Certyfikowany tester
Sylabus dla poziomu zaawansowanego
Techniczny analityk testowy

Wersja 2012

International Software Testing Qualifications Board

Podgrupa robocza poziomu zaawansowanego dla Technicznego Analityka Testów: Graham Bath (przewodniczący), Paul Jorgensen, Jamie Mitchell; 2010-2012.

Historia zmian

Wersja	Data	Uwagi
ISEB 1.1	04.09.2001	Sylabus ISEB Practitioner
ISTQB 1.2E	09.2003	Sylabus ISTQB dla poziomu zaawansowanego EOQ-SG
V2007	12.10.2007	Sylabus certyfikowanego testera dla poziomu zaawansowanego, wersja 2007
D100626	26.06.2010	Uwzględnienie zmian zaakceptowanych w 2009 r., oddzielenie rozdziałów dotyczących poszczególnych modułów
D101227	27.12.2010	Zaakceptowanie zmian formatowania i poprawek niemających wpływu na znaczenie zdań
Pierwsza wersja robocza	17.09.2011	Pierwsza wersja nowego, odrębnego sylabusa dla technicznych analityków testowych, oparta na uzgodnionym dokumencie z opisem zakresu. Przegląd w ramach grupy roboczej poziomu zaawansowanego
Druga wersja robocza	20.11.2011	Wersja do przeglądu przez komisje krajowe
Alfa 2012	09.03.2012	Uwzględnienie komentarzy komisji krajowych zgłoszonych do październikowego wydania
Beta 2012	07.04.2012	Wersja beta przekazana zgromadzeniu ogólnemu
Beta 2012	08.06.2012	Zredagowana wersja udostępniona komisjom krajowym
Beta 2012	27.06.2012	Uwzględnienie komentarzy do glosariusza i EWG
RC 2012	15.08.2012	Wersja kandydująca z uwzględnionymi ostatecznymi modyfikacjami zgłoszonymi przez komisje krajowe
RC 2012	02.09.2012	Uwzględnienie komentarzy BNLTB i Stuarta Reida. Sprawdzenie przez Paula Jorgensena
GA 2012	19.10.2012	Ostateczne poprawki redakcyjne i oczyszczenie dokumentu przed wydaniem firmowanym przez zgromadzenie ogólne (GA)

Spis treści

Podziękowania

Niniejszy dokument został opracowany przez zespół członków podgrupy roboczej poziomu zaawansowanego ds. technicznej analizy testów International Software Testing Qualifications Board w składzie: Graham Bath (przewodniczący), Paul Jorgensen, Jamie Mitchell.

Zespół składa podziękowania zespołowi weryfikatorów oraz komisjom krajowym za sugestie i wskazówki. W chwili zakończenia prac nad sylabusem poziomu zaawansowanego członkami grupy roboczej poziomu zaawansowanego byli (w kolejności alfabetycznej):

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (wiceprzewodniczący), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (przewodniczący), Geoff Thompson, Erik van Veenendaal, Tsuyoshi Yumoto.

Następujące osoby uczestniczyły w przeglądach, zgłaszały komentarze i głosowały nad niniejszym sylabusem:

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng, Shaomin Zhu.

Niniejszy dokument został formalnie wydany przez zgromadzenie ogólne ISTQB® 19 października 2012 r.

Polskie tłumaczenie zostało wykonane przez biuro tłumaczeń BTInfo.

Edycja i przegląd tłumaczenia zespół pod kierunkiem Jana Sabaka w składzie Damian Brzeczek, Adam Ścierski, Bartosz Walter, Artur Zwoliński

0. Wprowadzenie

0.1 Przeznaczenie dokumentu

Niniejszy sylabus stanowi podstawę egzaminu International Software Testing Qualification dla technicznych analityków testowych na poziomie zaawansowanym. ISTQB® udostępnia ten sylabus następującym odbiorcom:
1. Komisjom krajowym (National Boards) -- w celu tłumaczenia na języki lokalne i akredytacji dostawców szkoleń. Komisje krajowe mogą dostosowywać sylabus do potrzeb danego języka i modyfikować odwołania do literatury tak, aby wskazywały na publikacje lokalne.
2. Komisjom egzaminacyjnym (Exam Boards) – jako podstawę do formułowania pytań egzaminacyjnych w języku lokalnym, odpowiadających celom nauczania danego sylabusa.
3. Dostawcom szkoleń -- w celu tworzenia materiałów szkoleniowych i doboru odpowiednich metod nauczania.
4. Kandydatom ubiegającym się o certyfikat – w celu przygotowania do egzaminu (w ramach szkoleń zorganizowanych lub samodzielnie).
5. Międzynarodowej społeczności specjalistów od oprogramowania i inżynierii systemów – w celu rozwijania zawodu testera oprogramowania i systemów oraz jako podstawę książek i artykułów.

ISTQB® może zezwolić innym podmiotom na korzystanie z niniejszego sylabusa do innych celów, o ile te podmioty uprzednio złożą wniosek o pisemne zezwolenie na takie korzystanie i otrzymają je.

0.2 Przegląd

Na poziomie zaawansowanym są dostępne trzy osobne sylabusy:

- „Kierownik testów”,
- „Analityk testowy”,
- „Techniczny analityk testowy”.

Dokument podsumowania poziomu zaawansowanego [ISTQB_AL_OVIEW] zawiera następujące informacje:

- cele biznesowe dla każdego sylabusa,
- podsumowanie każdego sylabusa,
- powiązania między sylabusami,
- opis poziomów poznawczych,
- załączniki.

0.3 Cele nauczania podlegające egzaminowaniu

Cele nauczania wspierają cele biznesowe i służą do formułowania egzaminów certyfikacyjnych „Techniczny analityk testowy — poziom zaawansowany”. Co do zasady, wszystkie części niniejszego sylabusa podlegają weryfikacji na poziomie K1, czyli kandydat musi potrafić rozpoznać, zapamiętać i przypomnieć sobie dane pojęcie lub koncepcję. Cele nauczania na poziomach K2, K3 i K4 są przedstawione na początku każdego z rozdziałów.

0.4 Wymagania wstępne

Niektóre z celów nauczania określonych dla technicznego analityka testowego zakładają posiadanie podstawowej wiedzy w następujących obszarach:

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

1. Zadania technicznego analityka testowego w testowaniu opartym na ryzyku — 30 minut

Słowa kluczowe
ryzyko produktowe, analiza ryzyka, ocena ryzyka, identyfikacja ryzyka, poziom ryzyka, łagodzenie ryzyka, testowanie oparte na ryzyku

Cele nauczania związane z zadaniami technicznego analityka testowego w testowaniu opartym na ryzyku

1.3 Ocena ryzyka

TTA-1.3.1 (K2) Kandydat potrafi omówić ogólne czynniki ryzyka, które zwykle musi wziąć pod uwagę techniczny analityk testowy

Wspólne cele nauczania
Poniższy cel nauczania wiąże się z zagadnieniami przedstawionymi w kilku punktach niniejszego rozdziału.

TTA-1.x.1 (K2) Kandydat potrafi omówić czynności wykonywane przez technicznego analityka testowego w ramach podejścia opartego na ryzyku, związane z planowaniem i wykonywaniem testów.

1.1 Wprowadzenie

Jednym z zadań kierownika testów jest ogólny nadzór nad ustanowieniem strategii testów opartej na ryzyku i zarządzaniem tą strategią. Kierownik testów zwykle angażuje technicznego analityka testowego w prace nad zapewnieniem poprawnej implementacji podejścia opartego na ryzyku.

Ze względu na specyficzne kompetencje techniczne, techniczni analitycy testowi aktywnie uczestniczą w następujących zadaniach związanych z testowaniem opartym na ryzyku:

- identyfikacja ryzyka,
- łagodzeniu ryzyka.

Te zadania są wykonywane cyklicznie w trakcie całego projektu, tak aby zespół projektowy mógł reagować na pojawiające się ryzyka produktowe i zmieniające się priorytety oraz regularnie oceniać i informować interesariuszy o statusie ryzyka.

Techniczny analityk testowy działa w ramach podejścia do testowania opartego na ryzyku ustanowionego przez kierownika testów dla potrzeb danego projektu. Powinien przy tym korzystać ze swojej znajomości czynników ryzyka technicznego występujących w projekcie, na przykład dotyczących bezpieczeństwa, niezawodności systemów i wydajności.

1.2 Identyfikacja ryzyka

W procesie identyfikacji ryzyka szanse na wykrycie jak największej liczby potencjalnych istotnych ryzyk są tym większe, im większa jest liczba zaangażowanych interesariuszy. Techniczni analitycy testowi dysponują unikatowymi kompetencjami technicznymi, dlatego są szczególnie predysponowani do przeprowadzania wywiadów z ekspertami, prowadzenia sesji „burzy mózgów” ze współpracownikami oraz analizowania

aktualnych uwarunkowań i nabytych doświadczeń w celu zidentyfikowania prawdopodobnych obszarów ryzyka produktowego. Podczas pracy nad określaniem takich obszarów mogą w szczególności ściśle współpracować z innymi osobami o kompetencjach technicznych np. programistami, architektami i specjalistami ds. eksploatacji.

Przykładami czynników ryzyka, jakie mogą zostać zidentyfikowane, są:

- ryzyko związane z wydajnością (np. brak możliwości uzyskania właściwych czasów odpowiedzi w warunkach dużego obciążenia);
- ryzyko związane z bezpieczeństwem (np. ujawnienie wrażliwych danych w wyniku ataku);
- ryzyko związane z niezawodnością (np. brak możliwości uzyskania przez aplikację niezawodności określonej w umowie dotyczącej poziomu usług).

Obszary ryzyka związane z poszczególnymi atrybutami jakościowymi oprogramowania zostały opisane w odpowiednich rozdziałach niniejszego sylabusa.

1.3 Ocena ryzyka

Celem identyfikacji ryzyka jest znalezienie jak największej liczby czynników ryzyka dotyczących danego projektu, przedmiotem oceny ryzyka jest natomiast analiza tych czynników zmierzająca do ich sklasyfikowania i ustalenia prawdopodobieństwa wystąpienia i wpływu każdego z nich.

Ustalenie poziomu ryzyka obejmuje z reguły ocenę prawdopodobieństwa wystąpienia danego ryzyka i jego wpływu w przypadku wystąpienia. Prawdopodobieństwo wystąpienia zwykle jest rozumiane jako prawdopodobieństwo, że dany potencjalny problem istnieje w testowanym systemie.

Wkład technicznego analityka testowego powinien polegać na wyszukiwaniu i ocenie czynników ryzyka technicznego, a wkład analityka testowego — na ocenie potencjalnego wpływu biznesowego wystąpienia danego problemu.

Wśród ogólnych czynników, które zwykle należy uwzględnić, znajdują się:

- złożoność technologii,
- złożoność struktury kodu,
- konflikty między interesariuszami dotyczące wymagań technicznych,
- problemy w komunikacji wynikające z rozproszenia geograficznego jednostek organizacyjnych odpowiedzialnych za tworzenie oprogramowania,
- narzędzia i technologia,
- presja czasowa, ograniczone zasoby i naciski ze strony kierownictwa,
- brak wcześniejszego zapewnienia jakości,
- duża ilość zmian wymagań technicznych,
- duża liczba znalezionych defektów związanych z technicznymi atrybutami jakościowymi,
- problemy techniczne związane z interfejsami i integracją.

Na podstawie dostępnych informacji o ryzyku techniczny analityk testowy wyznacza poziomy ryzyka zgodnie z wytycznymi otrzymanymi od kierownika testów. Kierownik testów może, na przykład, określić, że czynniki ryzyka należy klasyfikować za pomocą wartości od 1 do 10, przy czym wartość 1 oznacza największe ryzyko.

1.4 Łagodzenie ryzyka

W czasie trwania projektu techniczny analityk testowy wpływa na sposób, w jaki w procesie testowania uwzględniane są zidentyfikowane czynniki ryzyka. Obejmuje to zwykle następujące elementy:

- Ograniczanie ryzyka poprzez wykonywanie najważniejszych testów i zastosowanie odpowiednich działań związanych z łagodzeniem i ograniczaniem skutków ryzyka, określonych w strategii testów i planie testów.
- Ocena ryzyka w oparciu o dodatkowe informacje zgromadzone w toku projektu oraz wykorzystanie takich informacji do wdrożenia działań związanych z łagodzeniem ryzyka, zmierzających do zmniejszenia prawdopodobieństwa wystąpienia oraz ograniczenia wpływu uprzednio zidentyfikowanych i przeanalizowanych czynników ryzyka.

