Certyfikowany Tester

Poziom Zaawansowany
Techniczny Analityk Testów
(CTAL-TTA)

Sylabus

wersja 4.0

wersja tłumaczenia PL 4.0.0

International Software Testing Qualifications Board

Informacja o prawach autorskich

Tłumaczenie z języka angielskiego: Adam Roman.

Przegląd tłumaczenia, korekta: Monika Petri-Starego.

Przegląd: Adam Ścierski.

Historia zmian

Wersja	Data	Uwagi
4.0	30.06.2021	Publikacja dokumentu w wersji 4.0
4.0	28.04.2021	Aktualizacja projektu na podstawie opinii z przeglądu wersji beta
4.0 beta	1.03.2021	Projekt zaktualizowany na podstawie opinii z przeglądu wersji alfa
4.0 alfa	7.12.2020	Projekt do przeglądu alfa z następującymi zmianami: • Poprawiono tekst w całym dokumencie • Usunięcie podsekcji związanej z K3 TTA-2.6.1 (2.6 Testowanie ścieżek podstawowych) i usunięcie LO • Usunięcie podsekcji związanej z K2 TTA-3.2.4 (3.2.4 Grafy wywołań) i usunięcie LO • Przepisanie podsekcji związanej z TTA-3.2.2 (3.2.2 Analiza przepływu danych) i przekształcenie jej w K3 • Przepisanie sekcji związanej z TTA-4.4.1 i TTA-4.4.2 (4.4. Testowanie niezawodności) • Przepisanie sekcji związanej z TTA-4.5.1 i TTA-4.5.2 (4.5 Testowanie wydajności) • Dodanie sekcji 4.9 dotyczącej profili operacyjnych • Przepisanie sekcji związanej z TTA-2.8.1 (2.7 Wybór technik testowania białoskrzynkowego) • Przepisanie TTA-3.2.1 w celu uwzględnienia złożoności cyklomatycznej (bez wpływu na pytania egzaminacyjne) • Przepisanie TTA-2.4.1 (MC/DC) w celu zapewnienia spójności z innymi LO dot. testowania białoskrzynkowego (bez wpływu na pytania egzaminacyjne)
2019	18.10.2019	Publikacja dokumentu w wersji 2019
2012	19.10.2012	Publikacja dokumentu w wersji 2012

Historia zmian polskiej wersji dokumentu

Wersja	Data	Uwagi
4.0.0	23.05.2026	Opublikowanie polskiego tłumaczenia sylabusa

Spis treści

Podziękowania

Wersja niniejszego dokumentu z 2019 r. została opracowana przez zespół ISTQB® – Grupę Roboczą ds. Poziomu Zaawansowanego: Graham Bath (wiceprzewodniczący), Rex Black, Judy McKay, Kenji Onoshi, Mike Smith (przewodniczący), Erik van Veenendaal.

W przeglądzie, komentowaniu i głosowaniu nad wersją niniejszego sylabusa z 2019 r. uczestniczyły następujące osoby: Dani Almog, Andrew Archer, Rex Black, Armin Born, Sudeep Chatterjee, Tibor Csöndes, Wim Decoutere, Klaudia Dusser-Zieger, Melinda Eckrich-Brájer, Peter Foldhazi, David Frei, Karol Frühauf, Jan Giesen, Attila Gyuri, Matthias Hamburg, Tamás Horváth, N. Khimanand, Jan te Kock, Attila Kovács, Claire Lohr, Rik Marselis, Marton Matyas, Judy McKay, Dénes Medzihradszky, Petr Neugebauer, Ingvar Nordström, Pálma Polyák, Meile Posthuma, Stuart Reid, Lloyd Roden, Adam Roman, Jan Sabak, Péter Sótér, Benjamin Timmermans, Stephanie van Dijck, Paul Weymouth.

Niniejszy dokument został opracowany przez zespół ISTQB® – Grupę Roboczą ds. Poziomu Zaawansowanego: Armin Born, Adam Roman, Stuart Reid.

Zaktualizowana wersja 4.0 niniejszego dokumentu została opracowana przez zespół ISTQB® – Grupę Roboczą ds. Poziomu Zaawansowanego: Armin Born, Adam Roman, Christian Graf, Stuart Reid.

W przeglądzie, komentowaniu i głosowaniu nad zaktualizowaną wersją 4.0 niniejszego sylabusa uczestniczyły następujące osoby:

Adél Vécsey-Juhász, Ágota Horváth, Benjamin Timmermans, Erwin Engelsma, Gary Mogyorodi, Geng Chen, Gergely Ágnecz, Jane Nash, Lloyd Roden,Matthias Hamburg, Meile Posthuma, Nishan Portoyan, Joan Killeen, Ole Chr. Hansen, Pálma Polyák, Paul Weymouth, Péter Földházi Jr., Rik Marselis, Sebastian Małyska, Tal Pe'er, Wang Lijuan, Zuo Zhenlei.

Zespół dziękuje zespołowi recenzentów i radom krajowym za sugestie i uwagi.

Niniejszy dokument został oficjalnie opublikowany przez Walne Zgromadzenie ISTQB® w dniu 30 czerwca 2021 r.

Wstęp

Cel niniejszego sylabusa

Niniejszy sylabus stanowi podstawę egzaminu ISTQB® Certyfikowany Tester Poziom Zaawansowany Techniczny Analityk Testów (CTAL-TTA). ISTQB® udostępnia sylabus:

1. Radom krajowym, w celu przetłumaczenia go na języki lokalne oraz dokonania akredytacji dostawców szkoleń (rady krajowe mogą dostosować sylabus do specyfiki języka oraz zmienić odniesienia do literatury, aby uwzględnić publikacje lokalne).
2. Organom certyfikacyjnym, w celu przygotowania pytań egzaminacyjnych w językach lokalnych, zgodnych z celami nauczania określonymi w niniejszym sylabusie.
3. Akredytowanym przez ISTQB® dostawcom szkoleń, w celu opracowania materiałów dydaktycznych oraz określenia odpowiednich metod nauczania.
4. Kandydatom do certyfikacji, w celu przygotowania do egzaminu certyfikacyjnego (ISTQB® rekomenduje udział w szkoleniu akredytowanym przez ISTQB® przed przystąpieniem do egzaminu na poziomie zaawansowanym).
5. Międzynarodowej społeczności specjalistów w dziedzinie inżynierii oprogramowania i systemów, w celu wspierania rozwoju zawodu testera oprogramowania i systemów oraz tworzenia publikacji, takich jak książki i artykuły.

ISTQB® może zezwolić innym podmiotom na wykorzystanie tego sylabusa w innych celach pod warunkiem uzyskania uprzedniej pisemnej zgody.

Certyfikowany tester – poziom zaawansowany w testowaniu oprogramowania

Kwalifikacja na poziomie zaawansowanym obejmuje odrębne sylabusy związane z następującymi rolami:

- kierownik testów,
- analityk testów,
- techniczny analityk testów,
- inżynier automatyzacji testów.

„Certyfikowany Tester ISTQB® Sylabus Poziomu Zaawansowanego Techniczny Analityk Testów (TTA). Omówienie sylabusa wersja 4.0” to oddzielny dokument [ISTQB_AL_TTA_OVERVIEW], w którym zawarto następujące informacje:

- rezultaty biznesowe,
- macierz powiązań między rezultatami biznesowymi a celami nauczania,
- podsumowanie.

Cele nauczania oraz poziomy wiedzy

Cele nauczania wspierają osiąganie rezultatów biznesowych i służą do przygotowania egzaminów certyfikacyjnych Certyfikowany Tester Poziom Zaawansowany Techniczny Analityk Testów. Poziomy wiedzy związane z poszczególnymi celami nauczania na poziomie K2, K3 oraz K4 przedstawiono na początku każdego rozdziału.

Poziomy te sklasyfikowano następująco:

- K2– zrozumieć,
- K3 – zastosować,
- K4 – przeanalizować.

Oczekiwane doświadczenie

Niektóre z celów nauczania określonych dla kwalifikacji Techniczny Analityk Testów zakładają posiadanie podstawowej wiedzy w następujących obszarach:

- ogólne koncepcje dotyczące programowania,
- ogólne koncepcje dotyczące architektury systemów.

Egzamin certyfikacyjny dla Technicznego Analityka Testów

Zakres egzaminu umożliwiającego uzyskanie certyfikatu Certyfikowany Tester Poziom Zaawansowany Techniczny Analityk Testów opiera się na niniejszym sylabusie. Przy udzielaniu odpowiedzi na pytania egzaminacyjne może być konieczne skorzystanie z materiału obejmującego więcej niż jeden rozdział tego sylabusa. Przedmiotem egzaminu może być treść wszystkich rozdziałów niniejszego sylabusa z wyjątkiem wstępu i załączników. W dokumencie znajdują się również odwołania do innych sylabusów ISTQB®, norm i książek, ale ich treść nie może być przedmiotem egzaminu w zakresie wykraczającym poza informacje streszczone w samym sylabusie.

Egzamin ma formę testu wielokrotnego wyboru i składa się z 45 pytań. Do zdania egzaminu niezbędne jest uzyskanie co najmniej 65% punktów.

Egzamin można zdawać w ramach akredytowanego szkolenia lub samodzielnie (np. w ośrodku egzaminacyjnym lub w ramach egzaminu otwartego). Ukończenie akredytowanego szkolenia nie jest warunkiem przystąpienia do egzaminu.

Wymagania dla osób przystępujących do egzaminu

Przed przystąpieniem do egzaminu certyfikacyjnego Certyfikowany Tester Poziom Zaawansowany Techniczny Analityk Testów należy wcześniej zdać egzamin certyfikacyjny związany z sylabusem Certyfikowany Tester Poziom Podstawowy.

Akredytacja szkoleń

Rada Krajowa ISTQB® może dokonywać akredytacji dostawców szkoleń, którzy oferują materiały dydaktyczne zgodne z niniejszym sylabusem. Wytyczne dotyczące akredytacji należy uzyskać od rady krajowej lub organu dokonującego akredytacji. Akredytowane szkolenie jest uznawane za zgodne z niniejszym sylabusem i może obejmować egzamin ISTQB® jako część szkolenia.

Poziom szczegółowości

Poziom szczegółowości niniejszego sylabusa umożliwia prowadzenie szkoleń i egzaminów spójnych na poziomie międzynarodowym.

Aby osiągnąć ten cel, w sylabusie uwzględniono:

- cele biznesowe opisujące intencje Poziomu Zaawansowanego Techniczny Analityk Testów,
- listę słów kluczowych, które kandydaci muszą być w stanie zapamiętać,
- cele nauczania dla każdego obszaru wiedzy, opisujące efekty uczenia się, które należy osiągnąć,
- opis kluczowych pojęć, w tym odniesienia do uznanych źródeł takich jak zatwierdzona literatura lub normy.

Treść sylabusa nie stanowi opisu całego obszaru wiedzy o testowaniu oprogramowania; odzwierciedla ona poziom szczegółowości, jaki należy uwzględnić w szkoleniach Certyfikowany Tester Poziom Zaawansowany Techniczny Analityk Testów. Koncentruje się ona na koncepcjach i technikach testowania, które mogą mieć zastosowanie do wszystkich projektów wytwarzania oprogramowania, bez względu na przyjęty model wytwarzania oprogramowania.

Sylabus nie zawiera żadnych konkretnych celów nauczania dotyczących specyficznych modeli wytwarzania oprogramowania, ale omawia sposób, w jaki koncepcje te znajdują zastosowanie w zwinnym wytwarzaniu oprogramowania, innych rodzajach iteracyjnych i przyrostowych modeli wytwarzania oprogramowania oraz w modelach sekwencyjnych.

Struktura sylabusa

Sylabus zawiera sześć rozdziałów, których treść może podlegać egzaminowaniu. Nagłówek najwyższego poziomu dla każdego rozdziału zawiera informację o minimalnym czasie trwania szkolenia obejmującego dany rozdział (nie podano czasu trwania dla podrozdziałów i mniejszych jednostek redakcyjnych). W przypadku akredytowanych szkoleń na przekazanie wiedzy zawartej w sylabusie potrzeba co najmniej 20 godzin wykładów. Czas ten podzielono na poszczególne rozdziały w następujący sposób:

- Rozdział 1: Zadania technicznego analityka testów w testowaniu opartym na ryzyku (30 minut).
- Rozdział 2: Białoskrzynkowe techniki testowania (300 minut).
- Rozdział 3: Analiza statyczna i dynamiczna (180 minut).
- Rozdział 4: Charakterystyki jakościowe w testach technicznych (345 minut).
- Rozdział 5: Przeglądy (165 minut).
- Rozdział 6: Narzędzia testowe i automatyzacja testów (180 minut).

Cele biznesowe

W tej sekcji wymieniono cele biznesowe (ang. *business outcomes*) oczekiwane od kandydata, który uzyskał certyfikat technicznego analityka testów na poziomie zaawansowanym.

Techniczny analityk testów potrafi:

TTA1	Rozpoznawać i klasyfikować typowe ryzyka związane z wydajnością, zabezpieczeniami, niezawodnością, przenaszalnością i utrzymywalnością systemów oprogramowania.
TTA2	Zapewniać elementy techniczne do planowania, projektowania i wykonywania testów w celu ograniczenia ryzyka związanego z wydajnością, zabezpieczeniami, niezawodnością, przenaszalnością i utrzymywalnością.
TTA3	Wybierać i stosować odpowiednie białoskrzynkowe techniki testowania, aby zapewnić odpowiedni poziom pewności testów w oparciu o zadane kryteria pokrycia.
TTA4	Skutecznie uczestniczyć w przeglądach z programistami i architektami oprogramowania, wykorzystując wiedzę na temat typowych defektów w kodzie i architekturze.
TTA5	Poprawiać charakterystyki jakościowe kodu i architektury, wykorzystując różne techniki analizy.
TTA6	Przedstawiać koszty i korzyści, jakich można oczekiwać po wprowadzeniu określonych rodzajów automatyzacji testów.
TTA7	Wybierać odpowiednie narzędzia do automatyzacji zadań związanych z testowaniem technicznym.
TTA8	Rozumieć zagadnienia techniczne i koncepcje związane ze stosowaniem automatyzacji testów.

1. Zadania technicznego analityka testów w testowaniu opartym na ryzyku (30 min.)

Słowa kluczowe

identyfikacja ryzyka, łagodzenie ryzyka, ocena ryzyka, ryzyko produktowe, ryzyko projektowe, testowanie oparte na ryzyku

Cele nauczania dla rozdziału 1

1.2 Zadania związane z testowaniem opartym na ryzyku

TTA-1.2.1 (K2) Omówić ogólne ryzyka, które zwykle musi wziąć pod uwagę techniczny analityk testów.

TTA-1.2.2 (K2) Omówić czynności wykonywane przez technicznego analityka testów w ramach podejścia do testowania opartego na ryzyku, związane z czynnościami testowymi.

1.1 Wprowadzenie

Jednym z obowiązków kierownika testów jest sprawowanie ogólnego nadzoru nad zdefiniowaniem strategii testów opartej na ryzyku oraz bieżące zarządzanie nią. Kierownik testów zazwyczaj włącza technicznego analityka testów w działania mające na celu zagwarantowanie prawidłowego wdrożenia podejścia opartego na ryzyku.

Techniczny analityk testów funkcjonuje w ramach struktury testowania (ang. *testing framework*) opartego na ryzyku, przygotowanej przez kierownika testów na potrzeby konkretnego projektu. Wnosi on swoją specjalistyczną wiedzę dotyczącą technicznych ryzyk produktowych, które są nierozerwalnie związane z projektem, takich jak ryzyka w obszarze zabezpieczeń, niezawodności oraz wydajności. Powinien także uczestniczyć w identyfikacji ryzyk projektowych oraz określaniu sposobów ich traktowania, w szczególności tych związanych ze środowiskami testowymi, na przykład w zakresie pozyskiwania i konfigurowania środowisk testowych do testów wydajnościowych, niezawodności czy testów zabezpieczeń.

1.2 Zadania związane z testowaniem opartym na ryzyku

Techniczni analitycy testów biorą czynny udział w realizacji następujących działań związanych z testowaniem opartym na ryzyku:

- identyfikacja ryzyka,
- ocena ryzyka,
- łagodzenie ryzyka.

Działania te są prowadzone w sposób iteracyjny przez cały czas trwania projektu, aby zespół projektowy mógł skutecznie reagować na nowe ryzyka oraz zmieniające się priorytety ryzyk, a także systematycznie oceniać poziom ryzyka i przekazywać informacje o jego statusie interesariuszom.

1.2.1 Identyfikacja ryzyka

W trakcie identyfikacji ryzyka prawdopodobieństwo wykrycia jak najszerszego zestawu potencjalnie istotnych ryzyk rośnie wraz z liczbą interesariuszy zaangażowanych w ten proces. Techniczni analitycy testów posiadają specyficzne kompetencje techniczne, dzięki czemu są szczególnie dobrze przygotowani do prowadzenia wywiadów z ekspertami, organizowania sesji „burzy mózgów” z członkami zespołu oraz analizowania ich doświadczeń w celu wskazania najbardziej prawdopodobnych obszarów ryzyka produktowego. W szczególności techniczni analitycy testów ściśle współdziałają z innymi interesariuszami, takimi jak programiści, architekci, specjaliści ds. eksploatacji, właściciele produktów, lokalne zespoły wsparcia, eksperci techniczni oraz technicy serwisowi, aby zidentyfikować obszary ryzyka technicznego wpływające na produkt i projekt. Zaangażowanie różnych grup interesariuszy umożliwia uwzględnienie wszystkich perspektyw, a koordynację tej współpracy zazwyczaj prowadzą kierownicy testów.

Ryzyka rozpoznawane przez technicznego analityka testów są zazwyczaj powiązane z charakterystykami jakościowymi produktu zdefiniowanymi w normie [ISO 25010] i opisanymi w rozdziale 4 niniejszego sylabusa.

1.2.2 Ocena ryzyka

Celem identyfikacji ryzyka jest wskazanie możliwie największej liczby istotnych ryzyk, natomiast ocena ryzyka polega na analizie tych zidentyfikowanych ryzyk w celu sklasyfikowania poszczególnych ryzyk oraz określenia prawdopodobieństwa ich wystąpienia i ich wpływu.

Prawdopodobieństwo wystąpienia ryzyka produktowego jest najczęściej rozumiane jako prawdopodobieństwo pojawienia się awarii w testowanym systemie. Techniczny analityk testów wspiera określenie prawdopodobieństwa wystąpienia każdego ryzyka produktowego, podczas gdy analityk testów wnosi wkład w ocenę potencjalnego wpływu biznesowego wystąpienia danego problemu.

Wystąpienie ryzyk projektowych może niekorzystnie wpłynąć na realizację całego projektu. W typowych przypadkach należy brać pod uwagę następujące ogólne ryzyka projektowe:

- spory między interesariuszami dotyczące wymagań technicznych,
- trudności komunikacyjne wynikające z geograficznego rozproszenia jednostek organizacyjnych odpowiedzialnych za wytwarzanie oprogramowania,
- narzędzia i technologie (w tym kluczowe umiejętności),
- presja czasu, ograniczenia zasobów oraz naciski ze strony kierownictwa,
- brak wcześniejszych działań w zakresie zapewnienia jakości,
- wysoka dynamika zmian wymagań technicznych.

Występowanie ryzyk produktowych może skutkować zwiększoną liczbą defektów. W typowych sytuacjach należy rozważyć następujące ogólne ryzyka produktowe:

- złożoność technologii,
- złożoność kodu,
- zakres zmian w kodzie źródłowym (dodanie, usunięcie, modyfikacja),
- duża liczba wykrytych defektów powiązanych z technicznymi charakterystykami jakościowymi (historia defektów),
- techniczne aspekty interfejsów oraz integracji.

Na podstawie dostępnych informacji dotyczących ryzyka techniczny analityk testów proponuje wstępne określenie prawdopodobieństwa wystąpienia ryzyka zgodnie z wytycznymi przekazanymi przez kierownika testów. Wartość ta może zostać zmieniona przez kierownika testów po uwzględnieniu opinii wszystkich interesariuszy. Wpływ ryzyka jest zazwyczaj określany przez analityka testów.

1.2.3 Łagodzenie ryzyka

W trakcie realizacji projektu techniczny analityk testów oddziałuje na sposób, w jaki zidentyfikowane ryzyka są uwzględniane w procesie testowania. Zazwyczaj obejmuje to następujące obszary:

- projektowanie przypadków testowych ukierunkowanych na obszary o wysokim poziomie ryzyka oraz wsparcie w ocenie ryzyka rezydualnego (resztkowego),
- łagodzenie ryzyka poprzez wykonywanie zaprojektowanych przypadków testowych oraz zastosowanie odpowiednich działań łagodzących i awaryjnych zgodnie z planem testów,

- ocenę ryzyk na podstawie dodatkowych informacji pozyskanych w trakcie projektu oraz wykorzystanie tych danych do wdrażania metod łagodzenia ryzyka, mających na celu obniżenie prawdopodobieństwa wystąpienia poszczególnych ryzyk.

Techniczny analityk testów często współdziała ze specjalistami, na przykład w obszarze zabezpieczeń i wydajności, w celu zdefiniowania metod łagodzenia ryzyka oraz elementów strategii testów. Dodatkowe informacje w tym zakresie można znaleźć w specjalistycznych sylabusach ISTQB®, takich jak sylabus Certyfikowany Tester – Tester Zabezpieczeń [CT_SEC_SYL] oraz sylabus Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

2. Białoskrzynkowe techniki testowania (300 min.)

Słowa kluczowe

białoskrzynkowa technika testowania, poziom nienaruszalności bezpieczeństwa, przepływ sterowania, testowanie API, testowanie decyzji, testowanie instrukcji, testowanie kombinacji warunków, warunek atomowy, zmodyfikowane testowanie warunkowo-decyzyjne (MC/DC)[1]

Cele nauczania dla rozdziału 2

2.2 Testowanie instrukcji

TTA-2.2.1 (K3) Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania instrukcji, aby osiągnąć zdefiniowany poziom pokrycia.

2.3 Testowanie decyzji

TTA-2.3.1 (K3) Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania decyzji, aby osiągnąć zdefiniowany poziom pokrycia.

2.4. Testowanie MC/DC

TTA-2.4.1 (K3) Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania MC/DC, aby osiągnąć pełne pokrycie MC/DC.

2.5. Testowanie kombinacji warunków

TTA-2.5.1 (K3) Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania kombinacji warunków, aby osiągnąć zdefiniowany poziom pokrycia.

2.6. Testowanie ścieżek podstawowych
(*podrozdział usunięty począwszy od wersji 4.0 niniejszego sylabusa*)

Cel nauczania TTA-2.6.1. został usunięty z niniejszej wersji sylabusa

2.7. Testowanie API

TTA-2.7.1 (K2) Rozumieć obszary zastosowania testów API i rodzaje defektów wykrywanych w takich testach.

