i testowanie, otrzymujemy ostateczną odpowiedź: wykorzystując metodę proporcji, czyli zakładając, że wysiłek w nowym projekcie rozłoży się proporcjonalnie do wysiłku w projekcie zakończonym:

- na projektowanie musimy przeznaczyć $x = 6$ dni,
- na programowanie musimy przeznaczyć $y = 36$ dni,
- na testowanie musimy przeznaczyć $66 - x - y = 24$ dni.

Ćwiczenie 5.3

(FL-5.2.4, K3)

Gdybyśmy uwzględniali tylko priorytety, kolejność wykonywania przypadków testowych byłaby następująca:

$$001 \rightarrow 003 \rightarrow 002 \rightarrow 004.$$

Musimy jednak brać pod uwagę także zależności logiczne — nie możemy wykonać przypadku testowego 001 przed przypadkami 002 i 003.

Zależności logiczne wymuszają, byśmy zaczęli od przypadku P002 (wprowadzanie danych osobowych), ponieważ jedynie on nie jest zależny od żadnego innego przypadku. Nadal nie możemy wykonać przypadku 001 o najwyższym priorytecie, gdyż jest on zależny nie tylko od 002, ale również od 003. Musimy zatem wykonać w kolejnym kroku przypadek 003, co odblokuje możliwość wykonania przypadku 001. Na samym końcu uruchamiamy przypadek 004.

Zatem ostateczna kolejność to: $002 \rightarrow 003 \rightarrow 001 \rightarrow 004$.

Ćwiczenie 5.4

(FL-5.2.4, K3)

Gdyby opierać się wyłącznie na priorytetach klienta, najpierw powinniśmy zaimplementować i przetestować wymagania Wym2, Wym3 i Wym6 (priorytet wysoki), następnie Wym1 i Wym5 (priorytet średni), a na końcu Wym4 (priorytet niski). Jednak niezależnie od priorytetów pierwszym implementowanym i testowanym wymaganiem musi być Wym1, ponieważ jest jedynym wymaganiem, które nie zależy od innych wymagań. Jedyne wymaganie, które może być implementowane i przetestowane w dalszej kolejności, to Wym3 (ponieważ jako jedyne zależy tylko

od już zaimplementowanego i przetestowanego wymagania, Wym1). Mając Wym1 i Wym3, w dalszej kolejności implementować i testować musimy wymaganie Wym2, gdyż jest jedynym wymaganiem, które zależy od już zaimplementowanych i przetestowanych wymagań.

Zauważmy, że do tego momentu priorytety nie grały żadnej roli. Teraz jednak możemy implementować i testować zarówno Wym4, jak i Wym5. Zwróćmy uwagę, że spośród jeszcze niezaimplementowanych i nieprzetestowanych wymagań wymaganiem o najwyższym dla klienta priorytecie jest Wym6, a ono jest zależne od Wym4. Zatem implementujemy i testujemy najpierw Wym4, aby jak najszybciej „odblokować” możliwość implementacji i testowania wysoko spriorytetyzowanego Wym6. Na samym końcu implementujemy i testujemy Wym5.

Ostateczna kolejność wymagań jest taka:

Wym1, Wym3, Wym2, Wym4, Wym6, Wym5.

Kolejność ta uwzględnia priorytety klienta, z jednoczesnym uwzględnieniem koniecznych zależności logicznych pomiędzy wymaganiami.

Ćwiczenie 5.5

(FL-5.5.1, K3)

Raport o defekcie powinien zawierać przynajmniej informacje opisane w tabeli 8.6.

TABELA 8.6. Zawartość raportu o defekcie

UNIKATOWY IDENTYFIKATOR	34.810
Tytuł, podsumowanie zgłoszanego defektu	Nieprawidłowe wyliczenie kwoty do zapłaty
Data zgłoszenia (= data odkrycia błędu)	07.04.2019
Autor	Tester_AAA
Opis defektu umożliwiający jego odtworzenie i usunięcie	przypadek testowy PT003
Wynik rzeczywisty	48 PLN
Wynik oczekiwany	47,50 PLN
Priorytet usunięcia defektu	Normalny
Opis niezgodności ułatwiający określenie jej przyczyny	Nieprawidłowe zaokrąglenie do pełnych złotych w kwocie „do zapłaty”

CZĘŚĆ IV

OFICJALNY PRZYKŁADOWY EGZAMIN

Niniejsza część podręcznika zawiera przykładowy egzamin. Jest to oficjalny egzamin próbny opublikowany przez ISTQB®. Czas trwania egzaminu to 60 minut, a w przypadku, gdy egzamin nie jest zdawany w języku ojczystym — 75 minut.

Egzamin

Pytanie nr 1 (1 punkt)

Która z poniższych odpowiedzi opisuje poprawny cel testów?

- A. Udowodnienie, że w systemie podlegającym testowaniu nie występują żadne nieusunięte defekty.
- B. Udowodnienie, że po wprowadzeniu systemu do eksploatacji nie będą występować żadne awarie.
- C. Obniżenie poziomu ryzyka związanego z przedmiotem testów i zwiększenie zaufania do jego jakości.
- D. Sprawdzenie, czy nie pozostały żadne nieprzetestowane kombinacje danych wejściowych.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 2 (1 punkt)

Które z poniższych stwierdzeń zawiera przykład czynności testowych, które przy czynią się do powodzenia projektu?

- A. Zaangażowanie testerów w różne działania podejmowane w ramach cyklu wytwarzania oprogramowania pomaga wykryć defekty w produktach pracy.
- B. Testerzy starają się nie przeszkadzać programistom na etapie pisania kodu, aby umożliwić im tworzenie kodu o wyższej jakości.
- C. Współpraca testerów z użytkownikami pozwala podnieść jakość raportów o defektach podczas testowania integracji modułów i testowania systemowego.
- D. Certyfikowani testerzy projektują znacznie lepsze przypadki testowe niż testerzy, którzy nie posiadają certyfikatu.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 3 (1 punkt)

Rozpoczynasz pracę jako tester w zespole opracowującym nowy system metodą przyrostową. Zauważasz, że od kilku iteracji nie wprowadzono żadnych zmian w przypadkach testowych używanych do testowania regresji, a także nie zidentyfikowano żadnych nowych defektów związanych z regresją. Twój kierownik jest zadowolony — w przeciwieństwie do ciebie. Która zasada testowania uzasadnia twój sceptyczyzm?

- A. Testy ulegają zużiciu.
- B. Przekonanie o braku defektów jest błędem.
- C. Defekty mogą się kumulować.
- D. Testowanie gruntowne jest niemożliwe.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 4 (1 punkt)

Pracujesz w zespole **tworzącym** aplikację mobilną do zamawiania posiłków. Zespół postanowił, że w bieżącej iteracji będzie zaimplementowana funkcjonalność obsługi płatności.

Która z wymienionych czynności jest elementem analizy testów?

- A. Oszacowanie, że testowanie integracji z usługą płatniczą potrwa 8 osobodni.
- B. Podjęcie decyzji, że zespół powinien przetestować, czy istnieje możliwość prawidłowego podziału płatności między kilku użytkowników.
- C. Zastosowanie metody analizy wartości brzegowych w celu opracowania danych testowych na potrzeby przypadków testowych, które sprawdzają prawidłowość przetwarzania płatności w minimalnej dozwolonej kwocie.
- D. Przeanalizowanie rozbieżności między rzeczywistym a oczekiwanym rezultatem po wykonaniu przypadku testowego sprawdzającego przetwarzanie płatności kartą kredytową, a następnie zgłoszenie defektu.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 5 (1 punkt)

Które z poniższych czynników (i–v) mają ISTOTNY wpływ na proces testowy?

- i. Cykl wytwarzania oprogramowania.
 - ii. Liczba defektów wykrytych w poprzednich projektach.
 - iii. Zidentyfikowane ryzyka produktowe.
 - iv. Nowe wymagania wynikające z przepisów.
 - v. Liczba certyfikowanych testerów w organizacji.
- A. Czynniki i oraz ii mają istotny wpływ, a czynniki iii, iv oraz v go nie mają.
 - B. Czynniki i, iii oraz iv mają istotny wpływ, a czynniki ii oraz v go nie mają.
 - C. Czynniki ii, iv oraz v mają istotny wpływ, a czynniki i oraz iii go nie mają.
 - D. Czynniki iii oraz v mają istotny wpływ, a czynniki i, ii oraz iv go nie mają.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 6 (1 punkt)

Wskaż DWA zdania, które są przypisane GŁÓWNIE do roli związanej z testowaniem.

- A. Konfigurowanie środowiska testowego.
- B. Prowadzenie backlogu produktu.
- C. Projektowanie rozwiązań zgodnie z nowymi wymaganiami.
- D. Tworzenie planu testów.
- E. Raportowanie na temat uzyskanego pokrycia.

Wybierz DWIE odpowiedzi.

Pytanie nr 7 (1 punkt)

Które z poniższych umiejętności (i–v) są NAJWAŻNIEJSZYMİ umiejętnościami testera?

- i. Posiadanie wiedzy merytorycznej.
 - ii. Tworzenie wizji produktu.
 - iii. Umiejętność sprawnej pracy w zespole.
 - iv. Planowanie i organizowanie pracy zespołu.
 - v. Krytyczne myślenie.
-
- A. Umiejętności ii oraz iv są ważne; umiejętności i, iii oraz v nie są istotne.
 - B. Umiejętności i, iii oraz v są ważne; umiejętności ii oraz iv nie są istotne.
 - C. Umiejętności i, ii oraz v są ważne; umiejętności iii oraz iv nie są istotne.
 - D. Umiejętności iii oraz iv są ważne; umiejętności i, ii oraz v nie są istotne.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 8 (1 punkt)

W jaki sposób podejście „cały zespół” uwidacznia się w kontaktach pomiędzy testerami a przedstawicielami jednostek biznesowych?

- A. Przedstawiciele jednostek biznesowych decydują o podejściu do automatyzacji testów.
- B. Testerzy pomagają przedstawicielom jednostek biznesowych w określaniu strategii testów.
- C. Przedstawiciele jednostek biznesowych nie są objęci podejściem opartym na zaangażowaniu całego zespołu (podejście „cały zespół”).
- D. Testerzy pomagają przedstawicielom jednostek biznesowych w tworzeniu odpowiednich testów akceptacyjnych.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 9 (1 punkt)

Zastanów się nad następującą zasadą: „Do każdej czynności związanej z wytwarzaniem oprogramowania powinna być przypisana odpowiadająca jej czynność testowa”. W których modelach cyklu wytwarzania oprogramowania obowiązuje ta zasada?

- A. Tylko w sekwencyjnych modelach wytwarzania oprogramowania.
- B. Tylko w iteracyjnych modelach wytwarzania oprogramowania.
- C. Tylko w iteracyjnych i przyrostowych modelach wytwarzania oprogramowania.
- D. W sekwencyjnych, przyrostowych i iteracyjnych modelach wytwarzania oprogramowania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 10 (1 punkt)

Która z poniższych odpowiedzi NAJLEPIEJ opisuje wytwarzanie sterowane testami akceptacyjnymi (ATDD)?

- A. W ATDD kryteria akceptacji są zwykle tworzone w formacie Given/When/Then.
- B. W ATDD przypadki testowe są zwykle tworzone na etapie testowania modułowego i są ukierunkowane na kod.
- C. W ATDD testy są tworzone na podstawie kryteriów akceptacji i określają sposób tworzenia związanego z nimi oprogramowania.
- D. W ATDD testy są tworzone na podstawie pożądanego zachowania oprogramowania, co ułatwia członkom zespołu ich zrozumienie.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 11 (1 punkt)

Która z poniższych odpowiedzi NIE jest przykładem przesunięcia w lewo (ang. *shift-left*)?

- A. Dokonywanie przeglądu wymagań użytkowników przed ich formalnym zaakceptowaniem przez interesariuszy.
- B. Pisanie testu modułowego przed napisaniem odpowiadającego mu kodu.
- C. Wykonywanie testu wydajności modułu w trakcie testowania modułowego.
- D. Pisanie skryptu testowego przed ustanowieniem procesu zarządzania konfiguracją.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 12 (1 punkt)

Którego z argumentów użyć, aby przekonać kierownika do organizowania retrospektyw na zakończenie każdego cyklu przekazywania oprogramowania do eksploatacji?

- A. Retrospektwy są obecnie bardzo popularne, dlatego klienci byliby zadowoleni, gdybyśmy uwzględnili je w naszych procesach.
- B. Organizowanie retrospektyw przyniesie organizacji wymierne oszczędności, ponieważ przedstawiciele użytkowników nie przekazują natychmiastowych informacji zwrotnych na temat produktu.
- C. Słabe punkty procesów zidentyfikowane podczas retrospektwy można przeanalizować, a następnie wykorzystać do opracowania listy działań, które będą podejmowane w ramach prowadzonego przez organizację programu ciągłego doskonalenia procesów.
- D. Retrospektwy realizują pięć wartości (do których należą między innymi odwaga i szacunek), które są kluczowe dla utrzymania procesu ciągłego doskonalenia w organizacji.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 13 (1 punkt)

Które typy awarii (1–4) NAJLEPIEJ odpowiadają poszczególnym poziomom testów (A–D)?

- 1. Awarie związane z zachowaniem systemu, polegające na niezgodności z potrzebami biznesowymi użytkownika.
 - 2. Awarie dotyczące wymiany danych między modułami.
 - 3. Awarie dotyczące logiki danego modułu.
 - 4. Awarie związane z nieprawidłowo zaimplementowanymi regułami biznesowymi.
-
- A. Testowanie modułowe.
 - B. Testowanie integracji modułów.
 - C. Testowanie systemowe.
 - D. Testowanie akceptacyjne.
-
- a. 1D, 2B, 3A, 4C
 - b. 1D, 2B, 3C, 4A
 - c. 1B, 2A, 3D, 4C
 - d. 1C, 2B, 3A, 4D

Wybierz JEDNĄ odpowiedź.

Pytanie nr 14 (1 punkt)

Testujesz historyjkę użytkownika, z którą są związane trzy kryteria akceptacji: KA1, KA2 i KA3. Kryterium KA1 jest pokryte przez przypadek testowy PT1, kryterium KA2 przez przypadek PT2, a kryterium KA3 przez przypadek PT3. Z historii wykonywania testów wynika, że wykonano trzy przebiegi testów w trzech kolejnych wersjach oprogramowania:

	PRZEBIEG 1	PRZEBIEG 2	PRZEBIEG 3
PT1	(1) Niezaliczony	(4) Zaliczony	(7) Zaliczony
PT2	(2) Zaliczony	(5) Niezaliczony	(8) Zaliczony
PT3	(3) Niezaliczony	(6) Niezaliczony	(9) Zaliczony

Testy są powtarzane dopiero po otrzymaniu informacji, że wszystkie defekty wykryte w ramach poprzednich przebiegów testów zostały usunięte i że dostępna jest nowa wersja oprogramowania.