2. Testowanie w oparciu o strukturę — 225 minut

Słowa kluczowe

warunek atomowy, testowanie warunków, testowanie przepływu sterowania, testowanie warunków i decyzji, testowanie wielu warunków, testowanie ścieżek, zwarcie, testowanie instrukcji, technika oparta o strukturę.

Cele nauczania dotyczące testowania w oparciu o strukturę

2.2 Testowanie warunków

TTA-2.2.1 (K2) Kandydat zna sposoby uzyskania pokrycia warunków i przyczyny, dla których może się z nim wiązać mniej rygorystyczne testowanie niż w przypadku pokrycia decyzji.

2.3 Testowanie warunków i decyzji

TTA-2.3.1 (K3) Kandydat potrafi zaprojektować przypadki testowe korzystając z techniki projektowania testów „testowanie warunków i decyzji” w celu uzyskania zdefiniowanego poziomu pokrycia.

2.4 Testowanie zmodyfikowanego pokrycia warunków i decyzji (ZPWD)

TTA-2.4.1 (K3) Kandydat potrafi zaprojektować przypadki testowe korzystając z techniki projektowania testów „zmodyfikowane pokrycie warunków i decyzji” w celu uzyskania zdefiniowanego poziomu pokrycia.

2.5 Testowanie wielokrotne warunków

TTA-2.5.1 (K3) Kandydat potrafi zaprojektować przypadki testowe korzystając z techniki projektowania testów „testowanie wielokrotne warunków” w celu uzyskania zdefiniowanego poziomu pokrycia.

2.6 Testowanie ścieżek

TTA-2.6.1 (K3) Kandydat potrafi zaprojektować przypadki testowe korzystając z techniki projektowania testów „testowanie ścieżek”.

2.7 Testowanie API

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

2.8 Wybór techniki opartej o strukturę

TTA-2.8.1 (K4) Kandydat potrafi wybrać odpowiednią technikę opartą na strukturze zgodnie z daną sytuacją projektową.

2.1 Wprowadzenie

W niniejszym rozdziale w ogólny sposób opisano techniki projektowania testów oparte o strukturę, nazywane również białoskrzynkowymi technikami projektowania testów lub technikami projektowania testów na podstawie kodu. W technikach tego typu do projektowania testów wykorzystywane są kod, dane oraz architektura i/lub przepływ sterowania w systemie. Poszczególne techniki umożliwiają systematyczne tworzenie przypadków testowych i koncentrują się na konkretnych aspektach struktury, które należy uwzględnić. Techniki określają kryteria pokrycia, które muszą zostać zmierzone i odniesione do celów zdefiniowanych w ramach danego projektu lub jednostki organizacyjnej. Uzyskanie pełnego pokrycia nie oznacza, że zbiór testów jest kompletny, tylko że w ramach używanej techniki nie można już opracować dalszych przydatnych testów badanej struktury.

Techniki projektowania testów w oparciu o strukturę przedstawione w niniejszym sylabusie (z wyjątkiem pokrycia warunków) są bardziej rygorystyczne niż techniki pokrycia instrukcji kodu i pokrycia decyzji opisane w sylabusie poziomu podstawowego [ISTQB_FL_SYL].

W niniejszym sylabusie przedstawiono następujące techniki:

- testowanie warunków,
- testowanie warunków i decyzji,
- testowanie pokrycia zmodyfikowanego warunków decyzji (ZPWD),
- testowanie wielu warunków,
- testowanie ścieżek,
- testowanie API.

Cztery techniki wymienione na początku powyższej listy są oparte na predykatach decyzji i, ogólnie rzecz biorąc, służą do wykrywania defektów podobnego rodzaju. Niezależnie od stopnia skomplikowania predykatu, będzie on mieć wartość PRAWDA lub FAŁSZ, a kod obsłuży tylko jedną ze ścieżek. Defekt zostanie wykryty wówczas, gdy zakładana ścieżka nie będzie wybrana ze względu na brak zgodności wartości złożonego predykatu decyzji z wartością oczekiwaną.

W ogólności, pierwsze cztery wymienione techniki charakteryzują się rosnącym stopniem szczegółowości. Wymagają zdefiniowania coraz większej liczby testów w celu uzyskania zamierzonego pokrycia i znalezienia bardziej złożonych instancji defektów tego typu.

Więcej informacji na ten temat można znaleźć w publikacjach [Bath08], [Beizer90], [Beizer95], [Copeland03] i [Koomen06].

2.2 Testowanie warunków

W porównaniu z testowaniem decyzji (gałęzi), w którym brana jest pod uwagę cała decyzja, a wartości PRAWDA oraz FAŁSZ rozpatrywane są w odrębnych przypadkach testowych, w testowaniu warunków rozpatruje się sposób podejmowania decyzji. Każdy predykat decyzji składa się z jednego lub wielu prostych warunków „atomowych”, z których każdy przyjmuje wartość logiczną. Warunki te są łączone za pomocą operatorów logicznych w celu określenia ostatecznego wyniku decyzji. Aby uzyskać ten poziom pokrycia, należy w przypadkach testowych uwzględnić obie wartości każdego z warunków atomowych.

Obszar zastosowania
Ze względu na opisane poniżej trudności, testowanie warunków jest techniką interesującą właściwie tylko z abstrakcyjnego punktu widzenia, jednak należy się z nią zapoznać w celu uzyskania wyższego poziomu pokrycia w innych opartych na niej technikach.
Ograniczenia/trudności
Jeśli w decyzji występują co najmniej dwa warunki atomowe, niewłaściwy wybór danych testowych w trakcie projektowania testów pozwala uzyskać pokrycie warunków, ale bez uzyskania pokrycia decyzji. Załóżmy na przykład, że predykat decyzji to „A ORAZ B".

	A	B	A ORAZ B
Test 1	FAŁSZ	PRAWDA	FAŁSZ
Test 2	PRAWDA	FAŁSZ	FAŁSZ

Aby uzyskać 100% pokrycia warunków, należy wykonać dwa testy przedstawione w tabeli. Testy te wprawdzie pozwalają uzyskać 100% pokrycia warunków, nie umożliwiają jednak osiągnięcia pokrycia decyzji, ponieważ w obu przypadkach wartość predykatu to FAŁSZ.

Gdy decyzja składa się z jednego warunku atomowego, testowanie warunków jest równoważne testowaniu decyzji.

2.3 Testowanie warunków i decyzji

Testowanie warunków i decyzji zakłada konieczność uzyskania pokrycia warunków (patrz powyżej), ale wymaga także uzyskania pokrycia decyzji (patrz sylabus poziomu podstawowego [ISTQB_FL_SYL]). Efektem przemyślanego wyboru wartości danych testowych dla warunków atomowych może być osiągnięcie tego poziomu pokrycia bez konieczności tworzenia dodatkowych przypadków testowych (w odniesieniu do przypadków niezbędnych do uzyskania pokrycia warunków).
W poniższym przykładzie testowany jest ten sam predykat decyzji co powyżej („A ORAZ B"). Pokrycie warunków decyzji można osiągnąć za pomocą tej samej liczby testów, jednak przy wyborze innych wartości testowych.

	A	B	A ORAZ B
Test 1	PRAWDA	PRAWDA	PRAWDA
Test 2	FAŁSZ	FAŁSZ	FAŁSZ

Technika ta może zatem okazać się bardziej efektywna.

Obszar zastosowania
Należy wziąć pod uwagę ten poziom pokrycia wówczas, gdy testowany kod jest istotny, ale nie ma znaczenia krytycznego.

Ograniczenia/trudności
Zastosowanie tej techniki może okazać się problematyczne, jeśli w projekcie występują ograniczenia czasowe, ponieważ wymagane jest utworzenie większej liczby przypadków testowych niż w testowaniu na poziomie decyzji.

2.4 Testowanie zmodyfikowane pokrycia warunków i decyzji (ZPWD)

Ta technika odznacza się wyższym poziomem pokrycia sterowania. Jeśli występuje N unikalnych warunków atomowych, testowanie ZPWD wymaga zwykle wykonania N+1 unikalnych przypadków testowych. Testowanie ZPWD pozwala jednocześnie uzyskać pokrycie warunków i decyzji przy spełnieniu następujących założeń:

1. Istnieje co najmniej jeden test, w którym wynik decyzji ulegnie zmianie, jeśli wartość warunku atomowego X to PRAWDA.
2. Istnieje co najmniej jeden test, w którym wynik decyzji ulegnie zmianie, jeśli wartość warunku atomowego X to FAŁSZ.
3. Dla każdego z warunków atomowych istnieją testy spełniające wymagania 1 i 2.

	A	B	C	(A LUB B) ORAZ C
Test 1	PRAWDA	FAŁSZ	PRAWDA	PRAWDA
Test 2	FAŁSZ	PRAWDA	PRAWDA	PRAWDA
Test 3	FAŁSZ	FAŁSZ	PRAWDA	FAŁSZ
Test 4	PRAWDA	FAŁSZ	FAŁSZ	FAŁSZ

W powyższym przykładzie uzyskano pokrycie decyzji (wynik predykatu decyzji to zarówno PRAWDA, jak i FAŁSZ) oraz pokrycie warunków (A, B i C mają wartości zarówno PRAWDA, jak i FAŁSZ).

W teście 1 warunek A ma wartość PRAWDA, a wynik decyzji to PRAWDA. Jeśli zmienimy A na FAŁSZ (jak w teście 3, bez zmiany innych wartości), wynik zmienia się na FAŁSZ.
W teście 2 warunek B ma wartość PRAWDA, a wynik decyzji to PRAWDA. Jeśli zmienimy B na FAŁSZ (jak w teście 3, bez zmiany innych wartości), wynik zmienia się na FAŁSZ.
W teście 1 warunek C ma wartość PRAWDA, a wynik decyzji to PRAWDA. Jeśli zmienimy C na FAŁSZ (jak w teście 4, bez zmiany innych wartości), wynik zmienia się na FAŁSZ.

Obszar zastosowania
Ta technika jest powszechnie stosowana w testach oprogramowania w przemyśle lotniczym, a także w wielu innych systemach o kluczowym znaczeniu dla bezpieczeństwa. Należy z niej skorzystać w przypadku tego typu oprogramowania, gdy ewentualna awaria może stać się przyczyną katastrofy.

Ograniczenia/trudności
Uzyskanie pokrycia ZPWD może okazać się skomplikowane, jeśli predykat zawiera wiele wystąpień tego samego warunku atomowego. Warunek taki nazywany jest „powiązanym”. W konkretnej instrukcji kodu zawierającej decyzję zmiana wartości powiązanego elementu w taki sposób, aby jedynie z tego powodu zmienił się wynik całej decyzji, może okazać się niemożliwa. Jedną z metod rozwiązania tego problemu jest przyjęcie założenia, że na poziomie ZPWD musimy testować wyłącznie niepowiązane warunki atomowe. Inny sposób polega na indywidualnym analizowaniu każdego przypadku występowania decyzji, w której pojawiają się elementy tego typu.

Niektóre języki programowania i interpretery zaprojektowano w taki sposób, aby podczas wyliczania wartości złożonego wyrażenia decyzyjnego w kodzie następowało tzw. zwarcie. Oznacza to, że w uruchomionym kodzie całe wyrażenie nie musi być obliczane, jeśli końcowy wynik obliczeń można określić już po ustaleniu wartości części wyrażenia. Na przykład, w trakcie obliczania wartości decyzji „A ORAZ B” nie ma powodu obliczać wartości warunku B, jeśli wartością warunku A jest FAŁSZ. Żadna z dostępnych wartości B nie jest w stanie zmienić wartości końcowej, zatem da się skrócić czas wykonywania kodu, rezygnując z obliczania wartości tego warunku. Zwarcie może mieć wpływ na zdolność do uzyskania pokrycia ZPWD, ponieważ niektóre wymagane

testy mogą nie zostać nigdy wykonane.