2.8. Wybór białoskrzynkowej techniki testowania

TTA-2.8.1 (K4) Wybrać odpowiednią białoskrzynkową technikę testowania zgodnie z daną sytuacją projektową.

[1] W dalszej części sylabusa będą stosowane terminy: „testowanie MC/DC” oraz „pokrycie MC/DC” jako terminy powszechnie używane.

2.1 Wstęp

Niniejszy rozdział dotyczy białoskrzynkowych technik testowania. Techniki te są stosowane wobec kodu oraz innych struktur posiadających przepływ sterowania, takich jak diagramy przepływu procesów biznesowych.

Poszczególne techniki umożliwiają metodyczne wyprowadzanie przypadków testowych i skupiają się na określonych elementach struktury przedmiotu testów. Przypadki testowe tworzone z ich wykorzystaniem spełniają zdefiniowane kryteria pokrycia, które stanowią cele testowania. Poziom pokrycia jest mierzony i odnoszony do tych celów. Osiągnięcie pełnego (100%) pokrycia nie oznacza, że zestaw testów jest kompletny, lecz wskazuje, że w ramach danej techniki nie ma już możliwości zaprojektowania dodatkowych użytecznych testów dla analizowanej struktury.

Dane wejściowe do testów są konstruowane tak, aby wykonanie testu prowadziło do sprawdzenia konkretnej części kodu (np. instrukcji lub wyniku decyzji). Wyznaczenie danych wejściowych, które spowodują wykonanie określonego fragmentu kodu, może być skomplikowane, zwłaszcza gdy fragment ten znajduje się na końcu długiej ścieżki przepływu sterowania zawierającej wiele punktów decyzyjnych. Oczekiwane wyniki testów są ustalane na podstawie źródeł zewnętrznych wobec badanej struktury, takich jak wymagania, specyfikacja projektowa lub inna podstawa testów.

W niniejszym sylabusie opisano następujące techniki białoskrzynkowe:

- testowanie instrukcji,
- testowanie decyzji,
- testowanie MC/DC,
- testowanie kombinacji warunków,
- testowanie API.

Sylabus Certyfikowany Tester Poziom Podstawowy [CTFL_SYL] omawia testowanie instrukcji oraz testowanie decyzji[2]. Testowanie instrukcji koncentruje się na weryfikacji wykonywalnych instrukcji zawartych w kodzie, natomiast testowanie decyzji sprawdza wyniki decyzji.

Wymienione wyżej techniki bazują na predykatach decyzyjnych złożonych z warunków logicznych i umożliwiają wykrywanie podobnych typów defektów. Niezależnie od złożoności predykatu decyzji, przyjmuje on wartość prawdy lub fałszu, co determinuje wybór określonej ścieżki w kodzie. Defekt zostaje ujawniony w sytuacji, gdy oczekiwana ścieżka nie zostanie zrealizowana z powodu niezgodności wartości predykatu decyzji z założeniami.

Dodatkowe informacje dotyczące białoskrzynkowych technik testowania można znaleźć w [ISO 29119] oraz [Roman18].

2.2 Testowanie instrukcji

Testowanie instrukcji jest wykorzystywane do weryfikacji instrukcji wykonywalnych zawartych w kodzie. Pokrycie określa się jako iloraz liczby instrukcji uruchomionych podczas testów przez

[2] w sylabusie Certyfikowany Tester Poziom Podstawowy w. 4.0 testowanie gałęzi zastąpiło testowanie decyzji.

całkowitą liczbę instrukcji wykonywalnych w przedmiocie testów. Pokrycie to jest zazwyczaj wyrażane w procentach.

Obszar zastosowania. Uzyskanie pełnego pokrycia instrukcji należy traktować jako minimalny poziom oczekiwany dla całego testowanego kodu, chociaż w praktyce jego osiągnięcie nie zawsze jest możliwe.

Ograniczenia/trudności. Pełne pokrycie instrukcji powinno być postrzegane jako minimalny wymóg w odniesieniu do całego testowanego kodu, jednak w praktyce może być trudne do zrealizowania z powodu ograniczeń czasowych oraz nakładu pracy niezbędnego do jego uzyskania. Nawet wysoki poziom pokrycia instrukcji nie gwarantuje wykrycia wszystkich defektów, w szczególności tych związanych z logiką zawartą w kodzie. W wielu sytuacjach osiągnięcie 100% pokrycia instrukcji jest niemożliwe ze względu na występowanie kodu nieosiągalnego. Choć tworzenie nieosiągalnego kodu zasadniczo nie jest uznawane za dobrą praktykę programistyczną, w pewnych przypadkach takie instrukcje mogą się pojawić, na przykład w postaci kodu dla domyślnego bloku (ang. *default*) w instrukcji SWITCH-CASE, gdy wszystkie możliwe przypadki są poprawnie obsłużone przez poprzedzające go bloki CASE.

2.3 Testowanie decyzji

Testowanie decyzji jest stosowane w celu weryfikacji wyników decyzji występujących w kodzie. W tym podejściu projektuje się przypadki testowe odzwierciedlające przepływy sterowania następujące po punktach decyzyjnych. Przykładowo, dla instrukcji IF istnieje jeden przepływ sterowania w przypadku spełnienia warunku (*true*) oraz jeden w przypadku jego niespełnienia (*false*); dla instrukcji SWITCH-CASE może występować wiele przepływów odpowiadających wszystkim możliwym wynikom; natomiast dla pętli WHILE wyróżnia się przepływ, w którym warunek pętli jest prawdziwy, oraz przepływ, w którym warunek jest fałszywy.

Pokrycie oblicza się jako iloraz liczby wyników decyzji wykonanych przez testy przez całkowitą liczbę wyników wszystkich decyzji w przedmiocie testów. Pokrycie to jest zazwyczaj wyrażane w procentach. Należy przy tym pamiętać, że jeden przypadek testowy może obejmować sprawdzenie wielu wyników decyzji.

W odróżnieniu od opisanych poniżej technik testowania MC/DC oraz testowania kombinacji warunków, w testowaniu decyzji rozpatruje się decyzję jako całość. Jej rezultatem jest zawsze pojedyncza wartość PRAWDA lub FAŁSZ, niezależnie od stopnia złożoności wewnętrznej struktury danej decyzji.

Termin „testowanie gałęzi” (ang. *branch testing*) bywa często używany zamiennie z określeniem „testowanie decyzji”, ponieważ pełne pokrycie wszystkich gałęzi oraz pokrycie wszystkich wyników decyzji mogą zostać osiągnięte za pomocą tych samych testów. Testowanie gałęzi koncentruje się na gałęziach w kodzie, rozumianych jako krawędzie w grafie przepływu sterowania, czyli bezpośrednie przejścia pomiędzy instrukcjami. W programach pozbawionych decyzji powyższa definicja pokrycia decyzji prowadzi do ilorazu 0/0, który jest wartością niezdefiniowaną, niezależnie od liczby uruchomionych testów. W przypadku pokrycia gałęzi każdy test obejmuje jedną sekwencję gałęzi prowadzącą od wejścia do wyjścia (przy założeniu pojedynczego wejścia i pojedynczego wyjścia), co skutkuje osiągnięciem 100% pokrycia gałęzi. Aby uzgodnić różnice pomiędzy tymi dwiema miarami pokrycia, norma ISO 29119-4 wymaga, by dla kodu bez decyzji uruchomić co najmniej jeden test w celu uzyskania 100% pokrycia decyzji.

W rezultacie 100% pokrycia decyzji oraz 100% pokrycia gałęzi są pojęciami równoważnymi dla niemal wszystkich programów. Wiele narzędzi testowych oferujących mechanizmy pomiaru pokrycia, w tym narzędzia stosowane do testowania systemów krytycznych ze względów bezpieczeństwa, stosuje podobne podejście.

Obszar zastosowania. Ten poziom pokrycia należy rozważyć w sytuacjach, gdy testowany kod jest istotny lub bardzo istotny, czyli ma charakter krytyczny (patrz tabele w sekcji 2.8.2. dotyczące systemów krytycznych ze względów bezpieczeństwa). Technika ta może być stosowana zarówno wobec kodu, jak i dowolnego modelu wykorzystującego punkty decyzyjne, na przykład modeli procesów biznesowych.

Ograniczenia/trudności. Testowanie decyzji nie uwzględnia sposobu podejmowania decyzji opartych na wielu warunkach, dlatego defekty wynikające z określonych kombinacji wyników warunków mogą nie zostać wykryte.

2.4 Testowanie MC/DC

W odróżnieniu od testowania decyzji, w którym analizowana jest decyzja jako całość i sprawdzane są jej wyniki PRAWDA i FAŁSZ, testowanie MC/DC (zmodyfikowane pokrycie warunkowo-decyzyjne, ang. *modified condition/decision coverage*) koncentruje się na sposobie budowy decyzji złożonej z wielu warunków atomowych. Jeżeli decyzja zawiera tylko jeden warunek atomowy, wówczas technika ta w praktyce nie różni się od testowania decyzji.

Każda decyzja składa się z jednego lub więcej warunków atomowych, z których każdy przyjmuje wartość logiczną (ang. *Boolean value*). Warunki te są łączone za pomocą operatorów logicznych w celu wyznaczenia wyniku całej decyzji. W ramach tej techniki dla każdego warunku atomowego weryfikuje się poprawność jego niezależnego wpływu na wynik całej decyzji.

Technika MC/DC zapewnia silniejszy poziom pokrycia niż pokrycie instrukcji oraz pokrycie decyzji w sytuacjach, gdy decyzje zawierają wiele warunków. Przy założeniu, że decyzja składa się z N unikalnych, wzajemnie niezależnych warunków atomowych, uzyskanie MC/DC dla pojedynczej decyzji zazwyczaj wymaga wykonania testów dla N+1 elementów pokrycia. Testowanie MC/DC opiera się na parach testów, które pokazują, że zmiana wartości logicznej jednego warunku atomowego, przy niezmienionych pozostałych warunkach, powoduje zmianę wyniku decyzji. Należy przy tym zauważyć, że jeden przypadek testowy może prowadzić do wielokrotnego wykonania tej samej decyzji dla różnych kombinacji wartości warunków atomowych, dlatego pełne pokrycie MC/DC bywa często możliwe do osiągnięcia przy użyciu mniejszej liczby niż N+1 oddzielnych przypadków testowych.

Obszar zastosowania. Technika ta jest wykorzystywana w testach w przemyśle lotniczym, samochodowym oraz w innych branżach, w których stosowane są systemy krytyczne ze względów bezpieczeństwa. Stosuje się ją do testowania oprogramowania, którego awaria może prowadzić do katastrofalnych skutków. Testowanie MC/DC może stanowić rozsądny kompromis pomiędzy testowaniem decyzji a testowaniem kombinacji warunków (patrz podrozdział 2.5), ze względu na liczbę kombinacji warunków atomowych wymagających przetestowania. Kryterium to jest silniejsze niż testowanie decyzji, a jednocześnie wymaga znacznie mniejszej liczby przypadków testowych niż testowanie kombinacji warunków w przypadku decyzji złożonych z wielu warunków atomowych.

Ograniczenia/trudności. Uzyskanie pokrycia MC/DC może być trudne w sytuacjach, gdy w decyzji z wieloma warunkami wielokrotnie występuje ta sama zmienna. W takich przypadkach warunki mogą być ze sobą powiązane (ang. *coupled*), co może uniemożliwić zmianę wartości jednego warunku w sposób prowadzący do zmiany wyniku decyzji wyłącznie z tego powodu. Jednym ze sposobów radzenia sobie z tym problemem jest przyjęcie założenia, że w testowaniu MC/DC analizowane są wyłącznie niepowiązane warunki atomowe. Alternatywnie można indywidualnie analizować każde wystąpienie decyzji zawierającej warunki powiązane.

Niektóre kompilatory i/lub interpretery są zaprojektowane w taki sposób, że podczas obliczania wartości złożonych wyrażeń decyzyjnych stosowane jest tzw. zwarcie (ang. *short-circuiting*). Oznacza to, że nie oblicza się całego wyrażenia logicznego, jeśli końcowy wynik może zostać określony już na podstawie obliczonej części wyrażenia. Przykładowo, przy ocenie decyzji „A i B" nie ma potrzeby obliczania wartości warunku B, jeżeli warunek A ma wartość FAŁSZ, ponieważ żadna wartość B nie zmieni rezultatu końcowego, zatem da się skrócić czas wykonywania kodu rezygnując z obliczania wartości warunku B. Zwarcie może wpływać na możliwość osiągnięcia pokrycia MC/DC, gdyż niektóre wymagane testy mogą okazać się niewykonalne. Zazwyczaj istnieje możliwość skonfigurowania kompilatora w celu wyłączenia zwarcia na potrzeby testów, jednak w przypadku aplikacji krytycznych ze względów bezpieczeństwa takie rozwiązanie może być niedopuszczalne, ponieważ często wymaga się, aby kod testowany był identyczny z kodem produkcyjnym.

2.5 Testowanie kombinacji warunków

W wyjątkowych sytuacjach może zaistnieć potrzeba sprawdzenia wszystkich możliwych kombinacji warunków atomowych występujących w decyzji. Taki zakres testów określa się mianem testowania kombinacji warunków. Przy założeniu istnienia N unikalnych, wzajemnie niezależnych warunków atomowych, pełne pokrycie kombinacji warunków dla danej decyzji można uzyskać poprzez sprawdzenie 2^n elementów pokrycia, czyli dla każdej możliwej kombinacji wartości warunków atomowych tej decyzji. Należy przy tym uwzględnić, że jeden przypadek testowy może obejmować sprawdzenie kilku takich kombinacji, dlatego osiągnięcie pełnego pokrycia kombinacji warunków bywa możliwe przy użyciu mniejszej liczby niż 2^n odrębnych przypadków testowych. Pokrycie oblicza się jako iloraz liczby przetestowanych kombinacji wartości warunków atomowych przez całkowitą liczbę takich kombinacji we wszystkich decyzjach w przedmiocie testów. Pokrycie to jest zazwyczaj wyrażane w procentach.

Obszar zastosowania. Technika ta jest wykorzystywana do testowania oprogramowania obarczonego wysokim ryzykiem oraz oprogramowania wbudowanego, które powinno działać w sposób niezawodny i bezawaryjny przez długi czas.

Ograniczenia/trudności. Ponieważ liczba wymaganych przypadków testowych wynika bezpośrednio z tablicy prawdy (ang. *truth table*) zawierającej wszystkie kombinacje wartości warunków atomowych, określenie tego poziomu pokrycia jest stosunkowo proste. Jednak liczba przypadków testowych niezbędnych do przeprowadzenia testowania kombinacji warunków może być bardzo duża, dlatego w większości sytuacji bardziej odpowiednim rozwiązaniem jest zastosowanie techniki testowania MC/DC. Jeżeli używany kompilator stosuje mechanizm zwarcia, rzeczywista liczba kombinacji wartości warunków atomowych wymagających przetestowania zazwyczaj ulega zmniejszeniu. Zależy to od kolejności oraz sposobu grupowania operacji logicznych wykonywanych na warunkach atomowych.

2.6 Testowanie ścieżek podstawowych

Ten podrozdział został usunięty z sylabusa w wersji 4.0.

2.7 Testowanie API

Interfejs programowania aplikacji (ang. API – *Application Programming Interface*) to dobrze określony interfejs umożliwiający komunikację z innym systemem, który udostępnia określone usługi, takie jak dostęp do zasobów zdalnych. Typowe przykłady takich usług obejmują: usługi sieciowe (ang. *web services*), korporacyjne magistrale usług (ang. *enterprise service buses* – ESB), bazy danych, komputery głównego szeregu (ang. *mainframes*) oraz webowe interfejsy użytkownika (ang. *Web UIs*).

Testowanie API jest bardziej opisem rodzaju testów niż konkretną techniką. W pewnym zakresie przypomina testowanie graficznego interfejsu użytkownika (ang. *graphical user interface* – GUI), gdyż skupia się na analizie wartości wejściowych i danych zwracanych przez interfejs.

Ważnym elementem testowania API jest często testowanie negatywne. Programiści korzystający z interfejsów API do dostępu do usług zewnętrznych mogą próbować używać ich niezgodnie z przeznaczeniem. Wymaga to wprowadzenia odpornych mechanizmów obsługi błędów, które zapobiegają niepoprawnemu działaniu systemu. Czasami konieczne jest także testowanie kombinatoryczne wielu interfejsów, ponieważ API są często wykorzystywane razem, a każda funkcja API może posiadać różne parametry, których wartości można łączyć w różny sposób.

Interfejsy API są często luźno powiązane (ang. *loosely coupled*), co zwiększa ryzyko utraty transakcji i zakłóceń czasowych. Dlatego kluczowe jest dokładne przetestowanie mechanizmów odtwarzania i ponawiania operacji. Organizacja udostępniająca interfejs API musi zapewnić wysoką dostępność wszystkich usług, co zwykle wymaga rygorystycznego testowania niezawodności i zapewnienia obsługi infrastruktury przez dostawcę API.

Obszar zastosowania. Testowanie API jest szczególnie istotne w systemach złożonych z wielu podsystemów (systemy systemów), w których coraz częściej stosuje się rozproszone przetwarzanie w celu przekierowania części zadań na różne procesory. Przykładowe zastosowania obejmują:

- wywołania systemu operacyjnego,
- architektury zorientowane na usługi (ang. *service-oriented architectures* – SOA),
- zdalne wywołania procedur (ang. *remote procedure calls* – RPC),
- usługi internetowe (ang. *Web services*).

W przypadku konteneryzacji oprogramowania program jest podzielony na kilka kontenerów, które komunikują się między sobą przy użyciu powyższych mechanizmów. Testowanie API powinno obejmować również takie interfejsy.

Ograniczenia/trudności. Bezpośrednie testowanie API zazwyczaj wymaga zastosowania specjalistycznych narzędzi przez technicznego analityka testów. Ponieważ interfejs API zwykle nie posiada powiązanego GUI, do skonfigurowania środowiska, serializacji danych, wywołania funkcji API i określenia wyników niezbędne są odpowiednie narzędzia testowe.

Pokrycie. Testowanie API opisuje typ testów, a nie konkretny poziom pokrycia. Test interfejsu API powinien obejmować co najmniej wywołania funkcji API z realistycznymi, poprawnymi wartościami parametrów wejściowych oraz z wartościami niepoprawnymi w celu weryfikacji obsługi wyjątków. W bardziej szczegółowych testach interfejsów API można przyjąć, że wszystkie wywoływalne obiekty powinny być przetestowane przynajmniej raz, lub że każda funkcja API została wywołana przynajmniej raz we wszystkich możliwych wariantach.

Architektura REST (ang. *Representational State Transfer*) stosowana w usługach sieciowych (RESTful Web services) umożliwia dostęp do zasobów sieciowych poprzez jednolity zestaw bezstanowych operacji. Dla interfejsów API usług sieciowych RESTful istnieje wiele kryteriów pokrycia testowego [Web-7], które można podzielić na kryteria pokrycia wejścia i kryteria pokrycia wyjścia. Kryteria wejścia obejmują między innymi: wykonanie wszystkich możliwych operacji API, użycie wszystkich możliwych parametrów funkcji API oraz pokrycie sekwencji wywołań funkcji. Kryteria wyjścia wymagają natomiast wygenerowania wszystkich możliwych kodów statusu operacji (poprawnych i błędnych) oraz wygenerowania odpowiedzi zawierających zasoby o wszystkich możliwych właściwościach lub typach właściwości.

Typy defektów. Podczas testowania API można wykryć różne rodzaje defektów, często związane z interfejsem. Należą do nich między innymi problemy z obsługą danych, zależnościami czasowymi, utratą lub duplikacją transakcji oraz błędy w obsłudze sytuacji wyjątkowych.

2.8 Wybór białoskrzynkowej techniki testowania

Wybór białoskrzynkowej techniki testowania zazwyczaj wynika z wymaganego poziomu pokrycia, który ma zostać osiągnięty dzięki zastosowaniu danej techniki. Przykładowo, wymóg uzyskania 100% pokrycia instrukcji zazwyczaj prowadzi do wykorzystania techniki testowania instrukcji. Zwykle proces przebiega tak, że najpierw stosuje się techniki czarnoskrzynkowe, a następnie mierzy uzyskane pokrycie; technika białoskrzynkowa jest wprowadzana tylko wtedy, gdy zakładany poziom pokrycia nie został osiągnięty. W pewnych sytuacjach techniki białoskrzynkowe mogą być stosowane w sposób mniej formalny, aby zidentyfikować obszary wymagające dodatkowego pokrycia – np. poprzez tworzenie dodatkowych testów w miejscach o szczególnie niskim poziomie pokrycia. W takich przypadkach testowanie instrukcji zwykle jest wystarczające jako nieformalna miara pokrycia.

Dobrym zwyczajem jest ustalanie wymaganego pokrycia białoskrzynkowego na poziomie 100%. Niższe wartości pokrycia często oznaczają, że nieprzetestowane fragmenty kodu są najtrudniejsze do pokrycia i jednocześnie najbardziej złożone oraz podatne na błędy. W efekcie osiągnięcie np. 80% pokrycia może oznaczać, że znacząca część defektów pozostaje niewykryta. Z tego powodu normy zazwyczaj wymagają 100% pokrycia. Ścisłe definicje poziomów pokrycia mogą czasami utrudniać lub uniemożliwiać osiągnięcie pełnego pokrycia, jednak norma ISO 29119-4 pozwala wykluczyć z obliczeń elementy nieosiągalne, dzięki czemu 100% pokrycia staje się realnym, osiągalnym celem.

Przy określaniu wymaganego pokrycia białoskrzynkowego dla przedmiotu testów wystarczy ustalić je dla jednego kryterium pokrycia – nie jest konieczne jednoczesne wymaganie np. 100% pokrycia instrukcji i 100% pokrycia MC/DC. Skupienie się na kryteriach na poziomie 100% pozwala tworzyć hierarchię kryteriów, w której jedne kryteria subsumują inne. Kryterium K1 subsumuje kryterium K2, jeśli dla każdego przedmiotu testów każdy zestaw przypadków

testowych osiągający 100% pokrycia K1 osiąga również 100% pokrycia K2. Przykładowo, pokrycie gałęzi subsumuje pokrycie instrukcji, ponieważ osiągnięcie 100% pokrycia gałęzi wymaga w szczególności wykonania każdej instrukcji, co oznacza 100% pokrycia instrukcji. W przypadku białoskrzynkowych technik opisanych w niniejszym sylabusie: pokrycie decyzji i pokrycie gałęzi subsumuje pokrycie instrukcji, pokrycie MC/DC subsumuje zarówno pokrycie decyzji, jak i gałęzi, a pokrycie kombinacji warunków subsumuje pokrycie MC/DC (jeśli przyjąć utożsamienie pokrycia gałęzi i decyzji na poziomie 100%, można uznać, że subsumują się nawzajem).