Które z powyższych testów są wykonywane jako testy regresji?

- A. Tylko 4, 7, 8 i 9.
- B. Tylko 5 i 7.
- C. Tylko 4, 6, 8 i 9.
- D. Tylko 5 i 6.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 15 (1 punkt)

Które z poniższych stwierdzeń NIE opisuje korzyści wynikające z testowania statycznego?

- A. Obniżenie kosztów zarządzania defektami z uwagi na łatwość wykrywania defektów na późniejszych etapach cyklu wytwarzania oprogramowania.
- B. Niższe koszty usuwania defektów wykrytych podczas testowania statycznego w porównaniu z usuwaniem defektów wykrytych podczas testowania dynamicznego.
- C. Wykrywanie defektów w kodzie, które mogłyby nie zostać wykryte w przypadku wykonania tylko testowania dynamicznego.
- D. Wykrywanie luk i niespójności w wymaganiach.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 16 (1 punkt)

Które z poniższych jest korzyścią wynikającą z wczesnego i częstego otrzymywania informacji zwrotnych?

- A. Usprawnienie procesu testowego na potrzeby przyszłych projektów.
- B. Zmuszanie klientów do ustalania priorytetów wymagań na podstawie uzgodnionych ryzyk.
- C. Jest to jedyny sposób mierzenia jakości zmian.
- D. Możliwość uniknięcia nieporozumień w kwestii wymagań.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 17 (1 punkt)

Przeglądy przeprowadzane w organizacji mają następujące atrybuty:

- wyznaczono osobę pełniąca rolę protokolanta;
- głównym celem jest dokonywanie oceny jakości;
- spotkanie związane z przegladem prowadzi autor produktu pracy;
- uczestnicy przygotowują się indywidualnie;
- sporządzany jest raport z przeglądu.

Który z poniższych typów przeglądu jest NAJPRAWDOPODOBNIEJ stosowany?

- A. Przegląd nieformalny.
- B. Przejrzenie.
- C. Przegląd techniczny.
- D. Inspekcja.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 18 (1 punkt)

Który z wymienionych elementów NIE jest czynnikiem przyczyniającym się do powodzenia przeglądu?

- A. Przeznaczenie przez uczestników odpowiedniej ilości czasu na przeprowadzenie przeglądu.
- B. Podzielenie dużych produktów pracy na mniejsze części w celu zmniejszenia wymaganych nakładów pracy.
- C. Uczestnicy powinni unikać zachowań, które mogłyby wskazywać na znużenie, irytację bądź wrogie nastawienie wobec innych uczestników.
- D. Przyjmowanie do wiadomości, potwierdzanie i rozpatrywanie wykrytych awarii w obiektywny sposób.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 19 (1 punkt)

Która z poniższych cech charakteryzuje techniki testowania oparte na doświadczeniu?

- A. Przypadki testowe są tworzone na podstawie szczegółowych informacji projektowych.
- B. Do mierzenia pokrycia używane są elementy testowane w kodzie implementującym interfejsy.
- C. Stosowane techniki bazują w dużej mierze na wiedzy testera dotyczącej oprogramowania i dziedziny biznesowej.
- D. Za pomocą przypadków testowych identyfikowane są odchylenia od wymagań.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 20 (1 punkt)

Testujesz uproszczony formularz wyszukiwania mieszkań, w którym występują tylko dwa kryteria wyszukiwania:

- kondygnacja (trzy możliwe opcje: parter; pierwsze piętro; drugie lub wyższe piętro);
- typ ogródka (trzy możliwe opcje: brak ogródka; mały ogródek; duży ogródek).

Ogródki są dostępne tylko w przypadku mieszkań na parterze. Formularz zawiera wbudowany mechanizm walidacji, który nie pozwala użyć kryteriów wyszukiwania niezgodnych z powyższą zasadą.

Każdy test ma dwie wartości wejściowe: kondygnacja i typ ogródka. Aby pokryć w swoich testach każdą kondygnację i każdy typ ogródka, chcesz zastosować podział na klasy równoważności.

Jaka jest **minimalna** liczba przypadków testowych potrzebna do uzyskania stu-procentowego pokrycia klas równoważności?

- A. 3.
- B. 4.
- C. 5.
- D. 6.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 21 (1 punkt)

Testujesz system, który oblicza ocenę końcową z przedmiotu dla danego ucznia.

Ocena końcowa jest ustalana na podstawie wyniku końcowego zgodnie z następującymi zasadami:

- 0–50 punktów: ocena niedostateczna;
- 51–60 punktów: ocena dopuszczająca;
- 61–70 punktów: ocena dostateczna;
- 71–80 punktów: ocena dobra;
- 81–90 punktów: ocena bardzo dobra;
- 91–100 punktów: ocena celująca.

Przygotowałaś/przygotowałaś następujący zbiór przypadków testowych:

	WYNIK KOŃCOWY	OCENA KOŃCOWA
PT1	91	celująca
PT2	50	niedostateczna
PT3	81	bardzo dobra
PT4	60	dopuszczająca
PT5	70	dostateczna
PT6	80	dobra

Jakie pokrycie wyniku końcowego można uzyskać metodą dwupunktowej analizy wartości brzegowych przy wykorzystaniu dotychczasowych przypadków testowych?

- A. 50%.
- B. 60%.
- C. 33,3%.
- D. 100%.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 22 (1 punkt)

Twoja ulubiona wypożyczalnia rowerów właśnie wprowadziła nowy system zarządzania relacjami z klientami i poprosiła cię (jako jednego z najbardziej lojalnych klientów) o jego przetestowanie.

W systemie zaimplementowano następujące funkcjonalności:

- Rower może wypożyczyć każdy, ale uczestnicy programu lojalnościowego otrzymują 20% rabatu.
- Niezwrocenie roweru w terminie oznacza brak rabatu.
- Po 15. wypożyczeniu uczestnik programu otrzymuje prezent w postaci koszulki.

Tablica decyzyjna opisująca zaimplementowane funkcje wygląda następująco:

Warunki	R1	R2	R3	R4	R5	R6	R7	R8
Udział w programie	T	T	T	T	F	F	F	F
Niedotrzymanie terminu	T	F	T	F	T	F	F	T
15. wypożyczenie	F	F	T	T	F	F	T	T
Akcje								
Rabat 20%		X		X				
Darmowa koszulka			X	X				X

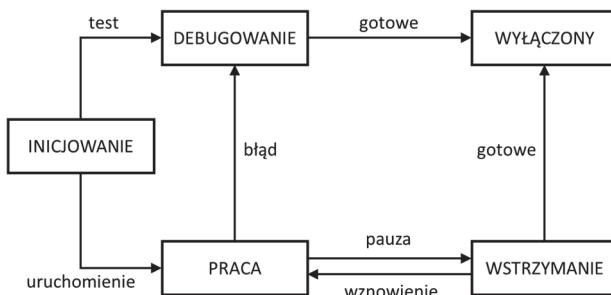
Opierając się WYŁĄCZNIE na opisie funkcji systemu zarządzania relacjami z klientami, wskaź, która z podanych reguł opisuje sytuację niemożliwą.

- A. R4.
- B. R2.
- C. R6.
- D. R8.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 23 (1 punkt)

Testujesz system, którego cykl działania opisano na przedstawionym poniżej diagramie przejść pomiędzy stanami. System rozpoczyna pracę w stanie INICJOWANIE, a kończy pracę w stanie WYŁĄCZONY.



Jaka jest MINIMALNA liczba przypadków testowych niezbędna do uzyskania pokrycia poprawnych przejść?

- A. 4.
- B. 2.
- C. 7.
- D. 3.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 24 (1 punkt)

Twój zestaw testowy osiągnął stuprocentowe pokrycie instrukcji kodu. Co to oznacza w praktyce?

- A. Każda instrukcja zawierająca defekt została wykonana przynajmniej raz.
- B. Dowolny zestaw testowy zawierający więcej przypadków testowych niż twój zestaw również osiągne stuprocentowe pokrycie instrukcji kodu.
- C. Każda ścieżka w kodzie musiała zostać wykonana co najmniej raz.
- D. Każda kombinacja wartości wejściowych musiała zostać przetestowana co najmniej raz.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 25 (1 punkt)

Które z poniższych stwierdzeń NIE jest zgodne z prawdą w odniesieniu do testowania białośkrzynkowego?

- A. Podczas testowania białośkrzynkowego uwzględniana jest cała implementacja oprogramowania.
- B. Miary pokrycia stosowane w testowaniu białośkrzynkowym pomagają wskazać dodatkowe testy umożliwiające zwiększenie pokrycia kodu.
- C. Białośkrzynkowe techniki testowania można stosować w testowaniu statycznym.
- D. Testowanie białośkrzynkowe pozwala rozpoznać luki w implementacji wymagań.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 26 (1 punkt)

Które z poniższych stwierdzeń NAJLEPIEJ opisuje koncepcję zgadywania błędów?

- A. Zgadywanie błędów polega na wykorzystaniu wiedzy i doświadczenia w zakresie dotychczas wykrytych defektów oraz typowych pomyłek popełnianych przez programistów.
- B. Zgadywanie błędów polega na wykorzystaniu własnego doświadczenia w wytwarzaniu oprogramowania i wiedzy na temat pomyłek popełnionych podczas pracy na stanowisku programisty.
- C. Zgadywanie błędów wymaga wyobrażenia sobie, że jest się użytkownikiem przedmiotu testów, i zgadywania, jakie pomyłki mógłby popełnić korzystający z niego użytkownik.
- D. Zgadywanie błędów wymaga szybkiego powielenia zadania związanego z wytwarzaniem oprogramowania w celu zidentyfikowania rodzajów pomyłek, jakie mógłby popełnić programista.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 27 (1 punkt)

W projekcie, w którym uczestniczysz, doszło do opóźnienia w pracach nad zupełnie nową aplikacją, przez co wykonywanie testów również rozpoczęło się zbyt późno. Masz jednak bardzo szczegółową wiedzę merytoryczną i dobre umiejętności analityczne. Chociaż zespół nie otrzymał jeszcze pełnej listy wymagań, kierownictwo domaga się przedstawienia wstępnych wyników testów.

Jaka technika testowania NAJLEPIEJ sprawdzi się w takiej sytuacji?

- A. Testowanie w oparciu o listę kontrolną.
- B. Zgadywanie błędów.
- C. Testowanie eksploracyjne.
- D. Testowanie gałęzi.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 28 (1 punkt)

Które z poniższych stwierdzeń NAJLEPIEJ opisuje sposób, w jaki można udokumentować kryteria akceptacji?

- A. Przeprowadzenie retrospektyny w celu ustalenia rzeczywistych potrzeb interesariuszy dotyczących danej historyjki użytkownika.
- B. Opisanie przykładowego warunku testowego związanego z daną historyjką użytkownika w formacie GIVEN/WHEN/THEN.
- C. Słowne przekazanie informacji w celu zmniejszenia ryzyka błędnego zrozumienia kryteriów akceptacji przez inne osoby.
- D. Udokumentowanie ryzyk związanych z daną historyjką użytkownika w planie testów, co ułatwi wykonanie testowania opartego na ryzyku w odniesieniu do tej historyjki.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 29 (1 punkt)

Rozważmy następującą historyjkę użytkownika:

*Jako redaktor
chcę weryfikować treść przed jej opublikowaniem,
aby upewnić się, że nie występują w niej błędy gramatyczne.*

Kryteria akceptacji związane z tą historyjką są następujące:

- Użytkownik może zalogować się do systemu zarządzania treścią przy użyciu konta z przypisaną rolą „redaktora”.
- Redaktor może wyświetlać istniejące strony z treścią.
- Redaktor może edytować treść stron.

- Redaktor może dodawać komentarze w formie adiustacji.
- Redaktor może zapisywać zmiany.
- Redaktor może zlecać osobie pełniącej rolę „właściciela treści” dokonywanie aktualizacji treści.

Która z poniższych opcji jest NAJLEPSZYM przykładem zastosowania modelu wytwarzania sterowanego testami akceptacyjnymi (ATDD) w celu przetestowania tej historyjki użytkownika?

- A. Przetestowanie, czy redaktor może zapisać dokument po usunięciu treści strony.
- B. Przetestowanie, czy właściciel treści może zalogować się i zaktualizować treść.
- C. Przetestowanie, czy redaktor może wyznaczyć termin publikacji zredagowanej treści.
- D. Przetestowanie, czy redaktor może zlecić innemu redaktorowi dokonanie aktualizacji treści.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 30 (1 punkt)

Jaki jest wkład testerów w planowanie iteracji i wydań?

- A. Testerzy ustalają priorytety opracowywanych historyjek użytkownika.
- B. Testerzy koncentrują się wyłącznie na aspektach funkcjonalnych systemu będącego przedmiotem testów.
- C. Testerzy uczestniczą w procesie identyfikacji i oceny ryzyka w odniesieniu do historyjek użytkownika.
- D. Testerzy gwarantują przekazanie do eksploatacji wysokiej jakości oprogramowania poprzez projektowanie testów na wczesnym etapie — w ramach planowania wydań.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 31 (1 punkt)

Które DWIE z poniższych opcji stanowią kryteria wyjścia w przypadku testowania systemu?

- A. Gotowość środowiska testowego.
- B. Możliwość zalogowania się przez testera do przedmiotu testów.
- C. Osiągnięcie szacowanej gęstości defektów.
- D. Przekształcenie wymagań do formatu GIVEN/WHEN/THEN.
- E. Zautomatyzowanie testów regresji.

Wybierz DWIE odpowiedzi.

Pytanie nr 32 (1 punkt)

Twój zespół szacuje pracochłonność testowania nowej funkcjonalności obarczonej dużym ryzykiem, korzystając w tym celu z techniki szacowania trójpunktowego. Przygotowano następujące oszacowania:

- najbardziej optymistyczne: 2 osobogodziny;
- najbardziej prawdopodobne: 11 osobogodzin;
- najbardziej pesymistyczne: 14 osobogodzin.

Jaka jest ostateczna szacowana wartość?

- A. 9 osobogodzin.
- B. 14 osobogodzin.
- C. 11 osobogodzin.
- D. 10 osobogodzin.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 33 (1 punkt)

Testujesz aplikację mobilną, która umożliwia użytkownikom znajdowanie pobliżunych restauracji na podstawie rodzaju serwowanej kuchni. Rozpatrz poniższą listę przypadków testowych, priorytetów (gdzie mniejsza liczba oznacza wyższy priorytet) oraz zależności:

NUMER PRZYPADKU TESTOWEGO	POKRYWANY WARUNEK TESTOWY	PRIORYTET	ZALEŻNOŚĆ LOGICZNA
PT 001	Wybór rodzaju kuchni	3	brak
PT 002	Wybór restauracji	2	PT 001
PT 003	Uzyskanie wskazówek dojazdu	1	PT 002
PT 004	Zatelefonowanie do restauracji	2	PT 002
PT 005	Dokonanie rezerwacji	3	PT 002

Który z poniższych przypadków testowych powinien zostać wykonany jako trzeci?