2.5 Testowanie wielokrotne warunków

W rzadkich przypadkach może okazać się konieczne przetestowanie wszystkich kombinacji wartości, które mogą pojawić się w decyzji. Taki gruntowny poziom testowania nazywany jest pokryciem wielokrotnym warunków. Liczba wymaganych testów zależy od liczby warunków atomowych w wyrażeniu decyzyjnym i wynosi 2^n, gdzie n jest liczbą niepowiązanych warunków atomowych. W przykładzie podanym w poprzednim punkcie, w celu uzyskania pokrycia warunków wielokrotnych wymagane jest wykonanie następujących testów:

	A	B	C	(A LUB B) ORAZ C
Test 1	PRAWDA	PRAWDA	PRAWDA	PRAWDA
Test 2	PRAWDA	PRAWDA	FAŁSZ	FAŁSZ
Test 3	PRAWDA	FAŁSZ	PRAWDA	PRAWDA
Test 4	PRAWDA	FAŁSZ	FAŁSZ	FAŁSZ
Test 5	FAŁSZ	PRAWDA	PRAWDA	PRAWDA
Test 6	FAŁSZ	PRAWDA	FAŁSZ	FAŁSZ
Test 7	FAŁSZ	FAŁSZ	PRAWDA	FAŁSZ
Test 8	FAŁSZ	FAŁSZ	FAŁSZ	FAŁSZ

Jeśli w używanym języku programowania występuje mechanizm zwarcia, faktyczna liczba przypadków testowych zwykle ulega zmniejszeniu, w zależności od kolejności i zgrupowania operacji logicznych wykonywanych na warunkach atomowych.

Obszar zastosowania
Ta technika była tradycyjnie stosowana do testowania oprogramowania wbudowanego, które powinno działać w niezawodny, bezawaryjny sposób przez długi okres czasu (np. zakładano, że centrale telefoniczne mają funkcjonować przez 30 lat). W przypadku większości newralgicznych aplikacji ten rodzaj testowania prawdopodobnie zostanie zastąpiony testowaniem ZPWD.

Ograniczenia/trudności
Ponieważ liczba przypadków testowych wynika bezpośrednio z tablicy prawdy zawierającej wszystkie warunki atomowe, łatwo określić ten poziom pokrycia. Liczba wymaganych przypadków może być jednak bardzo duża, dlatego w większości sytuacji wystarczające jest zastosowanie pokrycia ZPWD.

2.6 Testowanie ścieżek

Testowanie ścieżek polega na zidentyfikowaniu ścieżek w kodzie, a następnie na opracowaniu testów w celu ich pokrycia. Teoretycznie rzecz biorąc, należałoby przetestować wszystkie unikatowe ścieżki w ramach systemu. Jednak w każdym nietrywialnym systemie liczba przypadków testowych może okazać się ogromna ze względu na występowanie struktur zapętlonych.

Pomijając kwestię nieskończonych pętli, użycie testowania ścieżek w pewnym zakresie jest realistyczne. W publikacji [Beizer90] znalazło się zalecenie, aby w ramach tej techniki tworzyć testy przechodzące przez różne ścieżki w module, od punktu wejścia do punktu wyjścia. Autor sugeruje uproszczenie tego potencjalnie złożonego zadania dzięki systematycznemu podejściu, polegającemu na zastosowaniu następującej procedury:

1. Wybierz jako pierwszą najprostszą sensowną z funkcjonalnego punktu widzenia ścieżkę od punktu wejścia do punktu wyjścia.

2. Wybieraj każdą kolejną, dodatkową ścieżkę tak, aby różniła się w niewielkim stopniu od poprzedniej. W kolejnych testach próbuj zmieniać tylko jedną różniącą się od poprzedniej gałąź w ścieżce. Jeśli to możliwe, wybieraj raczej krótsze ścieżki. Staraj się wybierać ścieżki, które mają sens z funkcjonalnego punktu widzenia.
3. Wybieraj ścieżki, które nie mają sensu z funkcjonalnego punktu widzenia jedynie wówczas, gdy jest to wymagane do uzyskania pokrycia. Beizer zwraca uwagę, że takie ścieżki mogą być nadmiarowe i należy zastanowić się wówczas nad ich uwzględnieniem.
4. Dokonując wyboru ścieżek, kieruj się intuicją (np. zastanów się, które ścieżki mają największe prawdopodobieństwo wykonania).

Należy zwrócić uwagę, że przy użyciu tej strategii niektóre segmenty ścieżek mogą być wykonywane więcej niż jeden raz. Kluczowym elementem strategii jest przetestowanie wszystkich możliwych gałęzi kodu przynajmniej raz, a jeśli to możliwe – wielokrotnie.

Obszar zastosowania
Opisane powyżej częściowe testowanie ścieżek jest często stosowane w przypadku oprogramowania o kluczowym znaczeniu dla bezpieczeństwa. Stanowi dobre uzupełnienie innych metod przedstawionych w niniejszym rozdziale, ponieważ dotyczy ścieżek w oprogramowaniu, a nie tylko sposobu podejmowania decyzji.

Ograniczenia/trudności
Wprawdzie w celu określenia ścieżek można skorzystać z diagramu przepływu sterowania, w praktyce to podejście wymaga zastosowania narzędzia, które znajdzie takie ścieżki w przypadku skomplikowanych modułów.

Pokrycie
Utworzenie testów wystarczających do pokrycia wszystkich ścieżek (z wyjątkiem pętli) gwarantuje uzyskanie pokrycia instrukcji kodu i pokrycia gałęzi. Testowanie ścieżek pozwala zapewnić większą dokładność testowania niż pokrycie gałęzi, a liczba testów wzrasta jedynie w stosunkowo niewielkim stopniu. [NIST 96]

2.7 Testowanie API

Interfejs programowania aplikacji (API) to kod umożliwiający komunikację między różnymi procesami, programami i/lub systemami. API są często stosowane do obsługi relacji klient/serwer, gdy jeden z procesów udostępnia pewien rodzaj funkcjonalności innym procesom.

W pewnym zakresie testowanie API przypomina testowanie graficznego interfejsu użytkownika (GUI). Technika koncentruje się na analizowaniu wartości wejściowych i zwracanych danych.

Testowanie negatywne jest często istotnym elementem badania API. Programiści, którzy wykorzystują API do uzyskiwania dostępu do usług zewnętrznych w stosunku do tworzonego przez nich kodu, mogą podejmować próby użycia API w sposób niezgodny z ich przeznaczeniem. Oznacza to konieczność wprowadzenia odpornych mechanizmów obsługi błędów, zapobiegających nieprawidłowemu działaniu. Może być niezbędne przeprowadzenie testowania kombinatorycznego różnych interfejsów, ponieważ API są często używane wspólnie, a każdy z nich może zawierać różne parametry, których wartości da się połączyć na różne sposoby.

API są często luźno powiązane, co zwiększa prawdopodobieństwo wystąpienia utraconych transakcji i zakłóceń

czasowych. Niezbędne jest zatem dokładne przetestowanie mechanizmów odtwarzania i ponawiania. Organizacja, która udostępnia API, musi zagwarantować bardzo wysoką dostępność wszystkich usług. Często wymaga to rygorystycznego testowania niezawodności przez dostawcę usług oraz zespół obsługujący infrastrukturę.

Obszar zastosowania
Testowanie API staje się coraz ważniejszą techniką, ponieważ pojawia się coraz więcej systemów rozproszonych i korzystających ze zdalnego przetwarzania w celu przekierowania części zadań na inne procesory. Można tu wspomnieć na przykład o wywołaniach systemów operacyjnych, architekturach zorientowanych na usługi (SOA), zdalnych wywołaniach procedur (RPC) i właściwie wszystkich innych rodzajach aplikacji rozproszonych. Testowanie interfejsów API jest szczególnie przydatne w przypadku systemów złożonych z podsystemów.

Ograniczenia/trudności
Bezpośrednie testowanie API na ogół wymaga zastosowania specjalistycznych narzędzi przez technicznego analityka testowego. Ponieważ zwykle nie istnieje interfejs graficzny bezpośrednio powiązany z API, do skonfigurowania początkowego środowiska, serializacji danych, wywołania funkcji API i określenia wyniku wykorzystuje się odpowiednie narzędzia.

Pokrycie
Testowanie API to typ prowadzenia testów, a nie opis konkretnego poziomu pokrycia. Test interfejsu API powinien obejmować co najmniej wszystkie wywołania funkcji API oraz uwzględniać wszystkie poprawne wartości i sensowne wartości niepoprawne.

Typy defektów
W trakcie testów interfejsów API można wykryć różne rodzaje błędów. Są to między innymi problemy dotyczące obsługi danych, zależności czasowych, utraty transakcji i duplikowania transakcji.

2.8 Wybór techniki opartej o strukturę

Kontekst testowanego systemu określa poziom pokrycia testowania w oparciu o strukturę, który należy osiągnąć. Im bardziej newralgiczny jest dany system, tym wyższy stopień pokrycia jest wymagany. W ogólności, osiągnięcie wyższego stopnia pokrycia wiąże się z koniecznością poświęcenia większej ilości czasu i zasobów.

Czasami wymagany poziom pokrycia jest określony przez odpowiednie standardy dotyczące danego typu oprogramowania. Na przykład, jeśli oprogramowanie ma być stosowane w awionice, musi być zgodne ze standardem DO-178B (w Europie: ED-12B). Standard ten określa następujące poziomy warunków awarii:

A. Katastroficzny: awaria może spowodować brak działania podstawowych funkcji niezbędnych do bezpiecznego kontynuowania lotu lub lądowania.
B. Niebezpieczny: awaria może mieć poważny, negatywny wpływ na bezpieczeństwo i funkcjonowanie systemu.
C. Wysoki: awaria jest istotna, ale mniej poważna niż w przypadku poziomu A lub B.
D. Niski: awaria jest zauważalna, ale ma mniejszy wpływ niż w przypadku poziomu C.
E. Bez konsekwencji: awaria nie ma wpływu na bezpieczeństwo.

Jeśli system ma kategorię A, musi zostać przetestowany z pokryciem ZPWD. Jeśli został sklasyfikowany na poziomie B, należy uzyskać w testach pokrycie na poziomie decyzji, a uzyskanie pokrycia ZPWD jest opcjonalne. Dla poziomu C wymagane jest co najmniej pokrycie instrukcji kodu.

Podobnie, norma IEC-61508 jest międzynarodowym standardem dotyczącym bezpieczeństwa funkcjonalnego programowalnych elektronicznych systemów związanych z bezpieczeństwem. Standard został przyjęty w wielu dziedzinach, m.in. w przemyśle motoryzacyjnym i kolejnictwie, w procesach produkcyjnych, elektrowniach jądrowych i w przemyśle maszynowym. Krytyczność definiuje się przy użyciu skali poziomów nienaruszalności bezpieczeństwa (ang. *Safety Integrity Level*; SIL), przy czym 1 oznacza najniższą, a 4 najwyższy jego poziom. Zalecane są następujące poziomy pokrycia:

1. Zalecane pokrycie instrukcji kodu i gałęzi.
2. Zdecydowanie zalecane pokrycie instrukcji kodu oraz zalecane pokrycie gałęzi.
3. Zdecydowanie zalecane pokrycie instrukcji kodu i gałęzi.
4. Zdecydowanie zalecane pokrycie ZPWD.

W nowoczesnych środowiskach rzadko się zdarza, by całe przetwarzanie odbywało się w obrębie jednego systemu. Należy przetestować interfejsy API zawsze wtedy, gdy część przetwarzania ma być realizowana zdalnie. Zakres działań dotyczących testowania API powinien zależeć od tego, jak newralgiczny jest dany system.
Jak zwykle, techniczny analityk testowy, który chce wybrać metody używane w testach, musi wziąć pod uwagę kontekst testowanego systemu.

3. Techniki analityczne — 255 minut

Słowa kluczowe
analiza przepływu sterowania, złożoność cyklomatyczna, analiza przepływu danych, para definicja-użycie, analiza dynamiczna, wyciek pamięci, testowanie integracji parami, testowanie integracji sąsiadująco, analiza statyczna, dziki wskaźnik

Cele nauczania dotyczące technik analitycznych

3.2 Analiza statyczna

TTA-3.2.1 (K3) Kandydat potrafi zastosować analizę przepływu sterowania w celu wykrycia ewentualnych anomalii związanych z tym przepływem.
TTA-3.2.2 (K3) Kandydat potrafi zastosować analizę przepływu danych w celu wykrycia ewentualnych anomalii związanych z tym przepływem.
TTA-3.2.3 (K3) Kandydat potrafi zaproponować sposoby zwiększenia pielęgnowalności kodu za pomocą analizy statycznej.
TTA-3.2.4 (K2) Kandydat potrafi wyjaśnić zasady użycia grafów wywołań do określenia strategii testowania integracyjnego.