Często ustalając poziomy pokrycia białoskrzynkowego, definiuje się różne wartości dla różnych części systemu, ponieważ poszczególne obszary mają różny wpływ na ryzyko. Na przykład w systemach lotniczych podsystem rozrywki pokładowej może mieć niższy poziom ryzyka niż podsystem kontroli lotu. Testowanie interfejsów jest wspólne dla wszystkich typów systemów i zwykle wymaga się go dla wszystkich poziomów nienaruszalności bezpieczeństwa w systemach krytycznych ze względów bezpieczeństwa. Poziom pokrycia testów API zazwyczaj rośnie wraz ze wzrostem powiązanego ryzyka – np. publiczny interfejs API może wymagać bardziej rygorystycznego testowania.

Wybór odpowiedniej białoskrzynkowej techniki testowania opiera się zwykle na charakterystyce przedmiotu testów i związanych z nim postrzegalnych ryzyk. W przypadku systemów krytycznych ze względów bezpieczeństwa (gdzie awaria może zagrażać życiu, zdrowiu lub środowisku) stosuje się obowiązujące normy i precyzyjnie definiuje poziomy pokrycia białoskrzynkowego (patrz sekcja 2.8.2). Dla systemów niekrytycznych (niezwiązanych z bezpieczeństwem) wybór poziomów pokrycia jest bardziej elastyczny, lecz wciąż powinien uwzględniać postrzegane ryzyka, zgodnie z zasadami opisanymi w sekcji 2.8.1.

2.8.1 Systemy niekrytyczne (niezwiązane z bezpieczeństwem)

Przy wyborze techniki białoskrzynkowej dla systemów niekrytycznych zazwyczaj uwzględnia się następujące czynniki (kolejność nie ma znaczenia):

- Kontrakt – jeśli umowa wymaga osiągnięcia określonego poziomu pokrycia, jego nieosiągnięcie może stanowić naruszenie warunków umowy.
- Klient – jeśli klient oczekuje konkretnego poziomu pokrycia, np. w ramach planowania testów, nieosiągnięcie go może prowadzić do problemów lub konfliktów z klientem.
- Normy regulacyjne – w niektórych branżach, np. finansowej, dla systemów istotnych dla powodzenia projektu obowiązują regulacje określające wymagane kryteria pokrycia białoskrzynkowego (patrz sekcja 2.8.2. dotycząca norm regulacyjnych dla systemów krytycznych ze względów bezpieczeństwa).
- Strategia testów – jeśli obowiązująca w organizacji strategia testów określa wymagania w zakresie pokrycia białoskrzynkowego, ich nieprzestrzeganie może skutkować krytyką ze strony kierownictwa.
- Styl kodowania – jeśli kod jest pisany tak, by decyzje nie zawierały wielu warunków atomowych, stosowanie pokrycia typu MC/DC czy pokrycie kombinacji warunków mogłoby być nieefektywne i czasochłonne.
- Dane historyczne o defektach – jeśli wcześniejsze doświadczenia wskazują, że określony poziom pokrycia był skuteczny dla podobnych przedmiotów testów, ignorowanie tych danych może zwiększać ryzyko. Takie informacje mogą pochodzić z projektu, organizacji lub z danych przemysłowych.

- Umiejętności i doświadczenie – jeśli testerzy nie mają wystarczającego doświadczenia lub umiejętności w danej technice białoskrzynkowej, jej zastosowanie może być źle zrozumiane i wprowadzić dodatkowe ryzyko.
- Narzędzia – pokrycie białoskrzynkowe można w praktyce mierzyć tylko przy użyciu odpowiednich narzędzi. Brak dostępnych narzędzi może sprawić, że wymaganie pomiaru pokrycia stanie się ryzykowne.

W przypadku systemów niekrytycznych techniczny analityk testów dysponuje większą swobodą w rekomendowaniu poziomu pokrycia białoskrzynkowego niż w systemach krytycznych ze względów bezpieczeństwa. Wybór techniki i poziomu pokrycia jest zazwyczaj kompromisem pomiędzy postrzeganymi ryzykami a kosztami, zasobami i czasem potrzebnym do jego ograniczenia za pomocą testów białoskrzynkowych. W niektórych sytuacjach skuteczniejsze mogą być alternatywne metody testowania lub inne podejścia do wytwarzania oprogramowania.

2.8.2 Systemy krytyczne (związane z bezpieczeństwem)

Gdy testowane oprogramowanie stanowi część systemu krytycznego ze względów bezpieczeństwa, zazwyczaj obowiązuje odpowiednia norma regulacyjna określająca wymagane poziomy pokrycia. Normy te zwykle nakładają obowiązek przeprowadzenia analizy zagrożeń (ang. *hazard analysis*), a zidentyfikowane ryzyka służą do przypisania poszczególnym elementom systemu poziomów nienaruszalności bezpieczeństwa (ang. *Safety Integrity Levels* – SIL). Poziomy wymaganego pokrycia są definiowane oddzielnie dla każdego poziomu SIL.

Norma IEC 61508 („Bezpieczeństwo funkcjonalne elektrycznych, elektronicznych, programowalnych elektronicznych systemów związanych z bezpieczeństwem") [IEC 61508] jest ogólną normą stosowaną w tym celu. Teoretycznie może obejmować dowolne systemy krytyczne ze względów bezpieczeństwa, jednak niektóre branże opracowały własne warianty – np. ISO 26262 [ISO 26262] w przemyśle samochodowym – a inne stworzyły całkowicie odrębne normy, takie jak DO-178C [DO-178C] dla systemów lotniczych. Więcej informacji o ISO 26262 zawiera sylabus ISTQB® Tester Oprogramowania Automotive [CT_AuT_SYL].

Norma IEC 61508 definiuje cztery poziomy SIL. Każdy poziom określa względny stopień redukcji ryzyka zapewnianego przez funkcję bezpieczeństwa, powiązany z częstotliwością (ang. *frequency*) i krytycznością (ang. *severity*) postrzeganych zagrożeń. Gdy przedmiot testów wykonuje funkcję związaną z bezpieczeństwem, im wyższe ryzyko awarii, tym większa powinna być niezawodność testowanego oprogramowania. W praktyce oznacza to, że wyższe poziomy SIL wymagają bardziej rygorystycznego testowania i wyższej niezawodności systemu. Poniższa tabela pokazuje poziomy niezawodności związane z poszczególnymi poziomami nienaruszalności bezpieczeństwa SIL. Na przykład poziom SIL 4 dla operacji ciągłych odpowiada ekstremalnie wysokiej niezawodności, gdzie średni czas między awariami (ang. *Mean Time Between Failures*, MTBF) przekracza 10 000 lat (patrz tabela poniżej).

IEC 61508 SIL	Praca ciągła (prawdopodobieństwo awarii na godzinę)	Praca na żądanie (prawdopodobieństwo awarii dla żądania)
1	$\geq 10^{-6}$ do $<10^{-5}$	$\geq 10^{-2}$ do $<10^{-1}$
2	$\geq 10^{-7}$ do $<10^{-6}$	$\geq 10^{-3}$ do $<10^{-2}$
3	$\geq 10^{-8}$ do $<10^{-7}$	$\geq 10^{-4}$ do $<10^{-3}$
4	$\geq 10^{-9}$ do $<10^{-8}$	$\geq 10^{-5}$ do $<10^{-4}$

Rekomendacje dotyczące poziomów pokrycia białoskrzynkowego przypisanych do poszczególnych poziomów SIL przedstawiono w poniższej tabeli. Określenie „wysoce rekomendowane" w praktyce oznacza, że osiągnięcie wskazanego poziomu pokrycia jest obowiązkowe. Natomiast określenie „rekomendowane" jest często interpretowane przez praktyków jako wskazanie opcjonalne – osiągnięcie tego poziomu pokrycia nie jest wymuszone, pod warunkiem, że istnieje uzasadnienie dla braku konieczności jego nieosiągnięcia. Na przykład, dla przedmiotu testów przypisanego do poziomu SIL 3 zwykle dąży się do osiągnięcia 100% pokrycia gałęzi, co automatycznie zapewnia również 100% pokrycia instrukcji z powodu relacji subsumpcji.

Poziom SIL wg IEC 61508	100% pokrycia instrukcji	100% pokrycia gałęzi	100% pokrycia MC/DC
1	rekomendowane	rekomendowane	rekomendowane
2	wysoce rekomendowane	rekomendowane	rekomendowane
3	wysoce rekomendowane	wysoce rekomendowane	rekomendowane
4	wysoce rekomendowane	wysoce rekomendowane	wysoce rekomendowane

Należy podkreślić, że poziomy SIL i odpowiadające im wymagania dotyczące pokrycia określone w normie IEC 61508 różnią się od tych zdefiniowanych w normie ISO 26262, które z kolei nie są identyczne z normą DO-178C.

3. Analiza statyczna i analiza dynamiczna (180 min.)

Słowa kluczowe

anomalia, analiza dynamiczna, analiza przepływu danych, analiza przepływu sterowania, analiza statyczna, dziki wskaźnik, para definicja-użycie, wyciek pamięci, złożoność cyklomatyczna

Cele nauczania dla rozdziału 3

3.2 Analiza statyczna

TTA-3.2.1 (K3) Obliczyć złożoność cyklomatyczną i zastosować analizę przepływu sterowania w celu wykrycia ewentualnych anomalii w kodzie związanych z tym przepływem.

TTA-3.2.2 (K3) Wykorzystać analizę przepływu danych w celu wykrycia ewentualnych anomalii w kodzie związanych z tym przepływem.

TTA-3.2.3 (K3) Zaproponować sposoby zwiększenia utrzymywalności kodu za pomocą analizy statycznej.

Uwaga: cel nauczania TTA-3.2.4. został usunięty z wersji 4.0 niniejszego sylabusa

3.3 Analiza dynamiczna

TTA-3.3.1 (K3) Zastosować analizę dynamiczną, aby osiągnąć określony cel.

3.1 Wstęp

Analiza statyczna (patrz podrozdział 3.2) to forma testowania realizowana bez uruchamiania oprogramowania. Ocena jakości oprogramowania odbywa się poprzez analizę jego konstrukcji, struktury, zawartości lub dokumentacji, wykonywaną przez człowieka lub przy użyciu narzędzi. Taki „statyczny" widok oprogramowania pozwala na dogłębną analizę bez konieczności przygotowywania danych testowych ani spełniania warunków wstępnych wymaganych do wykonania testów.

Oprócz analizy statycznej, do technik testowania statycznego zalicza się również różne formy przeglądów. Te, które mają znaczenie z perspektywy technicznego analityka testów, zostały omówione w rozdziale 5.

Analiza dynamiczna (patrz podrozdział 3.3) polega natomiast na rzeczywistym uruchamianiu kodu i służy do wykrywania defektów, które ujawniają się dopiero podczas jego wykonywania, takich jak wycieki pamięci. Podobnie jak w analizie statycznej, może ona być wspierana przez narzędzia lub opierać się na obserwacji systemu przez człowieka, np. w celu wychwycenia symptomów takich jak gwałtowny wzrost zużycia pamięci.

3.2 Analiza statyczna

Celem analizy statycznej jest wykrycie rzeczywistych i potencjalnych defektów oraz zwiększenie utrzymywalności kodu i architektury.

3.2.1 Analiza przepływu sterowania

Analiza przepływu sterowania polega na badaniu sekwencji kroków wykonywanych przez program z wykorzystaniem grafu przepływu sterowania, najczęściej przy wsparciu dedykowanych narzędzi. Metoda ta umożliwia identyfikację różnych typów anomalii, takich jak błędnie zaprojektowane pętle (np. posiadające wiele punktów wejścia lub pozbawione warunku zakończenia), niejednoznaczne cele wywołań funkcji w niektórych językach programowania, niepoprawna kolejność operacji, fragmenty kodu, do których nie ma dostępu w trakcie wykonania, czy funkcje, które nigdy nie są wywoływane.

Analiza przepływu sterowania może być również wykorzystana do obliczania złożoności cyklomatycznej. Metryka ta jest dodatnią liczbą całkowitą określającą liczbę ścieżek liniowo niezależnych w grafie przepływu sterowania programu.

Złożoność cyklomatyczna jest powszechnie stosowana jako miara stopnia złożoności modułu. Zgodnie z koncepcją zaproponowaną przez T. McCabe'a [McCabe76] im większa złożoność systemu, tym wyższe prawdopodobieństwo występowania defektów oraz trudniejsze utrzymanie kodu. Liczne badania potwierdzają zależność między poziomem złożoności a liczbą defektów. Dlatego moduły o wysokiej złożoności powinny zostać poddane przeglądowi, na przykład w celu refaktoryzacji lub podziału na mniejsze, prostsze elementy.

3.2.2 Analiza przepływu danych

Analiza przepływu danych obejmuje zbiór technik skupiających się na zbieraniu i analizie informacji dotyczących zmiennych występujących w systemie. Na podstawie grafu przepływu

danych śledzony jest pełny cykl życia każdej zmiennej, czyli miejsca jej zadeklarowania, zdefiniowania, użycia oraz usuwania. Pozwala to na wykrycie potencjalnych anomalii, które mogą pojawić się, gdy te operacje są wykonywane w niewłaściwej kolejności [Beizer 90].

Jedna z powszechnie stosowanych technik polega na klasyfikowaniu każdej operacji na zmiennej jako jednego z trzech podstawowych działań:

- definiowanie, deklarowanie lub inicjalizowanie zmiennej (np. x := 3),
- użycie (odczyt) zmiennej w wyrażeniu bądź w predykacie (np. if x > 0),
- usunięcie zmiennej, jej zniszczenie lub utratę dostępności w danym zakresie (np. uchwyt do pliku po zamknięciu pliku lub iterator po zakończeniu pętli).

Pewne sekwencje tych działań mogą sygnalizować istnienie potencjalnych anomalii, takich jak:

- ponowna definicja lub usunięcie zmiennej bez jej wcześniejszego użycia,
- definicja, po której nie następuje usunięcie zmiennej (co w przypadku alokacji dynamicznej zmiennych może prowadzić do wycieków pamięci),
- użycie bądź usunięcie zmiennej przed jej zdefiniowaniem,
- użycie lub usunięcie zmiennej po jej wcześniejszym usunięciu.

W zależności od używanego języka programowania część takich anomalii może zostać wykryta przez kompilator. W wielu przypadkach konieczne jest jednak przeprowadzenie dodatkowej analizy statycznej z wykorzystaniem specjalistycznych narzędzi w celu wykrycia przepływu sterowania anomalii. Przykładowo, ponowna definicja zmiennej bez jej wcześniejszego użycia jest w większości języków dozwolona i bywa stosowana celowo, lecz narzędzia analizy statycznej często oznaczają takie sytuacje jako potencjalnie ryzykowne i wymagające weryfikacji.

Analizowanie sekwencji operacji na zmiennych na podstawie ścieżek w grafie przepływu danych może prowadzić do wskazania anomalii, które w rzeczywistości nigdy nie wystąpią. Wynika to z faktu, że narzędzia do analizy statycznej nie zawsze potrafią ocenić, czy dana ścieżka wykonania jest faktycznie osiągalna, ponieważ zależy to od wartości zmiennych w trakcie działania programu. Dodatkowym wyzwaniem są przypadki, w których analizowane dane są strukturami dynamicznymi, a także współbieżność, czyli współdzielenie zmiennych przez równolegle wykonywane wątki – wtedy kolejność operacji może być szczególnie trudna do przewidzenia.

W przeciwieństwie do analizy przepływu danych, która jest metodą testowania statycznego, testowanie przepływu danych jest techniką testowania dynamicznego. Polega ono na projektowaniu przypadków testowych w taki sposób, aby pokryć pary definicja-użycie zmiennych w kodzie. Technika ta opiera się na tych samych założeniach co analiza przepływu danych, ponieważ pary definicja-użycie odpowiadają ścieżkom w grafie przepływu sterowania prowadzącym od miejsca definicji zmiennej do jej kolejnego użycia.

3.2.3 Analiza statyczna jako sposób poprawy utrzymywalności

Analizę statyczną można wykorzystywać na wiele sposobów w celu poprawy utrzymywalności kodu, architektury systemu oraz stron internetowych.

Kod napisany w sposób nieczytelny, pozbawiony komentarzy i odpowiedniej struktury jest znacznie trudniejszy w dalszym utrzymaniu. Wyszukiwanie i analizowanie defektów w takim

kodzie wymaga większego wysiłku ze strony programistów, a wprowadzanie poprawek lub nowych funkcji zwiększa ryzyko popełniania kolejnych błędów i wprowadzania defektów.

Zastosowanie analizy statycznej umożliwia sprawdzenie zgodności kodu z obowiązującymi standardami i wytycznymi programistycznymi. Wykrycie odstępstw od tych zasad pozwala ulepszyć jakość kodu i zwiększyć jego podatność na dalsze modyfikacje. Standardy i wytyczne opisują wymagane praktyki projektowania i kodowania takie jak konwencje nazewnictwa, zasady komentowania, sposób formatowania kodu oraz wymagania dotyczące modularyzacji. Warto podkreślić, że narzędzia do analizy statycznej zazwyczaj generują ostrzeżenia, a nie jednoznacznie wykrywają defekty. Takie ostrzeżenia (np. o nadmiernej złożoności) mogą pojawić się nawet wtedy, gdy kod jest poprawny składniowo.

Tworzenie zmodularyzowanego kodu zwykle sprzyja jego lepszej utrzymywalności. Narzędzia do analizy statycznej wspierają to podejście między innymi poprzez:

- identyfikowanie powielonych fragmentów kodu, które mogą zostać zrefaktoryzowane i wydzielone do osobnych funkcji (choć w systemach czasu rzeczywistego należy brać pod uwagę koszt wywołań funkcji),
- obliczanie metryk pomagających w podziale kodu na moduły, takich jak sprzężenie (ang. *coupling*) i spójność (ang. *cohesion*); system łatwy w utrzymaniu cechuje się zwykle niskim sprzężeniem między modułami oraz wysoką spójnością każdego z modułów (stopień, w jakim dany moduł jest samowystarczalny i skoncentrowany na jednym zadaniu),
- wskazywanie w kodzie obiektowym miejsc, w których zakres widoczności klas bazowych dla klas pochodnych jest niewłaściwie dobrany,
- identyfikowanie fragmentów kodu lub elementów architektury o nadmiernej złożoności strukturalnej.

Narzędzia do analizy statycznej znajdują również zastosowanie w utrzymaniu serwisów internetowych. W takim przypadku analizuje się m.in. strukturę drzewa strony, sprawdzając jej zrównoważenie oraz wykrywając nieprawidłowości, które mogą:

- utrudniać proces testowania,
- zwiększać nakład pracy związany z utrzymaniem serwisu.

Oprócz oceny utrzymywalności, narzędzia do analizy statycznej mogą być wykorzystywane do badania kodu odpowiedzialnego za działanie stron internetowych pod kątem potencjalnych podatności zabezpieczeń. Dotyczy to między innymi podatności na wstrzykiwanie kodu (ang. *code injection*), zagrożeń związanych z obsługą ciasteczek (ang. *cookie security*), ataków typu *cross-site scripting*, manipulowania zasobami (ang. *resource tampering*) czy ataków SQL *injection*. Szczegółowe informacje na ten temat można znaleźć w podrozdziale 4.3 oraz w sylabusie Certyfikowany Tester – Tester Zabezpieczeń [CT_SEC_SYL].

3.3 Analiza dynamiczna

3.3.1 Przegląd

Analiza dynamiczna jest wykorzystywana do identyfikowania awarii, które ujawniają się dopiero podczas wykonywania programu. Przykładem są wycieki pamięci: choć analiza statyczna może

czasem wskazać potencjalne zagrożenie (np. fragmenty kodu alokujące pamięć bez jej zwalniania), to właśnie analiza dynamiczna pozwala je jednoznacznie i łatwo zaobserwować w trakcie działania systemu.

Awarie trudne do natychmiastowego odtworzenia, występujące nieregularnie, znacząco zwiększają nakład pracy związany z testowaniem oraz mogą opóźniać wdrożenie oprogramowania lub utrudniać jego eksploatację. Do ich przyczyn zalicza się m.in. wycieki pamięci i zasobów, błędne użycie wskaźników czy inne problemy, takie jak uszkodzenia stosu systemowego [Kaner02]. Tego typu awarie często prowadzą do stopniowego pogarszania się wydajności, a w skrajnych przypadkach do całkowitej awarii systemu. Dlatego w strategii testów należy uwzględnić ryzyka związane z takimi defektami i tam, gdzie jest to uzasadnione, przeprowadzić analizę dynamiczną, zazwyczaj wspieraną przez specjalistyczne narzędzia, aby ograniczyć ich występowanie. Ponieważ tego rodzaju awarie należą do najtrudniejszych i najdroższych w usuwaniu, rekomenduje się rozpoczynanie analizy dynamicznej już na wczesnym etapie projektu.

Analiza dynamiczna może być stosowana w celu:

- zapobiegania awariom poprzez wykrywanie wycieków pamięci (patrz sekcja 3.3.2) i tzw. dzikich wskaźników (patrz sekcja 3.3.3),
- badania awarii, które trudno jednoznacznie odtworzyć,
- oceny działania sieci,
- poprawy wydajności systemu dzięki użyciu profilerów kodu (ang. *code profilers*), które dostarczają informacji o zachowaniu aplikacji w czasie jej wykonywania i umożliwiają wprowadzanie usprawnień.

Metody analizy dynamicznej można wykorzystywać na dowolnym poziomie testów. Wymagają one jednak odpowiednich kompetencji technicznych i systemowych, pozwalających:

- zdefiniować cele analizy dynamicznej,
- określić właściwy moment jej rozpoczęcia i zakończenia,
- prawidłowo interpretować uzyskane wyniki.

Narzędzia do analizy dynamicznej mogą być używane także przez technicznych analityków testów o ograniczonym doświadczeniu technicznym, ponieważ zazwyczaj generują one szczegółowe logi, które mogą być następnie analizowane przez specjalistów posiadających odpowiednią wiedzę techniczną i analityczną.

3.3.2 Wykrywanie wycieków pamięci

Wyciekiem pamięci nazywa się sytuację, w której program rezerwuje obszary pamięci operacyjnej (RAM), lecz nie zwalnia ich po zakończeniu wykorzystania. Taka pamięć staje się niedostępna do ponownego użycia. Jeśli zjawisko to występuje często lub system dysponuje ograniczoną ilością pamięci, aplikacja może w końcu wyczerpać dostępne zasoby. W przeszłości pełna odpowiedzialność za prawidłowe zarządzanie pamięcią spoczywała na programiście – każdy dynamicznie przydzielony obszar musiał zostać jawnie zwolniony, aby zapobiec wyciekom. Współcześnie wiele środowisk programistycznych oferuje automatyczne lub półautomatyczne mechanizmy zarządzania pamięcią, takie jak odśmiecacze (ang. *garbage collectors*), które odzyskują nieużywaną pamięć bez bezpośredniego udziału programisty. W takich warunkach

wykrywanie wycieków pamięci bywa jednak znacznie trudniejsze, ponieważ zwalnianie pamięci odbywa się w sposób pośredni i automatyczny.