- A. PT 003.
- B. PT 005.
- C. PT 002.
- D. PT 001.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 34 (1 punkt)

Rozważ następujące kategorie testów (1–4) i kwadranty testowe w testowaniu zwinnym (A–D):

1. Testowanie użyteczności.
 2. Testowanie modułowe.
 3. Testowanie funkcjonalne.
 4. Testowanie niezawodności.
-
- A. Kwadrant testowania zwanego Q1: cel technologiczny, wspieranie zespołu tworzącego oprogramowanie.
 - B. Kwadrant testowania zwanego Q2: cel biznesowy, wspieranie zespołu tworzącego oprogramowanie.
 - C. Kwadrant testowania zwanego Q3: cel biznesowy, krytyka produktu.
 - D. Kwadrant testowania zwanego Q4: cel technologiczny, krytyka produktu.

W jaki sposób należy przypisać podane kategorie testów do poszczególnych kwadrantów testowych w testowaniu zwinnym?

- A. 1C, 2A, 3B, 4D.
- B. 1D, 2A, 3C, 4B.
- C. 1C, 2B, 3D, 4A.
- D. 1D, 2B, 3C, 4A.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 35 (1 punkt)

Podczas analizy ryzyka zidentyfikowano i oceniono następujące ryzyko:

- Ryzyko: Czas odpowiedzi podczas generowania raportu jest zbyt długi.
- Prawdopodobieństwo wystąpienia ryzyka: średnie; wpływ ryzyka: duży.
- Reakcja na ryzyko:
 - Niezależny zespół testowy wykonuje testowanie wydajnościowe w ramach testowania systemowego.
 - Wybrana grupa użytkowników wykonuje testy akceptacyjne alfa i beta przed przekazaniem oprogramowania do eksploatacji.

Jak nazywa się działanie zaproponowane w odpowiedzi na zidentyfikowane podczas analizy ryzyko?

- A. Akceptacja ryzyka.
- B. Planowanie awaryjne.
- C. Łagodzenie ryzyka.
- D. Przeniesienie ryzyka.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 36 (1 punkt)

Które z wymienionych narzędzi umożliwi zespołowi zwinnemu wykazanie, jaka ilość pracy została już wykonana oraz jaka ilość pracy pozostaje jeszcze do wykonania w ramach danej iteracji?

- A. Kryteria akceptacji.
- B. Raport o defekcie.
- C. Sumaryczny raport z testów.
- D. Wykres spalania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 37 (1 punkt)

Musisz zaktualizować jeden ze skryptów testów automatycznych, aby dostosować go do nowego wymagania. Który proces pozwala zarejestrować w repozytorium testów utworzenie nowej wersji skryptu testowego?

- A. Zarządzanie śledzeniem powiązań.
- B. Testowanie pielęgnacyjne.
- C. Zarządzanie konfiguracją.
- D. Inżynieria wymagań.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 38 (1 punkt)

Otrzymujesz od programistów następujący raport o defekcie wraz z informacją, że anomalii opisanej w raporcie z testów nie da się odtworzyć.

Aplikacja zawiesza się

3 maja 2022 r. – Jan Kowalski – Odrzucono

Aplikacja zawiesza się po wprowadzeniu ciągu „Dane wejściowe do testów: \$ä” w polu Nazwa na ekranie tworzenia nowego użytkownika. Próbowałem się wylogować, a następnie ponownie zalogować na konto test_admin01, ale problem nadal występuje. Próbowałem również korzystać z innych kont administratora, ale pojawił się ten sam problem. Nie jest wyświetlany żaden komunikat o błędzie, a dziennik (w załączniku) zawiera informację o błędzie krytycznym. Zgodnie z przypadkiem testowym PT-1305 aplikacja powinna zaakceptować podane dane wejściowe i utworzyć użytkownika. Proszę o pilne wprowadzenie poprawek, ponieważ funkcjonalność ta jest związana z wymaganiem W-0012, które jest nowym wymaganiem biznesowym o znaczeniu krytycznym.

Jakie kluczowe informacje, które byłyby przydatne dla programistów, NIE zostały uwzględnione w powyższym raporcie o defekcie?

- A. Oczekiwany rezultat i rzeczywisty rezultat.
- B. Odwołania i status defektu.
- C. Środowisko testowe i element testowy.
- D. Priorytet i krytyczność.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 39 (1 punkt)

W ramach której z podanych czynności testowych jest przydatne narzędzie do przygotowywania danych testowych?

- A. Monitorowanie testów i nadzór nad testami.
- B. Analiza i projektowanie testów.
- C. Implementacja i wykonywanie testów.
- D. Ukończenie testów.

Wybierz JEDNĄ odpowiedź.

Pytanie nr 40 (1 punkt)

Która z poniższych odpowiedzi poprawnie wskazuje potencjalne ryzyko związane z automatyzacją testów?

- A. Automatyzacja może spowodować wprowadzenie nieznanych regresji w środowisku produkcyjnym.
- B. Nakłady pracy przeznaczone na utrzymanie testaliów mogą być niewystarczające.
- C. Narzędzia do testowania i związane z nimi testalia mogą nie cieszyć się wystarczającym zaufaniem.
- D. Automatyzacja może spowodować skrócenie czasu przeznaczonego na testowanie manualne.

Wybierz JEDNĄ odpowiedź.

Dodatkowe przykładowe pytania

Ogólna zasada publikowania przykładowych pytań przez ISTQB wymaga, aby do każdego celu nauczania było opublikowane przynajmniej jedno przykładowe pytanie. Jeśli więc celów nauczania jest więcej niż pytań egzaminacyjnych, pytania pokrywające cele nauczania nieuwzględnione w przykładowym zestawie egzaminacyjnym są publikowane osobno, jako pytania dodatkowe. W tej części podręcznika znajdują się oficjalne, dodatkowe pytania.

Pytanie nr A1 (1 punkt)

Twoim zadaniem jest przeanalizowanie i usunięcie przyczyn awarii w nowym systemie, który ma zostać przekazany do eksplotacji.

Którą z wymienionych czynności wykonujesz?

- A. Debugowanie.
- B. Testowanie oprogramowania.
- C. Pozyskiwanie wymagań.
- D. Zarządzanie defektami.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A2 (1 punkt)

W wielu organizacjach, które wytwarzają oprogramowanie, dział odpowiedzialny za testowanie jest nazywany działem zapewnienia jakości. Czy to zdanie jest poprawne, a jeśli nie, to dlaczego?

- A. Zdanie jest poprawne. Testowanie i zapewnienie jakości to dwa różne określenia na ten sam proces.
- B. Zdanie jest poprawne. Określenia te mogą być używane wymiennie, ponieważ działania wykonywane w ramach testowania i działania wykonywane w ramach zapewnienia jakości koncentrują się na tych samych problemach związanych z jakością.
- C. Zdanie jest niepoprawne. Testowanie jest szerszym procesem, który obejmuje wszystkie działania związane z jakością, a zapewnienie jakości koncentruje się na procesach związanych z jakością.
- D. Zdanie jest niepoprawne. Zapewnienie jakości koncentruje się na procesach związanych z jakością, a testowanie — na wykazaniu, że dany moduł lub system jest zdatny do użytku zgodnie z przeznaczeniem, oraz na wykryciu ewentualnych defektów.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A3 (1 punkt)

Telefon dzwoniący w sąsiednim pomieszczeniu chwilowo rozproszył programistę, przez co niewłaściwie zaprogramował on logikę sprawdzającą górną wartość brzegową jednej ze zmiennych wejściowych. Później, w trakcie testowania systemowego, tester zauważył, że system akceptuje nieprawidłowe wartości wejściowe wpisywane w polu wprowadzania danych.

Nieprawidłowo zakodowana logika sprawdzania górnej wartości brzegowej to:

- A. Podstawowa przyczyna.
- B. Awaria.
- C. Pomyłka.
- D. Defekt.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A4 (1 punkt)

Przeanalizuj podany poniżej fragment testaliów:

Karta opisu testu nr 04.018 Czas trwania sesji: 1 h	
Badany obiekt:	Strona rejestracji
Badanie za pomocą:	Różnych zestawów niepoprawnych danych wejściowych
Wykrywane defekty:	Defekty związane z akceptacją rejestracji w przypadku podania nieprawidłowych danych wejściowych

W ramach której czynności testowej powstały powyższe testalia?

- A. Planowanie testów.
- B. Monitorowanie testów i nadzór nad testami.
- C. Analiza testów.
- D. Projektowanie testów.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A5 (1 punkt)

Które z poniższych stwierdzeń jest NAJLEPSZYM przykładem na to, jak śledzenie powiązań pomaga w testowaniu?

- A. Przeprowadzenie analizy wpływu zmiany pozwala uzyskać informacje na temat ukończenia testów.
- B. Przeanalizowanie powiązań między przypadkami testowymi a wynikami testów pozwala uzyskać informacje na temat szacowanego poziomu ryzyka resztkowego (rezydualnego).

- C. Przeprowadzenie analizy wpływu zmiany pomaga w wyborze właściwych przypadków testowych na potrzeby testowania regresji.
- D. Przeanalizowanie powiązań między podstawą testów, przedmiotami testów a przypadkami testowymi ułatwia wybór danych testowych umożliwiających osiągnięcie zakładanego pokrycia przedmiotu testów.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A6 (1 punkt)

Które z poniższych stwierdzeń NAJLEPIEJ wyjaśnia korzyści wynikające z niezależności testowania?

- A. Korzystanie z niezależnego zespołu testowego umożliwia kierownictwu projektu przeniesienie odpowiedzialności za jakość finalnego produktu na ten zespół.
- B. Jeśli organizacja może sobie pozwolić na skorzystanie z zewnętrznego zespołu testowego, rozwiązywanie takie może przynieść konkretne korzyści, ponieważ zewnętrzny zespół nie ulegnie łatwo presji związanej z obawami kierownictwa projektu dotyczącymi sposobu realizacji oraz koniecznością przestrzegania ścisłych terminów realizacji.
- C. Niezależny zespół testowy może pracować niezależnie od programistów. Dzięki temu zmiany wymagań związanych z projektem nie rozpraszają uwagi testerów, a komunikacja z programistami ogranicza się do raportowania o defektach za pośrednictwem systemu zarządzania defektami.
- D. Interpretacja specyfikacji, w których występują niejasności i niespójności, wymaga przyjęcia określonych założeń, a niezależni testerzy mogą zakwestionować powyższe założenia i interpretację dokonaną przez programistę.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A7 (1 punkt)

Pracujesz jako tester w zespole, który stosuje model V. Jak wybór powyższego modelu cyklu wytwarzania oprogramowania wpływa na to, kiedy wykonywane jest testowanie?

- A. W tym modelu nie można wykonywać testowania dynamicznego na wcześniejszym etapie cyklu wytwarzania oprogramowania.
- B. W tym modelu nie można wykonywać testowania statycznego na wcześniejszym etapie cyklu wytwarzania oprogramowania.
- C. W tym modelu nie można planować testów na wcześniejszym etapie cyklu wytwarzania oprogramowania.
- D. Model ten umożliwia wykonywanie testów akceptacyjnych na wcześniejszym etapie cyklu wytwarzania oprogramowania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A8 (1 punkt)

Które z poniższych stwierdzeń opisują zalety metodyki DevOps?

- i. Przyspieszenie wprowadzania produktów do eksploatacji i na rynek.
 - ii. Zwiększenie zapotrzebowania na powtarzalne testowanie manualne.
 - iii. Zapewnienie stałej dostępności wykonywalnego oprogramowania.
 - iv. Zmniejszenie liczby testów regresji związanych z refaktoryzacją kodu.
 - v. Niskie koszty konfigurowania struktury (*framework*) do testów automatycznych z uwagi na automatyzację całego procesu.
-
- A. i, ii oraz iv są zaletami — w przeciwnieństwie do iii oraz v.
 - B. iii oraz v są zaletami — w przeciwnieństwie do i, ii oraz iv.
 - C. i oraz iii są zaletami — w przeciwnieństwie do ii, iv oraz v.
 - D. ii, iv oraz v są zaletami — w przeciwnieństwie do i oraz iii.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A9 (1 punkt)

Jesteś testerem w projekcie, w ramach którego na zlecenie jednego z klientów powstaje aplikacja mobilna do zamawiania posiłków. Klient przesłał listę wymagań. Jedno z nich, oznaczone wysokim priorytetem, brzmi następująco:

„Zamówienie musi zostać przetworzone w czasie krótszym niż 10 sekund w 95% przypadków”.

W związku z tym utworzyłeś/utworzyłaś zbiór przypadków testowych obejmujących: złożenie losowej liczby zamówień, zmierzenie czasu przetwarzania oraz sprawdzenie wyników testów pod kątem zgodności z wymaganiami.

Jaki typ testów wykonałeś/wykonałaś?

- A. Testy funkcjonalne, ponieważ przypadki testowe pokrywają wymaganie biznesowe użytkownika dotyczące systemu.
- B. Testy niefunkcjonalne, ponieważ mierzona była wydajność systemu.
- C. Testy funkcjonalne, ponieważ przypadki testowe przewidują interakcję z interfejsem użytkownika.
- D. Testy strukturalne, ponieważ do zmierzenia czasu przetwarzania zamówień niezbędna jest znajomość struktury wewnętrznej programu.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A10 (1 punkt)

Strategia testów obowiązująca w twojej organizacji zaleca przetestowanie migracji danych w przypadku podjęcia decyzji o wycofaniu systemu z eksploatacji. W ramach jakiego typu testów zostanie to NAJPRAWDOPODOBNIEJ wykonane?

- A. Testowanie pielęgnacyjne.
- B. Testowanie regresji.
- C. Testowanie modułowe.
- D. Testowanie integracyjne.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A11 (1 punkt)

Poniżej przedstawiono listę produktów pracy, które powstały w ramach cyklu wytwarzania oprogramowania.

- i. Wymagania biznesowe.
- ii. Harmonogram.
- iii. Budżet testów.
- iv. Kod wykonywalny innych firm.
- v. Historyjki użytkownika i związane z nimi kryteria akceptacji.

Które z nich można objąć przeglądem?

- A. i oraz iv można objąć przeglądem, natomiast ii, iii oraz v nie mogą być przedmiotem przeglądu.
- B. i, ii, iii oraz iv można objąć przeglądem, natomiast v nie może być przedmiotem przeglądu.
- C. i, ii, iii oraz v można objąć przeglądem, natomiast iv nie może być przedmiotem przeglądu.
- D. iii, iv oraz v można objąć przeglądem, natomiast i oraz ii nie mogą być przedmiotem przeglądu.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A12 (1 punkt)

Które z poniższych stwierdzeń (i-v) są prawdziwe w odniesieniu do testowania dynamicznego, a które są prawdziwe w odniesieniu do testowania statycznego?