3.3 Analiza dynamiczna

TTA-3.3.1 (K3) Kandydat potrafi określić cele, które można osiągnąć z wykorzystaniem analizy dynamicznej.

3.1 Wprowadzenie

Istnieją dwa rodzaje analizy: analiza statyczna i analiza dynamiczna.

Analiza statyczna (punkt 3.2) obejmuje testowanie analityczne odbywające się bez uruchamiania oprogramowania. Badanie oprogramowania jest wykonywane przez człowieka lub przez narzędzie. Celem jest sprawdzenie, czy przetwarzanie będzie odbywać się w prawidłowy sposób po uruchomieniu produktu. Statyczny obraz oprogramowania umożliwia przeprowadzenie szczegółowej analizy bez konieczności tworzenia danych i spełniania warunków wstępnych niezbędnych do realizacji scenariusza.

Należy tu wspomnieć, że różne rodzaje przeglądów istotne z punktu widzenia technicznego analityka testowego zostały opisane w rozdziale 5.

Analiza dynamiczna (punkt 3.3) wymaga faktycznego wykonania kodu i jest stosowana do wykrywania usterek, które łatwiej znaleźć, jeśli dany kod zostanie uruchomiony (np. wycieków pamięci). Podobnie jak w przypadku analizy statycznej, analiza dynamiczna może opierać się na zastosowaniu narzędzi lub monitorowaniu wykonywanego systemu przez użytkownika pod kątem występowania takich objawów, jak szybki przyrost używanej pamięci.

3.2 Analiza statyczna

Celem analizy statycznej jest wykrycie rzeczywistych i potencjalnych usterek w kodzie i architekturze systemu oraz zwiększenie pielęgnowalności kodu i architektury. Analiza statyczna jest zwykle wykonywana za pomocą narzędzi.

3.2.1 Analiza przepływu sterowania

Analiza przepływu sterowania to technika statyczna, w której przepływ sterowania w programie jest analizowany przy użyciu diagramu przepływu sterowania lub odpowiedniego narzędzia. Istnieją różne rodzaje anomalii, które można wykryć w systemie za pomocą tej techniki. To m.in. źle zaprojektowane pętle (np. pętle z wieloma punktami wejścia), niejednoznaczne obiekty docelowe wywołań funkcji w pewnych językach (np. Scheme) oraz niepoprawna kolejność wykonywanych operacji.

Jednym z najczęstszych powodów stosowania analizy przepływu sterowania jest określenie złożoności cyklomatycznej. Złożoność cyklomatyczna to dodatnia liczba całkowita określająca liczbę niezależnych ścieżek w silnie spójnym grafie, przy czym pętle i iteracje traktowane są jako pojedyncze przejście. Każda niezależna ścieżka, od punktu wejścia do punktu wyjścia, reprezentuje unikalną ścieżkę w module. Każda z takich ścieżek powinna zostać przetestowana.

Wartość złożoności cyklomatycznej jest na ogół wykorzystywana do określenia ogólnej złożoności modułu. Według teorii przedstawionej przez Thomasa McCabe'a [McCabe 76], im bardziej złożony jest system, tym więcej zawiera defektów i tym trudniejsza jest jego pielęgnacja. W wielu pracach opublikowanych w kolejnych latach zauważono tego typu korelację między złożonością i liczbą defektów. Według rekomendacji National Institute of Standards and Technology (NIST), najwyższa dopuszczalna wartość złożoności wynosi 10. Moduł, który odznacza się większym stopniem złożoności, prawdopodobnie powinien zostać podzielony na wiele modułów.

3.2.2 Analiza przepływu danych

Analiza przepływu danych obejmuje wiele technik, które koncentrują się na gromadzeniu informacji o różnych zmiennych związanych z systemem. Szczegółowo bada się cykl życia zmiennych, tj. miejsca ich deklaracji, definicji, odczytu, obliczania wartości i niszczenia, ponieważ anomalie mogą wystąpić w każdej z takich operacji.

Jedną z często używanych technik jest notacja definicja-użycie, w której cykl życia każdej zmiennej dzieli się na trzy atomowe działania:

- d (define): gdy zmienna jest deklarowana, definiowana lub inicjowana,
- u (use): gdy zmienna jest używana lub odczytywana w obliczeniach lub w predykacie,
- k (kill): gdy zmienna jest usuwana lub niszczona albo przestaje być dostępna w danym zasięgu.

Trzy powyższe działania atomowe łączone są w pary (tzw. „pary definicja-użycie”), które opisują przepływ danych. Na przykład, ścieżka „du” reprezentuje fragment kodu, w którym zmienna danych jest definiowana, a następnie używana.

Anomalie dotyczące danych to, na przykład, wykonanie poprawnego działania na zmiennej w niewłaściwym momencie lub wykonanie niepoprawnego działania na danych w zmiennej. Przykłady:

- przypisanie niepoprawnej wartości zmiennej,
- brak przypisania wartości zmiennej przed jej użyciem,
- wybór nieprawidłowej ścieżki z powodu niepoprawnej wartości predykatu sterującego,
- próba użycia wartości po zniszczeniu zmiennej,
- odwołanie do zmiennej znajdującej się poza zasięgiem,
- zadeklarowanie i zniszczenie zmiennej bez jej użycia,
- zmiana definicji zmiennej przed jej użyciem,
- brak usunięcia dynamicznie zaalokowanej zmiennej (a w konsekwencji możliwy wyciek pamięci),
- modyfikacja wartości powodująca nieoczekiwane skutki uboczne (np. wynikające ze zmiany wartości zmiennej globalnej bez uwzględniania wszystkich miejsc, w których użyto tej zmiennej).

Używany język programowania może mieć wpływ na zasady stosowane w trakcie analizy przepływu danych. Niektóre języki programowania mogą umożliwiać programistom wykonywanie pewnych działań na zmiennych, które nie są niedozwolone, jednak w pewnych okolicznościach mogą spowodować pracę systemu niezgodną z oczekiwaniami. Na przykład, na jednej ze ścieżek zmienna może być zdefiniowana dwukrotnie i w ogóle nie używana. W trakcie analizy przepływu danych takie sytuacje zostaną sklasyfikowane jako „podejrzane”. Wprawdzie mogą to być dozwolone operacje przypisania wartości zmiennej, jednak w przyszłości są w stanie spowodować problemy związane z pielęgnowalnością kodu.

Testowanie przepływu danych „wykorzystuje diagram przepływu sterowania do wykrycia niespodziewanych sytuacji, w jakich mogą znaleźć się dane” [Beizer90], zatem pozwala wykryć inne rodzaje błędów niż testowanie przepływu sterowania. Techniczny analityk testowy powinien uwzględnić tę technikę podczas planowania testów, ponieważ wiele defektów może doprowadzić do nieregularnych awarii, które trudno wykryć podczas testowania dynamicznego.

Analiza przepływu danych jest jednak techniką statyczną; można w niej pominąć pewne problemy, które dotyczą danych w czasie wykonywania. Na przykład, statyczna zmienna danych może zawierać wskaźnik do

dynamicznie tworzonej tablicy, która istnieje jedynie w czasie wykonywania. W przypadku korzystania z wielu procesorów i wielozadaniowości z wywłaszczaniem mogą pojawić się tzw. wyścigi, które nie zostaną wykryte w ramach analizy przepływu danych i przepływu sterowania.

3.2.3 Analiza statyczna jako sposób poprawy pielęgnowalności

Istnieją różne sposoby zastosowania analizy statycznej do poprawy pielęgnowalności kodu, architektury i stron internetowych.

Źle napisany, nieskomentowany i nieustrukturyzowany kod jest trudniejszy w utrzymaniu. Odnalezienie i przeanalizowanie defektów w kodzie wymaga od programistów większego nakładu pracy, a dodanie nowej funkcji może wiązać się z wprowadzeniem do kodu kolejnych defektów.

Analiza statyczna wspomagana odpowiednim narzędziem pozwala poprawić pielęgnowalność kodu dzięki weryfikacji zgodności ze standardami i wytycznymi dotyczącymi kodowania. Te standardy i wytyczne opisują wymagane procedury tworzenia kodu, m.in. konwencje nazewnictwa, sposób komentowania, zasady tworzenia wcięć w tekście i podział kodu na moduły. Narzędzia do analizy statycznej zwykle zgłaszają ostrzeżenia, a nie błędy, w tym także w sytuacji, gdy kod jest poprawny pod względem składniowym.

Modułowa konstrukcja zwykle pozwala poprawić jakość kodu. Narzędzia do analizy statycznej ułatwiają tworzenie kodu modułowego w następujący sposób:

- Wyszukują powtarzający się kod. Takie fragmenty kodu potencjalnie nadają się do refaktoryzacji i przekształcenia w moduły (chociaż narzut związany z wywołaniami modułów w czasie wykonania może stanowić problem w przypadku systemów czasu rzeczywistego).
- Obliczają metryki, które są przydatnymi wskaźnikami podziału kodu na moduły, między innymi zależność i spójność kodu. System odznaczający się dobrą pielęgnowalnością na ogół ma niski wskaźnik zależności (czyli stopnia powiązania modułów z innymi modułami w czasie wykonywania) i wysoki wskaźnik spójności (określającej stopień, w jakim moduły są samodzielne i przeznaczone do realizacji jednego zadania).
- Wskazują w kodzie obiektowym miejsca, w których klasy nadrzędne są widoczne dla obiektów pochodnych w zbyt dużym lub zbyt małym zakresie.
- Wskazują obszary kodu lub architektury odznaczające się dużą złożonością strukturalną, co zwykle wiąże się ze zmniejszeniem stopnia pielęgnowalności i wzrostem prawdopodobieństwa występowania usterek. Wytyczne mogą określać dopuszczalne poziomy złożoności cyklomatycznej (patrz punkt 3.2.1), tak aby kod był tworzony w sposób modułowy w celu zwiększenia łatwości jego utrzymania i w celu zapobiegania powstawaniu defektów. Fragmenty kodu o dużej złożoności cyklomatycznej potencjalnie nadają się do podziału na moduły.

Narzędzia do analizy statycznej mogą również służyć do pielęgnowania serwisu internetowego. W takim wypadku należy sprawdzać, czy drzewo struktury serwisu jest zrównoważone i czy nie występują nieprawidłowości, które mogą prowadzić do:

- zwiększenia stopnia trudności zadań związanych z testowaniem,
- zwiększenia nakładu pracy związanego z pielęgnowaniem serwisu,
- utrudnień w nawigacji dla użytkowników.

3.2.4 Grafy wywołań

Grafy wywołań stanowią statyczną reprezentację złożoności komunikacji. Są to grafy skierowane, w których węzły oznaczają jednostki programów, a krawędzie -- komunikację między tymi jednostkami.

Grafy wywołań mogą znaleźć zastosowanie w testach jednostkowych, w których istnieją wzajemne wywołania różnych funkcji i modułów, w testach integracyjnych i systemowych, w których istnieją wywołania między modułami, a także w testach integracji systemów, w których istnieją wywołania między systemami.

Grafy wywołań można stosować w następujących celach:

- projektowanie testów wywołujących konkretny moduł lub system,
- określanie liczby miejsc w oprogramowaniu, w których następuje wywołanie modułu lub systemu,
- analiza struktury kodu i architektury systemu,
- określenie zaleceń dotyczących integracji (integracja parami lub sąsiadująco); opcje te zostały szerzej opisane w dalszej części rozdziału.