Problemy związane z wyciekami pamięci zazwyczaj ujawniają się dopiero po dłuższym czasie, gdy utracona zostanie znaczna ilość pamięci. Bezpośrednio po instalacji aplikacji lub restarcie systemu wycieki są zwykle niewidoczne. Podczas testów, gdzie alokacja pamięci następuje często i intensywnie, ich identyfikacja również może być utrudniona. Z tego względu negatywne skutki wycieków pamięci są nierzadko zauważane dopiero w środowisku produkcyjnym, w trakcie rzeczywistej eksploatacji systemu.

Najczęstszym symptomem wycieku pamięci jest stopniowe pogarszanie się czasu odpowiedzi systemu, co w skrajnych przypadkach prowadzi do jego awarii. Choć problem można czasami tymczasowo rozwiązać poprzez ponowne uruchomienie systemu, takie działanie bywa uciążliwe, a w pewnych systemach wręcz niemożliwe.

Wiele narzędzi do analizy dynamicznej umożliwia wskazanie fragmentów kodu odpowiedzialnych za wycieki pamięci, co pozwala na skuteczne usunięcie wycieków. Prostsze narzędzia monitorujące zużycie pamięci mogą natomiast sygnalizować stopniowy spadek dostępnych zasobów, choć dokładne ustalenie przyczyny problemu wymaga wówczas przeprowadzenia bardziej szczegółowej analizy.

3.3.3 Wykrywanie dzikich wskaźników

Dzikie wskaźniki to wskaźniki, które utraciły swoją poprawność i nie powinny być dalej wykorzystywane. Może to być na przykład wskaźnik, który nie wskazuje już na właściwy obiekt lub funkcję, albo taki, który odwołuje się do niezamierzonego obszaru pamięci, np. poza dozwolonym zakresem tablicy. Wykorzystanie takich wskaźników może mieć różne konsekwencje:

- program może nadal działać poprawnie, jeśli wskaźnik odnosi się do obszaru pamięci, który aktualnie nie jest używany lub zawiera wartości niepowodujące awarii,
- może dojść do awarii aplikacji, gdy wskaźnik naruszy pamięć istotną dla prawidłowego działania programu, na przykład obszary zarezerwowane dla systemu operacyjnego,
- program może funkcjonować w ograniczonym zakresie (może działać niepoprawnie), lecz zgłaszać błędy z powodu braku dostępu do wymaganych danych lub obiektów,
- może nastąpić uszkodzenie danych w pamięci, co skutkuje wykorzystaniem nieprawidłowych wartości, a w skrajnych przypadkach stanowi zagrożenie dla bezpieczeństwa.

Należy mieć na uwadze, że każda zmiana wpływająca na sposób zarządzania pamięcią, taka jak ponowna kompilacja po modyfikacjach kodu, może ujawnić dowolny z powyższych problemów. Jest to szczególnie niebezpieczne w sytuacjach, gdy aplikacja początkowo działa poprawnie mimo obecności dzikich wskaźników, a dopiero po pewnym czasie lub po wprowadzeniu zmian niespodziewanie ulega awarii, często już w środowisku produkcyjnym. Narzędzia diagnostyczne umożliwiają wykrywanie takich wskaźników niezależnie od tego, czy aktualnie powodują one zauważalne problemy. Dodatkowo wiele systemów operacyjnych oferuje mechanizmy monitorujące i kontrolujące dostęp do pamięci w trakcie działania programu, np. zgłaszając wyjątek w przypadku próby odwołania się do niedozwolonego obszaru pamięci.

3.3.4 Analiza wydajności

Analiza dynamiczna jest użyteczna nie tylko do identyfikowania awarii i ustalania przyczyn związanych z defektami. Narzędzia stosowane w dynamicznej analizie wydajności (profilery kodu) umożliwiają wykrywanie wąskich gardeł wydajnościowych oraz generowanie różnorodnych metryk, które mogą być wykorzystane przez programistów do optymalizacji działania systemu. Przykładowo pozwalają one ustalić, jak często poszczególne funkcje są wykonywane w czasie pracy programu, co umożliwia skoncentrowanie działań optymalizacyjnych na elementach wywoływanych najczęściej. W praktyce często potwierdza się tu zasada Pareto, zgodnie z którą program zużywa większość czasu wykonania (około 80%) przebywając w niewielkiej części swoich modułów (około 20%) [Andrist20].

Dynamiczna analiza wydajności jest zazwyczaj realizowana na etapie testów systemowych, jednak może być również prowadzona wcześniej, podczas testowania pojedynczych podsystemów, z użyciem jarzm testowych. Dalsze informacje w tym zakresie są zawarte w sylabusie Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

4. Charakterystyki jakościowe w testach technicznych (345 min.)

Słowa kluczowe

adaptowalność, analizowalność, charakterystyka jakościowa, dojrzałość, instalowalność, integralność, kompatybilność, model wzrostu niezawodności, modułowość, modyfikowalność, niezaprzeczalność, niezawodność, odtwarzalność, osiągalność, pojemność, poufność, profil operacyjny, przenaszalność[3], reużywalność, rozliczalność, testowalność, tolerowanie usterek, utrzymywalność, wiarygodność, współistnienie, wydajność, zabezpieczenia, zachowanie w czasie, zastępowalność, zużycie zasobów

Cele nauczania dla rozdziału 4

4.2 Ogólne planowanie

TTA-4.2.1 (K4) Dla konkretnego scenariusza przeanalizować wymagania niefunkcjonalne i napisać odpowiednie fragmenty planu testów.

TTA-4.2.2 (K3) Określić dla danego ryzyka produktowego konkretne typy testów niefunkcjonalnych, które są najbardziej odpowiednie.

TTA-4.2.3 (K2) Rozpoznać i omówić etapy w cyklu życia aplikacji, w których należy w typowych sytuacjach przeprowadzić testowanie niefunkcjonalne.

TTA-4.2.4 (K3) Dla konkretnego scenariusza zdefiniować typy defektów, których wykrycia należy się spodziewać w przypadku użycia różnych typów testów niefunkcjonalnych.

4.3 Testowanie zabezpieczeń

TTA-4.3.1 (K2) Omówić przyczyny uwzględnienia testowania zabezpieczeń w podejściu do testowania.

TTA-4.3.2 (K2) Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów zabezpieczeń.

4.4 Testowanie niezawodności

TTA-4.4.1 (K2) Omówić przyczyny uwzględnienia testowania niezawodności w podejściu do testowania.

TTA-4.4.2 (K2) Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów niezawodności.

[3] W normie ISO/IEC 25010:2023 termin *flexibility* (elastyczność) zastąpił termin *portability* (przenaszalność) obecny w starszej wersji normy z 2011 roku, zatem elastyczność i przenaszalność należy traktować jak synonimy.

4.5 Testowanie wydajnościowe

TTA-4.5.1 (K2) Omówić przyczyny uwzględnienia testowania wydajnościowego w podejściu do testowania.

TTA-4.5.2 (K2) Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów wydajnościowych.

4.6 Testowanie utrzymywalności

TTA-4.6.1 (K2) Omówić przyczyny uwzględnienia testowania utrzymywalności w podejściu do testowania.

4.7 Testowanie przenaszalności

TTA-4.7.1 (K2) Omówić przyczyny uwzględnienia testowania przenaszalności w podejściu do testowania.

4.8 Testowanie kompatybilności

TTA-4.8.1 (K2) Omówić przyczyny uwzględnienia testowania współistnienia w podejściu do testowania.

4.1 Wstęp

Zasadniczo techniczny analityk testów skupia się przede wszystkim na sposobie działania produktu, a nie na jego funkcjach, czyli na tym, jak system pracuje, a nie co realizuje. Tego typu testy mogą być wykonywane na każdym poziomie testów. Przykładowo, podczas testowania modułowego w systemach czasu rzeczywistego oraz systemach wbudowanych kluczowe znaczenie mają testy porównawcze wydajności oraz testy zużycia zasobów. Z kolei w operacyjnym testowaniu akceptacyjnym prowadzonym w środowisku produkcyjnym oraz w testowaniu systemowym istotne jest sprawdzanie aspektów niezawodnościowych, takich jak możliwość odtworzenia działania systemu po awarii. Testy realizowane na tym poziomie odnoszą się do konkretnej konfiguracji systemu, rozumianej jako połączenie sprzętu i oprogramowania. Dany testowany system może obejmować różne elementy, takie jak serwery, stacje robocze, bazy danych, sieci oraz inne zasoby. Niezależnie od poziomu testów, działania te powinny być planowane i wykonywane z uwzględnieniem priorytetów wynikających z ryzyka oraz dostępnych zasobów.

Zarówno testowanie dynamiczne, jak i testowanie statyczne, w tym przeglądy (omówione w rozdziałach 2, 3 i 5), mogą być stosowane do weryfikacji niefunkcjonalnych charakterystyk jakościowych opisanych w niniejszym rozdziale.

Opis charakterystyk jakościowych produktu oraz ich podcharakterystyk opiera się na normie ISO 25010 [ISO 25010] i został zaprezentowany w poniższej tabeli wraz z informacją, które z nich są omawiane w sylabusie Certyfikowany Tester Poziom Zaawansowany Analityk Testów, a które w niniejszym sylabusie.

Charakterystyka	Podcharakterystyki	analityk testów	techniczny analityk testów
Funkcjonalność (przydatność funkcjonalna)	Poprawność funkcjonalna, adekwatność funkcjonalna, kompletność funkcjonalna	X	
Niezawodność	Dojrzałość, tolerowanie usterek, odtwarzalność, osiągalność		X
Użyteczność	Rozpoznawalność, łatwość nauki, łatwość obsługi, estetyka interfejsu użytkownika, błędoodporność, dostępność (ułatwienia dostępu)	X	
Wydajność	Zachowanie w czasie, zużycie zasobów, pojemność		X
Utrzymywalność	Analizowalność, modyfikowalność, testowalność, modułowość, reużywalność		X
Przenaszalność	Adaptowalność, instalowalność, zastępowalność	X	X
Zabezpieczenia	Poufność, integralność, niezaprzeczalność, rozliczalność, wiarygodność		X
Zgodność	Współistnienie		X
	Współdziałanie	X	

Załącznik A zawiera tabelę, w której porównano charakterystyki jakościowe opisane w (obecnie wycofanej) normie ISO 9126-1, używanej w wersji 2012 niniejszego sylabusa, z tymi z nowszej normy ISO 25010.

W celu opracowania odpowiedniego podejścia do testów należy zidentyfikować typowe ryzyka dla wszystkich charakterystyk jakościowych oraz ich podcharakterystyk omówionych w tej sekcji.

Testowanie charakterystyk jakościowych wymaga szczególnie starannego doboru momentu w cyklu życia, dostępnych narzędzi, obowiązujących norm, a także odpowiednich wersji oprogramowania i dokumentacji oraz wiedzy technicznej. Jeśli nie zaplanuje się sposobu postępowania z każdą charakterystyką oraz jej specyficznymi wymaganiami testowymi, tester może nie mieć w harmonogramie przewidzianego odpowiedniego czasu na planowanie, przygotowanie i przeprowadzenie testów.

Niektóre testy, np. wydajnościowe, wymagają szczegółowego planowania, specjalistycznego sprzętu i narzędzi, zaawansowanych umiejętności testerskich oraz często znacznej ilości czasu. Testowanie charakterystyk jakościowych i podcharakterystyk musi być uwzględnione w ogólnym harmonogramie testów i muszą być zapewnione odpowiednie zasoby do jego realizacji.

Kierownik testów odpowiada za zbieranie informacji o metrykach i raportowanie wyników podsumowujących dla wszystkich charakterystyk jakościowych i podcharakterystyk, natomiast analityk testów lub techniczny analityk testów gromadzi dane dotyczące poszczególnych metryk, zgodnie z przypisaniem w powyższej tabeli.

Pomiary charakterystyk jakościowych przeprowadzane w testach przedprodukcyjnych przez technicznego analityka testów mogą stanowić podstawę do ustalenia poziomu usług (ang. *Service Level Agreement* – SLA) między dostawcą systemu a interesariuszami, np. klientami lub operatorami. W niektórych przypadkach testy są kontynuowane także po wdrożeniu produkcyjnym, zwykle przez odrębny zespół lub organizację, co jest szczególnie istotne w testach wydajności i niezawodności, ponieważ wyniki w środowisku produkcyjnym mogą różnić się od wyników uzyskanych w środowisku testowym.

4.2 Ogólne planowanie

Niezaplanowanie testowania niefunkcjonalnego niesie ze sobą poważne ryzyko niepowodzenia projektu. Kierownik testów może poprosić technicznego analityka testów o zidentyfikowanie głównych ryzyk dla określonych charakterystyk jakościowych (patrz tabela w podrozdziale 4.1) i rozwiązanie ewentualnych problemów dotyczących planowania, związanych z zaproponowanymi testami. Zagadnienia te można wykorzystać podczas opracowywania głównego planu testów.

Podczas wykonywania opisanych zadań należy uwzględnić:

- wymagania interesariuszy,
- wymagania dotyczące środowiska testowego,
- zakup wymaganych narzędzi i szkolenia,
- kwestie organizacyjne,
- zagadnienia dotyczące bezpieczeństwa danych.

4.2.1 Wymagania interesariuszy

Wymagania niefunkcjonalne często bywają niedokładnie określone, a niekiedy w ogóle nie są zdefiniowane. W fazie planowania techniczny analityk testów powinien pozyskać od właściwych interesariuszy informacje na temat oczekiwanych poziomów jakości związanych z charakterystykami niefunkcjonalnymi, a następnie ocenić ryzyka związane z tymi wymaganiami.

Często przyjmuje się, że jeśli klient jest zadowolony z obecnej wersji systemu, będzie także zadowolony z nowej wersji, pod warunkiem utrzymania dotychczasowych poziomów jakości. W związku z tym istniejąca wersja systemu może służyć jako punkt odniesienia (ang. *benchmark*). Takie podejście jest szczególnie użyteczne przy niefunkcjonalnych charakterystykach jakościowych takich jak wydajność, gdzie interesariusze mogą mieć trudności z precyzyjnym określeniem wymagań.

Podczas zbierania wymagań niefunkcjonalnych warto uwzględnić różne perspektywy, pozyskując je od przedstawicieli różnych grup interesariuszy takich jak klienci i użytkownicy, właściciele produktu oraz personel operacyjny i serwisowy. Pominięcie istotnych interesariuszy zwiększa ryzyko pominięcia części wymagań. Szczegółowe informacje o rejestrowaniu wymagań znajdują się w sylabusie Certyfikowany Tester Poziom Zaawansowany Kierownik Testów [CT_TM_SYL].

W metodykach zwinnych wymagania niefunkcjonalne mogą być dokumentowane jako historyjki użytkownika lub dołączane do funkcjonalności opisanych w przypadkach użycia w formie ograniczeń niefunkcjonalnych.

4.2.2 Wymagania dotyczące środowiska testowego

Wiele testów technicznych takich jak testy zabezpieczeń czy testy wydajności wymaga przygotowania środowiska testowego, które w jak największym stopniu odwzorowuje środowisko produkcyjne, aby uzyskać wiarygodne wyniki pomiarów. W zależności od rozmiaru i złożoności testowanego systemu, takie wymagania mogą znacząco wpłynąć na planowanie i budżet testów. Ze względu na wysokie koszty utrzymania środowisk testowych, warto rozważyć następujące podejścia:

- korzystanie z rzeczywistego środowiska produkcyjnego,
- użycie ograniczonej wersji systemu, przy zapewnieniu, że wyniki testów nadal będą w odpowiednim stopniu odzwierciedlać działanie systemu produkcyjnego,
- wykorzystanie zasobów chmurowych zamiast ich bezpośredniego zakupu,
- stosowanie środowisk wirtualnych.

W harmonogramie testów należy starannie zaplanować moment przeprowadzania takich testów, ponieważ często możliwe jest ich wykonanie tylko w określonych terminach, np. w okresach mniejszego obciążenia systemu.

4.2.3 Zakup wymaganych narzędzi i szkolenia

Narzędzia stanowią integralną część środowiska testowego. Szczególnie istotne są komercyjne narzędzia lub symulatory przy testach wydajnościowych oraz niektórych testach zabezpieczeń. Do zadań technicznego analityka testów należy oszacowanie kosztów i czasu potrzebnego na zakup, opanowanie i wdrożenie takich narzędzi. W przypadku korzystania ze specjalistycznych narzędzi należy uwzględnić czas potrzebny na naukę nowych produktów lub koszty zatrudnienia ekspertów zewnętrznych.

Tworzenie złożonego symulatora może stanowić osobny projekt programistyczny, co trzeba uwzględnić w planowaniu. Harmonogram i alokacja zasobów powinny przewidywać testowanie i dokumentowanie opracowanego narzędzia. W razie zmian w symulowanym systemie konieczne będzie zarezerwowanie czasu i budżetu na aktualizację oraz ponowne przetestowanie symulatora. Dla systemów krytycznych ze względów bezpieczeństwa plan prac nad symulatorami

powinien także uwzględniać testy akceptacyjne i ewentualną certyfikację narzędzia przez niezależną jednostkę.

4.2.4 Kwestie organizacyjne

Testy techniczne często obejmują pomiar zachowania wielu modułów całego systemu takich jak serwery, bazy danych czy sieci. Gdy moduły te są rozproszone w różnych lokalizacjach lub zarządzane przez różne organizacje, planowanie i koordynacja testów może wymagać znacznego nakładu pracy. Na przykład niektóre moduły mogą być dostępne do testów systemowych tylko w określonych godzinach, a wsparcie ze strony innych organizacji może być ograniczone do wybranych dni. Brak wcześniejszego potwierdzenia dostępności modułów i personelu zewnętrznego może prowadzić do poważnych zakłóceń w harmonogramie zaplanowanych testów.

4.2.5 Zagadnienia dotyczące zabezpieczeń i ochrony danych

Podczas planowania testów należy uwzględnić wbudowane mechanizmy zabezpieczeń systemu, aby wszystkie czynności testowe mogły zostać prawidłowo przeprowadzone. Na przykład stosowanie szyfrowania danych może utrudniać generowanie danych testowych oraz weryfikację wyników.

Dodatkowo obowiązujące przepisy i regulacje dotyczące ochrony danych mogą uniemożliwiać użycie rzeczywistych danych produkcyjnych, takich jak dane osobowe czy informacje o kartach kredytowych. W związku z tym przygotowanie danych testowych wymaga często ich anonimizacji, co jest zadaniem złożonym i powinno być uwzględnione w planie implementacji testów.

4.3 Testowanie zabezpieczeń

4.3.1 Powody przeprowadzenia testów zabezpieczeń

Testowanie zabezpieczeń polega na sprawdzaniu podatności systemu na zagrożenia poprzez próby przełamania mechanizmów ochronnych wynikających z przyjętej polityki bezpieczeństwa. W trakcie takich testów należy brać pod uwagę różne potencjalne zagrożenia, w tym:

- Nieuprawnione kopiowanie aplikacji lub danych.
- Brak kontroli dostępu, czyli możliwość wykonywania operacji bez odpowiednich uprawnień. Weryfikacja ról użytkowników, zasad dostępu i uprawnień stanowi kluczowy element testów i powinna opierać się na dokumentacji systemu.
- Niepożądane efekty uboczne działania oprogramowania, które mimo poprawnej realizacji podstawowych funkcji generuje dodatkowe, niezamierzone zachowania. Przykładem może być aplikacja multimedialna zapisująca dane w nieszyfrowanej pamięci tymczasowej, co może zostać wykorzystane przez osoby trzecie.
- Wstrzykiwanie kodu w aplikacjach webowych, który może zostać uruchomiony przez innych użytkowników (ang. *cross-site scripting* – XSS) i prowadzić do szkód. Ten kod może być złośliwy.
- Przepełnienie bufora wynikające z przetwarzania danych wejściowych przekraczających założone limity. Podatność zabezpieczeń polegająca na przepełnieniu bufora stwarza możliwość wykonania złośliwego kodu.

- Odmowa usługi (ang. *Denial of Service* – DoS), czyli sytuacje, w których aplikacja przestaje być dostępna dla użytkowników, na przykład w wyniku przeciążenia serwera ciągłymi żądaniami.
- Przechwytywanie, podszywanie się lub modyfikowanie, a następnie przekazywanie treści komunikatów (transakcji kartą kredytową) przez osobę trzecią w taki sposób, że użytkownik nie zdaje sobie sprawy z obecności tej osoby (atak „człowiek pośrodku" – ang. *man-in-the-middle*).
- Łamanie mechanizmów kryptograficznych zabezpieczających poufne informacje.
- Bomby logiczne, czyli celowo umieszczone w kodzie fragmenty aktywujące się tylko w określonych warunkach (np. w konkretnym dniu), które po uruchomieniu mogą wykonywać destrukcyjne działania, takie jak usuwanie danych czy formatowanie nośników.

4.3.2 Planowanie testów zabezpieczeń

Podczas przygotowywania planu testów zabezpieczeń warto szczególnie rozważyć następujące kwestie:

- Problemy z zabezpieczeniami mogą pojawiać się już na etapie projektowania architektury, tworzenia projektu czy implementacji systemu. Dlatego testy zabezpieczeń można planować na poziomie testów modułowych, integracyjnych oraz systemowych. Ze względu na zmienność zagrożeń warto także wykonywać regularne testy zabezpieczeń po wdrożeniu systemu produkcyjnego, szczególnie w przypadku dynamicznych i otwartych środowisk, takich jak IoT (ang. *Internet of Things*), gdzie częste aktualizacje oprogramowania i sprzętu mogą wprowadzać nowe ryzyka.
- Podejście do testów technicznego analityka testów może obejmować przeglądy architektury, projektu i kodu, a także analizę statyczną z wykorzystaniem narzędzi do sprawdzania zabezpieczeń. Takie narzędzia często pozwalają wykryć problemy z zabezpieczeniami, które mogą umknąć podczas testów dynamicznych.
- Techniczny analityk testów może być zaangażowany w projektowanie i przeprowadzanie określonych „ataków" na zabezpieczenia (patrz poniżej), które wymagają starannego planowania i koordynacji z interesariuszami, w tym ekspertami ds. zabezpieczeń. Inne testy zabezpieczeń, np. dotyczące ról użytkowników, zasad dostępu i uprawnień, można realizować we współpracy z programistami lub innymi analitykami testów.
- Kluczowe jest uzyskanie przez technicznego analityka testów formalnej zgody od kierownika testów na przeprowadzenie testów zabezpieczeń. Testy wykonywane bez wyraźnej autoryzacji mogą zostać potraktowane jako rzeczywiste ataki, narażając testera na konsekwencje prawne. Dlatego każda aktywność powinna mieć pisemne potwierdzenie od kierownika testów. Wyjaśnienie „przeprowadzamy test zabezpieczeń" może być trudne do przekonującego usprawiedliwienia zaistniałej sytuacji.
- W organizacjach posiadających stanowisko dyrektora ds. bezpieczeństwa informacji, plan testów należy uzgadniać z tą osobą.
- Wprowadzanie udoskonaleń zabezpieczeń może negatywnie oddziaływać na wydajność lub niezawodność systemu. W takim przypadku warto rozważyć przeprowadzenie testów wydajności i niezawodności po wprowadzeniu zmian (patrz podrozdziały 4.5 oraz 4.4).