- i. Ten typ testowania pozwala łatwiej zidentyfikować nieprawidłowe zewnętrzne zachowania systemu.
- ii. Ten typ testowania pozwala łatwiej wykryć odstępstwa od standardu kodowania.
- iii. Ten typ testowania umożliwia identyfikowanie awarii spowodowanych przez defekty podczas uruchamiania oprogramowania.

- iv. Celem tego testowania jest jak najwcześniejsze zidentyfikowanie defektów.
 - v. Ten typ testowania pozwala łatwiej znaleźć i skorygować braki w pokryciu krytycznych wymagań w zakresie zabezpieczeń.
- A. i, iv oraz v są prawdziwe w odniesieniu do testowania statycznego, natomiast ii oraz iii — w odniesieniu do testowania dynamicznego.
 - B. i, iii oraz iv są prawdziwe w odniesieniu do testowania statycznego, natomiast ii oraz v — w odniesieniu do testowania dynamicznego.
 - C. ii oraz iii są prawdziwe w odniesieniu do testowania statycznego, natomiast i, iv oraz v — w odniesieniu do testowania dynamicznego.
 - D. ii, iv oraz v są prawdziwe w odniesieniu do testowania statycznego, natomiast i, iii oraz iv — w odniesieniu do testowania dynamicznego.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A13 (1 punkt)

Które z poniższych stwierdzeń dotyczących przeglądów formalnych jest PRAWDZIWE?

- A. Niektóre przeglądy nie wymagają wyznaczenia więcej niż jednej roli.
- B. Proces przeglądu składa się z kilku czynności.
- C. Dokumentacja będąca przedmiotem przeglądu nie jest przekazywana uczestnikom przed rozpoczęciem spotkania związanego z przeglądem (z wyjątkiem produktu pracy w przypadku określonych typów przeglądów).
- D. Defekty wykryte podczas przeglądu nie są zgłasiane, ponieważ nie zostały wykryte w ramach testowania dynamicznego.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A14 (1 punkt)

Jakie zadania może wykonywać kierownictwo podczas przeglądu formalnego?

- A. Przyjęcie ogólnej odpowiedzialności za przegląd.
- B. Decydowanie o tym, co ma być przedmiotem przeglądu.
- C. Dbanie o sprawny przebieg spotkań związanych z przeglądem i występowanie w roli mediatora, jeśli zachodzi taka potrzeba.
- D. Protokołowanie informacji związanych z przeglądem, w tym informacji o podjętych decyzjach.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A15 (1 punkt)

W systemie do przechowywania wina zastosowano układ sterowania, który mierzy temperaturę T w winiarce (wyrażoną w °C w zaokrągleniu do najbliższego pełnego stopnia) i powiadamia użytkownika o odchyleniach od optymalnej wartości wynoszącej 12 zgodnie z następującymi regułami:

- jeśli $T = 12$, system wysyła komunikat „Temperatura jest optymalna”;
- jeśli $T < 12$, system wysyła komunikat „Temperatura jest zbyt niska!”;
- jeśli $T > 12$, system wysyła komunikat „Temperatura jest zbyt wysoka!”.

Chcesz zweryfikować zachowanie układu sterowania metodą trójpunktowej analizy wartości brzegowych. Dane wejściowe do testu to temperatura w °C podawana przez urządzenie.

Jaki jest MINIMALNY zbiór danych wejściowych do testów, który pozwala uzyskać 100% wymaganego pokrycia?

- A. 11, 12, 13.
- B. 10, 12, 14.
- C. 10, 11, 12, 13, 14.
- D. 10, 11, 13, 14.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A16 (1 punkt)

Które z poniższych stwierdzeń dotyczących testowania gałęzi jest POPRAWNE?

- A. Jeśli program zawiera tylko gałęzie bezwarunkowe, stuprocentowe pokrycie gałęzi można uzyskać bez wykonywania żadnych przypadków testowych.
- B. Jeśli przypadki testowe sprawdzają wszystkie gałęzie bezwarunkowe w kodzie, pokrycie gałęzi wynosi 100%.
- C. Uzyskanie stuprocentowego pokrycia instrukcji kodu oznacza również uzyskanie stuprocentowego pokrycia gałęzi.
- D. Uzyskanie stuprocentowego pokrycia gałęzi oznacza, że sprawdzono wszystkie wyniki decyzji w każdej instrukcji decyzyjnej w kodzie.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A17 (1 punkt)

Testujesz aplikację mobilną, która umożliwia klientom dostęp do kont bankowych i zarządzanie nimi. Uruchamiasz zestaw testowy, który przewiduje dokonanie oceny każdego ekranu i każdego pola na każdym ekranie w oparciu o ogólną listę najlepszych praktyk w dziedzinie tworzenia interfejsów użytkownika. Lista pochodzi z popularnej książki na ten temat i ma na celu zapewnienie maksymalnej atrakcyjności, łatwości obsługi oraz dostępności tego typu aplikacji. Która z poniższych opcji NAJ-LEPIEJ opisuje stosowaną przez ciebie technikę testowania?

- A. Testowanie czarnoskrzynkowe.
- B. Testowanie eksploracyjne.
- C. Testowanie w oparciu o listę kontrolną.
- D. Zgadywanie błędów.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A18 (1 punkt)

Która z poniższych odpowiedzi NAJLEPIEJ opisuje wspólne pisanie historyjek użytkownika?

- A. Historyjki użytkownika są tworzone przez testerów i programistów, a następnie akceptowane przez przedstawicieli jednostek biznesowych.
- B. Historyjki użytkownika są tworzone wspólnie przez przedstawicieli jednostek biznesowych, programistów i testerów.
- C. Historyjki użytkownika są tworzone przez przedstawicieli jednostek biznesowych, a następnie weryfikowane przez programistów i testerów.
- D. Historyjki użytkownika są tworzone tak, aby były niezależne, negocjalne, wartościowe, możliwe do oszacowania, zwięzłe i testowalne.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A19 (1 punkt)

Weźmy pod uwagę następujący fragment planu testów:

Testowanie obejmie testowanie modułowe i testowanie integracji modułów. Zgodnie z wymogami obowiązujących przepisów należy wykazać osiągnięcie stuprocentowego pokrycia gałęzi w odniesieniu do każdego modułu sklasyfikowanego jako krytyczny.

W której części planu testów powinien znajdować się ten fragment?

- A. Wymiana informacji.
- B. Rejestr ryzyka.
- C. Kontekst testowania.
- D. Podejście do testowania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A20 (1 punkt)

Twój zespół szacuje metodą pokera planistycznego pracochnonność testowania funkcjonalności, która musi zostać wprowadzona zgodnie z nowymi wymaganiami. W zespole obowiązuje zasada, że jeśli brakuje czasu na osiągnięcie pełnego porozumienia, a rozbieżności między wynikami są niewielkie, dopuszczalne jest przyjęcie wartości, która uzyskała największą liczbę głosów.

Po dwóch rundach nie osiągnięto konsensusu, w związku z czym rozpoczęto trzecią rundę. Wyniki szacowania testów przedstawiono w poniżej tabeli.

	OSZACOWANIA PODANE PRZEZ CZŁONKÓW ZESPOŁU						
Runda 1	21	2	5	34	13	8	2
Runda 2	13	8	8	34	13	8	5
Runda 3	13	8	13	13	13	13	8

Które z poniższych stwierdzeń NAJTRAFNIEJ wskazuje, jaki powinien być następny krok?

- A. Właściciel produktu musi wkroczyć i podjąć ostateczną decyzję.
- B. Wartość 13 uzyskała najczęściej głosów, w związku z czym należy ją przyjąć jako ostateczne oszacowanie pracochności testów.
- C. Nie są wymagane żadne dalsze działania. Osiągnięto konsensus.
- D. Z powodu braku konsensusu należy usunąć nową funkcjonalność z bieżącego wydania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A21 (1 punkt)

Które z poniższych stwierdzeń NIE jest zgodne z prawdą w odniesieniu do piramidy testów?

- A. W modelu piramidy testów podkreśla się fakt, że na niższych poziomach testów powinna być wykonywana większa liczba testów.
- B. Im bliżej szczytu piramidy, tym bardziej sformalizowana powinna być automatyzacja testów.
- C. Do automatyzacji testowania modułowego i testowania integracji modułów używa się zwykle narzędzi opartych na interfejsach API.
- D. W przypadku testowania systemowego i testowania akceptacyjnego testy automatyczne tworzy się zwykle przy użyciu narzędzi wyposażonych w graficzny interfejs użytkownika.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A22 (1 punkt)

W trakcie analizy ryzyka zespół rozważał następujące ryzyko: „*System zezwala na udzielenie klientowi zbyt wysokiego rabatu*”. Członkowie zespołu oszacowali, że wpływ ryzyka jest bardzo duży.

Co można na tej podstawie powiedzieć o prawdopodobieństwie wystąpienia ryzyka?

- A. Jest również bardzo duże. Duży wpływ ryzyka zawsze oznacza duże prawdopodobieństwo wystąpienia ryzyka.
- B. Jest bardzo małe. Duży wpływ ryzyka zawsze oznacza małe prawdopodobieństwo wystąpienia ryzyka.
- C. Nie można nic powiedzieć na temat prawdopodobieństwa wystąpienia ryzyka. Wpływ ryzyka i prawdopodobieństwo wystąpienia ryzyka są od siebie niezależne.
- D. Prawdopodobieństwo wystąpienia ryzyka nie jest istotne w przypadku tak dużego wpływu ryzyka, w związku z czym nie trzeba go określać.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A23 (1 punkt)

Na poniższej liście wymieniono ryzyka zidentyfikowane w odniesieniu do nowego oprogramowania, które ma zostać wytworzone.

- i. Kierownictwo przesuwa dwóch doświadczonych testerów do innego projektu.
- ii. System nie spełnia standardów w dziedzinie bezpieczeństwa funkcjonalnego.
- iii. Czas odpowiedzi systemu przekracza wartość określoną w wymaganiach użytkownika.
- iv. Interesariusze mają nieprecyzyjnie określone oczekiwania.
- v. Osoby niepełnosprawne mają problemy z korzystaniem z systemu.

Które z powyższych ryzyk to ryzyka projektowe?

- A. i oraz iv to ryzyka projektowe, natomiast ii, iii oraz v nimi nie są.
- B. iv oraz v to ryzyka projektowe, natomiast i, ii oraz iii nimi nie są.
- C. i oraz iii to ryzyka projektowe, natomiast ii, iv oraz v nimi nie są.
- D. ii oraz v to ryzyka projektowe, natomiast i, iii oraz iv nimi nie są.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A24 (1 punkt)

Które z poniższych stwierdzeń zawiera przykład na to, jak analiza ryzyka produk-towego wpływa na staranność i zakres testowania?

- A. Kierownik testów codziennie monitoruje poziom wszystkich znanych ryzyk i przesyła raporty na ten temat, aby umożliwić interesariuszom podjęcie świadomej decyzji co do terminu przekazania do eksploatacji.
- B. Jednym ze zidentyfikowanych ryzyk był „*brak obsługi baz danych o otwartym kodzie*”, w związku z czym zespół postanowił zintegrować system z taką bazą danych.
- C. W ramach analizy ryzyka przeprowadzanej metodą ilościową zespół oszacowały łączny poziom wszystkich zidentyfikowanych ryzyk i wykazały go w raporcie jako łączone ryzyko resztkowe (rezydualne) przed rozpoczęciem testowania.
- D. Ocena ryzyka ujawniła bardzo wysoki poziom ryzyka związanego z wydajnością, w związku z czym podjęto decyzję o wykonaniu szczegółowego te-stowania wydajnościowego we wczesnej fazie cyklu wytwarzania oprogra-mowania.

Wybierz JEDNĄ odpowiedź.

Pytanie nr A25 (1 punkt)

Wskaż DWIE z poniższych opcji, które odpowiadają metrykom powszechnie uży-wanym w raportach na temat poziomu jakości przedmiotu testów.

- A. Liczba defektów wykrytych podczas testowania systemowego.
- B. Iloraz łącznych nakładów pracy na projektowanie testów przez liczbę zapro-jektowanych przypadków testowych.
- C. Liczba wykonanych procedur testowych.
- D. Iloraz liczby wykrytych defektów przez wielkość produktu pracy.
- E. Czas niezbędny do usunięcia defektu.

Wybierz DWIE odpowiedzi.

Pytanie nr A26 (1 punkt)

Która z poniższych informacji zawartych w raporcie o postępie testów jest NAJMNIĘJ przydatna z punktu widzenia przedstawicieli jednostek biznesowych?

- A. Przeszkody w testowaniu.
- B. Uzyskane pokrycie gałęzi.
- C. Postęp testów.
- D. Nowe ryzyka zaobserwowane w cyklu testowym.

Wybierz JEDNĄ odpowiedź.

Egzamin — odpowiedzi

Pytanie 1.

FL-1.1.1 (K1)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Nie da się udowodnić, że w systemie podlegającym testowaniu nie występują już żadne defekty. Patrz zasada testowania nr 1.
- B. Odpowiedź niepoprawna. Patrz zasada testowania nr 7.
- C. Odpowiedź poprawna. Testowanie pozwala wykryć defekty i awarie, co przekłada się na obniżenie poziomu ryzyka, a jednocześnie na zwiększenie zaufania do jakości przedmiotu testów.
- D. Odpowiedź niepoprawna. Przetestowanie wszystkich kombinacji danych wejściowych nie jest możliwe (patrz zasada testowania nr 2).

Pytanie 2.

FL-1.2.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Należy zadbać o zaangażowanie testerów od samego początku cyklu wytwarzania oprogramowania, ponieważ umożliwia to lepsze zrozumienie decyzji projektowych i sprzyja wczesnemu wykrywaniu defektów.
- B. Odpowiedź niepoprawna. Programiści i testerzy powinni wzajemnie poznawać swoje produkty pracy, ponieważ dzięki temu poszerzają swoją wiedzę na temat sposobów testowania kodu.
- C. Odpowiedź niepoprawna. Jeżeli testerzy ściśle współpracują z projektantami systemu, pozwala im to lepiej zrozumieć, jak testować.
- D. Odpowiedź niepoprawna. Testowanie nie zakończy się pomyślnie, jeśli wymagania prawne nie zostaną przetestowane pod kątem zgodności.

Pytanie 3.

FL-1.3.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Zasada ta oznacza, że ciągłe powtarzanie tych samych testów prowadzi do sytuacji, w której przestają one w pewnym momencie wykrywać nowe defekty. Prawdopodobnie z tego powodu wszystkie testy zostały zaliczone również w bieżącym wydaniu.