W sylabusie poziomu podstawowego [ISTQB_FL_SYL] przedstawiono dwie kategorie testowania integracyjnego: przyrostowe (zstępujące, wstępujące itp.) oraz nieprzyrostowe („wielkiego wybuchu”). Stwierdzono tam, że preferowane są metody przyrostowe, ponieważ kod jest analizowany w przyrostach, co ułatwia wykrywanie błędów, ponieważ ilość uwzględnianego każdorazowo kodu jest ograniczona.
W niniejszym sylabusie dla poziomu zaawansowanego prezentowane są trzy dodatkowe metody nieprzyrostowe wykorzystujące grafy wywołań. Mogą one okazać się bardziej przydatne od metod przyrostowych, w których do realizacji testów prawdopodobnie niezbędne będzie budowanie dodatkowych wersji i tworzenie pomocniczego kodu nieprzeznaczonego do dystrybucji. Te metody to:

- Testowanie integracji parami (nie należy mylić tego pojęcia z czarnoskrzynkową techniką testowania o nazwie „testowanie sposobem par”): forma testowania integracyjnego, która uwzględnia pary współpracujących modułów, zgodnie z ich rozmieszczeniem w grafie wywołań. Metoda ta pozwala ograniczyć liczbę budowanych wersji tylko w niewielkim stopniu, jednak umożliwia zmniejszenie rozmiaru niezbędnego kodu tzw. jarzma testowego.
- Testowanie integracji sąsiadująco: forma testowania integracyjnego, w której wszystkie węzły połączone z danym węzłem uczestniczą w testowaniu integracyjnym. Wszystkie poprzedniki i następniki danego węzła w grafie wywołań są uwzględniane w testach.
- Podejście McCabe’a oparte na predykatach projektowych korzysta z teorii złożoności cyklomatycznej w odniesieniu do grafu wywołań modułów. Wymaga ono skonstruowania grafu wywołań, w którym zostaną uwzględnione różne sposoby wzajemnych wywołań modułów, a w szczególności:
  - wywołania bezwarunkowe: wywołania jednego modułu przez inny moduł, które zawsze mają miejsce;
  - wywołania warunkowe: wywołania jednego modułu przez inny moduł, które występują tylko w pewnych sytuacjach;
  - wykluczające się wzajemnie wywołania warunkowe: moduł wywołuje dokładnie jeden spośród pewnej liczby innych modułów;
  - wywołania iteracyjne: moduł wywołuje inny moduł co najmniej raz, choć może również wywoływać go wielokrotnie;
  - iteracyjne wywołania warunkowe: moduł może wywołać inny moduł wiele razy, jednak może nie

wywołać go w ogóle.

Po utworzeniu grafu wywołań oblicza się złożoność integracji i tworzy testy celem pokrycia grafu.

W pracy [Jorgensen07] można znaleźć więcej informacji na temat używania grafów wywołań i testowania integracyjnego parami.

3.3 Analiza dynamiczna

3.3.1 Przegląd

Analiza dynamiczna służy do wykrywania awarii, których objawy nie zawsze są natychmiast widoczne. Przykładem mogą być wycieki pamięci. Wprawdzie analiza statyczna pozwala niekiedy wykryć ryzyko ich wystąpienia (poprzez znalezienie kodu, który przydziela pamięć, ale nigdy jej nie zwalnia), natomiast to dzięki analizie dynamicznej wycieki takie są łatwe do rozpoznania.
Awarie, których nie da się natychmiast odtworzyć, mogą w istotny sposób zwiększyć pracochłonność procesu testowania oraz utrudnić wprowadzenie oprogramowania do sprzedaży lub jego eksploatację. Wśród przyczyn takich awarii można wymienić wycieki pamięci, nieprawidłowe stosowanie wskaźników oraz inne nieprawidłowości (np. uszkodzenie stosu systemowego) [Kaner02]. Takie awarie mogą powodować stopniowy spadek wydajności systemu, a nawet jego załamanie. W strategiach testowania należy zatem uwzględnić ryzyko związane z tego rodzaju błędami i — w uzasadnionych przypadkach — przewidzieć również przeprowadzenie analizy dynamicznej w celu ograniczenia tego ryzyka (zwykle przy użyciu odpowiednich narzędzi). Awarie tego rodzaju są często najkosztowniejsze do zlokalizowania i skorygowania, dlatego zaleca się przeprowadzenie analizy dynamicznej już na wczesnym etapie projektu.
Analizę dynamiczną można stosować w celu:

- zapobiegania wystąpieniu awarii poprzez wykrywanie dzikich wskaźników i przypadków utraty pamięci systemowej;
- analizowania trudnych do odtworzenia awarii systemu;
- dokonywania oceny funkcjonowania sieci;
- zwiększania wydajności systemu poprzez dostarczenie informacji na temat jego funkcjonowania w czasie wykonywania.

Analizę dynamiczną można przeprowadzić na dowolnym poziomie testowania. Wymaga ona jednak kwalifikacji technicznych i systemowych, które pozwalają na:

- określenie celów testowania realizowanych w ramach analizy dynamicznej;
- ustalenie właściwego czasu rozpoczęcia i zakończenia analizy;
- przeanalizowanie wyników.

Podczas testowania systemu z narzędzi do analizy dynamicznej mogą korzystać nawet analitycy mający minimalne kwalifikacje techniczne, ponieważ narzędzia te tworzą zwykle obszerne logi, które mogą być następnie analizowane przez osoby dysponujące niezbędnymi kwalifikacjami technicznymi.

3.3.2 Wykrywanie wycieków pamięci

Z wyciekiem pamięci mamy do czynienia w sytuacji, w której program przydziela dostępne obszary pamięci operacyjnej (RAM), ale nie zwalnia ich, gdy przestają być potrzebne. Dany obszar pamięci pozostaje wówczas

przydzielony i nie można go ponownie wykorzystać. Jeśli dzieje się tak często lub jeśli dostępne zasoby pamięci są bardzo ograniczone, programowi może zabraknąć możliwej do wykorzystania pamięci. Dawniej odpowiedzialność za właściwe posługiwanie się pamięcią ponosił programista. Program dokonujący alokacji musiał zwalniać we właściwym zakresie wszelkie dynamicznie przydzielane obszary pamięci w celu uniknięcia wycieku. Obecnie wiele środowisk programistycznych zawiera automatyczne lub półautomatyczne funkcje „odśmiecania" pamięci, które pozwalają odzyskać alokowaną pamięć bez bezpośredniej interwencji programisty. W takim przypadku, gdy dotychczas przydzielona pamięć jest zwalniana w ramach procesu automatycznego czyszczenia, zlokalizowanie wycieków pamięci może być bardzo trudne.

Wycieki pamięci powodują problemy, które narastają z czasem i nie zawsze są natychmiast widoczne. Taka sytuacja może mieć miejsce na przykład krótko po zainstalowaniu oprogramowania lub zrestartowaniu systemu, co często zdarza się podczas testowania. Z tego powodu negatywne skutki wycieków pamięci są często dostrzegane dopiero po rozpoczęciu eksploatacji programu.

Jednym z objawów wycieku pamięci jest stopniowe wydłużanie się czasu reakcji systemu, co może ostatecznie doprowadzić do jego awarii. Skutki takiej awarii można co prawda usunąć poprzez ponowne uruchomienie systemu, ale jest to niewygodne, a niekiedy może nawet okazać się niemożliwe.

Wiele narzędzi do analizy dynamicznej pozwala zidentyfikować obszary kodu, w których występują wycieki pamięci, dzięki czemu można wprowadzić odpowiednie poprawki. Proste programy do monitorowania pamięci również pozwalają się zorientować, czy ilość dostępnej pamięci z czasem maleje, chociaż ustalenie dokładnej przyczyny pogorszenia wydajności wymaga w takim przypadku przeprowadzenia dalszej analizy.
Należy również wziąć pod uwagę inne przyczyny wycieków, na przykład związane z uchwytami plików, semaforami czy pulami połączeń wykorzystywanymi przez zasoby.

3.3.3 Wykrywanie dzikich wskaźników

W programie nie mogą w żadnym razie występować tzw. „dzikie" wskaźniki. Dzikim wskaźnikiem może być na przykład wskaźnik, który „utracił" swój obiekt lub funkcję docelową, bądź też wskaźnik, który odwołuje się do innego niż zamierzony obszaru pamięci (np. do obszaru znajdującego się poza przydzielonymi granicami tablicy). Użycie w programie dzikich wskaźników może mieć różne konsekwencje:

- Program może działać zgodnie z oczekiwaniami. Sytuacja taka może mieć miejsce, jeśli dziki wskaźnik odwołuje się do pamięci, która nie jest obecnie używana przez program (przez co jest teoretycznie „wolna") i/lub zawiera wartość niepowodującą problemów.
- Program może ulec awarii. Dzieje się tak, jeśli dziki wskaźnik spowoduje nieprawidłowe użycie obszaru pamięci mającego krytyczne znaczenie dla działania programu (np. obszaru zastrzeżonego dla systemu operacyjnego).
- Program może nie działać prawidłowo z powodu braku dostępu do wymaganych obiektów. W takiej sytuacji program może nadal funkcjonować, ale zostanie wyświetlony komunikat o błędzie.
- Wskaźnik może uszkodzić dane znajdujące się w określonym obszarze pamięci, czego skutkiem jest użycie niepoprawnych wartości.

Należy przy tym pamiętać, że każda zmiana związana z użyciem pamięci przez program (np. nowa kompilacja po wprowadzeniu modyfikacji w oprogramowaniu) może pociągnąć za sobą każdy z wyżej wymienionych skutków. Ma to szczególnie istotne znaczenie w sytuacji, w której program na początku działa zgodnie z oczekiwaniami pomimo użycia dzikich wskaźników, a następnie niespodziewanie ulega załamaniu (być może dopiero podczas

eksploatacji) po wprowadzeniu zmiany. Warto również zwrócić uwagę na to, że tego rodzaju awarie są często symptomami występowania głębszych, ukrytych defektów (właśnie takich jak dzikie wskaźniki) (patrz [Kaner02], „Lesson 74"). Narzędzia ułatwiają identyfikowanie dzikich wskaźników używanych przez program niezależnie od tego, czy wskaźniki takie wpływają na jego działanie. Niektóre systemy operacyjne zawierają wbudowane funkcje umożliwiające wykrywanie naruszeń zasad dostępu do pamięci w czasie wykonywania. System operacyjny może na przykład zgłosić wyjątek, gdy aplikacja próbuje uzyskać dostęp do miejsca w pamięci znajdującego się poza obszarem dozwolonym dla tej aplikacji.

3.3.4 Analiza wydajności

Analiza dynamiczna przydaje się nie tylko do wykrywania błędów. Narzędzia używane podczas analizy dynamicznej wydajności programu pozwalają rozpoznać wąskie gardła związane z wydajnością oraz obliczyć wiele różnych metryk, które mogą posłużyć programistom do dostrojenia wydajności systemu. W ten sposób można, na przykład, uzyskać informacje o tym, ile razy poszczególne moduły są wywoływane w trakcie wykonywania programu, a następnie skupić się na zwiększaniu wydajności elementów najczęściej wywoływanych.

Scalenie informacji na temat dynamicznego funkcjonowania oprogramowania z informacjami uzyskanymi dzięki grafom wywołań na etapie analizy statycznej (patrz punkt 3.2.4) pozwala również testerowi rozpoznać moduły, które należałoby poddać szczegółowym i wnikliwym testom (np. moduły, które są często wywoływane i mają wiele interfejsów).

Analizę dynamiczną wydajności programu często przeprowadza się w ramach testów systemu, ale można ją również wykonać podczas testowania pojedynczego podsystemu we wcześniejszych fazach testowania (z wykorzystaniem jarzm testowych).

4. Atrybuty jakości w testach technicznych — 405 minut

Słowa kluczowe
zdolność adaptacyjna, zdolność do analizy, modyfikowalność, koegzystencja, efektywność, instalowalność, testowanie pielęgnowalności, dojrzałość, produkcyjne testy akceptacyjne, profil produkcyjny, testowanie wydajnościowe, testowanie przenaszalności, testowanie odtwarzalności, model wzrostu niezawodności, testowanie niezawodności, zastępowalność, testowanie zużycia zasobów, odporność, testowanie zabezpieczeń, stabilność, testowalność

Cele nauczania dotyczące atrybutów jakości w testach technicznych

4.2 Ogólne planowanie
TTA-4.2.1 (K4) Kandydat potrafi przeanalizować wymagania niefunkcjonalne i napisać odpowiednie fragmenty planu testów dla konkretnego projektu i systemu podlegającego testowaniu.

4.3 Testowanie zabezpieczeń
TTA-4.3.1 (K3) Kandydat potrafi zdefiniować podejście do testowania zabezpieczeń i zaprojektować przypadki testowe wysokiego poziomu.

4.4 Testowanie niezawodności
TTA-4.4.1 (K3) Kandydat potrafi zdefiniować podejście oraz zaprojektować przypadki testowe wysokiego poziomu do testowania atrybutu niezawodności i jej odpowiednich atrybutów podrzędnych według ISO 9126.

4.5 Testowanie wydajnościowe
TTA-4.5.1 (K3) Kandydat potrafi zdefiniować podejście i zaprojektować profile produkcyjne wysokiego poziomu dla testowania wydajnościowego.