W niektórych branżach testowanie zabezpieczeń powinno uwzględniać wymagania określone w normach, np. [IEC 62443-3-2] dla systemów automatyki przemysłowej i systemów sterowania.

Szczegółowe wytyczne dotyczące elementów planu testów zabezpieczeń można znaleźć w sylabusie Certyfikowany Tester – Tester Zabezpieczeń [CT_SEC_SYL].

4.3.3 Specyfikacja testów zabezpieczeń

Testy zabezpieczeń można klasyfikować – zgodnie z podejściem zaproponowanym w pracy [Whittaker04] – według źródła potencjalnego ryzyka zabezpieczeń. Wśród nich wyróżnia się m.in.:

- Ryzyka wynikające z interfejsu użytkownika – obejmujące próby uzyskania nieuprawnionego dostępu oraz wprowadzanie danych wejściowych prowadzących do niepożądanych lub szkodliwych efektów.
- Ryzyka związane z systemem plików – dotyczące dostępu do poufnych informacji zapisanych w plikach lub repozytoriach danych.
- Ryzyka na poziomie systemu operacyjnego – np. przechowywanie wrażliwych danych (takich jak hasła) w postaci niezaszyfrowanej w pamięci, która może zostać ujawniona po awarii systemu wywołanej złośliwymi danymi wejściowymi.
- Ryzyka wynikające z użycia oprogramowania zewnętrznego – związane z interakcjami pomiędzy systemem a zewnętrznymi modułami, których system używa; problemy te mogą występować zarówno na poziomie sieci (np. błędne pakiety lub komunikaty), jak i na poziomie modułów aplikacji (np. awaria krytycznego modułu).

Podstawą do definiowania testów zabezpieczeń mogą być także podcharakterystyki zabezpieczeń określone w normie ISO 25010 [ISO 25010]. Skupiają się one na następujących aspektach zabezpieczeń:

- poufność,
- integralność,
- niezaprzeczalność,
- rozliczalność,
- wiarygodność.

Przy projektowaniu testów zabezpieczeń można zastosować następujące podejście [Whittaker04]:

- Zgromadzenie informacji pomocniczych – takich jak dane personalne pracowników, adresy fizyczne, szczegóły dotyczące sieci wewnętrznych, adresy IP, sygnatury sprzętu i oprogramowania oraz wersje systemów operacyjnych.
- Skanowanie podatności – z użyciem powszechnie dostępnych narzędzi, które nie służą do bezpośrednich ataków, lecz do identyfikacji podatności i słabych punktów mogących naruszać zasady zabezpieczeń. Dodatkowym źródłem wiedzy o określonych podatnościach są listy kontrolne i publikacje, m.in. NIST (National Institute of Standards and Technology) [Web-1] oraz OWASP [Web-4].
- Opracowanie „planów ataków” – na podstawie zebranych informacji, obejmujących scenariusze testowe mające na celu złamanie zasad zabezpieczeń systemu. Plany te powinny uwzględniać różnorodne dane wejściowe oraz różne interfejsy (np. interfejs użytkownika czy system plików), aby umożliwić wykrycie najpoważniejszych defektów

zabezpieczeń. Opisane w [Whittaker04] techniki ataków usterek stanowią użyteczny przegląd metod dedykowanych testom zabezpieczeń.

Tak przygotowane plany ataków mogą być wykorzystane w testach penetracyjnych.

Dodatkowe informacje dotyczące testowania zabezpieczeń można znaleźć w podrozdziale 3.2 poświęconym analizie statycznej oraz w sylabusie Certyfikowany Tester – Tester Zabezpieczeń [CT_SEC_SYL].

4.4 Testowanie niezawodności

4.4.1 Wprowadzenie

Klasyfikacja charakterystyk jakościowych produktu opisana w normie ISO 25010 definiuje następujące podcharakterystyki niezawodności: dojrzałość, osiągalność, tolerowanie usterek i odtwarzalność. Testowanie niezawodności dotyczy zdolności systemu lub oprogramowania do wykonywania określonych funkcji w określonych warunkach przez określony czas.

4.4.2 Testowanie dojrzałości

Dojrzałość to stopień spełnienia wymagań dotyczących niezawodności w normalnym użytkowaniu, zwykle określanym przez profil operacyjny (patrz podrozdział 4.9). Miary dojrzałości, jeśli są stosowane, często pełnią rolę kryterium wyjścia, np. przy udostępnianiu wersji produkcyjnej systemu.

Tradycyjnie dojrzałość jest oceniana w przypadku systemów wymagających wysokiej niezawodności, takich jak te o znaczeniu krytycznym dla bezpieczeństwa (np. systemy kontroli lotu samolotu), gdzie jej poziom jest określany w normach regulacyjnych. Wymagania dojrzałości dla takich systemów mogą obejmować bardzo wysoki średni czas między awariami (MTBF), np. rzędu 10^9 godzin, choć w praktyce jego bezpośrednie zmierzenie jest niemożliwe.

Typowe badanie dojrzałości w systemach o wysokiej niezawodności polega na modelowaniu wzrostu niezawodności (ang. *reliability growth modelling*) i przeprowadza się je zwykle pod koniec testów systemowych, po zakończeniu oceny innych charakterystyk jakościowych i usunięciu wykrytych awarii. Podejście to opiera się na metodach statystycznych i jest realizowane w środowisku testowym jak najbardziej zbliżonym do operacyjnego. Aby zmierzyć określony MTBF (średni czas między awariami), dane wejściowe do testów generowane są zgodnie z profilem operacyjnym, system jest uruchamiany i wszystkie wystąpienia awarii są rejestrowane i następnie korygowane. Obserwacja spadku częstości awarii pozwala przy użyciu modelu wzrostu niezawodności oszacować MTBF.

W przypadku systemów o niższej niezawodności (np. niekrytycznych dla bezpieczeństwa) dojrzałość można określić na podstawie liczby awarii występujących w określonym czasie użytkowania operacyjnego, np. maksymalnie dwóch awarii o dużym wpływie na tydzień, i uwzględnić ją w umowie dotyczącej poziomu usług (SLA).

4.4.3 Testowanie osiągalności

Osiągalność systemu określa się zwykle jako czas, w którym jest on gotowy do pracy dla użytkowników i innych systemów w normalnych warunkach eksploatacji. System może mieć

niską dojrzałość, a jednocześnie zapewniać wysoką osiągalność. Przykładem jest sieć telefoniczna – może być w stanie obsłużyć tylko kilka połączeń naraz (co wskazuje na niską dojrzałość), ale jeśli awarie są szybko usuwane i użytkownicy mogą ponawiać próby połączeń, większość z nich pozostaje zadowolona. Z drugiej strony pojedyncza awaria trwająca kilka godzin może skutkować nieakceptowalnym poziomem osiągalności.

Osiągalność jest często ujęta w umowach SLA i mierzona w systemach produkcyjnych, takich jak strony internetowe czy aplikacje typu SaaS. Może być wyrażona w procentach, np. 99,999% („pięć dziewiątek"), co odpowiada maksymalnie 5 minutom nieosiągalności rocznie, lub w kategoriach maksymalnego czasu niedostępności, np. nie dłużej niż 60 minut w skali miesiąca.

Przed dopuszczeniem systemu do użytkowania osiągalność często ocenia się przy użyciu testów podobnych do tych stosowanych w testowaniu dojrzałości. Testy te oparte są na profilu operacyjnym i przeprowadzane w środowisku jak najbardziej zbliżonym do produkcyjnego.

Osiągalność można obliczyć jako MTTF / (MTTF + MTTR), gdzie MTTF (*Mean Time To Failure*) to średni czas do awarii, a MTTR (*Mean Time To Repair*) to średni czas potrzebny na naprawę, często mierzony w testach utrzymywalności. W systemach o wysokiej niezawodności, które mają zdolność do odzyskiwania danych po awarii (patrz sekcja 4.4.5), zamiast MTTR można użyć średniego czasu odzyskania danych po awarii.

4.4.4 Testowanie tolerowania usterek

Systemy lub oprogramowanie o bardzo wysokich wymaganiach dotyczących niezawodności często projektuje się w sposób odporny na usterki, co pozwala im w idealnych warunkach kontynuować działanie mimo wystąpienia awarii, bez zauważalnych przestojów. Głównym miernikiem odporności systemu na awarie jest jego zdolność do radzenia sobie z awariami, dlatego testowanie odporności (tolerowania usterek przez system) obejmuje symulację awarii, aby sprawdzić, czy system nadal funkcjonuje prawidłowo.

Projekt odporny na usterki zwykle obejmuje jeden lub więcej zduplikowanych podsystemów, zapewniających redundancję w sytuacji awarii. W przypadku oprogramowania zduplikowane systemy muszą być tworzone niezależnie, aby uniknąć tzw. awarii wspólnego trybu – podejście to nazywa się programowaniem N-wersji. Na przykład systemy sterowania lotem mogą posiadać trzy lub cztery poziomy redundancji, realizując najważniejsze funkcje w wielu wariantach. Gdy problem dotyczy niezawodności sprzętu, system wbudowany może działać na różnych procesorach, a krytyczne moduły mogą korzystać z serwera lustrzanego, gotowego do przejęcia działania w razie awarii serwera głównego. Niezależnie od zastosowanego rozwiązania, testowanie tolerowania usterek wymaga wykrycia awarii i sprawdzenia reakcji systemu.

Testy wstrzykiwania usterek (ang. *fault injection*) służą do oceny odporności systemu zarówno na usterki w środowisku (np. wadliwe zasilanie, niepoprawne komunikaty wejściowe, niedostępne procesy lub pliki, brak pamięci), jak i wewnątrz samego systemu (np. odwrócony bit spowodowany promieniowaniem kosmicznym, defekty projektowe lub w kodzie). Są one rodzajem testów negatywnych – celowo wprowadza się defekty, aby sprawdzić, czy system reaguje zgodnie z oczekiwaniami, np. w sposób bezpieczny dla systemów krytycznych ze względów bezpieczeństwa. Niektóre scenariusze defektów, które się testuje, nigdy nie powinny wystąpić w normalnym działaniu (np. zadanie oprogramowania nie powinno „umrzeć" ani utknąć w nieskończonej pętli). Takie sytuacje trudno odtworzyć tradycyjnym testowaniem systemu, ale

dzięki testom wstrzykiwania usterek można je zasymulować i ocenić reakcję systemu, upewniając się, że wykrywa awarię i radzi sobie z nią w sposób przewidziany.

4.4.5 Testowanie odtwarzalności

Odtwarzalność określa zdolność systemu lub oprogramowania do przywrócenia danych po awarii, uwzględniając zarówno czas potrzebny na odzyskanie danych (co może ograniczać się do częściowego przywrócenia funkcjonalności), jak i ilość danych utraconych w wyniku awarii. Testowanie odtwarzalności obejmuje między innymi weryfikację przejścia na zasilanie awaryjne oraz sprawdzanie procedur tworzenia kopii zapasowych i ich przywracania. W obu przypadkach często stosuje się próbne przebiegi kodu (ang. *dry runs*) oraz okazjonalnie, najlepiej niezapowiedziane, testy w środowiskach produkcyjnych.

Testowanie kopii zapasowych koncentrują się na minimalizowaniu wpływu awarii na dane systemowe. Sprawdzane są zarówno procedury tworzenia kopii, jak i proces przywracania danych. Chociaż tworzenie kopii zapasowych jest stosunkowo proste, odtworzenie systemu z kopii może być bardziej skomplikowane i wymaga dokładnego planowania, aby ograniczyć zakłócenia w działaniu systemu. Stosowane typowe miary obejmują czas wykonania pełnej i przyrostowej kopii zapasowej, czas potrzebny na przywrócenie danych (docelowy czas odtworzenia) oraz dopuszczalny poziom utraty danych (docelowy punkt odtworzenia).

Testowanie pracy mimo awarii dotyczy sytuacji, gdy system podstawowy współpracuje z systemem awaryjnym, który przejmuje jego zadania w przypadku awarii. W kontekście poważnych incydentów, takich jak klęski żywiołowe, ataki terrorystyczne czy ataki *ransomware*, testy te nazywa się testami odzyskiwania danych po awarii, a system awaryjny często znajduje się w innej lokalizacji geograficznej. Przeprowadzenie pełnego testu w środowisku produkcyjnym wymaga szczegółowego planowania ze względu na ryzyko i konieczność zaangażowania kadry kierowniczej. W przypadku niepowodzenia testu system podstawowy pozostaje dostępny i może natychmiast przejąć działanie. Testy pracy mimo awarii obejmują zarówno przełączenie na system awaryjny, jak i weryfikację utrzymania wymaganego poziomu usług po przejęciu kontroli.

4.4.6 Planowanie testów niezawodności

Podczas planowania testów niezawodności warto szczególnie zwrócić uwagę na następujące kwestie:

- Terminy. Testy niezawodności zwykle wymagają sprawdzenia całego systemu i muszą być poprzedzone zakończeniem innych typów testów, a ich realizacja może być czasochłonna.
- Koszty. Systemy o wysokiej niezawodności generują wysokie koszty testów, ponieważ długotrwałe testy bez awarii są potrzebne do oszacowania oczekiwanego wysokiego MTBF.
- Czas trwania. Pomiar dojrzałości przy użyciu modeli wzrostu niezawodności opiera się na rejestrowaniu awarii. Przy wysokich wymaganiach niezawodności uzyskanie statystycznie wiarygodnych danych zajmuje dużo czasu.
- Środowisko testowe. Musi ono możliwie wiernie odzwierciedlać środowisko produkcyjne. Można też wykorzystać samo środowisko produkcyjne, ale może to być uciążliwe dla użytkowników i wiązać się z wysokim ryzykiem, np. podczas testów przywracania systemu po awarii.

- Zakres. Różne podsystemy i moduły mogą wymagać testowania przy różnych poziomach i rodzajach niezawodności.
- Kryteria wyjścia. Wymagania dotyczące niezawodności powinny być określone przez obowiązujące normy i regulacje, zwłaszcza dla systemów związanych z bezpieczeństwem.
- Definicja awarii. Pomiar niezawodności opiera się na liczbie awarii, dlatego wcześniej należy uzgodnić, co dokładnie klasyfikuje się jako awarię.
- Współpraca z programistami. W testach dojrzałości, korzystając z modeli wzrostu niezawodności, konieczne jest porozumienie z programistami w zakresie szybkiego usuwania wykrytych defektów.
- Pomiar niezawodności operacyjnej. Jest stosunkowo prosty w porównaniu z pomiarem przed wydaniem, ponieważ wymaga jedynie śledzenia awarii, choć często potrzebna jest współpraca z zespołem operacyjnym.
- Wczesne testowanie. Uzyskanie wysokiej niezawodności wymaga rozpoczęcia testów jak najwcześniej, wraz z rygorystycznymi przeglądami wczesnej dokumentacji i statyczną analizą kodu.

4.4.7 Specyfikacja testów niezawodności

W testach dojrzałości i osiągalności główny nacisk kładzie się na sprawdzanie systemu w normalnych warunkach pracy. Do takich testów potrzebny jest profil operacyjny, który określa sposób użytkowania systemu (patrz podrozdział 4.9).

Natomiast w testach odporności na usterki i odtwarzalności często tworzy się scenariusze symulujące awarie zarówno w środowisku, jak i w samym systemie, aby ocenić jego reakcję. W tym celu powszechnie stosuje się testowanie wstrzykiwania usterek (ang. *fault injection testing*). Istnieją różne techniki i listy kontrolne, np. analiza drzewa usterek czy analiza przyczyn i skutków awarii (ang. *Failure Mode and Effect Analysis* – FMEA), które pomagają zidentyfikować potencjalne defekty i odpowiadające im awarie.

4.5 Testowanie wydajnościowe

4.5.1 Wprowadzenie

Norma ISO 25010 definiuje następujące podcharakterystyki wydajności: zachowanie w czasie, zużycie zasobów i pojemność. Testowanie wydajności dotyczy pomiaru wydajności systemu lub oprogramowania w określonych warunkach w odniesieniu do ilości wykorzystanych zasobów. Typowe zasoby obejmują czas, który upłynął, czas procesora, pamięć i przepustowość.

4.5.2 Testowanie zachowania w czasie

Testowanie zachowania w czasie polega na pomiarze następujących aspektów systemu (lub oprogramowania) w określonych warunkach pracy:

- czas od otrzymania żądania do momentu rozpoczęcia odpowiedzi (czas rozpoczęcia działania, a nie czas zakończenia żądanej czynności (czas odpowiedzi),
- czas potrzebny na wykonanie całej czynności, od startu do zakończenia (czas przetwarzania),

- liczba wykonanych czynności w jednostce czasu, np. operacji na bazie danych na sekundę (przepustowość).

W wielu systemach wymagania określają maksymalne czasy odpowiedzi dla poszczególnych funkcji. W takich przypadkach czas odpowiedzi obejmuje zarówno czas do rozpoczęcia reakcji, jak i czas przetwarzania. Jeśli realizacja zadania wymaga wykonania kilku etapów (np. w przetwarzaniu potokowym, ang. *pipeline*), przydatne jest zmierzenie czasu każdego kroku, aby zidentyfikować potencjalne wąskie gardła w działaniu systemu.

4.5.3 Testowanie zużycia zasobów

Testowanie zużycia zasobów polega na pomiarze następujących parametrów systemu (lub oprogramowania) w określonych warunkach działania:

- zużycie procesora, zazwyczaj wyrażane jako procent dostępnego czasu CPU,
- zużycie pamięci operacyjnej, zwykle w procentach dostępnej pamięci,
- użycie urządzeń wejścia/wyjścia, mierzone jako procent dostępnego czasu pracy tych urządzeń,
- wykorzystanie pasma sieciowego, zazwyczaj podawane jako procent dostępnej szerokości pasma.

4.5.4 Testowanie pojemności

Testowanie pojemności polega na określeniu maksymalnych granic dla różnych aspektów systemu (lub oprogramowania) w określonych warunkach pracy:

- liczba transakcji obsługiwanych w jednostce czasu (np. maksymalnie 687 przetłumaczonych słów na minutę),
- liczba użytkowników jednocześnie korzystających z systemu (np. maksymalnie 1223 użytkowników),
- liczba nowych użytkowników dodawanych do systemu w określonym czasie (np. maksymalnie 400 użytkowników na sekundę).

4.5.5 Typowe aspekty testowania wydajności

Podczas testowania zachowania w czasie, zużycia zasobów lub pojemności zazwyczaj wykonuje się wiele pomiarów, a jako wynik raportowany przyjmuje się średnią lub medianę. Wynika to z faktu, że wartości mogą się różnić w zależności od zadań wykonywanych w tle przez system. W niektórych przypadkach pomiary analizuje się bardziej szczegółowo, uwzględniając np. wariancję lub inne miary statystyczne, a wartości odstające (ang. *outliers*) mogą być badane i eliminowane, jeśli jest to uzasadnione.

Analiza dynamiczna (patrz sekcja 3.3.4) może być użyta do wykrywania modułów stanowiących wąskie gardła, monitorowania zużycia zasobów podczas testów oraz ustalania maksymalnych limitów w testach pojemności.

4.5.6 Rodzaje testowania wydajnościowego

Testy wydajnościowe mają dwa główne cele. Pierwszy polega na sprawdzeniu, czy oprogramowanie spełnia ustalone kryteria akceptacji, np. czy strona internetowa ładuje się

w określonym czasie (np. maksymalnie 4 sekund). Drugi cel to dostarczenie informacji programistom, które pozwolą poprawić wydajność systemu, np. poprzez wykrycie wąskich gardeł i określenie, które elementy architektury są najbardziej obciążone przy dużej liczbie jednoczesnych użytkowników.

Rodzaje testów wydajnościowych omówione w sekcjach 4.5.2, 4.5.3 oraz 4.5.4 służą zarówno do weryfikacji spełnienia kryteriów akceptacji, jak i do ustalania wartości bazowych, które mogą być wykorzystywane przy późniejszych zmianach systemu. Opisane testy są szczególnie przydatne, gdy celem jest zrozumienie, jak system zachowuje się w różnych warunkach użytkowania.

Testowanie obciążeniowe. Testowanie obciążeniowe (ang. *load testing*) koncentruje się na tym, jak system radzi sobie z różnym poziomem obciążenia. Obciążenie zwykle definiuje się w kategoriach liczby użytkowników korzystających jednocześnie z systemu lub liczby procesów działających równocześnie, często w oparciu o profile operacyjne (patrz podrozdział 4.9). Wydajność systemu przy określonym obciążeniu mierzy się przede wszystkim pod kątem czasów reakcji oraz zużycia zasobów, np. sprawdzając, jak podwojenie liczby użytkowników wpływa na czas odpowiedzi. Testy obciążeniowe przeprowadza się zazwyczaj rozpoczynając od niskiego obciążenia i stopniowo je zwiększając, przy jednoczesnym monitorowaniu zachowania w czasie i zużycia zasobów. Wyniki takich testów mogą być bardzo pomocne dla programistów, ponieważ ujawniają np. niespodziewane opóźnienia lub nadmierne wykorzystanie zasobów przy określonym poziomie obciążenia.

Testowanie przeciążeniowe. Testowanie przeciążeniowe (ang. *stress testing*) można podzielić na dwa typy: jeden przypomina testowanie obciążeniowe, a drugi jest powiązany z testowaniem odporności systemu (ang. *robustness testing*).

W pierwszym przypadku testy rozpoczynają się od ustawienia obciążenia na maksymalny przewidywany poziom, a następnie stopniowo zwiększa się je, aż system zacznie zawodzić (na przykład czasy odpowiedzi stają się zbyt długie lub system ulega zawieszeniu). Czasami zamiast doprowadzać system do awarii, stosuje się duże obciążenie w celu jego przeciążenia, po czym obciążenie jest zmniejszane do normalnego poziomu, aby sprawdzić, czy system odzyska wcześniejszą wydajność.

Drugi typ testów przeciążeniowych polega na celowym ograniczeniu zasobów dostępnych dla systemu, takich jak dostępna pamięć czy przepustowość, aby sprawdzić jego zachowanie w warunkach niedoboru zasobów. Wyniki takich testów pozwalają programistom zidentyfikować najbardziej krytyczne elementy systemu, tzw. słabe ogniwa, które mogą wymagać usprawnienia lub wzmocnienia.