- B. Odpowiedź niepoprawna. Ta zasada mówi o mylnym przekonaniu, że samo wykrycie i usunięcie dużej liczby defektów zapewni pomyślne wdrożenie systemu.
- C. Odpowiedź niepoprawna. Ta zasada mówi, że w niewielkiej liczbie modułów zwykle znajdujemy większość defektów.
- D. Odpowiedź niepoprawna. Ta zasada mówi, że przetestowanie wszystkich kombinacji danych wejściowych i warunków wstępnych nie jest możliwe.

Pytanie 4.

FL-1.4.1 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Szacowanie pracochłonności testów jest elementem planowania testów.
- B. Odpowiedź poprawna. Jest to przykład definiowania warunków testowych, co wchodzi w zakres analizy testów.
- C. Odpowiedź niepoprawna. Wyprowadzanie elementów pokrycia przy użyciu technik testowania wchodzi w zakres projektowania testów.
- D. Odpowiedź niepoprawna. Zgłaszanie defektów wykrytych podczas testowania dynamicznego jest elementem wykonywania testów.

Pytanie 5.

FL-1.4.2 (K2)

Poprawna odpowiedź: B

- i. Prawda. Cykl wytwarzania oprogramowania ma wpływ na proces testowy.
- ii. Fałsz. Liczba defektów wykrytych w poprzednich projektach może mieć pewien wpływ, ale nie jest on tak istotny, jak wpływ czynników i, iii oraz iv.
- iii. Prawda. Zidentyfikowane ryzyka produktowe należą do najważniejszych czynników wpływających na proces testowy.
- iv. Prawda. Wymagania prawne są istotnymi czynnikami wpływającymi na proces testowy.
- v. Fałsz. Środowisko testowe powinno być kopią środowiska eksploatacyjnego, ale nie ma to istotnego wpływu na proces testowy.

W związku z tym poprawna jest odpowiedź B.

Pytanie 6.

FL-1.4.5 (K2)

Poprawna odpowiedź: A, E

- A. Odpowiedź poprawna. Zadanie to jest wykonywane przez testerów.
- B. Odpowiedź niepoprawna. Backlog produktu tworzy i utrzymuje właściciel produktu.

- C. Odpowiedź niepoprawna. Zadanie to jest wykonywane przez zespół tworzący oprogramowanie.
- D. Odpowiedź niepoprawna. Zadanie to wchodzi w zakres obowiązków kierownika.
- E. Odpowiedź poprawna. Zadanie to jest wykonywane przez testerów.

Pytanie 7.

FL-1.5.1 (K2)

Poprawna odpowiedź: B

- i. Prawda. Posiadanie wiedzy merytorycznej to ważna umiejętność testera.
- ii. Fałsz. Jest to zadanie analityka biznesowego i przedstawiciela jednostki biznesowej.
- iii. Prawda. Umiejętność sprawnej pracy w zespole jest ważną umiejętnością.
- iv. Fałsz. Planowanie i organizowanie pracy zespołu jest zadaniem kierownika testów lub (głównie w projektach zwanego wytwarzania oprogramowania) całego zespołu, a nie tylko testera.
- v. Prawda. Umiejętność krytycznego myślenia jest jedną z najważniejszych umiejętności testera.

W związku z tym poprawna jest odpowiedź B.

Pytanie 8.

FL-1.5.2 (K1)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Podejście do automatyzacji testów określają testerzy z pomocą programistów i przedstawicieli jednostek biznesowych.
- B. Odpowiedź niepoprawna. Decyzję o wyborze strategii testów podejmuje się w porozumieniu z programistami.
- C. Odpowiedź niepoprawna. Podejście oparte na zaangażowaniu całego zespołu obejmuje testerów, programistów i przedstawicieli jednostek biznesowych.
- D. Odpowiedź poprawna. Testerzy ściśle współpracują z przedstawicielami jednostek biznesowych, aby zagwarantować osiągnięcie wymaganych poziomów jakości. Współpraca ta obejmuje wsparcie i współdziałanie w zakresie tworzenia odpowiednich testów akceptacyjnych.

Pytanie 9.

FL-2.1.2 (K1)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Patrz uzasadnienie poprawnej odpowiedzi.
- B. Odpowiedź niepoprawna. Patrz uzasadnienie poprawnej odpowiedzi.
- C. Odpowiedź niepoprawna. Patrz uzasadnienie poprawnej odpowiedzi.
- D. Odpowiedź poprawna. Ta reguła odnosi się do wszystkich modeli cyklu wytwarzania oprogramowania.

Pytanie 10.

FL-2.1.3 (K1)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Podejście to jest częściej stosowane w modelu wytwarzania sterowanego zachowaniem (ang. *Behaviour-Driven Development*, BDD).
- B. Odpowiedź niepoprawna. Jest to opis wytwarzania sterowanego testami (ang. *Test-Driven Development*, TDD).
- C. Odpowiedź poprawna. W modelu wytwarzania sterowanego testami akceptacyjnymi (ang. *Acceptance Test-Driven Development*, ATDD) testy są pisane na podstawie kryteriów akceptacji w ramach procesu projektowania.
- D. Odpowiedź niepoprawna. Metoda ta jest stosowana w modelu wytwarzania sterowanego zachowaniem (ang. *Behaviour-Driven Development*, BDD).

Pytanie 11.

FL-2.1.5 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Dokonywanie przeglądu na wczesnym etapie jest przykładem podejścia przesunięcie w lewo.
- B. Odpowiedź niepoprawna. Wytwarzanie sterowane testami (ang. *Test-Driven Development*, TDD) jest przykładem podejścia przesunięcie w lewo.
- C. Odpowiedź niepoprawna. Testowanie niefunkcjonalne na wczesnym etapie jest przykładem podejścia przesunięcie w lewo.
- D. Odpowiedź poprawna. Skrypty testowe powinny być objęte zarządzaniem konfiguracją, w związku z czym tworzenie ich przed ustanowieniem tego procesu nie ma sensu.

Pytanie 12.

FL-2.1.6 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Celem retrospektyw jest raczej wskazanie możliwości wprowadzania udoskonaleń, natomiast ich znaczenie dla klientów jest niewielkie.
- B. Odpowiedź niepoprawna. Przedstawiciele jednostek biznesowych nie przekazują informacji zwrotnych na temat samego produktu, w związku z czym nie przyniesie to organizacji żadnych korzyści finansowych.
- C. Odpowiedź poprawna. Regularnie przeprowadzane retrospektwy połączone z odpowiednimi działaniami weryfikacyjnymi mają kluczowe znaczenie dla ciągłego doskonalenia procesów wytwarzania i testowania oprogramowania.
- D. Odpowiedź niepoprawna. Odwaga i szacunek to wartości związane z programowaniem ekstremalnym (ang. *eXtreme Programming*, XP), które nie mają ścisłego związku z retrospektywami.

Pytanie 13.

FL-2.2.1 (K2)

Poprawna odpowiedź: A

Podstawą testów w testowaniu akceptacyjnym są potrzeby biznesowe użytkownika (1D). Wymianę danych między modułami testuje się na etapie testowania integracji modułów (2B). Awarie na poziomie logiki można wykryć w trakcie testowania modułowego (3A). Reguły biznesowe są podstawą testów w testowaniu systemowym (4C).

W związku z tym poprawna jest odpowiedź A.

Pytanie 14.

FL-2.2.3 (K2)

Poprawna odpowiedź: B

Przypadki testowe PT1 i PT3 nie zostały zaliczone w przebiegu 1 (test 1 i test 3), dlatego test 4 i test 6 to testy potwierdzające. Przypadki testowe PT2 i PT3 nie zostały zaliczone w przebiegu 2 (test 5 i test 6), dlatego test 8 i test 9 to również testy potwierdzające. Przypadek testowy PT2 został zaliczony w przebiegu 1 (test 2), dlatego test 5 to test regresji. Przypadek testowy PT1 został zaliczony w przebiegu 2 (test 4), dlatego test 7 to również test regresji.

W związku z tym poprawna jest odpowiedź B.

Pytanie 15.

FL-3.1.2 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Zarządzanie defektami nie jest mniej kosztowne. Koszty wykrywania i usuwania defektów na późniejszych etapach cyklu wytwarzania oprogramowania są w istocie wyższe.
- B. Odpowiedź niepoprawna. Jest to korzyść wynikająca z testowania statycznego.
- C. Odpowiedź niepoprawna. Jest to korzyść wynikająca z testowania statycznego.
- D. Odpowiedź niepoprawna. Jest to korzyść wynikająca z testowania statycznego.

Pytanie 16.

FL-3.2.1 (K1)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Informacje zwrotne pozwalają usprawnić proces testowy, ale jeśli usprawnienia mają dotyczyć jedynie przyszłych projektów, informacje takie nie muszą być przekazywane na wcześniejszym etapie ani z dużą częstotliwością.
- B. Odpowiedź niepoprawna. Informacje zwrotne nie są wykorzystywane do ustalania priorytetów wymagań.

- C. Odpowiedź niepoprawna. Jakość zmian można mierzyć na wiele sposobów.
- D. Odpowiedź poprawna. Przekazywane w odpowiednim czasie i z odpowiednią częstotliwością informacje zwrotne umożliwiają wcześnie sygnalizowanie potencjalnych problemów z jakością.

Pytanie 17.

FL-3.2.4 (K2)

Poprawna odpowiedź: B

Biorąc pod uwagę przedstawione atrybuty:

- Wyznaczono osobę pełniąącą rolę protokolanta. Rolę tę określono w odniesieniu do przejrzeń, przeglądów technicznych i inspekcji, w związku z czym przeprowadzane przeglądy nie mogą być przeglądami nieformalnymi.
- Celem jest dokonanie oceny jakości, co odpowiada jednemu z najważniejszych celów przejrzenia.
- Spotkanie związane z przeglądem prowadzi autor produktu pracy, co nie jest dozwolone w przypadku inspekcji i zwykle nie jest stosowane w przypadku przeglądów technicznych. W przypadku przejrzeń wymagane jest wyznanie moderatora; jest ono również dozwolone w przypadku przeglądów nieformalnych.
- Poszczególni przeglądający wykrywają potencjalne anomalie na etapie przygotowania. Przegląd indywidualny może być elementem wszystkich typów przeglądów (nawet nieformalnych).
- Sporządzany jest raport z przeglądu. Raport z przeglądu może powstać w ramach każdego typu przeglądu, chociaż w przypadku przeglądów nieformalnych dokumentacja nie jest wymagana.

W związku z tym poprawna jest odpowiedź B.

Pytanie 18.

FL-3.2.5 (K1)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Przeznaczenie przez uczestników odpowiedniej ilości czasu na przeprowadzenie przeglądu jest czynnikiem powodzenia przeglądu.
- B. Odpowiedź niepoprawna. Właściwy podział produktów pracy na mniejsze części jest czynnikiem powodzenia przeglądu.
- C. Odpowiedź niepoprawna. Unikanie zachowań, które mogłyby wskazywać na znudzenie, irytację itd., jest czynnikiem powodzenia przeglądu.
- D. Odpowiedź poprawna. Podczas przeglądów można wykryć defekty, a nie awarie.

Pytanie 19.

FL-4.1.1 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Jest to typowa cecha białośkrzynkowych technik testowania. Warunki testowe, przypadki testowe i dane testowe wyprowadza się z podstawy testów, która może obejmować kod, architekturę oprogramowania, szczegółowy projekt bądź dowolne inne źródło informacji o strukturze oprogramowania.
- B. Odpowiedź niepoprawna. Jest to typowa cecha białośkrzynkowych technik testowania. Pokrycie mierzy się na podstawie przetestowanych elementów wybranej struktury oraz techniki zastosowanej w odniesieniu do podstawy testów.
- C. Odpowiedź poprawna. Jest to typowa cecha technik testowych opartych na doświadczeniu. Do definiowania testów wykorzystywane są wiedza i doświadczenie dotyczące między innymi przewidywanego sposobu korzystania z oprogramowania, środowiska pracy oprogramowania oraz prawdopodobnych defektów i ich rozkładu.
- D. Odpowiedź niepoprawna. Jest to typowa cecha czarnoskrzynkowych technik testowania. Przypadki testowe mogą być wykorzystywane do wykrywania rozbieżności między wymaganiami a ich implementacją bądź odstępstw od wymagań.

Pytanie 20.

FL-4.2.1 (K3)

Poprawna odpowiedź: B

Opcje „mały ogródek” i „duży ogródek” mogą łączyć się tylko z opcją „parter”, w związku z czym potrzebne są dwa przypadki testowe z opcją „parter”, które pokrywają dwie klasy równoważności kryterium „typ ogródka”. Ponadto potrzebne są kolejne dwa przypadki testowe, które pokryją pozostałe dwie klasy równoważności kryterium „kondygnacja” oraz pozostałą klasę równoważności kryterium „typ ogródka”, czyli „brak ogródka”.

Łącznie potrzebne są cztery przypadki testowe, np.:

- PT1 (parter, mały ogródek);
- PT2 (parter, duży ogródek);
- PT3 (pierwsze piętro, brak ogródka);
- PT4 (drugie lub wyższe piętro, brak ogródka).

Zatem:

- A. Odpowiedź niepoprawna.
- B. Odpowiedź poprawna.
- C. Odpowiedź niepoprawna.
- D. Odpowiedź niepoprawna.

Pytanie 21.

FL-4.2.2 (K3)

Poprawna odpowiedź: A

Występuje 12 wartości brzegowych związkanych z wartościami wyniku końcowego: 0, 50, 51, 60, 61, 70, 71, 80, 81, 90, 91 oraz 100. Przypadki testowe pokrywają sześć z nich (PT1 – 91, PT2 – 50, PT3 – 81, PT4 – 60, PT5 – 70 oraz PT7 – 51). W związku z tym przypadki testowe pokrywają 6/12 wartości, czyli 50%. Zatem:

- A. Odpowiedź poprawna.
- B. Odpowiedź niepoprawna.
- C. Odpowiedź niepoprawna.
- D. Odpowiedź niepoprawna.

Pytanie 22.

FL-4.2.3 (K3)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Uczestnik programu, który oddawał rowery w terminie, może uzyskać rabat i otrzymać darmową koszulkę po 15. wypożyczeniu.
- B. Odpowiedź niepoprawna. Uczestnik programu, który oddawał rowery w terminie, może uzyskać rabat, ale nie może otrzymać darmowej koszulki, dopóki nie wypożyczy roweru 15 razy.
- C. Odpowiedź niepoprawna. Osoba nieuczestnicząca w programie lojalnościowym nie może uzyskać rabatu, nawet jeśli zawsze oddaje rowery w terminie.
- D. Odpowiedź poprawna. Brak rabatu jest prawidłowy, ponieważ dana osoba nie uczestniczy w programie ani nie oddała roweru w terminie, ale tylko uczestnicy mogą otrzymać darmową koszulkę. W związku z tym akcja nie jest poprawna.