Wspólne cele nauczania
Poniższe cele nauczania wiążą się z zagadnieniami przedstawionymi w kilku punktach niniejszego rozdziału.

TTA-4.x.1 (K2) Kandydat zna i potrafi uzasadnić przyczyny uwzględnienia testów pielęgnowalności, przenaszalności i zużycia zasobów w strategii testowania i/lub podejściu do testowania.
TTA-4.x.2 (K3) Kandydat potrafi zdefiniować konkretne typy testów niefunkcjonalnych, które są najbardziej dla odpowiednie dla podanego ryzyka produktowego.
TTA-4.x.3 (K2) Kandydat zna i potrafi omówić etapy w cyklu życia aplikacji, w których należy przeprowadzić testy niefunkcjonalne.
TTA-4.x.4 (K3) Kandydat potrafi zdefiniować typy defektów, których wykrycia należy się spodziewać w testach niefunkcjonalnych dla podanego scenariusza.

4.1 Wprowadzenie

W ogólności, techniczny analityk testowy koncentruje się raczej na tym, jak działa produkt, a nie na aspektach funkcjonalnych (co produkt robi). Takie testy można przeprowadzić na dowolnym poziomie. Na przykład, w trakcie testowania komponentów systemów czasu rzeczywistego i systemów wbudowanych istotne jest przeprowadzenie testów porównawczych wydajności oraz testów zużycia zasobów. W trakcie testowania systemowego i produkcyjnych testów akceptacyjnych właściwe jest testowanie aspektów związanych z niezawodnością, takich jak odtwarzalność. Testy na tym poziomie dotyczą konkretnego systemu,

tj. kombinacji sprzętu i oprogramowania. Konkretny testowany system może zawierać różne serwery, oprogramowanie klienckie, bazy danych, sieci i inne zasoby. Niezależnie od poziomu, testowanie należy przeprowadzić z uwzględnieniem priorytetów ryzyka i dostępnych zasobów.
Opis atrybutów jakości produktu bazuje na normie ISO 9126. Mogą w nim być również uwzględnione informacje pochodzące z innych standardów, takich jak ISO 25000 (który zastąpił normę ISO 9126). Atrybuty jakości ISO 9126 podzielono na atrybuty główne, z których każdy może zawierać atrybuty podrzędne. Są one przedstawione w poniższej tabeli wraz ze wskazaniem, które atrybuty i atrybuty podrzędne zostały opisane w sylabusie dla analityków testowych, a które w sylabusie dla technicznych analityków testowych.

Atrybut	Atrybut podrzędny	Analityk testowy	Techniczny analityk testowy
Funkcjonalność	Dokładność, dopasowanie, współdziałanie, zgodność	X	
	Zabezpieczenia		X
Niezawodność	Dojrzałość (odporność), tolerowanie awarii, odtwarzalność, zgodność		X
Użyteczność	Zrozumiałość, łatwość nauki, łatwość obsługi, atrakcyjność, zgodność	X	
Efektywność	Wydajność (w czasie), zużycie zasobów, zgodność		X
Pielęgnowalność	Zdolność do analizy, modyfikowalność, stabilność, testowalność, zgodność		X
Przenaszalność	Zdolność adaptacyjna, instalowalność, koegzystencja, zastępowalność, zgodność		X

Taki podział pracy jest stosowany w sylabusach ISTQB, jednak w różnych organizacjach może przybierać różne formy.

Dla każdego z atrybutów jakości wymieniono atrybut podrzędny „zgodność”. W pewnych środowiskach, których dotyczą szczególne standardy bezpieczeństwa lub uregulowania prawne, może być wymagane spełnienie określonych standardów lub norm przez każdy atrybut jakości. Standardy te mogą być bardzo różne dla poszczególnych branż i dlatego nie będą tutaj omawiane. Jeżeli techniczny analityk testowy pracuje w środowisku, którego dotyczą wymagania zgodności, powinien rozumieć te wymagania i zadbać o to, by zarówno testowanie, jak i dokumentacja testów je spełniły.
Należy zidentyfikować czynniki ryzyka typowe dla wszystkich atrybutów głównych i atrybutów podrzędnych jakości omówionych w tej sekcji, aby można było sformułować i udokumentować odpowiednią strategię testowania. Testowanie atrybutów jakości wymaga szczególnie starannego doboru właściwej fazy w cyklu życia, niezbędnych narzędzi, a także dostępności oprogramowania i dokumentacji do testowania oraz fachowej wiedzy technicznej. Bez odpowiedniej strategii dla każdego z atrybutów i specyficznych potrzeb związanych z jego testowaniem, tester może nie dysponować wystarczającą ilością czasu na odpowiednie zaplanowanie, przygotowanie i wykonanie testów [Bath08]. Część tych testów, np. testowanie wydajnościowe, wymaga szczegółowego zaplanowania, udostępnienia specjalnego sprzętu i konkretnych narzędzi, specjalistycznych kompetencji dotyczących testowania oraz, w większości przypadków, dużej ilości czasu. Testowanie atrybutów i atrybutów podrzędnych jakości musi być powiązane z ogólnym harmonogramem testów i muszą być do niego

przydzielone wystarczające zasoby. W każdym z tych obszarów występują specyficzne potrzeby i problemy, a testowanie może odbywać się w różnych punktach cyklu życia oprogramowania; te kwestie zostały dokładniej omówione w kolejnych punktach.

Kierownik testów zajmuje się kompilacją informacji o metrykach i raportowaniem sumarycznym dotyczącym atrybutów jakości i atrybutów podrzędnych, natomiast analityk testowy lub techniczny analityk testowy (zgodnie z powyższą tabelą) gromadzi informacje o każdej z metryk.

Pomiary atrybutów jakości dokonane w testach przedprodukcyjnych przez technicznego analityka testowego mogą stanowić podstawę umowy dotyczącej poziomu usług (umowy SLA) między dostawcą systemu i interesariuszami (np. klientami lub operatorami). W niektórych sytuacjach testy mogą być kontynuowane po wdrożeniu produkcyjnym oprogramowania, przy czym często są one wówczas wykonywane przez odrębny zespół lub organizację. Takie postępowanie ma na ogół miejsce w testach efektywności i niezawodności, których wyniki uzyskane w środowisku produkcyjnym mogą różnić się od wyników w środowisku testowym.

4.2 Planowanie ogólne

Brak planowania testów niefunkcjonalnych stwarza poważne ryzyko niepowodzenia wdrożenia aplikacji. Kierownik testów może poprosić technicznego analityka testowego o zidentyfikowanie głównych czynników ryzyka dla odpowiednich atrybutów jakości (patrz tabela w punkcie 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 następujące ogólne elementy:

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

4.2.1 Wymagania interesariuszy

Wymagania niefunkcjonalne są często niewystarczająco wyspecyfikowane, a czasami nawet w ogóle nie są zdefiniowane. W fazie planowania techniczny analityk testowy musi uzyskać od odpowiednich interesariuszy informacje o oczekiwanych poziomach związanych z atrybutami jakości, a następnie dokonać oceny czynników ryzyka, które się z nimi łączą.

Zwykle przyjmuje się, że jeśli klient jest zadowolony z aktualnej wersji systemu, będzie także zadowolony z nowych wersji, o ile osiągnięte poprzednio poziomy jakości pozostaną utrzymane. Dzięki temu istniejąca wersja systemu może być traktowana jako punkt odniesienia. Podejście to jest szczególnie przydatne w przypadku niektórych niefunkcjonalnych atrybutów jakości, takich jak wydajność, gdy interesariusze mogą mieć problemy z określeniem wymagań.

W trakcie zbierania wymagań niefunkcjonalnych należy uwzględnić różne punkty widzenia. Informacje muszą pochodzić od różnych interesariuszy, np. klientów i użytkowników oraz personelu operacyjnego i serwisowego, w przeciwnym razie istnieje prawdopodobieństwo pominięcia pewnych wymagań.

4.2.2 Zakup wymaganych narzędzi i szkolenia

Komercyjne narzędzia lub symulatory są szczególnie istotne w przypadku testów wydajnościowych i niektórych rodzajów testów zabezpieczeń. Do zadań technicznych analityków testowych należy oszacowanie kosztów i zarysowanie harmonogramu zakupu, nauki i wdrożenia takich narzędzi. Jeśli mają zostać zastosowane specjalistyczne narzędzia, w planowaniu należy wziąć pod uwagę konieczność przyswojenia wiedzy na temat nowych produktów i/lub koszt zaangażowania zewnętrznych specjalistów.

Opracowanie złożonego symulatora może być odrębnym projektem programistycznym, który należy uwzględnić w trakcie planowania. W szczególności, w harmonogramie i planie wykorzystania zasobów należy przewidzieć testowanie i dokumentowanie opracowanego narzędzia. Na aktualizację i ponowne przetestowanie symulatora wynikające ze zmian w symulowanym produkcie trzeba przeznaczyć odpowiedni budżet i czas. W przypadku aplikacji krytycznych ze względu na bezpieczeństwo, plan prac nad symulatorami musi uwzględniać testowanie akceptacyjne i ewentualną certyfikację symulatora przez niezależną jednostkę.

4.2.3 Wymagania dotyczące środowiska testowego

Wiele testów technicznych (np. testy zabezpieczeń i testy wydajnościowe) wymaga skonfigurowania środowiska testowego przypominającego środowisko produkcyjne w celu uzyskania realistycznych wartości pomiarów. W zależności od wielkości i złożoności testowanego systemu może mieć to istotny wpływ na planowanie i finansowanie testów. Ponieważ koszty takich środowisk mogą być wysokie, warto zastanowić się nad następującymi opcjami:

- wykorzystanie środowiska produkcyjnego;
- wykorzystanie ograniczonej wersji systemu. Należy wówczas zadbać o to, by wyniki testów w wystarczającej mierze odzwierciedlały działanie systemu produkcyjnego.

Harmonogram wykonywania takich testów powinien zostać szczegółowo zaplanowany. Istnieje duże prawdopodobieństwo, że testy tego rodzaju da się przeprowadzić jedynie w konkretnych terminach (np. w okresach zmniejszonego wykorzystania systemu).

4.2.4 Kwestie organizacyjne

Testy techniczne mogą wiązać się z pomiarami zachowania kilku komponentów pełnego systemu (np. serwerów, baz danych i sieci). Jeśli komponenty są rozproszone między różne lokalizacje i organizacje, nakład pracy związany z planowaniem i koordynacją testów może okazać się znaczący. Na przykład, niektóre komponenty oprogramowania mogą być dostępne na potrzeby testowania systemowego wyłącznie w określonych porach dnia lub roku. Może się także okazać, że organizacje są w stanie udzielić testerom wsparcia jedynie przez ograniczoną liczbę dni. Skutkiem braku potwierdzenia dostępności „na żądanie” komponentów systemu i personelu pochodzącego z innych organizacji („zewnętrznych” kompetencji) mogą być istotne zakłócenia przebiegu zaplanowanych testów.

4.2.5 Zagadnienia dotyczące bezpieczeństwa danych

W fazie planowania testów należy wziąć pod uwagę wdrożone zabezpieczenia systemu, tak aby możliwe było wykonanie wszystkich czynności związanych z testowaniem. Na przykład, jeśli używane jest szyfrowanie danych, wówczas tworzenie danych testowych i weryfikacja rezultatów mogą okazać się utrudnione.

Zasady i przepisy dotyczące ochrony danych mogą uniemożliwić generowanie wymaganych danych testowych opartych na danych produkcyjnych. Anonimizacja danych testowych jest zadaniem nietrywialnym, które należy

zaplanować w ramach implementacji testów.

4.3 Testowanie zabezpieczeń

4.3.1 Wprowadzenie

Testowanie zabezpieczeń różni się od innych form testowania funkcjonalnego w dwóch istotnych obszarach:

1. standardowe techniki wyboru testowych danych wejściowych mogą nie uwzględniać ważnych kwestii bezpieczeństwa;
2. objawy problemów związanych z bezpieczeństwem bardzo się różnią od symptomów wykrywanych w innych rodzajach testowania funkcjonalnego.