Testowanie skalowalności. System skalowalny potrafi dostosowywać swoje zasoby do zmieniającego się poziomu obciążenia. Na przykład skalowalna witryna internetowa może w miarę wzrostu ruchu uruchamiać dodatkowe serwery backendowe, a przy spadku ruchu zmniejszać ich liczbę. Testy skalowalności są zbliżone do testów obciążeniowych, ale koncentrują się na ocenie zdolności systemu do zwiększania lub zmniejszania zasobów w odpowiedzi na zmieniające się obciążenie, np. gdy liczba użytkowników przekracza możliwości aktualnego sprzętu.

Dalsze informacje na temat rodzajów testów wydajności można znaleźć w sylabusie Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

4.5.7 Planowanie testów wydajnościowych

Przy planowaniu testów wydajnościowych kluczowe są aspekty opisane poniżej.

- Czas. Testy wydajnościowe często wymagają kompletnego systemu, dlatego zwykle realizuje się je w ramach testów systemowych.
- Przeglądy. Przeglądy kodu, zwłaszcza w obszarach interakcji z bazą danych, współpracy modułów i obsługi błędów, mogą ujawnić problemy z wydajnością, takie jak nieefektywne zapytania czy logika „czekaj i spróbuj ponownie". Przeglądy powinny odbywać się możliwie wcześnie, zanim rozpocznie się testowanie dynamiczne.
- Wczesne testowanie. Niektóre pomiary wydajności, np. obciążenie procesora przez krytyczne moduły, mogą być wykonywane już w testowaniu modułowym. Moduły wskazane jako wąskie gardła mogą być poprawiane i ponownie testowane w izolacji jako część testowania modułowego.
- Zmiany w architekturze. Negatywne wyniki testów wydajnościowych mogą wymagać modyfikacji architektury systemu. Dlatego testowanie wydajnościowe powinno się rozpocząć możliwie wcześnie, aby zapewnić czas na wprowadzenie niezbędnych zmian.
- Koszty. Narzędzia i środowiska testowe bywają kosztowne. Często stosuje się tymczasowe środowiska w chmurze i licencje „na żądanie", co wymaga optymalizacji harmonogramu testów w celu minimalizacji kosztów.
- Środowisko testowe. Powinno jak najbardziej odzwierciedlać środowisko produkcyjne, aby wyniki testów wydajnościowych można było wiarygodnie przenieść na system rzeczywisty.
- Kryteria wyjścia. Wymagania dotyczące wydajności bywają trudne do pozyskania od klienta i często ustala się je na podstawie wartości bazowych z wcześniejszych systemów. W przypadku systemów wbudowanych o znaczeniu krytycznym dla bezpieczeństwa normy prawne mogą określać maksymalne zużycie procesora i pamięci.
- Narzędzia. Testy wydajnościowe wymagają narzędzi do generowania obciążenia. Na przykład, weryfikacja skalowalności popularnego serwisu internetowego może wymagać symulacji setek tysięcy wirtualnych użytkowników. Narzędzia symulujące ograniczenia zasobów są również przydatne w testach przeciążeniowych. Ważne jest, aby używane narzędzia były zgodne z protokołami komunikacyjnymi systemu.

Szczegółowe informacje na temat planowania testów wydajnościowych zawiera sylabus Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

4.5.8 Specyfikacja testów wydajnościowych

Testy wydajnościowe opierają się głównie na badaniu systemu w określonych warunkach użytkowania. Do ich przeprowadzenia potrzebny jest profil operacyjny (patrz podrozdział 4.9), który określa, jak system będzie wykorzystywany.

W testach wydajnościowych często konieczne jest modyfikowanie poziomu obciążenia systemu poprzez zmiany w profilu operacyjnym, aby zasymulować różne scenariusze użytkowania. Na przykład w testach pojemności zwykle modyfikuje się profil operacyjny dla zmiennej podlegającej testowi (np. liczby jednoczesnych użytkowników), aż do momentu, gdy system przestaje prawidłowo funkcjonować, aby określić maksymalną dopuszczalną pojemność. Podobnie w testach obciążeniowych stopniowo zwiększa się wolumen przetwarzanych transakcji.

Szczegółowe informacje dotyczące projektowania testów wydajnościowych zawiera sylabus Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

4.6 Testowanie utrzymywalności

4.6.1 Wprowadzenie

Czas poświęcany na utrzymanie oprogramowania jest często dłuższy niż czas jego wytworzenia. Aby zapewnić jak największą efektywność procesu utrzymania, wykonuje się testy utrzymywalności (ang. *maintainability testing*), które pozwalają zmierzyć stopień łatwości analizowania kodu, wprowadzania w nim zmian oraz testowania, modularyzacji i ponownego wykorzystania. Testowanie utrzymywalności nie powinno być mylone z testowaniem pielęgnacyjnym (ang. *maintenance testing*), które jest wykonywane w celu przetestowania zmian wprowadzonych do działającego oprogramowania.

Wśród celów związanych z utrzymywalnością, jakie chcą uzyskać interesariusze (np. właściciele lub operatorzy oprogramowania), znajdują się:

- minimalizacja kosztów posiadania lub eksploatacji oprogramowania,
- ograniczenie przestojów niezbędnych na pielęgnację oprogramowania.

Testy utrzymywalności powinny zostać uwzględnione w podejściu do testów, jeśli spełniony jest co najmniej jeden z następujących warunków:

- Prawdopodobne jest wprowadzanie zmian w oprogramowaniu w wersji produkcyjnej (np. w celu usunięcia defektów lub wprowadzenia zaplanowanych aktualizacji).
- Interesariusze uznają, że korzyści wynikające z osiągnięcia celów w obszarze utrzymywalności w cyklu życia oprogramowania przeważają nad kosztami przeprowadzenia testów utrzymywalności i wprowadzenia niezbędnych zmian.
- Ryzyka związane z niską utrzymywalnością (np. długie czasy reakcji na defekty zgłaszane przez użytkowników) uzasadniają przeprowadzenie testów tego typu.

4.6.2 Statyczne i dynamiczne testowanie utrzymywalności

Techniki wykorzystywane w statycznym testowaniu utrzymywalności obejmują m.in. analizę statyczną i przeglądy, zgodnie z opisem w podrozdziałach 3.2 i 5.2. Testowanie utrzymywalności warto rozpocząć zaraz po udostępnieniu dokumentacji projektowej i kontynuować równolegle z implementacją kodu. Ponieważ utrzymywalność odnosi się do jakości kodu i dokumentacji poszczególnych modułów kodu, można ją oceniać już we wczesnej fazie cyklu życia oprogramowania, bez konieczności posiadania pełnego, działającego systemu.

Dynamiczne testy utrzymywalności koncentrują się natomiast na weryfikacji udokumentowanych procedur utrzymania aplikacji, np. opisujących sposób aktualizacji oprogramowania. Scenariusze pielęgnacyjne służą jako przypadki testowe, pozwalając sprawdzić, czy dokumentacja umożliwia osiągnięcie wymaganych poziomów usług. Ten rodzaj testowania jest szczególnie istotny w środowiskach złożonych, gdzie procedury wsparcia obejmują wiele działów lub organizacji. Testy dynamiczne utrzymywalności mogą być realizowane w ramach operacyjnych testów akceptacyjnych.

4.6.3 Podcharakterystyki utrzymywalności

Utrzymywalność systemu można określić poprzez następujące cechy:

- analizowalność,
- modyfikowalność,
- testowalność.

Na te cechy wpływ mają m.in. dobre praktyki programistyczne, takie jak stosowanie komentarzy, logiczne nazewnictwo zmiennych oraz poprawne wcięcia w kodzie, a także dostępność dokumentacji technicznej, np. specyfikacji projektowej systemu i specyfikacji interfejsów.

Dodatkowo, istotnymi podcharakterystykami utrzymywalności [ISO 25010] są:

- modułowość,
- reużywalność.

Modułowość można testować przy pomocy analizy statycznej (patrz sekcja 3.2.3), natomiast testowanie reużywalności można przeprowadzać poprzez przeglądy architektury (patrz rozdział 5).

4.7 Testowanie przenaszalności

4.7.1 Wprowadzenie

Testowanie przenaszalności (ang. *portability testing*) dotyczy tego, jak łatwo moduł lub system można przenieść do docelowego środowiska (zarówno po raz pierwszy, jak i z istniejącego środowiska), dostosować do nowego otoczenia lub zastąpić nim inną jednostkę.

Zgodnie z normą ISO 25010 [ISO 25010], przenaszalność obejmuje następujące podcharakterystyki:

- instalowalność,
- adaptowalność,
- zastępowalność.

Testowanie przenaszalności można rozpocząć od pojedynczych modułów, np. weryfikując możliwość zastąpienia jednego modułu innym (np. przejście z jednej bazy danych na inną), a następnie rozszerzać zakres testów wraz z udostępnianiem kolejnych fragmentów kodu. Instalowalność zwykle można ocenić dopiero po wytworzeniu wszystkich modułów produktu.

Przenaszalność powinna być uwzględniona już na etapie projektowania systemu i tworzenia jego architektury. Przeglądy projektu i architektury (patrz rozdział 5) są szczególnie efektywnym sposobem wykrywania wymagań dotyczących przenaszalności oraz potencjalnych problemów, takich jak zależności od konkretnego systemu operacyjnego.

4.7.2 Testowanie instalowalności

Testowanie instalowalności (ang. *installability testing*) obejmuje ocenę zarówno samego oprogramowania, jak i procedur służących do jego instalacji w docelowym środowisku. Może

dotyczyć na przykład systemu produkcyjnego lub kreatora instalacji wykorzystywanego do wdrożenia produktu na komputerze użytkownika.

Główne cele testowania instalowalności to:

- Walidacja, czy oprogramowanie można zainstalować zgodnie z instrukcjami z podręcznika instalacji lub przy użyciu kreatora instalacji, uwzględniając różne konfiguracje sprzętowe i programowe oraz różne typy instalacji (np. instalacja początkowa, aktualizacja).
- Testowanie, czy ewentualne awarie podczas instalacji (np. brak wymaganej biblioteki DLL) są poprawnie obsługiwane i nie pozostawiają systemu w stanie nieokreślonym, np. z częściowo zainstalowanym oprogramowaniem lub błędną konfiguracją.
- Testowanie możliwości wykonania częściowej instalacji lub deinstalacji.
- Sprawdzenie, czy kreator instalacji poprawnie rozpoznaje nieobsługiwane platformy sprzętowe i niezgodne konfiguracje systemu operacyjnego.
- Ocena, czy proces instalacji można zakończyć w przewidzianym czasie lub w określonej liczbie kroków.
- Walidacja możliwości powrotu do starszej wersji oprogramowania lub jego pełnej deinstalacji.

Po zakończeniu instalacji zwykle przeprowadza się testy przydatności funkcjonalnej (ang. *functional suitability testing*), aby wykryć ewentualne problemy wynikające z instalacji, takie jak niepoprawne konfiguracje lub brak dostępności funkcji. Równolegle często realizuje się testy użyteczności, np. w celu sprawdzenia, czy użytkownicy otrzymują jasne instrukcje, komunikaty o postępie oraz komunikaty o błędach.

4.7.3 Testowanie adaptowalności

Testowanie adaptowalności (ang. *adaptability testing*) polega na sprawdzeniu, czy aplikacja działa prawidłowo we wszystkich przewidzianych środowiskach docelowych, obejmujących sprzęt, systemy operacyjne, oprogramowanie pośrednie i inne elementy. W ramach przygotowania specyfikacji testów adaptowalności należy zidentyfikować odpowiednie środowiska docelowe, skonfigurować je i udostępnić zespołowi testowemu. Następnie przeprowadza się testy funkcjonalne na wybranych przypadkach testowych, aby zweryfikować działanie poszczególnych modułów w tych środowiskach.

Adaptowalność obejmuje również możliwość przenoszenia oprogramowania między różnymi środowiskami zgodnie z określoną procedurą, a testy mogą posłużyć do sprawdzenia poprawności tej procedury.

4.7.4 Testowanie zastępowalności

Testowanie zastępowalności (ang. *replaceability testing*) skupia się na sprawdzeniu, czy dany moduł może skutecznie zastąpić istniejący moduł w systemie. Jest to szczególnie istotne w systemach wykorzystujących komercyjne oprogramowanie gotowe do użycia lub w aplikacjach Internetu rzeczy (IoT).

Testy zastępowalności można prowadzić równolegle z testami integracji podstawowej funkcjonalności systemu, jeśli w pełnej integracji dostępnych jest kilka alternatywnych modułów. Ocena zastępowalności może odbywać się także w formie przeglądu technicznego lub inspekcji

na poziomie architektury i projektu, gdzie kluczowe jest wyraźne określenie interfejsów modułu, który może pełnić rolę zamiennika.

4.8 Testowanie kompatybilności

4.8.1 Wprowadzenie

W testowaniu kompatybilności brane są pod uwagę następujące podcharakterystyki [ISO 25010]:

- współistnienie,
- współdziałanie.

4.8.2 Testowanie współistnienia

Systemy, które nie są bezpośrednio powiązane, nazywa się współistniejącymi, jeśli mogą działać w tym samym środowisku (np. na tym samym sprzęcie) bez wzajemnego zakłócania pracy, np. bez konfliktów w korzystaniu z zasobów. Testowanie współistnienia przeprowadza się, gdy nowe lub zaktualizowane oprogramowanie ma być wdrożone w środowisku, w którym już funkcjonują inne aplikacje.

Problemy mogą pojawić się, gdy aplikacja jest testowana w środowisku, gdzie jest jedyną zainstalowaną aplikacją i zgodność z innymi systemami nie jest weryfikowana, a następnie zostaje przeniesiona do środowiska produkcyjnego, gdzie działają także inne aplikacje.

Typowe cele testowania współistnienia obejmują:

- sprawdzenie, czy uruchomienie aplikacji w tym samym środowisku co inne systemy nie powoduje negatywnego wpływu na ich przydatność funkcjonalną, np. konfliktów w korzystaniu z zasobów przy wielu uruchomionych aplikacjach,
- ocenę, w jaki sposób instalacja poprawek lub aktualizacji systemu operacyjnego wpływa na działanie aplikacji.

Aspekty współistnienia należy uwzględnić już na etapie planowania docelowego środowiska produkcyjnego, natomiast same testy wykonuje się zwykle po zakończeniu testów systemowych.

4.9 Profile operacyjne

Profile operacyjne są wykorzystywane jako element specyfikacji testów dla różnych typów testów niefunkcjonalnych, takich jak testy niezawodności i wydajności. Są one szczególnie przydatne, gdy wymagania odnoszą się do działania systemu „w określonych warunkach", ponieważ pozwalają te warunki zdefiniować.

Profil operacyjny określa typowy sposób korzystania z systemu, zazwyczaj pod kątem liczby użytkowników i operacji wykonywanych przez system. Użytkownicy są zwykle klasyfikowani według liczby osób korzystających z systemu, godzin aktywności oraz rodzaju użytkownika (np. administrator lub zwykły użytkownik). Operacje wykonywane przez system definiuje się wraz z częstotliwością i prawdopodobieństwem ich wystąpienia. Dane te można uzyskać z narzędzi monitorujących działający system lub poprzez prognozowanie użytkowania na podstawie szacunków dostarczanych przez organizację biznesową.

Narzędzia testowe mogą generować dane wejściowe zgodnie z profilem operacyjnym, często w sposób pseudolosowy. Pozwala to tworzyć wirtualnych użytkowników w liczbie odpowiadającej profilowi operacyjnemu (np. do testów niezawodności i osiągalności) lub ją przekraczającej (np. do testów przeciążeniowych lub testów pojemności). Szczegóły dotyczące takich narzędzi omówiono w sekcji 6.2.3.

5. Przeglądy (165 min.)

Słowa kluczowe

przegląd, przegląd techniczny

Cele nauczania dla rozdziału 5

5.1 Zadania technicznego analityka testów w trakcie przeglądów

TTA-5.1.1 (K2) Wyjaśnić, dlaczego przygotowanie do przeglądu jest istotne w przypadku technicznego analityka testów.

5.2 Korzystanie z list kontrolnych podczas przeglądów

TTA-5.2.1 (K4) Przeanalizować projekt architektury i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.

TTA-5.2.2 (K4) Przeanalizować fragment kodu lub pseudokodu i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.

5.1 Zadania technicznego analityka testów w trakcie przeglądów

Techniczny analityk testów powinien aktywnie uczestniczyć w przeglądach technicznych, wnosząc własną, specjalistyczną perspektywę dotyczącą omawianych zagadnień. Wszyscy uczestnicy powinni być przeszkoleni w zakresie formalnych przeglądów, aby świadomie pełnić swoje role, a jednocześnie powinno im zależeć na maksymalnym wykorzystaniu korzyści płynących z dobrze przeprowadzonego przeglądu. Obejmuje to utrzymywanie konstruktywnej współpracy z autorami podczas zgłaszania i omawiania uwag. Szczegółowe wytyczne dotyczące przeglądów technicznych, wraz z przykładami list kontrolnych, znajdują się w [Wiegers02]. Techniczny analityk testów zazwyczaj uczestniczy zarówno w przeglądach technicznych, jak i inspekcjach, gdzie może przedstawić perspektywę operacyjną dotyczącą zachowania systemu, której programiści mogli nie uwzględnić. Ponadto odpowiada za definiowanie, stosowanie i aktualizowanie list kontrolnych przeglądów oraz za ocenę krytyczności wykrytych defektów.

Niezależnie od rodzaju wykonywanego przeglądu, techniczny analityk testów musi przeznaczyć odpowiednią ilość czasu na przygotowanie. Obejmuje to analizę produktu pracy, weryfikację powiązanej dokumentacji pod kątem spójności oraz zidentyfikowanie braków w produkcie pracy. Bez solidnego przygotowania przegląd może sprowadzić się jedynie do czynności redakcyjnych. Dobry przegląd wymaga zrozumienia zawartości produktu, określenia braków, weryfikacji poprawności technicznej oraz sprawdzenia spójności z innymi produktami – zarówno już istniejącymi, jak i tymi w trakcie tworzenia. Na przykład przy przeglądzie planu testów na poziomie integracji techniczny analityk testów musi uwzględnić integrowane elementy: czy są gotowe do integracji, czy istnieją zależności wymagające dokumentacji oraz czy dostępne są dane do testowania punktów integracji. Przegląd nie ogranicza się więc wyłącznie do samego produktu, ale obejmuje także jego interakcje z innymi częściami systemu.

5.2 Korzystanie z list kontrolnych podczas przeglądów

Podczas przeglądów stosuje się listy kontrolne, które przypominają uczestnikom o konieczności sprawdzenia określonych elementów. Dzięki nim można też ograniczyć wpływ subiektywnych opinii („ta sama lista jest używana we wszystkich przeglądach, nie tylko w odniesieniu do twojego produktu"). Listy kontrolne mogą mieć charakter ogólny lub skupiać się na konkretnych charakterystykach jakościowych lub obszarach. Na przykład lista do przeglądów dokumentacji wymagań może obejmować sprawdzenie poprawnego użycia zwrotów typu „będzie" i „powinien", weryfikację formatowania czy zgodność z przyjętym szablonem. Mogą one również koncentrować się na kwestiach związanych z zabezpieczeniami lub wydajnością.

Najbardziej efektywne są listy kontrolne opracowywane stopniowo przez poszczególne jednostki organizacyjne, ponieważ uwzględniają:

- charakter produktu,
- lokalne środowisko wytwórcze, w tym personel, używane narzędzia i priorytety,
- doświadczenia wynikające z wcześniejszych sukcesów i defektów,
- specyficzne problemy, np. związane z wydajnością lub zabezpieczeniami.

Listy kontrolne powinny być dostosowane do potrzeb organizacji, a w niektórych przypadkach również do wymagań konkretnego projektu. Przykłady podane w tym rozdziale należy traktować jako wskazówki, a nie jako obowiązkowy zestaw.

5.2.1 Przegląd architektury

Architektura oprogramowania definiuje podstawowe zasady i koncepcje systemu, obejmujące jego komponenty, wzajemne powiązania oraz reguły projektowania i rozwoju [ISO 42010]. Podczas przeglądów architektury, np. w kontekście zachowania w czasie stron internetowych, można posługiwać się listami kontrolnymi. Przykładowe punkty w takich listach[4] (źródło: [Web-2]) obejmują weryfikację poprawnej implementacji:

- zestawiania połączeń – pomagającego skrócić czas potrzebny na łączenie z bazą danych poprzez wykorzystanie współdzielonej puli połączeń,
- równoważenia obciążenia – służącego do równomiernego rozdzielenia obciążenia między elementy zasobów,
- przetwarzania rozproszonego,
- buforowania – czyli użycia lokalnych kopii danych w celu przyspieszenia dostępu,
- inicjowania z opóźnieniem (tzw. *lazy initialization*),
- współbieżności transakcji,
- separacji procesów przetwarzania transakcyjnego w czasie rzeczywistym (ang. *Online Transactional Processing* – OLTP) i analitycznego przetwarzania w czasie rzeczywistym (ang. *Online Analytical Processing* – OLAP),
- replikacji danych.

5.2.2 Przeglądy kodu

Listy kontrolne stosowane przy przeglądach kodu mają charakter niskopoziomowy i najlepiej sprawdzają się, gdy są dostosowane do konkretnego języka programowania. Uwzględnienie antywzorców (ang. *anti-patterns*) na poziomie kodu może być szczególnie pomocne dla mniej doświadczonych programistów.

W przeglądach kodu listy kontrolne[5] mogą obejmować następujące obszary:

1. Struktura kodu

- Czy kod w pełni i prawidłowo implementuje projekt?
- Czy przestrzegane są obowiązujące standardy kodowania?
- Czy struktura, styl i formatowanie kodu są jednolite?
- Czy nie występują nieużywane lub nieosiągalne fragmenty kodu?
- Czy w kodzie nie pozostały zaślepki lub procedury testowe?
- Czy fragmenty kodu można zastąpić wywołaniami do wielokrotnego użytku modułów lub funkcji z biblioteki?
- Czy występują powtarzające się fragmenty kodu, które można scalić w jedną procedurę?
- Czy wykorzystanie pamięci jest optymalne?
- Czy zamiast „magicznych liczb” lub stałych łańcuchowych stosowane są wartości symboliczne?

[4] W pytaniu egzaminacyjnym znajdzie się podzbiór elementów listy kontrolnej podanej w sylabusie, na podstawie którego należy udzielić odpowiedzi.
[5] W pytaniu egzaminacyjnym znajdzie się podzbiór elementów listy kontrolnej podanej w sylabusie, na podstawie którego należy udzielić odpowiedzi.

- Czy moduły o dużej złożoności wymagają restrukturyzacji lub podziału na mniejsze jednostki?