Pytanie 23.

FL-4.2.4 (K3)

Poprawna odpowiedź: D

Przejścia „test” i „błąd” nie mogą występować w jednym przypadku testowym. To samo dotyczy obu przejść „gotowe”. Oznacza to, że do uzyskania pokrycia przejść potrzebne są co najmniej trzy przypadki testowe, na przykład:

- PT1: test, gotowe;
- PT2: uruchomienie, błąd, gotowe;
- PT3: uruchomienie, pauza, wznowienie, pauza, gotowe.

W związku z tym:

- A. Odpowiedź niepoprawna.
- B. Odpowiedź niepoprawna.
- C. Odpowiedź niepoprawna.
- D. Odpowiedź poprawna.

Pytanie 24.

FL-4.3.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Uzyskanie stuprocentowego pokrycia instrukcji kodu oznacza, że każda instrukcja musiała zostać wykonana i oceniona co najmniej raz, a więc w szczególności także każda instrukcja zawierająca defekt.
- B. Odpowiedź niepoprawna. Pokrycie zależy od tego, co jest testowane, a nie od liczby przypadków testowych. Na przykład w przypadku kodu

```
IF (x==0)  
    y=1;
```

jeden przypadek testowy ($x = 0$) pozwala osiągnąć 100% pokrycia instrukcji kodu, a dwa przypadki testowe ($x = 1$) i ($x = 2$) łącznie — tylko 50% pokrycia.

- C. Odpowiedź niepoprawna. Jeśli w kodzie występuje pętla, liczba możliwych ścieżek może być nieskończona, w związku z czym nie da się wykonać wszystkich możliwych ścieżek w kodzie.
- D. Odpowiedź niepoprawna. Testowanie gruntowne nie jest możliwe (patrz podrozdział na temat siedmiu zasad testowania w syllabusie). Na przykład w przypadku kodu

```
input x;  
print x;
```

każdy pojedynczy test z dowolną wartością x pozwala uzyskać stuprocentowe pokrycie instrukcji kodu, ale pokrywa on tylko jedną wartość wejściową.

Pytanie 25.

FL-4.3.3 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Zasadniczą zaletą technik białoskrzynkowych jest fakt, że podczas testowania uwzględniana jest cała implementacja oprogramowania.
- B. Odpowiedź niepoprawna. Miary pokrycia stosowane w technikach białoskrzynkowych zapewniają obiektywny pomiar pokrycia i dostarczają niezbędnych informacji umożliwiających wygenerowanie dodatkowych testów w celu jego zwiększenia.
- C. Odpowiedź niepoprawna. Białoskrzynkowe techniki testowania można stosować podczas przeprowadzania przeglądów (będących formą testowania statycznego).

- D. Odpowiedź poprawna. Jest to słaby punkt białośkrzynkowych technik testowania. Nie pozwalają one rozpoznać braków w implementacji, ponieważ bazują wyłącznie na strukturze przedmiotu testów, nie zaś na specyfikacji wymagań.

Pytanie 26.

FL-4.4.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Podstawowym założeniem techniki zgadywania błędów jest to, że tester próbuje zgadnąć, jakie pomyłki mógł popełnić programista i jakie defekty mogą występować w przedmiocie testów, na podstawie dotychczasowego doświadczenia (oraz niekiedy list kontrolnych).
- B. Odpowiedź niepoprawna. Chociaż tester, który był kiedyś programistą, może wykorzystać swoje osobiste doświadczenie przy zgadywaniu błędów, technika ta nie opiera się na dotychczasowej wiedzy testera w dziedzinie tworzenia oprogramowania.
- C. Odpowiedź niepoprawna. Zgadywanie błędów nie jest techniką oceny użyteczności i nie służy do przewidywania potencjalnych problemów w zakresie interakcji użytkowników z przedmiotem testów.
- D. Odpowiedź niepoprawna. Metoda powielania zadań związanych z wytwarzaniem oprogramowania ma szereg wad, które sprawiają, że jest niepraktyczna, takich jak konieczność posiadania przez testera umiejętności równoważnych umiejętnościom programisty oraz czasochłonność prac programistycznych. Nie jest to technika zgadywania błędów.

Pytanie 27.

FL-4.4.2 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Masz do czynienia z nowym produktem. Prawdopodobnie nie dysponujesz jeszcze listą kontrolną, a warunki testowe mogą nie być znane z uwagi na brak wymagań.
- B. Odpowiedź niepoprawna. Masz do czynienia z nowym produktem. Prawdopodobnie nie dysponujesz wystarczającą ilością informacji, aby móc poprawnie zgadywać błędy.
- C. Odpowiedź poprawna. Testowanie eksploracyjne jest najbardziej przydatne w przypadku posiadania niepełnych specyfikacji bądź w przypadku wykonywania testów pod presją czasu.
- D. Odpowiedź niepoprawna. Testowanie gałęzi jest czasochłonne, a kierownictwo chce otrzymać pierwsze wyniki testów już teraz. Ponadto testowanie gałęzi nie wymaga wiedzy merytorycznej.

Pytanie 28.

FL-4.5.2 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Retrospektywy służą do rejestrowania wyciągniętych wniosków oraz usprawniania procesu wytwarzania i testowania oprogramowania, nie zaś do dokumentowania kryteriów akceptacji.
- B. Odpowiedź poprawna. Jest to standardowy sposób dokumentowania kryteriów akceptacji.
- C. Odpowiedź niepoprawna. Słowna wymiana informacji nie umożliwia fizycznego udokumentowania kryteriów akceptacji związanych z historyjką użytkownika, co odpowiada „karcie” (ang. *card*) w modelu „3C”.
- D. Odpowiedź niepoprawna. Kryteria akceptacji dotyczą historyjki użytkownika, a nie planu testów. Ponadto kryteria akceptacji określają warunki, które muszą zostać spełnione, aby można było uznać historyjkę użytkownika za kompletną, a ryzyka nie są takimi warunkami.

Pytanie 29.

FL-4.5.3 (K3)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Test ten pokrywa dwa kryteria akceptacji: kryterium dotyczące edycji dokumentu i kryterium dotyczące zapisywania zmian.
- B. Odpowiedź niepoprawna. Kryteria akceptacji pokrywają czynności wykonywane przez redaktora, a nie przez właściciela treści.
- C. Odpowiedź niepoprawna. Wyznaczanie terminu publikacji zredagowanej treści może być przydatną funkcją, ale nie jest objęte kryteriami akceptacji.
- D. Odpowiedź niepoprawna. W kryteriach akceptacji jest mowa o tym, że redaktor może zlecać dokonanie aktualizacji właścicielowi treści, nie zaś innemu redaktorowi.

Pytanie 30.

FL-5.1.2 (K1)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Priorytety historyjek użytkownika ustala przedstawiciel jednostki biznesowej we współpracy z zespołem tworzącym oprogramowanie.
- B. Odpowiedź niepoprawna. Testerzy koncentrują się zarówno na aspektach funkcjonalnych, jak i na aspektach niefunkcjonalnych systemu będącego przedmiotem testów.
- C. Odpowiedź poprawna. Zgodnie z syllabusem jest to jeden ze sposobów, w jakie testerzy wnoszą wkład w planowanie iteracji i wydań.

- D. Odpowiedź niepoprawna. Wczesne projektowanie testów nie jest elementem planowania wydań i nie gwarantuje automatycznie wysokiej jakości przekazywanego do eksploatacji oprogramowania.

Pytanie 31.

FL-5.1.3 (K2)

Poprawna odpowiedź: C, E

- A. Odpowiedź niepoprawna. Gotowość środowiska testowego jest kryterium związanym z dostępnością zasobów, a tym samym zalicza się do kryteriów wejścia.
- B. Odpowiedź niepoprawna. Kryterium to jest związane z dostępnością zasobów, w związku z czym zalicza się do kryteriów wejścia.
- C. Odpowiedź poprawna. Szacowana gęstość defektów jest miarą staranności, a tym samym zalicza się do kryteriów wyjścia.
- D. Odpowiedź niepoprawna. Przekształcenie wymagań do określonego formatu powoduje powstanie wymagań testowalnych, w związku z czym jest to kryterium wejścia.
- E. Odpowiedź poprawna. Automatyzacja testów regresji jest kryterium zakończenia testów, a tym samym zalicza się do kryteriów wyjścia.

Pytanie 32.

FL-5.1.4 (K3)

Poprawna odpowiedź: D

W technice szacowania trzypunktowego stosuje się następujący wzór:

$E = (\text{optymistyczne} + 4 \times \text{najbardziej prawdopodobne} + \text{pesymistyczne})/6$,
czyli $E = (2 + (4 \times 11) + 14)/6 = 10$. W związku z tym poprawna jest odpowiedź D.

Pytanie 33.

FL-5.1.5 (K3)

Poprawna odpowiedź: A

Przypadek testowy PT 001 musi zostać wykonany jako pierwszy, a przypadek PT 002 jako drugi z uwagi na występujące zależności. Następnie należy wykonać przypadek PT 003, co wynika z ustalonych priorytetów, a potem kolejno PT 004 i PT 005. Zatem:

- A. Odpowiedź poprawna.
- B. Odpowiedź niepoprawna.
- C. Odpowiedź niepoprawna.
- D. Odpowiedź niepoprawna.

Pytanie 34.

FL-5.1.7 (K2)

Poprawna odpowiedź: A

Testowanie użyteczności zalicza się do kwadrantu Q3 (1C).

Testowanie modułowe zalicza się do kwadrantu Q1 (2A).

Testowanie funkcjonalne zalicza się do kwadrantu Q2 (3B).

Testowanie niezawodności zalicza się do kwadrantu Q4 (4D).

W związku z tym poprawna jest odpowiedź A.

Pytanie 35.

FL-5.2.4 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Nie zaakceptowano ryzyka — wręcz przeciwnie, zaproponowano konkretne działania zaradcze.
- B. Odpowiedź niepoprawna. Nie zaproponowano żadnych planów awaryjnych.
- C. Odpowiedź poprawna. Proponowane działania są związane z testowaniem, co jest formą łagodzenia ryzyka.
- D. Odpowiedź niepoprawna. Ryzyko nie jest przenoszone, lecz łagodzone.

Pytanie 36.

FL-5.3.3 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Kryteria akceptacji to warunki służące do ustalania, czy historyjka użytkownika jest gotowa. Nie pozwalają one wykazać postępu pracy.
- B. Odpowiedź niepoprawna. Raporty o defektach służą do informowania o wystąpieniu defektów. Nie pozwalają one wykazać postępu pracy.
- C. Odpowiedź niepoprawna. Sumaryczny raport z testów można utworzyć po zakończeniu iteracji, w związku z czym nie pozwala on wykazywać postępu w sposób ciągły w trakcie jej wykonywania.
- D. Odpowiedź poprawna. Wykresy spalania stanowią graficzne odzwierciedlenie pozostałości do wykonania pracy i pozostałego czasu. Ponadto są aktualizowane codziennie, dzięki czemu pozwalają na bieżąco wykazywać postęp prac.

Pytanie 37.

FL-5.4.1 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Śledzenie powiązań dotyczy relacji między kilkoma różnymi produktami pracy, nie zaś relacji między poszczególnymi wersjami tego samego produktu pracy.
- B. Odpowiedź niepoprawna. Testowanie pielęgnacyjne dotyczy wprowadzanych zmian i nie jest ściśle związane z zarządzaniem wersjami.
- C. Odpowiedź poprawna. W kontekście testowania zarządzanie konfiguracją może obejmować kontrolę wersji wszystkich elementów testowych.
- D. Odpowiedź niepoprawna. Inżynieria wymagań polega na pozyskiwaniu i dokumentowaniu wymagań oraz zarządzaniu nimi, nie jest natomiast ściśle związana z zarządzaniem wersjami skryptów testowych.

Pytanie 38.

FL-5.5.1 (K3)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Oczekiwany rezultat to „aplikacja powinna zaakceptować podane dane wejściowe i utworzyć użytkownika”. Rzeczywisty rezultat to „aplikacja zawiesza się po wprowadzeniu ciągu «Dane wejściowe do testów: \$ä»”.
- B. Odpowiedź niepoprawna. Podano odwołanie do przypadku testowego i do związanego z nim wymagania, a także wskazano, że defekt został odrzucony. Informacja o statusie defektu nie byłaby zbyt przydatna dla programistów.
- C. Odpowiedź poprawna. Nie wiadomo, w którym środowisku testowym wykryto anomalię oraz której aplikacji (i wersji) ona dotyczy.
- D. Odpowiedź niepoprawna. Raport o defekcie zawiera informację o tym, że wymagane jest pilne usunięcie anomalii oraz że problem ma charakter globalny (tzn. dotyczy wielu, a potencjalnie nawet wszystkich testowych kont administratora), jak również o tym, że anomalia ma duży wpływ na interesariuszy biznesowych.

Pytanie 39.

FL-6.1.1 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Monitorowanie testów polega na ciągłym sprawdzaniu wszystkich czynności testowych i porównywaniu rzeczywistego postępu z założeniami przyjętymi w planie testów, a nadzór nad testami polega na podejmowaniu działań, które są niezbędne do osiągnięcia celów testów określonych w planie testów. W ramach tych czynności nie są przygotowywane żadne dane testowe.

- B. Odpowiedź niepoprawna. Analiza testów polega na przeanalizowaniu podstawy testów w celu zidentyfikowania warunków testowych i określenia ich priorytetów, a projektowanie testów polega na przekształceniu warunków testowych w przypadki testowe i inne testalia. W ramach tych czynności nie są przygotowywane dane testowe.
- C. Odpowiedź poprawna. Implementacja testów polega na utworzeniu lub pozyskaniu testaliów niezbędnych do wykonywania testów (np. danych testowych).
- D. Odpowiedź niepoprawna. Ukończenie testów obejmuje czynności, które są wykonywane w momencie osiągnięcia kamieni milowych projektu (takich jak przekazanie do eksploatacji, zakończenie iteracji lub ukończenie testów danego poziomu). Na tym etapie jest już za późno na przygotowywanie danych.

Pytanie 40.

FL-6.2.1 (K1)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Automatyzacja testów nie powoduje wprowadzenia nieznanych regresji w środowisku eksploatacyjnym.
- B. Odpowiedź poprawna. Niewłaściwy przydział nakładów pracy na utrzymanie testaliów jest ryzykiem.
- C. Odpowiedź niepoprawna. Narzędzia testowe należy dobierać w taki sposób, aby użytkownicy mieli zaufanie do nich i do związanych z nimi testaliów.
- D. Odpowiedź niepoprawna. Podstawowym celem automatyzacji testów jest ograniczenie testowania manualnego, dlatego jest to korzyść, a nie ryzyko.

Dodatkowe przykładowe pytania — odpowiedzi

Pytanie A1.