W ramach testowania zabezpieczeń dokonywana jest ocena podatności systemu na zagrożenia poprzez podjęcie próby naruszenia ochrony systemu określonej przez jego strategię bezpieczeństwa.
Poniższa lista zawiera listę potencjalnych zagrożeń, które należy uwzględnić podczas testowania zabezpieczeń:

- Kopiowanie aplikacji lub danych bez upoważnienia.
- Nieautoryzowany dostęp (np. możliwość wykonywania zadań, do których użytkownik nie ma uprawnień). Prawa użytkowników, zasady dostępu i uprawnienia są najważniejszymi elementami sprawdzanymi podczas testowania. Takie informacje powinny być dostępne w specyfikacji systemu.
- Oprogramowanie, którego zamierzonemu działaniu towarzyszą niezamierzone skutki uboczne. Na przykład, działaniu odtwarzacza plików multimedialnych, który poprawnie odtwarza pliki audio, ale zapisuje przy tym pliki w nieszyfrowanej pamięci tymczasowej, towarzyszy skutek uboczny, który może zostać wykorzystany przez cyberprzestępców.
- Kod umieszczany na stronie internetowej, który może zostać uruchomiony przez kolejnych użytkowników (tzw. *cross-site scripting*; XSS). Kod taki może wyrządzić szkody.
- Przepełnienie bufora, które może być spowodowane wprowadzaniem w polu wejściowym w interfejsie użytkownika łańcuchów o długości większej niż możliwa do poprawnego obsłużenia w kodzie. Występowanie tego zjawiska może stanowić okazję do uruchomienia szkodliwego kodu.
- Odmowa usługi, czyli sytuacja, gdy użytkownicy nie mogą nawiązać interakcji z aplikacją (przyczyną może być np. przeciążenie serwera WWW ustawicznie ponawianymi żądaniami).
- Przechwycenie, naśladowanie lub modyfikowanie, a następnie przekierowanie informacji (np. transakcji kartą kredytową) przez stronę trzecią w taki sposób, że użytkownik pozostaje nieświadomy istnienia strony trzeciej (atak typu *man in the middle*).
- Złamanie algorytmów szyfrujących używanych do ochrony wrażliwych danych.
- Bomby logiczne (nazywane czasami „kukułczymi jajami”), które mogą zostać rozmyślnie umieszczone w kodzie i aktywowane jedynie w pewnych okolicznościach (np. konkretnego dnia). Po aktywowaniu bomba logiczna może wykonać szkodliwe działania, takie jak usunięcie plików albo sformatowanie dysków.

4.3.2 Planowanie testów zabezpieczeń

Podczas planowania testów zabezpieczeń należy w szczególności zastanowić się nad następującymi

zagadnieniami:

- Ponieważ problemy dotyczące zabezpieczeń mogą się pojawić w trakcie tworzenia architektury, projektowania i implementacji systemu, można zaplanować testy zabezpieczeń na poziomie testowania jednostkowego, integracyjnego i systemowego. Ze względu na zmienność zagrożeń, testy zabezpieczeń można także wykonywać regularnie po wdrożeniu produkcyjnym systemu.
- Strategie testów zaproponowane przez technicznego analityka testowego mogą obejmować przeglądy kodu i analizę statyczną z wykorzystaniem narzędzi do zabezpieczeń. Narzędzia tego typu mogą być skutecznym środkiem wykrywania problemów związanych z bezpieczeństwem w architekturze, dokumentach projektowych i kodzie, które łatwo pominąć w trakcie testowania dynamicznego.
- Techniczny analityk testowy może zostać poproszony o zaprojektowanie i wykonanie pewnych typów „ataków" (patrz poniżej), które wymagają szczegółowego zaplanowania i skoordynowania działań z interesariuszami. Inne testy zabezpieczeń można wykonać we współpracy z programistami lub analitykami testowymi (np. testowanie praw użytkowników, zasad dostępu i uprawnień). W ramach planowania testów zabezpieczeń należy szczegółowo uwzględnić kwestie organizacyjne tego rodzaju.
- Kluczowym aspektem planowania testów zabezpieczeń jest uzyskanie akceptacji działań. Techniczny analityk testowy musi uzyskać wyraźne zezwolenie od kierownika testów na wykonanie zaplanowanych testów zabezpieczeń. Wszelkie dodatkowe, niezaplanowane testy mogą zostać uznane za rzeczywiste ataki, a osoba wykonująca takie testy jest narażona na podjęcie wobec niej działań prawnych. W przypadku braku potwierdzenia planów i uzyskania akceptacji w formie pisemnej wytłumaczenie typu „my tylko prowadzimy testy zabezpieczeń" może nie zabrzmieć zbyt przekonująco.
- Należy zwrócić uwagę, że udoskonalenia, których celem jest poprawa bezpieczeństwa systemu, mogą obniżyć jego wydajność. Po wprowadzeniu takich modyfikacji zalecane jest rozważenie konieczności wykonania testów wydajnościowych (patrz punkt 4.5 poniżej).

4.3.3 Specyfikacja testów zabezpieczeń

Testy zabezpieczeń można pogrupować (zgodnie z pracą [Whittaker04]) według źródła pochodzenia określonego zagrożenia:

- Związane z interfejsem użytkownika — nieautoryzowany dostęp i dane wejściowe wywołujące szkodliwe skutki.
- Związane z systemem plików — dostęp do wrażliwych danych przechowywanych w plikach lub repozytoriach.
- Związane z systemem operacyjnym — przechowywanie poufnych informacji (np. haseł) w postaci niezaszyfrowanej w pamięci, której zawartość może zostać ujawniona po załamaniu systemu wywołanym przez szkodliwe dane wejściowe.
- Związane z zewnętrznym oprogramowaniem — interakcje, które mogą wystąpić między zewnętrznymi komponentami używanymi przez system. Takie problemy mogą pojawić się na poziomie sieci (np. przesłanie niepoprawnych pakietów lub komunikatów) lub na poziomie modułów oprogramowania (np. awaria modułu oprogramowania niezbędnego do działania systemu).

Podczas projektowania testów zabezpieczeń można skorzystać z następującego podejścia [Whitaker04]:

- Zbierz informacje, które mogą okazać się przydatne podczas specyfikowania testów, np. nazwiska pracowników, adresy fizyczne, szczegółowe informacje o sieciach wewnętrznych, adresy IP, sygnatury

używanego oprogramowania i sprzętu oraz wersje systemu operacyjnego.

- Wykonaj skanowanie słabych punktów zabezpieczeń za pomocą ogólnie dostępnych narzędzi. Narzędzia tego rodzaju nie służą do podejmowania bezpośrednich ataków na system, ale do zidentyfikowania słabych punktów zabezpieczeń, które stanowią naruszenie zasad ochrony lub mogą takie naruszenie spowodować. Konkretne zagrożenia da się także zidentyfikować z wykorzystaniem list kontrolnych, takich jak listy opublikowane przez National Institute of Standards and Technology (NIST) [Web-2].
- Korzystając ze zgromadzonych informacji, opracuj „plany ataków” (tj. plany działań testowych zmierzających do naruszenia zasad ochrony konkretnego systemu). W planach ataków należy wyspecyfikować wykorzystanie różnych danych wejściowych wprowadzanych za pośrednictwem różnych interfejsów (np. interfejsu użytkownika i systemu plików), tak aby możliwe było wykrycie najpoważniejszych usterek związanych z zabezpieczeniami. Różne „ataki” opisane w pracy [Whittaker04] stanowią przydatny przegląd technik opracowanych specjalnie w celu testowania zabezpieczeń.

Problemy dotyczące zabezpieczeń można zidentyfikować także w trakcie przeglądów (patrz rozdział 5) lub zastosowania narzędzi do analizy statycznej (patrz punkt 3.2). Narzędzia do analizy statycznej zawierają rozbudowany zestaw reguł związanych z zagrożeniami bezpieczeństwa, służących do weryfikowania kodu. Narzędzie jest w stanie, na przykład,wykryć problemy z przepełnieniem bufora wynikające z braku sprawdzenia jego wielkości przed umieszczeniem w nim danych.

Narzędzi do analizy statycznej można użyć także w przypadku kodu aplikacji internetowej, aby wykryć potencjalne zagrożenia związane ze wstrzykiwaniem kodu, obsługą plików cookie, osadzaniem kodu pochodzącego z innych witryn (*cross-site scripting*), manipulacją zasobami i wstrzykiwaniem kodu SQL.

4.4 Testowanie niezawodności

Klasyfikacja ISO 9126 atrybutów jakości produktów definiuje następujące atrybuty podrzędne niezawodności:

- dojrzałość,
- tolerowanie awarii,
- odtwarzalność.

4.4.1 Pomiar dojrzałości oprogramowania

Celem testowania niezawodności jest monitorowanie statystycznych pomiarów dojrzałości oprogramowania na przestrzeni czasu i porównanie tych wartości z wymaganą niezawodnością docelową, która może mieć postać umowy dotyczącej poziomu usług (SLA). Używane miary mogą przybierać formę średniego czasu między awariami (MTBF), średniego czasu do naprawy (MTTR) lub dowolnych innych miar częstotliwości występowania awarii (np. liczby awarii o określonym stopniu ważności na tydzień). Uzyskane wyniki mogą zostać potraktowane jako kryteria wyjścia (np. związane z udostępnieniem wersji produkcyjnej).

4.4.2 Testy tolerowania awarii

W uzupełnieniu testowania funkcjonalnego, w którym badana jest zdolność oprogramowania do tolerowania awarii związanych z obsługą nieoczekiwanych wartości wejściowych (tzw. testowanie negatywne), niezbędne

jest przeprowadzenie dodatkowych testów w celu oceny tolerowania przez oprogramowanie usterek występujących na zewnątrz testowanej aplikacji. Tego rodzaju błędy są zwykle zgłaszane przez system operacyjny (np. przepełnienie dysku, niedostępny proces lub usługa, nie znaleziono pliku, pamięć nie jest dostępna). Testy tolerowania awarii na poziomie systemu można wykonywać z wykorzystaniem konkretnych narzędzi.

Warto wspomnieć, że w odniesieniu do tolerowania usterek często używa się także terminów „odporność" i „tolerowanie błędów" (szczegółowe informacje na ten temat podano w dokumencie [ISTQB_GLOSSARY]).

4.4.3 Testowanie odtwarzalności

W innych rodzajach testów niezawodności oceniana jest zdolność systemu do odzyskania możliwości działania po awariach sprzętu lub oprogramowania w ustalony sposób, który pozwala wznowić normalne działanie. Testy odtwarzalności obejmują testowanie pracy mimo awarii oraz testowanie tworzenia i odtwarzania kopii zapasowych.

Testy pracy mimo awarii wykonuje się wówczas, gdy konsekwencje awarii oprogramowania mogą być tak poważne, że wdrożono mechanizmy sprzętowe i/lub programowe, aby zapewnić ciągłość pracy systemu nawet w przypadku awarii. Tego rodzaju testy mogą być stosowane, na przykład, w przypadku występowania bardzo wysokiego ryzyka strat finansowych oraz w sytuacjach, w których istnieje poważne zagrożenie bezpieczeństwa. Jeśli awarie mogą być spowodowane zdarzeniami o charakterze katastrofy, ta odmiana testowania odtwarzalności nazywana bywa również „testowaniem odtwarzania po katastrofie".

Jednym z typowych środków zapobiegania awariom sprzętowym jest rozkładanie obciążenia między kilka serwerów w klastrze, procesorów lub dysków, w taki sposób, aby jeden z nich mógł natychmiast przejąć obciążenie w wypadku awarii drugiego elementu (tzw. systemy nadmiarowe). Typowym środkiem programowym jest wdrożenie więcej niż jednej niezależnej instancji systemu (np. w przypadku systemów kontroli lotu) w ramach tzw. zróżnicowanych systemów nadmiarowych. Systemy nadmiarowe stanowią zwykle kombinację elementów programowych i sprzętowych; wyróżniamy systemy podwójne, potrójne i poczwórne, w zależności od liczby niezależnych instancji (odpowiednio dwie, trzy lub cztery). Zróżnicowanie oprogramowania można osiągnąć poprzez przekazanie tych samych wymagań programowych dwóm lub większej liczbie niezależnych, niekomunikujących się zespołów w celu uzyskania takich samych usług realizowanych za pomocą różnego oprogramowania. Pozwala to podwyższyć poziom ochrony systemu, gdyż podanie analogicznych błędnych danych wejściowych z mniejszym prawdopodobieństwem spowoduje taki sam wynik. Kroki podjęte w celu poprawy odtwarzalności systemu mogą mieć bezpośredni wpływ na jego niezawodność, zatem powinny zostać wzięte pod uwagę podczas wykonywania testów niezawodności. Testowanie pracy mimo awarii polega na bezpośrednim testowaniu systemów za pomocą symulowania stanu awarii lub faktycznego wywoływania awarii w kontrolowanym środowisku. Po wystąpieniu awarii testowane są mechanizmy przełączania awaryjnego, aby uzyskać pewność, że dane nie zostaną utracone ani uszkodzone, a ewentualne uzgodnione poziomy usług zostaną utrzymane (np. dostępność funkcji i odpowiednie czasy reakcji). Więcej informacji na temat testowania pracy mimo awarii można znaleźć w serwisie [Web-1].