2. Dokumentacja

- Czy komentarze w kodzie są jasne, poprawne i ułatwiają wprowadzanie zmian?
- Czy treść komentarzy jest zgodna z kodem?
- Czy dokumentacja spełnia obowiązujące normy?

3. Zmienne

- Czy wszystkie zmienne są właściwie zdefiniowane i mają spójne, znaczące nazwy?
- Czy nie występują zmienne nadmiarowe lub nieużywane?

4. Operacje arytmetyczne

- Czy unika się porównywania liczb zmiennoprzecinkowych?
- Czy stosowane są mechanizmy zapobiegające błędom zaokrągleń?
- Czy nie wykonuje się operacji arytmetycznych (dodawania i odejmowania) na liczbach o różnych rzędach wielkości?
- Czy dzielniki są sprawdzane pod kątem zerowych lub niepoprawnych wartości?

5. Pętle i gałęzie

- Czy pętle, gałęzie i konstrukcje logiczne są kompletne, poprawne i właściwie zagnieżdżone?
- Czy w łańcuchach IF-ELSE IF najczęstsze przypadki sprawdzane są na początku?
- Czy wszystkie przypadki w IF-ELSE IF lub bloku CASE są uwzględnione, w tym klauzule ELSE/DEFAULT?
- Czy każda instrukcja CASE ma wartość domyślną?
- Czy warunki zakończenia pętli są oczywiste i zawsze osiągalne?
- Czy indeksy tablic i zmienne sterujące są właściwie inicjalizowane przed wejściem do pętli?
- Czy instrukcje możliwe do wyjęcia z pętli zostały przeniesione poza nią?
- Czy kod w pętli unika manipulowania zmienną sterującą lub korzystania z niej po wyjściu z pętli?

6. Programowanie defensywne

- Czy sprawdzany jest dostęp do tablic, rekordów i plików pod kątem przekroczenia granic?
- Czy importowane dane i argumenty wejściowe są weryfikowane pod kątem poprawności i kompletności?
- Czy wszystkie zmienne wyjściowe mają przypisane wartości?
- Czy w każdej instrukcji używany jest właściwy element danych?
- Czy pamięć przydzielona dynamicznie jest zwalniana?
- Czy przy dostępie do urządzeń zewnętrznych stosowane są limity czasu lub mechanizmy obsługi błędów?
- Czy przed dostępem do pliku sprawdzane jest jego istnienie?
- Czy po zakończeniu programu wszystkie pliki i urządzenia pozostają w poprawnym stanie?

6. Narzędzia testowe i automatyzacja testów (180 min.)

Słowa kluczowe

emulator, posiew usterek, rejestrowanie-odtwarzanie, symulator, testowanie oparte na słowach kluczowych, testowanie oparte na modelu (MBT), testowanie sterowane danymi, wstrzykiwanie usterek, wykonywanie testu

Cele nauczania dla rozdziału 6

6.1 Definiowanie projektu automatyzacji testów

TTA-6.1.1 (K2) Omówić czynności wykonywane przez technicznego analityka testów podczas konfigurowania projektu automatyzacji testów.

TTA-6.1.2 (K2) Omówić różnice między automatyzacją sterowaną danymi a automatyzacją opartą na słowach kluczowych.

TTA-6.1.3 (K2) Omówić często występujące problemy techniczne, z powodu których w projektach automatyzacji nie udaje się uzyskać zaplanowanego zwrotu z inwestycji.

TTA-6.1.4 (K3) Utworzyć słowa kluczowe na podstawie danego procesu biznesowego.

6.2 Kategorie narzędzi testowych

TTA-6.2.1 (K2) Omówić zastosowanie narzędzi do posiewu usterek i wstrzykiwania usterek.

TTA-6.2.2 (K2) Omówić główne cechy narzędzi do testów wydajnościowych oraz zagadnienia dotyczące ich wdrażania.

TTA-6.2.3 (K2) Przedstawić ogólne zastosowania narzędzi do testowania stron internetowych.

TTA-6.2.4 (K2) Omówić sposoby wspierania przez narzędzia koncepcji testowania opartego na modelu.

TTA-6.2.5 (K2) Omówić zastosowanie narzędzi używanych do obsługi testowania modułów i procesu budowania.

TTA-6.2.6 (K2) Omówić zastosowanie narzędzi używanych do obsługi testowania aplikacji mobilnych.

6.1 Definiowanie projektu automatyzacji testów

Narzędzia testowe, w szczególności te służące do automatyzacji wykonywania testów, muszą być starannie zaprojektowane i posiadać odpowiednią architekturę, aby ich wdrożenie było opłacalne. Próba automatyzacji testów bez przemyślanej architektury skutkuje systemem kosztownym w utrzymaniu, nieefektywnym i niezdolnym do realizacji założonych celów, co uniemożliwia osiągnięcie oczekiwanego zwrotu z inwestycji.

Projekt automatyzacji testów należy traktować jak typowy projekt programistyczny. Obejmuje to opracowanie dokumentacji architektury i szczegółowego projektu, przeprowadzanie przeglądów projektu i kodu, testowanie modułowe, integracji modułów oraz systemowe. Korzystanie z niestabilnego lub wadliwego kodu automatyzacji może niepotrzebnie wydłużyć i skomplikować proces testowania.

Techniczny analityk testów może pełnić wiele funkcji w zakresie automatyzacji testów, w tym:

- Ustalenie osób odpowiedzialnych za realizację testów, we współpracy z kierownikiem testów, jeśli jest to potrzebne.
- Wybór odpowiedniego narzędzia dla organizacji, określenie harmonogramu, wymagań dotyczących kompetencji zespołu i utrzymania narzędzia, co może obejmować decyzję o stworzeniu własnego narzędzia zamiast jego zakupu.
- Definiowanie wymagań dotyczących integracji narzędzia z innymi narzędziami, np. narzędziami do zarządzania testami, defektami czy ciągłą integracją.
- Opracowanie adapterów do połączenia narzędzia testowego z testowanym oprogramowaniem.
- Wybór podejścia do automatyzacji, np. opartego na słowach kluczowych lub sterowanego danymi (patrz sekcja 6.6.1).
- Określenie kosztów wdrożenia (w tym szkoleń) wspólnie z kierownikiem testów; w metodykach zwinnych dyskusja nad tymi kosztami odbywa się podczas spotkań planistycznych (planowanie projektu, planowanie sprintu).
- Opracowanie harmonogramu wdrożenia automatyzacji oraz planowanie czasu na utrzymanie narzędzia.
- Szkolenie analityków testów i analityków biznesowych w zakresie obsługi narzędzia i dostarczania danych.
- Ustalenie sposobu i momentu wykonywania testów automatycznych.
- Określenie metod łączenia wyników testów automatycznych z wynikami testów manualnych.

W projektach, w których automatyzacja odgrywa kluczową rolę, część z tych zadań może przejąć inżynier automatyzacji testów (szczegóły w sylabusie Certyfikowany Tester Inżynier Automatyzacji Testów [CT_TAE_SYL]), natomiast zadania organizacyjne może realizować kierownik testów, w zależności od potrzeb projektu. W metodykach zwinnych powiązanie ról z tymi zadaniami jest zwykle elastyczne i mniej formalne.

Decyzje i działania podejmowane w tym zakresie mają bezpośredni wpływ na skalowalność i utrzymywalność rozwiązania dla automatyzacji testów. Należy poświęcić wystarczająco dużo czasu na analizę dostępnych narzędzi, technologii i strategii oraz zrozumienie przyszłych planów organizacji.

6.1.1 Wybór podejścia do automatyzacji

W tej sekcji omówiono kluczowe zagadnienia związane z automatyzacją testów, obejmujące:

- wykorzystanie do automatyzacji interfejsu GUI, API i CLI,
- testowanie sterowane danymi,
- testowanie oparte na słowach kluczowych,
- obsługę awarii oprogramowania,
- zarządzanie stanem systemu.

Szczegółowe informacje dotyczące wyboru podejścia do automatyzacji znajdują się w sylabusie Certyfikowany Tester Inżynier Automatyzacji Testów [CT_TAE_SYL].

Automatyzacja przez GUI, API i CLI. Automatyzacja testów nie ogranicza się wyłącznie do interfejsu graficznego (GUI). Można również stosować narzędzia testowe działające na poziomie API, wiersza poleceń (CLI) lub innych punktów dostępowych w testowanym systemie. Jedną z pierwszych decyzji technicznego analityka testów jest wybór najlepszego interfejsu do automatyzacji. Standardowe narzędzia często wymagają opracowania adapterów do wybranych interfejsów, a czas potrzebny na ich stworzenie należy uwzględnić w planie prac.

Testowanie przez GUI może być problematyczne ze względu na zmiany w interfejsie w trakcie rozwoju oprogramowania. Skrypty nagrane podczas sesji testowej mogą przestać działać po modyfikacjach obiektów GUI, co zwiększa nakład pracy na ich utrzymanie. Dzieje się tak dlatego, że w nagranym skrypcie zapisane są interakcje z obiektami GUI, a jeśli poszczególne obiekty są modyfikowane, może zajść potrzeba aktualizacji skryptu.

Narzędzia rejestrująco-odtwarzające zapewniają wygodny punkt wyjścia do projektowania skryptów automatyzacji. Tester nagrywa sesję testową i powstały skrypt jest następnie modyfikowany w celu zwiększenia utrzymywalności kodu (np. poprzez zastąpienie fragmentów skryptu wywołaniami funkcji wielokrotnego użytku).

Zastosowanie podejścia „testowanie sterowane danymi”. Dane wykorzystywane w poszczególnych testach mogą różnić się w zależności od testowanego oprogramowania, choć wykonywane kroki pozostają zasadniczo takie same (np. przy testowaniu obsługi błędów w polach wejściowych poprzez wprowadzanie wielu niepoprawnych wartości i sprawdzanie zwracanych komunikatów błędów). Tworzenie i utrzymywanie osobnych zautomatyzowanych skryptów dla każdej testowanej wartości jest nieefektywne. Rozwiązaniem stosowanym w praktyce jest przeniesienie danych ze skryptów do zewnętrznej bazy, np. arkusza kalkulacyjnego lub bazy danych. Następnie tworzy się funkcje umożliwiające pobieranie odpowiednich danych przy każdym uruchomieniu skryptu, dzięki czemu jeden skrypt może obsłużyć zestaw danych zawierający zarówno wartości wejściowe, jak i oczekiwane wyniki (np. komunikaty wyświetlane w polach tekstowych lub komunikaty o błędach). Takie podejście określa się jako testowanie sterowane danymi.

Oprócz skryptów testowych przetwarzających dane, tworzy się również jarzmo testowe oraz infrastrukturę niezbędną do uruchamiania pojedynczych skryptów lub całych zestawów skryptów. Rzeczywiste dane przechowywane w arkuszu lub bazie danych przygotowują analitycy testów, którzy znają funkcje biznesowe realizowane przez oprogramowanie. W metodykach zwinnych w proces definiowania danych, szczególnie dla testów akceptacyjnych, może być zaangażowany

także przedstawiciel strony biznesowej (np. właściciel produktu). Taki podział obowiązków pozwala twórcom skryptów (np. technicznym analitykom testów) skupić się na implementacji skryptów automatyzacji, podczas gdy właścicielem testu pozostaje analityk testów. W większości przypadków to analityk testów odpowiada za uruchamianie skryptów po ich zaimplementowaniu i przetestowaniu automatyzacji.

Zastosowanie podejścia opartego na słowach kluczowych. W podejściu zwanym testowaniem opartym na słowach kluczowych (lub słowach akcji) oddziela się działania wykonywane na danych testowych od samego skryptu testowego [Buwalda01]. W tym celu tworzy się język wysokiego poziomu o charakterze opisowym, który nie jest przeznaczony do bezpośredniego uruchamiania kodu. Słowa kluczowe mogą obejmować zarówno działania wysokiego, jak i niskiego poziomu. Na przykład słowa kluczowe związane z procesami biznesowymi, takie jak „Zaloguj", „UtwórzUżytkownika" czy „UsuńUżytkownika", opisują działania wysokiego poziomu w obrębie aplikacji. Dodatkowo definiuje się słowa kluczowe niskiego poziomu, które odpowiadają interakcjom z interfejsem oprogramowania, np. „KliknijPrzycisk", „WybierzZListy" lub „WpiszTekst", używane przy testowaniu funkcji GUI niezwiązanych bezpośrednio z procesami biznesowymi. Słowa kluczowe mogą również przyjmować parametry, np. słowo „LogIn" może zawierać parametry „nazwa użytkownika" i „hasło".

Po zdefiniowaniu słów kluczowych i danych testowych osoba odpowiedzialna za automatyzację testów (np. techniczny analityk testów lub inżynier automatyzacji testów) tłumaczy je na kod automatyzacji. Słowa kluczowe, odpowiadające im działania oraz dane mogą być zapisane w arkuszach lub wprowadzone za pomocą narzędzi wspierających automatyzację opartą na słowach kluczowych. Struktura do testów automatycznych implementuje słowa kluczowe w postaci modułowych funkcji lub skryptów wykonywalnych. Narzędzia odczytują przypadki testowe zapisane w postaci słów kluczowych i wywołują odpowiednie funkcje lub skrypty. Modułowa budowa skryptów ułatwia odwzorowanie ich na konkretne słowa kluczowe i wymaga umiejętności programowania.

Oddzielenie wiedzy o logice biznesowej od rzeczywistego kodu implementującego automatyzację pozwala efektywnie wykorzystać zasoby testowe. Techniczny analityk testów może wykorzystać swoje kompetencje programistyczne, nie stając się ekspertem w wielu dziedzinach biznesowych. Dodatkowo oddzielenie kodu od danych podlegających zmianom pozwala odizolować automatyzację od wprowadzanych modyfikacji, poprawia utrzymywalność kodu i zwiększa zwrot z inwestycji w automatyzację.

Obsługa awarii oprogramowania. W projekcie automatyzacji testów należy przewidzieć możliwość wystąpienia awarii oprogramowania i określić sposób ich obsługi. Osoba odpowiedzialna za automatyzację powinna zdecydować, jak ma zachowywać się oprogramowanie wykonujące testy w przypadku awarii. Należy ustalić, czy awarię rejestrować i kontynuować testy, przerwać ich wykonywanie, czy też obsłużyć awarię poprzez konkretne działania (np. kliknięcie przycisku w oknie dialogowym) albo dodanie opóźnienia w teście. Brak właściwej obsługi awarii może spowodować problemy nie tylko w trakcie bieżącego testu, ale także wpłynąć na wyniki kolejnych testów.

Uwzględnianie stanu systemu. Ważne jest także uwzględnienie stanu systemu na początku i końcu każdego testu. Często konieczne jest przywrócenie systemu do określonego stanu po zakończeniu testów, co pozwala wielokrotnie uruchamiać zestawy testów automatycznych bez potrzeby ręcznej interwencji użytkownika. Aby to osiągnąć, testy automatyczne mogą

np. usuwać utworzone dane lub zmieniać status rekordów w bazie danych. Struktura automatyzacji powinna zapewnić, że zakończenie testów nastąpi w poprawny sposób, np. poprzez prawidłowe wylogowanie z systemu.

6.1.2 Modelowanie procesów biznesowych na potrzeby automatyzacji

Aby wdrożyć automatyzację testów opartą na słowach kluczowych, konieczne jest wcześniejsze odwzorowanie procesów biznesowych przeznaczonych do testowania w języku słów kluczowych wysokiego poziomu. Język ten powinien być łatwy do zrozumienia dla jego użytkowników, którymi zazwyczaj są analitycy testów, a w przypadku zwinnego wytwarzania oprogramowania – także przedstawiciele jednostek biznesowych (np. właściciel produktu).

Słowa kluczowe opisują głównie wysokopoziomowe interakcje biznesowe z systemem. Na przykład słowo kluczowe „AnulujZamówienie" może obejmować sprawdzenie istnienia zamówienia, weryfikację uprawnień osoby zgłaszającej anulowanie, wyświetlenie zamówienia oraz żądanie potwierdzenia akcji. Przypadki testowe definiuje analityk testów, łącząc odpowiednie słowa kluczowe (np. „Zaloguj", „WybierzZamówienie", „AnulujZamówienie") z właściwymi danymi testowymi.

W trakcie tworzenia języka słów kluczowych warto zwrócić uwagę na następujące aspekty:

- Im bardziej szczegółowe są słowa kluczowe, tym bardziej precyzyjne scenariusze można opracować, ale jednocześnie trudniej zarządzać całym językiem wysokiego poziomu.
- Pozwolenie analitykom testów na definiowanie działań niskiego poziomu (np. „KliknijPrzycisk", „WybierzZListy") umożliwia obsługę większej liczby różnych sytuacji testowych. Jednak takie działania są silnie powiązane z GUI, co może wymagać dodatkowej pielęgnacji testów po wprowadzeniu zmian w interfejsie.
- Tworzenie złożonych, zagregowanych słów kluczowych może uprościć projektowanie testów, ale utrudnić pielęgnację. Przykładowo sześć słów kluczowych mogących utworzyć rekord może zostać połączone w jedno słowo wysokiego poziomu, ale takie podejście nie zawsze jest najbardziej efektywne.
- Nawet po starannym opracowaniu języka słów kluczowych często zachodzi potrzeba dodania nowych słów lub modyfikacji istniejących. Każde słowo kluczowe obejmuje dwa aspekty: logikę biznesową oraz funkcję automatyzacji, która ją realizuje, dlatego należy wdrożyć proces obsługujący oba te elementy.

Automatyzacja testów oparta na słowach kluczowych może być bardziej ekonomiczna w utrzymaniu w porównaniu z innymi metodami. Początkowe wdrożenie może być kosztowne, ale przy długotrwałym projekcie przynosi ogólnie niższe koszty.

Sylabus Certyfikowany Tester Inżynier Automatyzacji Testów [CT_TAE_SYL] zawiera dodatkowe informacje dotyczące modelowania procesów biznesowych na potrzeby automatyzacji.

6.2 Kategorie narzędzi testowych

W tym podrozdziale przedstawiono informacje na temat narzędzi, których może użyć techniczny analityk testów, a które nie zostały omówione w sylabusie Certyfikowany Tester Poziom Podstawowy [CTFL_SYL].

Szczegółowe informacje na temat narzędzi można znaleźć w następujących sylabusach ISTQB®:

- Certyfikowany Tester – Tester Aplikacji Mobilnych [CT_MAT_SYL],
- Certyfikowany Tester – Tester Wydajności [CT_PT_SYL],
- Certyfikowany Tester Testowanie Oparte na Modelu [CT_MBT_SYL],
- Certyfikowany Tester Inżynier Automatyzacji Testów [CT_TAE_SYL].

6.2.1 Narzędzia do posiewu usterek

Narzędzia do posiewu usterek wprowadzają zmiany w testowanym kodzie (czasem korzystając z gotowych algorytmów) w celu sprawdzenia, jak dobrze określone testy pokrywają kod. Stosowane systematycznie pozwalają ocenić skuteczność testów, czyli ich zdolność do wykrywania wprowadzonych defektów, a w razie potrzeby – poprawić jakość testów.

Zwykle narzędzia te wykorzystuje techniczny analityk testów, ale mogą być także używane przez programistów podczas testowania nowo napisanego kodu.

6.2.2 Narzędzia do wstrzykiwania usterek

Narzędzia do wstrzykiwania usterek celowo wprowadzają do oprogramowania nieprawidłowe dane wejściowe, aby sprawdzić, czy system prawidłowo radzi sobie z obsługą błędów. Takie wartości powodują nieprawidłowości w działaniu programu, które powinny uruchomić mechanizmy obsługi błędów i zostać przetestowane. Zakłócenie normalnego przebiegu wykonywania kodu jednocześnie zwiększa pokrycie testowanego kodu.

Zazwyczaj z tych narzędzi korzystają techniczni analitycy testów, choć programiści mogą je również stosować podczas testowania nowo powstałego kodu.

6.2.3 Narzędzia do testów wydajnościowych

Narzędzia do testów wydajnościowych pełnią przede wszystkim trzy funkcje:

- generują obciążenie systemu,
- mierzą, monitorują, wizualizują i analizują reakcję systemu na to obciążenie,
- dostarczają informacji o wykorzystaniu zasobów w modułach systemu oraz w elementach sieciowych.

Generowanie obciążenia odbywa się poprzez implementację wcześniej zdefiniowanego profilu operacyjnego (patrz podrozdział 4.9) w postaci skryptu. Skrypt może zostać najpierw zarejestrowany dla pojedynczego użytkownika, np. przy użyciu narzędzia rejestrująco-odtwarzającego, a następnie wykonany zgodnie z określonym profilem operacyjnym przy pomocy narzędzia do testów wydajnościowych. W implementacji należy uwzględnić zróżnicowanie danych w poszczególnych transakcjach lub zestawach transakcji.

Narzędzia te generują obciążenie, symulując dużą liczbę wirtualnych użytkowników realizujących określone profile operacyjne i generujących dane wejściowe o ustalonej wielkości. Skrypty testów wydajnościowych zazwyczaj odtwarzają interakcje użytkowników z systemem na poziomie protokołu komunikacyjnego, a nie przez interfejs GUI, jak to ma miejsce w standardowych skryptach automatyzacji. Takie podejście pozwala zwykle zmniejszyć liczbę koniecznych „sesji” w trakcie testów. Niektóre narzędzia dodatkowo sterują działaniem aplikacji poprzez interfejs GUI, co umożliwia dokładniejsze pomiary czasów odpowiedzi systemu pod obciążeniem.

Podczas testów narzędzia do testów wydajnościowych wykonują liczne pomiary, które można analizować w trakcie testów lub po ich zakończeniu. Typowe metryki i raporty obejmują:

- liczbę symulowanych użytkowników przez cały czas trwania testu,
- liczbę i typ transakcji generowanych przez wirtualnych użytkowników oraz częstotliwość ich występowania,
- czasy odpowiedzi na poszczególne żądania transakcji wykonywane przez użytkowników,
- raporty i wykresy pokazujące obciążenie i czasy reakcji,
- raporty dotyczące wykorzystania zasobów, np. zmiany w użytkowaniu na przestrzeni czasu z podaniem wartości minimalnych i maksymalnych.

Przy wdrażaniu narzędzi do testów wydajnościowych należy uwzględnić istotne czynniki:

- sprzęt i przepustowość sieci potrzebne do wygenerowania obciążenia,
- zgodność narzędzia z protokołami komunikacyjnymi używanymi w testowanym systemie,
- elastyczność umożliwiającą implementację różnych profili operacyjnych,
- wymagania dotyczące monitorowania, analizy i raportowania.

Ze względu na dużą pracochłonność wytwarzania takich narzędzi, zazwyczaj są one nabywane, a nie opracowywane w ramach projektu. Wyjątkiem może być sytuacja, gdy istnieją ograniczenia techniczne uniemożliwiające użycie gotowego narzędzia lub gdy profil obciążenia i realizowane funkcje są prostsze niż te dostępne w narzędziach komercyjnych. Dodatkowe informacje o narzędziach do testów wydajnościowych znajdują się w sylabusie Certyfikowany Tester – Tester Wydajności [CT_PT_SYL].