FL-1.1.2 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. Proces wyszukiwania, analizowania i usuwania przyczyn awarii w module lub systemie jest nazywany debugowaniem.
- B. Odpowiedź niepoprawna. Testowanie to proces polegający na planowaniu, przygotowywaniu i dokonywaniu oceny modułu lub systemu oraz powiązanych z nim produktów pracy w celu ustalenia, czy spełniają one wyspecyfikowane wymagania, wykazania, że są one zdatne do użytku zgodnie z przeznaczeniem, oraz wykrycia ewentualnych defektów. Proces ten nie jest związany z usuwaniem przyczyn awarii.
- C. Odpowiedź niepoprawna. Pozyskiwanie wymagań to proces polegający na gromadzeniu, rejestrowaniu i scalaniu wymagań pochodzących z dostępnych źródeł. Proces ten nie jest związany z usuwaniem przyczyn awarii.
- D. Odpowiedź niepoprawna. Zarządzanie defektami to proces rozpoznawania, rejestrowania, klasyfikowania, badania, rozwiązywania i usuwania defektów. Proces ten nie jest związany z usuwaniem przyczyn awarii.

Pytanie A2.

FL-1.2.2 (K1)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Patrz uzasadnienie do odpowiedzi D.
- B. Odpowiedź niepoprawna. Patrz uzasadnienie do odpowiedzi D.
- C. Odpowiedź niepoprawna. Patrz uzasadnienie do odpowiedzi D.
- D. Odpowiedź poprawna. Testowanie i zapewnienie jakości nie są tożsame. Testowanie to proces obejmujący wszystkie czynności związane z cyklem wytwarzania oprogramowania (zarówno statyczne, jak i dynamiczne), polegający na planowaniu, przygotowywaniu i dokonywaniu oceny modułu lub systemu oraz powiązanych z nim produktów pracy w celu ustalenia, czy spełniają one wyspecyfikowane wymagania, wykazania, że są one zdatne do użytku zgodnie z przeznaczeniem, oraz wykrycia ewentualnych defektów. Zapewnienie jakości skupia się na ustanawianiu, wprowadzaniu, monitorowaniu i udoskonalaniu procesów związanych z jakością oraz na ich przestrzeganiu.

Pytanie A3.

FL-1.2.3 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Podstawową przyczyną jest rozproszenie programisty podczas pisania kodu.
- B. Odpowiedź niepoprawna. Akceptowanie nieprawidłowych danych wejściowych to awaria.
- C. Odpowiedź niepoprawna. Pomyłka to błąd w myśleniu, który spowodował umieszczenie defektu w kodzie.
- D. Odpowiedź poprawna. Problem w kodzie to defekt.

Pytanie A4.

FL-1.4.3 (K2)

Poprawna odpowiedź: D

Rozpatrywane testalia to karta opisu testu. Karty opisu powstają na etapie projektowania testów. W związku z tym poprawna jest odpowiedź D.

Pytanie A5.

FL-1.4.4 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Przeprowadzenie analizy wpływu nie pozwala uzyskać informacji na temat kompletności testów, pomaga natomiast w wyborze właściwych przypadków testowych, które mają zostać wykonane.
- B. Odpowiedź niepoprawna. Śledzenie powiązań nie dostarcza informacji na temat szacowanego poziomu ryzyka rezydualnego, jeśli przypadki testowe nie zostały powiązane z ryzykami.
- C. Odpowiedź poprawna. Przeprowadzenie analizy wpływu zmian pomaga w wyborze przypadków testowych na potrzeby testowania regresji.
- D. Odpowiedź niepoprawna. Przeanalizowanie powiązań między podstawą testów, przedmiotami testów i przypadkami testowymi nie ułatwia wyboru danych testowych umożliwiających osiągnięcie zakładanego pokrycia przedmiotu testów. Wybór danych testowych jest powiązany w większym stopniu z analizą i implementowaniem testów, nie zaś ze śledzeniem powiązań.

Pytanie A6.

FL-1.5.3 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Za jakość powinny odpowiadać wszystkie osoby pracujące przy projekcie, a nie tylko członkowie zespołu testowego.

- B. Odpowiedź niepoprawna. Po pierwsze, nieprzestrzeganie przez zewnętrzny zespół terminów realizacji nie jest korzyścią. Po drugie, nie ma podstaw, by sądzić, że zewnętrzni testerzy będą czuli się zwolnieni z obowiązku ścisłego przestrzegania terminów.
- C. Odpowiedź niepoprawna. Nie zaleca się, aby zespół testowy pracował w całkowitej izolacji. Ponadto oczekuje się, że taki zespół będzie zwracał uwagę na zmieniające się wymagania projektu i będzie komunikował się na bieżąco z programistami.
- D. Odpowiedź poprawna. Specyfikacje nigdy nie są doskonałe, co oznacza, że programista musi przyjąć pewne założenia. Zaletą korzystania z usług niezależnych testerów jest to, że mogą oni podważać i weryfikować takie założenia oraz ich interpretację dokonaną przez programistę.

Pytanie A7.

FL-2.1.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. W przypadku sekwencyjnych modeli wytwarzania oprogramowania w początkowych fazach procesu testerzy uczestniczą w przeglądach wymagań, analizie testów oraz projektowaniu testów. Kod wykonywalny powstaje z reguły w późniejszych fazach, co uniemożliwia przeprowadzenie testowania dynamicznego na wczesnym etapie cyklu wytwarzania oprogramowania.
- B. Odpowiedź niepoprawna. Testowanie statyczne można zawsze wykonywać we wczesnej fazie cyklu wytwarzania oprogramowania.
- C. Odpowiedź niepoprawna. Planowanie testów powinno odbywać się we wczesnej fazie cyklu wytwarzania oprogramowania, przed rozpoczęciem projektu testowania.
- D. Odpowiedź niepoprawna. Testowanie akceptacyjne można wykonywać dopiero po udostępnieniu działającego produktu, a w sekwencyjnych modelach cyklu wytwarzania oprogramowania jest on zwykle późno w cyklu wytwarzania.

Pytanie A8.

FL-2.1.4 (K2)

Poprawna odpowiedź: C

- i. Prawda. Szybsze wprowadzanie produktów do eksploatacji i na rynek to zaleta metodyki DevOps.
- ii. Fałsz. Automatyzacja testów pozwala zwykle zmniejszyć nakłady pracy związane z wykonywaniem testów manualnych.
- iii. Prawda. Zapewnienie stałej dostępności wykonywalnego oprogramowania jest zaletą tej metodyki.
- iv. Fałsz. W tym przypadku wymagana jest większa liczba testów regresji.

- v. Fałsz. Nie wszystkie czynności można zautomatyzować, a skonfigurowanie struktury do testów automatycznych jest kosztowne.

W związku z tym poprawna jest odpowiedź C.

Pytanie A9.

FL-2.2.2 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Z faktu, że wymaganie dotyczące wydajności systemu zostało określone bezpośrednio przez klienta oraz że wydajność jest istotna z biznesowego punktu widzenia (wysoki priorytet), nie wynika, że są to testy funkcjonalne, ponieważ nie sprawdzają one tego, „co” wykonuje system, ale to, „jak” jest to wykonywane (tzn. jak szybko przetwarzane są zamówienia).
- B. Odpowiedź poprawna. Jest to przykład testowania wydajnościowego, które jest typem testowania niefunkcjonalnego.
- C. Odpowiedź niepoprawna. Ze scenariusza nie wynika, czy jednym z warunków testowych jest interakcja z interfejsem użytkownika. Jednak nawet gdyby tak było, głównym celem testów pozostaje sprawdzenie wydajności, a nie użyteczności.
- D. Odpowiedź niepoprawna. Znajomość struktury wewnętrznej kodu nie jest niezbędna do wykonywania testów wydajnościowych. Testy wydajności można wykonać, nie znając struktury programu.

Pytanie A10.

FL-2.3.1 (K2)

Poprawna odpowiedź: A

- A. Odpowiedź poprawna. W przypadku wycofywania systemu może być konieczne przetestowanie migracji danych, co jest formą testowania pielęgnacyjnego.
- B. Odpowiedź niepoprawna. Testowanie regresji sprawdza, czy poprawka nie wpłynęła negatywnie na zachowanie innych części kodu, ale w tym przypadku jest mowa o migracji do nowego systemu.
- C. Odpowiedź niepoprawna. Testowanie modułowe koncentruje się na poszczególnych modułach sprzętowych lub programowych, a nie na migracji danych.
- D. Odpowiedź niepoprawna. Testowanie integracyjne skupia się na interakcjach między modułami i/lub systemami, a nie na migracji danych.

Pytanie A11.

FL-3.1.1 (K1)

Poprawna odpowiedź: C

Przeglądem nie można objąć tylko kodu wykonywalnego innych firm. W związku z tym poprawna jest odpowiedź C.

Pytanie A12.

FL-3.1.3 (K2)

Poprawna odpowiedź: D

- i. Zachowanie takie można łatwo wykryć podczas działania oprogramowania, w związku z czym używa się do tego testowania dynamicznego.
- ii. Odstępstwa od standardu są typowym przykładem defektu, który łatwiej jest wykryć w testowaniu statycznym.
- iii. Jeśli podczas testu jest uruchamiane oprogramowanie, mamy do czynienia z testowaniem dynamicznym.
- iv. Jak najwcześniejsze zidentyfikowanie defektów jest celem testów zarówno w testowaniu statycznym, jak i w testowaniu dynamicznym.
- v. Luki związane z możliwością śledzenia lub pokryciem podstawy testów są typowym przykładem defektu, który łatwiej jest wykryć w testowaniu statycznym.

W związku z tym poprawna jest odpowiedź D.

Pytanie A13.

FL-3.2.2 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. We wszystkich typach przeglądów, nawet w przeglądach nieformalnych, występuje więcej niż jedna rola.
- B. Odpowiedź poprawna. W procesie przeglądu formalnego wykonywanych jest kilka czynności.
- C. Odpowiedź niepoprawna. Dokumentacja będąca przedmiotem przeglądu powinna być przekazywana uczestnikom jak najwcześniej.
- D. Odpowiedź niepoprawna. Defekty wykryte podczas przeglądu powinny być zgłoszane.

Pytanie A14.

FL-3.2.3 (K1)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Jest to zadanie lidera przeglądu.
- B. Odpowiedź poprawna. Jest to zadanie kierownictwa w przeglądzie formalnym.
- C. Odpowiedź niepoprawna. Jest to zadanie moderatora.
- D. Odpowiedź niepoprawna. Jest to zadanie protokolanta.

Pytanie A15.

FL-4.2.2 (K3)

Poprawna odpowiedź: C

Występują trzy klasy równoważności: {..., 10, 11}, {12} oraz {13, 14, ...}. Wartości brzegowe to 11, 12 i 13. W przypadku trzypunktowej analizy wartości brzegowych dla każdej wartości brzegowej musimy przetestować tę wartość oraz obie wartości sąsiednie, czyli:

- dla wartości 11 testujemy wartości 10, 11 i 12;
- dla wartości 12 testujemy wartości 11, 12 i 13;
- dla wartości 13 testujemy wartości 12, 13 i 14.

Łącznie konieczne jest przetestowanie wartości 10, 11, 12, 13 oraz 14. Zatem:

- A. Odpowiedź niepoprawna.
- B. Odpowiedź niepoprawna.
- C. Odpowiedź poprawna.
- D. Odpowiedź niepoprawna.

Pytanie A16.

FL-4.3.2 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. W tej sytuacji wciąż wymagany jest co najmniej jeden przypadek testowy, ponieważ istnieje co najmniej jedna gałąź (bezwarkowa) wymagająca pokrycia.
- B. Odpowiedź niepoprawna. Pokrycie wyłącznie gałęzi bezwarunkowych nie oznacza pokrycia wszystkich gałęzi warunkowych.
- C. Odpowiedź niepoprawna. Uzyskanie stuprocentowego pokrycia gałęzi zapewnia uzyskanie stuprocentowego pokrycia instrukcji kodu, ale nie odwrotnie. Na przykład w przypadku decyzji IF bez bloku ELSE jeden test wystarczy do osiągnięcia stuprocentowego pokrycia instrukcji kodu, ale pozwala osiągnąć jedynie pięćdziesięcioprocentowe pokrycie gałęzi.
- D. Odpowiedź poprawna. Każdy wynik decyzji odpowiada gałęzi warunkowej, w związku z czym stuprocentowe pokrycie gałęzi oznacza również stuprocentowe pokrycie decyzji.

Pytanie A17.

FL-4.4.3 (K2)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Książka zawiera wskazówki o charakterze ogólnym i nie jest formalnym dokumentem z wymaganiami, specyfikacją ani zbiorem przypadków użycia, historyjek użytkownika lub procesów biznesowych.
- B. Odpowiedź niepoprawna. Chociaż listę można by było uznać za zbiór kart opisu testów, bardziej przypomina ona listę warunków testowych wymagających sprawdzenia.

- C. Odpowiedź poprawna. Lista najlepszych praktyk w dziedzinie tworzenia interfejsów użytkownika stanowi listę warunków testowych wymagających systematycznego sprawdzenia.
- D. Odpowiedź niepoprawna. Testy nie skupiają się na potencjalnych awariach, lecz na zapoznaniu się z kwestiami, które są ważne dla użytkownika (w kategoriach użyteczności).

Pytanie A18.

FL-4.5.1 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Wspólne pisanie historyjek użytkownika oznacza, że wszyscy interesariusze razem tworzą historyjki w celu wypracowania wspólnej wizji.
- B. Odpowiedź poprawna. Wspólne pisanie historyjek użytkownika oznacza, że wszyscy interesariusze razem tworzą historyjki w celu wypracowania wspólnej wizji.
- C. Odpowiedź niepoprawna. Wspólne pisanie historyjek użytkownika oznacza, że wszyscy interesariusze razem tworzą historyjki w celu wypracowania wspólnej wizji.
- D. Odpowiedź niepoprawna. Jest to lista cech, jakimi powinna się charakteryzować każda historyjka użytkownika, nie zaś opis podejścia opartego na współpracy.

Pytanie A19.

FL-5.1.1 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Podany fragment tekstu zawiera informacje na temat poziomów testów i kryteriów wyjścia, które są elementem podejścia do testowania.
- B. Odpowiedź niepoprawna. Podany fragment tekstu zawiera informacje na temat poziomów testów i kryteriów wyjścia, które są elementem podejścia do testowania.
- C. Odpowiedź niepoprawna. Podany fragment tekstu zawiera informacje na temat poziomów testów i kryteriów wyjścia, które są elementem podejścia do testowania.
- D. Odpowiedź poprawna. Podany fragment tekstu zawiera informacje na temat poziomów testów i kryteriów wyjścia, które są elementem podejścia do testowania.

Pytanie A20.