6.2.4 Narzędzia do testowania stron internetowych

Istnieje wiele wyspecjalizowanych narzędzi, zarówno komercyjnych, jak i *open source*, przeznaczonych do testowania stron internetowych. Poniżej przedstawiono przykłady zastosowań niektórych popularnych narzędzi w tej kategorii:

- narzędzia do testowania hiperłączy, umożliwiające skanowanie serwisu i wykrywanie brakujących lub nieprawidłowych linków,
- narzędzia do sprawdzania poprawności kodu HTML i XML pod kątem zgodności ze standardami dla poszczególnych stron,
- narzędzia do testów wydajnościowych, które badają reakcję serwera przy dużym obciążeniu użytkowników,
- lekkie narzędzia automatyzujące testy, działające w różnych przeglądarkach,
- narzędzia do skanowania kodu serwera, identyfikujące nieużywane lub „osierocone” pliki,
- narzędzia do kontroli pisowni obsługujące HTML,
- narzędzia do sprawdzania arkuszy CSS,
- narzędzia weryfikujące zgodność z normami, np. dotyczącymi dostępności (Section 508 w USA, M/376 w Europie),
- narzędzia do wykrywania różnorodnych problemów zabezpieczeń.

Dobre źródła narzędzi *open source* do testowania stron internetowych to między innymi:

- World Wide Web Consortium (W3C) [Web-3] – organizacja ustalająca standardy internetowe i udostępniająca narzędzia do wykrywania niezgodności z nimi;

- Web Hypertext Application Technology Working Group (WHATWG) [Web-5] – organizacja definiująca standardy HTML i oferująca narzędzie do sprawdzania poprawności kodu HTML [Web-6].

Niektóre narzędzia wyposażone w roboty indeksujące (ang. *web spider*) mogą także dostarczać dane o wielkości stron, czasie ich pobrania czy dostępności (np. występowaniu błędu HTTP 404). Informacje te są przydatne dla programistów, administratorów i testerów.

Analitycy testów oraz techniczni analitycy testów korzystają z tych narzędzi głównie podczas testów systemowych.

6.2.5 Narzędzia wspomagające testowanie oparte na modelu

Testowanie oparte na modelu (ang. *Model-Based Testing* – MBT) to podejście, w którym do opisu oczekiwanego zachowania systemu wykorzystuje się jego model, na przykład automat skończony. Komercyjne narzędzia MBT (patrz np. [Utting07]) często oferują funkcję umożliwiającą „wykonywanie" modelu. Interesujące scenariusze powstałe w trakcie wykonywania modelu mogą zostać zapisane i wykorzystane jako przypadki testowe. Podobnie działają inne wykonywalne modele, takie jak sieci Petriego czy maszyny stanów, które także pozwalają na ten typ testowania.

Modele i narzędzia MBT umożliwiają generowanie dużych zbiorów odrębnych ścieżek wykonywania. Narzędzia te pozwalają również ograniczyć ogromną liczbę możliwych ścieżek, które mogłyby zostać wygenerowane w obrębie modelu. Wykorzystanie takiego podejścia daje nową perspektywę na testowane oprogramowanie i umożliwia wykrycie defektów, które mogły pozostać niezauważone podczas testów funkcjonalnych.

Szczegółowe informacje na temat narzędzi do testowania opartego na modelu można znaleźć w sylabusie Certyfikowany Tester Testowanie Oparte na Modelu [CT_MBT_SYL].

6.2.6 Narzędzia do testowania modułowego i budowania wersji

Narzędzia do testowania modułowego i automatyzacji budowania wersji są pierwotnie przeznaczone dla programistów, jednak w wielu przypadkach techniczni analitycy testów również je wykorzystują i utrzymują, szczególnie w projektach opartych na metodykach zwinnych.

Narzędzia do testowania modułowego często są ściśle powiązane z językiem, w którym napisany jest dany moduł. Przykładowo, w przypadku języka Java automatyzację testów jednostkowych można realizować za pomocą środowiska JUnit. Podobne narzędzia istnieją dla innych języków i ogólnie określa się je mianem środowisk xUnit. W takich środowiskach przedmioty testów są tworzone dla każdej klasy, która została utworzona, co ułatwia programistom automatyzację testowania modułowego.

Niektóre narzędzia do automatyzacji budowania wersji pozwalają na automatyczne uruchomienie procesu tworzenia nowej wersji po wprowadzeniu zmian w module. Po jego zakończeniu inne narzędzia mogą automatycznie uruchamiać testy modułowe. Taki stopień automatyzacji jest typowy dla środowisk ciągłej integracji.

Odpowiednio skonfigurowany zestaw tych narzędzi może znacząco podnieść jakość wersji przekazywanych do testów. W sytuacji, gdy wprowadzone zmiany powodują defekty regresji, niektóre testy automatyczne zwykle zakończą się niezaliczeniem. Przed przekazaniem wersji

do środowiska testowego programista podejmuje wtedy próbę zidentyfikowania przyczyn awarii i ich usunięcia.

6.2.7 Narzędzia wspomagające testowanie aplikacji mobilnych

W testowaniu aplikacji mobilnych powszechnie wykorzystuje się symulatory oraz emulatory.

Symulatory. Symulator urządzenia mobilnego odwzorowuje środowisko wykonawcze określonej platformy mobilnej. Testowane aplikacje są kompilowane w specjalnej wersji przeznaczonej do uruchamiania na symulatorze, a nie na fizycznym urządzeniu. Narzędzia te pełnią rolę zastępczą wobec rzeczywistych urządzeń, jednak zazwyczaj wykorzystuje się je głównie do wczesnych testów funkcjonalnych oraz do symulowania wielu wirtualnych użytkowników podczas testów obciążeniowych. W porównaniu z emulatorami symulatory są prostsze w użyciu i pozwalają na szybsze wykonywanie testów. Należy jednak mieć na uwadze, że aplikacja testowana w symulatorze różni się od wersji przeznaczonej do dystrybucji.

Emulatory. Emulator urządzenia mobilnego odtwarza działanie sprzętu i korzysta z tego samego środowiska wykonawczego, które jest wykorzystywane przez fizyczne urządzenia. Aplikacje skompilowane w celu uruchomienia na emulatorze mogą być również instalowane i uruchamiane na docelowych urządzeniach fizycznych.

Emulator nie stanowi jednak pełnego odpowiednika rzeczywistego urządzenia, ponieważ jego zachowanie może się różnić od faktycznego. Dodatkowo może on nie obsługiwać wszystkich funkcji sprzętowych, takich jak ekran dotykowy (w tym obsługa wielu punktów dotyku) czy akcelerometr. Ograniczenia te wynikają m.in. z platformy, na której działa emulator.

Wspólne aspekty zastosowań. Zarówno symulatory, jak i emulatory są często wykorzystywane w celu obniżenia kosztów środowisk testowych poprzez ograniczenie potrzeby używania fizycznych urządzeń. Narzędzia te są szczególnie przydatne na wczesnych etapach cyklu wytwórczego, ponieważ zwykle integrują się ze środowiskami programistycznymi i umożliwiają szybkie wdrażanie, testowanie oraz monitorowanie aplikacji. Aby skorzystać z emulatora lub symulatora, należy go uruchomić i zainstalować na nim aplikację w sposób analogiczny do instalacji na rzeczywistym urządzeniu. Każde środowisko programistyczne dla mobilnych systemów operacyjnych zazwyczaj oferuje własny emulator lub symulator, choć dostępne są również rozwiązania pochodzące od innych dostawców.

Emulatory i symulatory umożliwiają zwykle konfigurowanie różnych parametrów użytkowych, takich jak symulacja sieci o zróżnicowanej przepustowości i sile sygnału, utrata pakietów, zmiana orientacji urządzenia, generowanie przerwań czy wykorzystanie danych lokalizacyjnych GPS. Część tych ustawień jest szczególnie cenna, ponieważ ich odtworzenie na rzeczywistych urządzeniach bywa trudne lub kosztowne, na przykład w przypadku określonej lokalizacji GPS lub poziomu sygnału sieciowego.

Dodatkowe informacje dotyczące tego zagadnienia można znaleźć w sylabusie poświęconym testowaniu aplikacji mobilnych Certyfikowany Tester – Tester Aplikacji Mobilnych [CT_MAT_SYL].

7. Bibliografia

Normy

[DO-178C]: Software Considerations in Airborne Systems and Equipment Certification, RTCA 2011.

[ISO 9126] ISO/IEC 9126-1:2001, Software Engineering –Software Product Quality

[ISO 25010] ISO/IEC 25010:2011, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models

[ISO 29119] ISO/IEC/IEEE 29119-4 Software and systems engineering – Software testing Part 4: Test techniques.

[ISO 42010] ISO/IEC/IEEE 42010:2011 Systems and software engineering – Architecture description

[IEC 61508] IEC 61508-5 (2010) Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Part 5: Examples of methods for the determination of safety integrity levels

[ISO 26262] ISO 26262-1:2018, Road vehicles – Functional safety, części od 1 do 12.

[IEC 62443-3-2] IEC 62443-3-2:2020, Security for industrial automation and control systems, część 3-2: Security risk assessment for system design

Sylabusy ISTQB®

[ISTQB_AL_TTA_OVERVIEW] Wprowadzenie do poziomu zaawansowanego – techniczny analityk testów, wersja 4.0

[CT_SEC_SYL] Sylabus Certyfikowany Tester – Tester Zabezpieczeń, wersja 2016

[CT_TAE_SYL] Sylabus Certyfikowany Tester Inżynier Automatyzacji Testów, wersja 2017

[CTFL_SYL] Sylabus Certyfikowany Tester Poziom Podstawowy, wersja 2018 v3.1

[CT_PT_SYL] Sylabus Certyfikowany Tester – Tester Wydajności, wersja 2018

[CT_MBT_SYL] Sylabus Certyfikowany Tester Testowanie Oparte na Modelu, wersja 2015

[CT_TM_SYL] Sylabus Certyfikowany Tester Poziom Zaawansowany Kierownik Testów, wersja 2012

[CT_MAT_SYL] Sylabus Certyfikowany Tester – Tester Aplikacji Mobilnych, wersja 2019

[ISTQB_GLOSSARY] Słownik terminów testowych ISTQB®, wersja 3.5, 2020

[CT_AuT_SYL] Sylabus Certyfikowany Tester – Tester Oprogramowania Automotive, wersja 2018

Książki i artykuły

[Andrist20] Björn Andrist and Viktor Sehr, C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition, Packt Publishing, 2020

[Beizer90] Boris Beizer, „Software Testing Techniques, Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3

[Buwalda01] Hans Buwalda, „Integrated Test Design and Automation", Addison-Wesley Longman, 2001, ISBN 0-201-73725-6

[Kaner02] Cem Kaner, James Bach, Bret Pettichord; „Lessons Learned in Software Testing"; Wiley, 2002, ISBN: 0-471-08112-4

[McCabe76] Thomas J. McCabe, „A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, 1976 r. PP 308-320

[Roman18] Adam Roman, „Testowanie i jakość oprogramowania. Modele, techniki, narzędzia", Polskie Wydawnictwo Naukowe PWN, 2018, ISBN 978-83-01-19644-8

[Utting07] Mark Utting, Bruno Legeard, „Practical Model-Based Testing: A Tools Approach", Morgan-Kaufmann, 2007, ISBN: 978-0-12-372501-1

[Whittaker04] James Whittaker i Herbert Thompson, „How to Break Software Security", Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0

[Wiegers02] Karl Wiegers, „Peer Reviews in Software: A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0

Inne źródła

Wymienione poniżej odwołania wskazują informacje dostępne w Internecie. Odwołania zostały sprawdzone w momencie publikacji niniejszego sylabusa poziomu zaawansowanego, ISTQB® nie ponosi jednak odpowiedzialności za ich ewentualną późniejszą niedostępność.

[Web-1] http://www.nist.gov (NIST National Institute of Standards and Technology)

[Web-2] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx

[Web-3] http://www.W3C.org

[Web-4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[Web-5] https://whatwg.org

[Web-6] https://validator.w3.org/

[Web-7] https://dl.acm.org/doi/abs/10.1145/3340433.3342822

8. Dodatek A – przegląd charakterystyk jakościowych

W poniższej tabeli przedstawiono porównanie charakterystyk jakościowych opisanych w nieaktualnej już normie ISO 9126-1 (używanych w sylabusie z 2012 r.) z charakterystykami pochodzącymi z nowej normy ISO 25010 [ISO 25010] (używanymi w najnowszej wersji sylabusa). Przydatność funkcjonalna i użyteczność są omówione w sylabusie Certyfikowany Tester Poziom Zaawansowany Analityk Testów.

ISO/IEC 25010	ISO/IEC 9126-1	Uwagi
Funkcjonalność (przydatność funkcjonalna)	Funkcjonalność	Nowa nazwa jest bardziej precyzyjna i pozwala uniknąć pomyłek z innymi znaczeniami "funkcjonalności".
Kompletność funkcjonalna		Pokrycie określonych potrzeb
Poprawność funkcjonalna	Dokładność	Bardziej ogólne niż dokładność
Adekwatność funkcjonalna	Przydatność	Pokrycie domniemanych potrzeb
	Współdziałanie	Przeniesione do Zgodności
	Zabezpieczenia	Teraz osobna charakterystyka
Wydajność	Efektywność	Zmieniona nazwa w celu uniknięcia konfliktu z definicją efektywności w ISO/IEC 25062
Zachowanie w czasie	Zachowanie w czasie	
Zużycie zasobów	Zużycie zasobów	
Pojemność		Nowa podcharakterystyka
Kompatybilność		Nowa charakterystyka
Współistnienie	Współistnienie	Charakterystyka przeniesiona z przenaszalności
Współdziałanie		Charakterystyka przeniesiona z funkcjonalności (analityk testów)
Użyteczność		Wyraźnie widoczna kwestia jakości
Rozpoznawalność	Zrozumiałość	Nowa nazwa jest bardziej precyzyjna
Łatwość nauki	Łatwość nauki	
Łatwość obsługi	Łatwość obsługi	
Błędoodporność		Nowa podcharakterystyka
Estetyka interfejsu użytkownika	Atrakcyjność	Nowa nazwa jest bardziej precyzyjna
Dostępność		Nowa podcharakterystyka
Niezawodność	Niezawodność	
Dojrzałość	Dojrzałość	
Dostępność		Nowa podcharakterystyka
Tolerowanie usterek	Tolerowanie usterek	
Odtwarzalność	Odtwarzalność	
Zabezpieczenia	Zabezpieczenia	
Poufność		Brak poprzednich podcharakterystyk
Integralność		Brak poprzednich podcharakterystyk
Niezaprzeczalność		Brak poprzednich podcharakterystyk
Odpowiedzialność		Brak poprzednich podcharakterystyk

Rozliczalność		Brak poprzednich podcharakterystyk
Wiarygodność		Brak poprzednich podcharakterystyk
Utrzymywalność	Utrzymywalność	
Modułowość		Nowa podcharakterystyka
Reużywalność		Nowa podcharakterystyka
Analizowalność	Analizowalność	
Modyfikowalność	Stabilność	Bardziej precyzyjna nazwa łącząca modyfikowalność i stabilność
Testowalność	Testowalność	
Przenaszalność	Przenaszalność	
Adaptowalność	Adaptowalność	
Instalowalność	Instalowalność	
	Współistnienie	Charakterystyka przeniesiona do kompabybilności
Zastępowalność	Zastępowalność	

9. Dodatek B – macierz powiązań między celami biznesowymi a celami nauczania

Cele nauczania	Poziom K	TTA1	TTA2	TTA3	TTA4	TTA5	TTA6	TTA7	TTA8
1. Zadania technicznego analityka testów w testowaniu opartym na ryzyku									
1.2 Zadania związane z testowaniem opartym na ryzyku									
TTA-1.2.1 Omówić ogólne ryzyka, które zwykle musi wziąć pod uwagę techniczny analityk testów.	K2	X							
TTA-1.2.2 Omówić czynności wykonywane przez technicznego analityka testów w ramach podejścia do testowania opartego na ryzyku, związane z czynnościami testowymi.	K2	X							
2. Białoskrzynkowe techniki testowania									
2.2 Testowanie instrukcji									
TTA-2.2.1 Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania instrukcji, aby osiągnąć zdefiniowany poziom pokrycia.	K3			X					
2.3 Testowanie decyzji									
TTA-2.3.1 Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania decyzji, aby osiągnąć zdefiniowany poziom pokrycia.	K3			X					
2.4. Testowanie MC/DC									
TTA-2.4.1 Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania MC/DC, aby osiągnąć pełne pokrycie MC/DC.	K3			X					
2.5. Testowanie kombinacji warunków									
TTA-2.5.1 Zaprojektować przypadki testowe dla podanego przedmiotu testów, korzystając z techniki testowania kombinacji warunków, aby osiągnąć zdefiniowany poziom pokrycia.	K3			X					
2.7. Testowanie API									
TTA-2.7.1 Rozumieć obszary zastosowania testów API i rodzaje defektów wykrywanych w takich testach.	K2			X					
2.8. Wybór białoskrzynkowej techniki testowania									
TTA-2.8.1 Wybrać odpowiednią białoskrzynkową technikę testowania zgodnie z daną sytuacją projektową.	K4			X					
3. Analiza statyczna i analiza dynamiczna									
3.2 Analiza statyczna									
TTA-3.2.1 Obliczyć złożoność cyklomatyczną i zastosować analizę przepływu sterowania w celu wykrycia ewentualnych anomalii w kodzie związanych z tym przepływem.	K3					X			
TTA-3.2.2 Wykorzystać analizę przepływu danych w celu wykrycia ewentualnych anomalii w kodzie związanych z tym przepływem.	K3					X			
TTA-3.2.3 Zaproponować sposoby zwiększenia utrzymywalności kodu za pomocą analizy statycznej.	K3					X			
3.3 Analiza dynamiczna									

Cele nauczania	Poziom K	TTA1	TTA2	TTA3	TTA4	TTA5	TTA6	TTA7	TTA8
TTA-3.3.1 Zastosować analizę dynamiczną, aby osiągnąć określony cel.	K3					X			
4. Charakterystyki jakościowe w testach technicznych									
4.2 Ogólne planowanie									
TTA-4.2.1 Dla konkretnego scenariusza przeanalizować wymagania niefunkcjonalne i napisać odpowiednie fragmenty planu testów.	K4	X							
TTA-4.2.2 Określić dla danego ryzyka produktowego konkretne typy testów niefunkcjonalnych, które są najbardziej odpowiednie.	K3	X							
TTA-4.2.3 Rozpoznać i omówić etapy w cyklu życia aplikacji, w których należy w typowych sytuacjach przeprowadzić testowanie niefunkcjonalne.	K2	X	X						
TTA-4.2.4 Dla konkretnego scenariusz zdefiniować typy defektów, których wykrycia należy się spodziewać w przypadku użycia różnych typów testów niefunkcjonalnych.	K3	X	X						
4.3 Testowanie zabezpieczeń									
TTA-4.3.1 Omówić przyczyny uwzględnienia testowania zabezpieczeń w podejściu do testowania.	K2		X						
TTA-4.3.2 Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów zabezpieczeń.	K2		X						
4.4 Testowanie niezawodności									
TTA-4.4.1 Omówić przyczyny uwzględnienia testowania niezawodności w podejściu do testowania.	K2		X						
TTA-4.4.2 Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów niezawodności.	K2		X						
4.5 Testowanie wydajnościowe									
TTA-4.5.1 Omówić przyczyny uwzględnienia testowania wydajnościowego w podejściu do testowania.	K2		X						
TTA-4.5.2 Omówić podstawowe aspekty, które należy uwzględnić podczas planowania i specyfikowania testów wydajnościowych.	K2		X						
4.6 Testowanie utrzymywalności									
TTA-4.6.1 Omówić przyczyny uwzględnienia testowania utrzymywalności w podejściu do testowania.	K2		X						
4.7 Testowanie przenaszalności									
TTA-4.7.1 Omówić przyczyny uwzględnienia testowania przenaszalności w podejściu do testowania.	K2		X						
4.8 Testowanie kompatybilności									
TTA-4.8.1 Omówić przyczyny uwzględnienia testowania współistnienia w podejściu do testowania.	K2		X						
5. Przeglądy									
5.1. Zadania technicznego analityka testów w trakcie przeglądów									
TTA-5.1.1 Wyjaśnić, dlaczego przygotowanie do przeglądu jest istotne w przypadku technicznego analityka testów.	K2				X				
5.2. Korzystanie z list kontrolnych podczas przeglądów									
TTA-5.2.1 Przeanalizować projekt architektury i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.	K4				X				
TTA-5.2.2 Przeanalizować fragment kodu lub pseudokodu i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.	K4				X				

Cele nauczania	Poziom K	TTA1	TTA2	TTA3	TTA4	TTA5	TTA6	TTA7	TTA8
6. Narzędzia testowe i automatyzacja testów									
6.1. Definiowanie projektu automatyzacji testów									
TTA-6.1.1 Omówić czynności wykonywane przez technicznego analityka testów podczas konfigurowania projektu automatyzacji testów.	K2								X
TTA-6.1.2 Omówić różnice między automatyzacją sterowaną danymi a automatyzacją opartą na słowach kluczowych.	K2						X		X
TTA-6.1.3 Omówić często występujące problemy techniczne, z powodu których w projektach automatyzacji nie udaje się uzyskać zaplanowanego zwrotu z inwestycji.	K2								X
TTA-6.1.4 Utworzyć słowa kluczowe na podstawie danego procesu biznesowego.	K3								X
6.2 Kategorie narzędzi testowych									
TTA-6.2.1 Omówić zastosowanie narzędzi do posiewu usterek i wstrzykiwania usterek.	K2						X		
TTA-6.2.2 Omówić główne cechy narzędzi do testów wydajnościowych oraz zagadnienia dotyczące ich wdrażania.	K2						X		X
TTA-6.2.3 Przedstawić ogólne zastosowania narzędzi do testowania stron internetowych.	K2						X	X	
TTA-6.2.4 Omówić sposoby wspierania przez narzędzia koncepcji testowania opartego na modelu.	K2						X	X	
TTA-6.2.5 Omówić zastosowanie narzędzi używanych do obsługi testowania modułów i procesu budowania.	K2						X	X	
TTA-6.2.6 Omówić zastosowanie narzędzi używanych do obsługi testowania aplikacji mobilnych.	K2						X	X	
Razem liczba celów nauczania pokrywających cel biznesowy		**6**	**11**	**6**	**3**	**4**	**7**	**4**	**5**

10. Indeks

Słowa kluczowe są zdefiniowane w Słowniku terminów testowych ISTQB® (glossary.istqb.org/).