FL-5.1.4 (K3)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Decyzja powinna zostać podjęta przez cały zespół i nie powinna być uchylna przez jedną osobę.
- B. Odpowiedź poprawna. Jeśli oszacowania dotyczące testów są różne, ale różnice między wynikami są niewielkie, dopuszczalne jest przyjęcie wartości, która uzyskała największą liczbę głosów.
- C. Odpowiedź niepoprawna. Nie osiągnięto jeszcze konsensusu, ponieważ niektóre osoby podają wartość 13, a inne — 8.
- D. Odpowiedź niepoprawna. Nie należy usuwać cech tylko dlatego, że zespół nie osiągnął porozumienia co do szacowanej pracochności testów.

Pytanie A21.

FL-5.1.6 (K1)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. W modelu piramidy testów podkreśla się fakt, że na niższych poziomach testów powinna być wykonywana większa liczba testów.
- B. Odpowiedź poprawna. Stwierdzenie, że automatyzacja testów powinna być bardziej sformalizowana w pobliżu szczytu piramidy, jest niezgodne z prawdą.
- C. Odpowiedź niepoprawna. Do automatyzacji testowania modułowego i testowania integracji modułów używa się zwykle narzędzi opartych na interfejsach API.
- D. Odpowiedź niepoprawna. W przypadku testowania systemowego i testowania akceptacyjnego testy automatyczne tworzy się zwykle przy użyciu narzędzi wyposażonych w graficzny interfejs użytkownika.

Pytanie A22.

FL-5.2.1 (K1)

Poprawna odpowiedź: C

- A. Odpowiedź niepoprawna. Wpływ ryzyka i prawdopodobieństwo ryzyka są od siebie niezależne.
- B. Odpowiedź niepoprawna. Wpływ ryzyka i prawdopodobieństwo ryzyka są od siebie niezależne.
- C. Odpowiedź poprawna. Wpływ ryzyka i prawdopodobieństwo ryzyka są od siebie niezależne.
- D. Odpowiedź niepoprawna. Do obliczenia poziomu ryzyka niezbędne są oba czynniki.

Pytanie A23.

FL-5.2.2 (K2)

Poprawna odpowiedź: A

- i. Ryzyko projektowe.
- ii. Ryzyko produktowe.
- iii. Ryzyko produktowe.
- iv. Ryzyko projektowe.
- v. Ryzyko produktowe.

W związku z tym poprawna jest odpowiedź A.

Pytanie A24.

FL-5.2.3 (K2)

Poprawna odpowiedź: D

- A. Odpowiedź niepoprawna. Jest to przykład czynności związanej z monitorowaniem ryzyka, nie zaś z analizą ryzyka.
- B. Odpowiedź niepoprawna. Jest to przykład decyzji dotyczącej architektury, a tym samym niezwiązanej z testowaniem.
- C. Odpowiedź niepoprawna. Jest to przykład wykonania analizy ryzyka metodą ilościową, co nie ma związku ze starannością ani zakresem testowania.
- D. Odpowiedź poprawna. Przykład ten pokazuje, jak analiza ryzyka wpływa na staranność (tj. szczegółowość) testowania.

Pytanie A25.

FL-5.3.1 (K1)

Poprawna odpowiedź: A, D

- A. Odpowiedź poprawna. Liczba wykrytych defektów ma związek z jakością przedmiotu testów.
- B. Odpowiedź niepoprawna. Jest to miara efektywności testów, a nie jakości przedmiotu testów.
- C. Odpowiedź niepoprawna. Liczba wykonanych przypadków testowych nic nam nie mówi o jakości. Informacji takich mogłyby dostarczyć wyniki testów.
- D. Odpowiedź poprawna. Gęstość defektów ma związek z jakością przedmiotu testów.
- E. Odpowiedź niepoprawna. Czas usuwania defektów jest miarą dotyczącą procesu, która nic nam nie mówi o jakości produktu.

Pytanie A26.

FL-5.3.2 (K2)

Poprawna odpowiedź: B

- A. Odpowiedź niepoprawna. Przeszkody w testowaniu mogą być przeszkodami wysokiego poziomu związanymi z kwestiami biznesowymi, w związku z czym jest to ważna informacja dla interesariuszy biznesowych.
- B. Odpowiedź poprawna. Pokrycie gałęzi jest miarą techniczną używaną przez programistów i testerów technicznych. Informacja ta nie ma znaczenia dla przedstawicieli jednostek biznesowych.
- C. Odpowiedź niepoprawna. Postęp testów jest czynnikiem związanym z projektem, w związku z czym informacja ta może być przydatna dla przedstawicieli jednostek biznesowych.
- D. Odpowiedź niepoprawna. Ryzyka wpływają na jakość produktu, w związku z czym informacja ta może być przydatna dla przedstawicieli jednostek biznesowych.

Bibliografia

Normy i standardy

- ISO 26262 (2011)
Road Vehicles — Functional Safety
- ISO 31000 (2018)
Risk Management
- ISO/IEC/IEEE 29119 (2013, 2015)
Software and systems engineering — Software testing
- ISO/IEC/IEEE 29119-1 (2013)
Software and systems engineering — Software testing — Part 1: Concepts and definitions
- ISO/IEC/IEEE 29119-2 (2013)
Software and systems engineering — Software testing — Part 2: Test processes
- ISO/IEC/IEEE 29119-3 (2013)
Software and systems engineering — Software testing — Part 3: Test documentation
- ISO/IEC/IEEE 29119-4 (2015)
Software and systems engineering — Software testing — Part 4: Test techniques
- [ISO/IEC 25010] (2011)
Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models
- [ISO/IEC 20246] (2017)
Software and systems engineering — Work product reviews

Dokumenty ISTQB

- [ISTQB-S]
ISTQB — Słownik terminów testowych, polska i angielska wersja online słownika: <https://glossary.istqb.org/>
- [ISTQB TA 2021]
ISTQB Certyfikowany Tester — Syllabus poziomu zaawansowanego — Analityk Testów (2021)

- [ISTQB TTA 2021]
ISTQB Certyfikowany Tester — Syabus poziomu zaawansowanego — Techniczny Analityk Testów (2021)
- [ISTQB FL 2023]
ISTQB Certyfikowany tester. Przegląd poziomu podstawowego, wersja 4.0, 2023
- [ISTQB SEC 2016]
ISTQB Certified Tester — Advanced Level Syllabus — Security Tester (2016)

Książki i artykuły

- [Adzic 2009]
Adzic G. (2009), *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*, Neuri Limited.
- [Adzic 2011]
Adzic G. (2011), *Specification by Example: How Successful Teams Deliver the Right Software*, Manning Publications: Shelter Island, NY.
- [Ammann 2016]
Ammann P., Offutt J. (2016), *Introduction to Software Testing* (2e), Cambridge University Press.
- [Andrews 2006]
Andrews M., Whittaker J. (2006), *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley Professional.
- [Beizer 1990]
Beizer B. (1990), *Software Testing Techniques* (2e), Van Nostrand Reinhold.
- [Boehm 1981]
Boehm B. (1981), *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, NJ.
- [Brykczynski 1999]
Brykczynski B. (1999), *A survey of software inspection checklists*, „ACM SIGSOFT Software Engineering Notes”, 24(1), s. 82 – 89.
- [Buxton 1970]
Buxton J.N., Randell B., eds (1970), *Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee*, Rome, Italy, 27 – 31 October 1969, s. 16.
- [Chelimsky 2010]
Chelimsky D. et al. (2010), *The Rspec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*, The Pragmatic Bookshelf, Raleigh, NC.

- [Cohn 2004]
Cohn M. (2004), *User Stories Applied For Agile Software Development*, Addison-Wesley.
- [Cohn 2009]
Cohn M. (2009), *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley.
- [Copeland 2004]
Copeland L. (2004), *A Practitioner's Guide to Software Test Design*, Artech House.
- [Craig 2002]
Craig, R., Jaskiel S. (2002), *Systematic Software Testing*, Artech House.
- [Crispin 2008]
Crispin L., Gregory J. (2008), *Agile Testing: A Practical Guide for Testers and Agile Teams*, Pearson Education, Boston MA.
- [Enders 1975]
Enders A. (1975), *An Analysis of Errors and Their Causes in System Programs*, „IEEE Transactions on Software Engineering” 1(2), s. 140 – 149.
- [Fagan 1976]
Fagan M. (1976), *Design and Code Inspection to Reduce Errors in Program Development*, „IBM Systems Journal” 15, 3 1976, s. 182 – 211.
- [Forgács 2019]
Forgács I., Kovács A. (2019), *Practical Test Design: Selection of traditional and automated test design techniques*, BCS, The Chartered Institute for IT.
- [Gawande 2009]
Gawande A. (2009), *The Checklist Manifesto: How to Get Things Right*, Metropolitan Books, New York, NY.
- [Gärtner 2011]
Gärtner M. (2011), *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*, Pearson Education, Boston MA.
- [Gilb 1993]
Gilb T., Graham D. (1993), *Software Inspection*, Addison-Wesley.
- [Hendrickson 2013]
Hendrickson E. (2013), *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*, The Pragmatic Programmers.
- [Hetzl 1988]
Hetzl B. (1988), *The Complete Guide to Software Testing*, 2nd ed., John Wiley and Sons.

- [Jackson 2007]
Jackson D., Thomas M., Millett L.I., eds (2007), *Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software Systems*, National Research Council, NAS.
- [Jeffries 2000]
Jeffries R., Anderson A., Hendrickson C. (2000), *Extreme Programming Installed*, Addison-Wesley Professional.
- [Johnson 1996]
Johnson P.M. (1996), *Introduction to formal technical reviews*, University College of London Press.
- [Jones 2012]
Jones C., Bonsignour O. (2012), *The Economics of Software Quality*, Addison-Wesley.
- [Jorgensen 2014]
Jorgensen P. (2014), *Software Testing, A Craftsman's Approach* (4e), CRC Press, Boca Raton FL.
- [Kan 2003]
Kan S. (2003), *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley.
- [Kaner 1999]
Kaner C., Falk J., Nguyen H.Q. (1999), *Testing Computer Software*, 2nd ed., Wiley.
- [Kaner 2011]
Kaner C., Bach J., Pettichord B. (2011), *Lessons Learned in Software Testing: A Context-Driven Approach*, 1st ed., Wiley.
- [Kim 2016]
Kim G., Humble J., Debois P., Willis J. (2016), *The DevOps Handbook*, Portland, OR.
- [Koomen 2006]
Koomen T., van der Aalst L., Broekman B., Vroon M. (2006), *TMap Next for result-driven testing*, UTN Publishers, The Netherlands.
- [Linz 2014]
Linz T. (2014), *Testing in Scrum: A Guide for Software Quality Assurance in the Agile World*, Rocky Nook.
- [Manna 1978]
Manna Z., Waldinger R. (1978), *The logic of computer programming*, „IEEE Transactions on Software Engineering” 4(3), s. 199 – 229.
- [Myers 2011]
Myers G. (2011), *The Art of Software Testing* (3e), John Wiley & Sons, New York NY.

- [Nazir 2020]
Nazir S., Fatima N., Chuprat S. (2020), *Modern Code Review Benefits-Primary Findings of A Systematic Literature Review*, „ICSIM '20: Proceedings of the 3rd International Conference on Software Engineering and Information Management”, s. 210 – 215.
- [Neeham 1969]
Neeham R. (1969), *Operational experience with the Cambridge multiple-access system*, „Computer Science and Technology, Conference Publication” 55, Institution of Electrical Engineers, London, s. 255 – 260.
- [Nielsen 1994]
Nielsen J. (1994), *Enhancing the explanatory power of usability heuristics*, „Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence”, s. 152 – 158, ACM Press.
- [O'Neill 1998]
O'Neill D. (1998), *National Software Quality Experiment. A lesson in measurement 1992–1997*, 23rd Annual Software Engineering Workshop, NASA Goddard Space Flight Center.
- [O'Regan 2019]
O'Regan G. (2019), *Concise Guide to Software Testing*, Springer Nature Switzerland.
- [Pressman 2019]
Pressman R.S. (2019), *Software Engineering. A Practitioner's Approach*, 9th ed., McGraw Hill.
- [Roman 2015]
Roman A. (2015), *Testowanie oprogramowania. Modele, techniki, narzędzia*, PWN.
- [Roman 2018]
Roman A. (2018), *Thinking-Driven Testing. The Most Reasonable Approach to Quality Control*, Springer Nature Switzerland.
- [Sauer 2000]
Sauer C. (2000), *The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research*, „IEEE Transactions on Software Engineering”, Volume 26 (1).
- [Schwaber 2002]
Schwaber K., Beedle M (2002), *Agile Software Development with Scrum*, Prentice-Hall, Upper Saddle River, NJ.
- [Stray 2021]
Stray V., Florea R., Paruch L. (2021), *Exploring human factors of the agile software tester*, „Software Quality Journal” 30(1), s. 1 – 27.
- [Veenendaal 2004]
Veenendaal E. van (ed.) (2004), *The Testing Practitioner*, UTN Publishers.

- [Veenendaal 2012]
Van Veenendaal, E (ed.) (2012) *Practical Risk-Based Testing, The PRISMA Approach*, UTN Publishers.
- [Watson 1996]
Watson A.H., Wallace D.R., McCabe T.J. (1996), *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, U.S. Dept. of Commerce, Technology Administration, NIST.
- [Westfall 2009]
Westfall L. (2009), *The Certified Software Quality Engineer Handbook*, ASQ Quality Press.
- [Whittaker 2002]
J.A. Whittaker (2002), *How to Break Software*, Pearson.
- [Whittaker 2003]
J.A. Whittaker, H.H. Thompson (2003), *How to Break Software Security*, Addison-Wesley.
- [Whittaker 2009]
J.A. Whittaker (2009), *Exploratory Software Testing. Tips, Tricks, Tours, and Techniques to Guide Test Design*, Addison-Wesley.
- [Wiegers 2001]
Wiegers K. (2001), *Peer Reviews in Software: A Practical Guide*, Addison-Wesley Professional

Źródła internetowe

- [Marick 2003]
Marick B. (2003) *Exploration through Example*,
<http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1>
- [UML 2017]
UML 2.5 — Unified Modeling Language Reference Manual,
<https://www.omg.org/spec/UML/2.5.1>
- [Web1]
<https://ssdip.bip.gov.pl/fobjects/download/13576/zalacznik-nr-1-do-opz-szablon-pt-a-pdf.html>
- [Web2]
Wake B. (2003), *INVEST in Good Stories, and SMART Tasks*,
<https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>
- [Web3]
Reid S. (2018) Software Reviews using ISO/IEC 20246,
<http://www.stureid.info/wp-content/uploads/2018/01/Software-Reviews.pdf>

Skorowidz

A

akceptacja, 253, 256
anityk 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
wpływ, 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

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
 produkту, 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 wytwarzania, 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
 retrospektyny, 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
 zступujące, 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****z**adania 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

PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

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

GRUPA
Helion