Testowanie tworzenia i odtwarzania kopii zapasowych opiera się na procedurach określonych w celu zminimalizowania skutków awarii. W takich testach dokonywana jest ocena procedur (zwykle udokumentowanych w podręcznikach) tworzenia różnych rodzajów kopii zapasowych i ich odtwarzania w przypadku utraty lub uszkodzenia danych. Przypadki testowe projektuje się tak, by uwzględnić przejście wszystkich ścieżek krytycznych poszczególnych procedur. Można także przeprowadzać przeglądy techniczne, aby „przećwiczyć" scenariusze i zweryfikować zgodność podręczników z faktycznie stosowanymi procedurami.

W trakcie produkcyjnych testów akceptacyjnych uruchamia się scenariusze w środowisku produkcyjnym lub zbliżonym do produkcyjnego, aby zweryfikować ich rzeczywistą przydatność.
Testy tworzenia i odtwarzania kopii zapasowych mogą uwzględniać:

- czas niezbędny na wykonanie różnych kopii zapasowych (np. pełnych i przyrostowych);
- czas potrzebny na odtworzenie danych;
- poziomy gwarantowanych kopii danych (np. odtworzenie wszystkich danych nie starszych niż 24 godziny, odtworzenie niektórych transakcji nie starszych niż godzina).

4.4.4 Planowanie testów niezawodności

Podczas planowania testów niezawodności należy w szczególności zastanowić się nad następującymi zagadnieniami:

- Niezawodność może być monitorowana także po wdrożeniu oprogramowania w środowisku produkcyjnym. Podczas gromadzenia wymagań dotyczących niezawodności w celu zaplanowania testów należy skonsultować się z jednostkami organizacyjnymi i personelem odpowiedzialnym za eksploatację oprogramowania.
- Techniczny analityk testowy może określić model wzrostu niezawodności, który opisuje oczekiwane poziomy niezawodności w miarę upływu czasu. Model wzrostu niezawodności może stanowić dla kierownika testów cenne źródło informacji, pozwalając mu porównać oczekiwany i osiągnięty poziom wydajności.
- Testy niezawodności należy przeprowadzić w środowisku zbliżonym do produkcyjnego. Używane środowisko powinno odznaczać się maksymalną stabilnością, aby możliwe było monitorowanie trendów związanych z niezawodnością.
- Ponieważ w testach niezawodności często wymagane jest użycie całego systemu, zwykle wykonuje się je w ramach testowania systemowego. Testami niezawodności można jednak również objąć pojedyncze moduły oraz zintegrowane zestawy modułów. Szczegółowe przeglądy architektury, projektu i kodu są również w stanie doprowadzić do ograniczenia ryzyka występowania problemów z niezawodnością we wdrożonym systemie.
- Aby wygenerować istotne statystycznie wyniki testów niezawodności, należy zwykle wydłużyć czas ich wykonywania. Skoordynowanie harmonogramu testowania z innymi zaplanowanymi testami może w związku z tym okazać się trudne.

4.4.5 Specyfikacja testów niezawodności

Testowanie niezawodności może odbywać się w formie powtarzalnego zestawu wcześniej zdefiniowanych testów. Mogą one być wybierane losowo z puli przypadków testowych generowanych w modelu statystycznym z wykorzystaniem metod losowych lub pseudolosowych. Testy mogą być również oparte na wzorcach użytkowania, nazywanych także profilami operacyjnymi (patrz punkt 4.5.4).

W niektórych testach niezawodności mogą występować operacje w intensywny sposób korzystające z pamięci, co pozwala wykryć ewentualne wycieki pamięci.

4.5 Testowanie wydajnościowe

4.5.1 Wprowadzenie

Klasyfikacja ISO 9126 atrybutów jakości produktów definiuje wydajność (atrybut czasowy) jako atrybut podrzędny efektywności. Testowanie wydajnościowe skupia się na zdolności modułu lub systemu do wygenerowania odpowiedzi na dane wejściowe przekazane przez użytkownika lub inny system w określonym czasie i w określonych warunkach.

Stosowane są różne typy pomiarów wydajności w zależności od celu testu. Dla pojedynczych modułów oprogramowania wydajność może być mierzona w cyklach procesora, natomiast dla systemów klienckich można ją wyrazić poprzez czas odpowiedzi systemu na konkretne żądanie użytkownika. W przypadku systemów, których architektura obejmuje różne komponenty (np. oprogramowanie klienckie, serwery i bazy danych), pomiary wydajności dotyczą transakcji między poszczególnymi komponentami, aby możliwe stało się zidentyfikowanie wąskich gardeł przetwarzania.

4.5.2 Typy testów wydajnościowych

4.5.2.1 Testowanie obciążeniowe

Testowanie obciążeniowe skupia się na zdolności systemu do obsługi wzrostu przewidywanego realistycznego obciążenia wynikającego z żądań transakcji generowanych przez jednocześnie pracujących użytkowników lub procesy. Pomiarom i analizie podlegają średnie czasy odpowiedzi dla użytkowników realizujących różne scenariusze typowego użycia (profile operacyjne). Patrz także [Splaine01].

4.5.2.2 Testowanie przeciążające

Testowanie przeciążające sprawdza zdolność systemu lub modułu do obsługi szczytowych wartości obciążenia na przewidywanej lub wyspecyfikowanej granicy (a nawet powyżej tej granicy) oraz w warunkach ograniczonej dostępności zasobów, np. mocy obliczeniowej lub przepustowości łącza. W miarę zwiększania obciążenia wydajność systemu powinna spadać powoli i w przewidywalny sposób, bez występowania awarii. Obciążony system powinien zostać w szczególności przetestowany pod kątem integralności funkcjonalnej w celu wykrycia ewentualnych usterek w przetwarzaniu lub niespójności danych. Jednym z celów testowania przeciążającego może być wykrycie wartości granicznych, powyżej których następuje faktyczna awaria systemu, tak aby można było zidentyfikować „najsłabsze ogniwo”. W testowaniu przeciążającym można w odpowiednich momentach uzupełniać system o dodatkowe zasoby (np. pamięć, moc obliczeniową procesora, miejsce w bazie danych).

4.5.2.3 Testowanie skalowalności

Testowanie skalowalności koncentruje się na sprawdzaniu zdolności systemu do realizacji przyszłych wymagań dotyczących efektywności, które mogą przekraczać bieżące założenia. Celem testów jest ocena zdolności systemu do rozbudowy (np. obsługi większej liczby użytkowników i przechowywania większej ilości danych) bez awarii i z zachowaniem bieżących wymagań dotyczących wydajności. Gdy zostaną określone ograniczenia skalowalności, wówczas można ustawić w środowisku produkcyjnym wartości progowe i monitorować je w celu wygenerowania ostrzeżeń o potencjalnych problemach. Środowisko produkcyjne można również rozszerzyć o odpowiednią liczbę jednostek sprzętowych.

4.5.3 Planowanie testów wydajnościowych

Oprócz ogólnych kwestii związanych z planowaniem, opisanych w punkcie 4.2, na planowania testów wydajnościowych mogą mieć wpływ następujące czynniki:

- W zależności od wykorzystywanego środowiska testowego i testowanego oprogramowania (patrz punkt 4.2.3) może się okazać, że do skutecznego przeprowadzenia testów wydajnościowych wymagana jest implementacja całego systemu. W takiej sytuacji testowanie wydajnościowe odbywa się zwykle w trakcie testów systemowych. Inne testy wydajnościowe, które da się skutecznie

przeprowadzić na poziomie modułów, można zaplanować w fazie testów jednostkowych.

- Ogólnie rzecz biorąc, zalecane jest możliwie najwcześniejsze wykonanie wstępnych testów wydajnościowych, nawet jeśli nie jest jeszcze dostępne środowisko zbliżone do produkcyjnego. Takie wczesne testy pozwalają wykryć problemy z wydajnością (np. wąskie gardła) i ograniczyć ryzyko projektowe dzięki uniknięciu konieczności wprowadzania czasochłonnych poprawek na późniejszych etapach tworzenia oprogramowania lub w środowisku produkcyjnym.
- Przeglądy kodu, w szczególności koncentrujące się na interakcji z bazą danych, interakcji między modułami i obsłudze błędów, pozwalają zidentyfikować problemy dotyczące wydajności (zwłaszcza związane z logiką „czekaj i ponów" oraz nieefektywnymi zapytaniami). Przeglądy powinny zostać zaplanowane we wczesnych etapach cyklu życia oprogramowania.
- Należy zaplanować i uwzględnić w budżecie sprzęt, oprogramowanie i przepustowość sieci niezbędne do uruchomienia testów wydajnościowych. Potrzebne zasoby są zależne przede wszystkim od generowanego obciążenia, które może wynikać z liczby symulowanych użytkowników wirtualnych oraz generowanego przez nich ruchu w sieci. Jeśli czynniki te nie zostaną uwzględnione, to uzyskane wartości pomiarów wydajności mogą być niereprezentatywne. Na przykład, weryfikacja wymagań dotyczących skalowalności często odwiedzanego serwisu internetowego może wymagać zasymulowania setek tysięcy wirtualnych użytkowników.
- Konieczność wygenerowania wymaganego obciążenia w testach wydajnościowych może oznaczać istotne zwiększenie kosztów zakupu sprzętu i narzędzi. Należy uwzględnić ten element podczas planowania testów wydajnościowych w celu uzyskania odpowiedniego finansowania.
- Koszty wygenerowania obciążenia w testach wydajnościowych można zminimalizować dzięki wynajęciu wymaganej infrastruktury testowej. Oznacza to, na przykład, wypożyczenie dodatkowych licencji na narzędzia albo skorzystanie z usług zewnętrznego dostawcy w celu zrealizowania potrzeb sprzętowych (np. z usług przetwarzania w chmurze). W takim przypadku, czas dostępny na przeprowadzenie testów wydajnościowych może być ograniczony, dlatego należy szczegółowo zaplanować testy.
- W fazie planowania trzeba zwrócić szczególną uwagę na to, czy używane narzędzie jest kompatybilne z protokołami komunikacyjnymi stosowanymi w testowanym systemie.
- Defekty związane z wydajnością często mają istotny wpływ na działanie testowanego systemu. Jeśli realizacja wymagań dotyczących wydajności ma najwyższy priorytet, warto wówczas wykonać testy wydajnościowe najważniejszych modułów (z wykorzystaniem sterowników i zaślepek), zamiast odkładać to zadanie do etapu testów systemowych.

4.5.4 Specyfikacja testów wydajnościowych

Specyfikacja testów w przypadku różnych rodzajów testów wydajnościowych, np. testów obciążeniowych i przeciążeniowych, opiera się na definicji profili operacyjnych. Opisują one różne rodzaje zachowań użytkowników podczas interakcji z aplikacją. Dla danej aplikacji może istnieć wiele profili produkcyjnych. Liczbę użytkowników przypadającą na poszczególne profile produkcyjne można określić za pomocą narzędzi monitorujących (o ile aplikacja ta lub aplikacja do niej podobna jest już dostępna) lub na podstawie przewidywań. Takie przewidywania mogą zostać podane przez jednostki biznesowe albo ustalone na podstawie algorytmów. Szczególnie istotne są one w przypadku specyfikowania profili produkcyjnych używanych podczas testowania skalowalności.

Na podstawie profili produkcyjnych ustala się liczbę i rodzaj przypadków testowych stosowanych podczas

testowania wydajnościowego. Testy odbywają się często pod kontrolą narzędzi testowych, które tworzą „wirtualnych", czyli symulowanych użytkowników w liczbie odpowiadającej testowanemu profilowi (patrz punkt 6.3.2).

4.6 Testowanie zużycia zasobów

Klasyfikacja ISO 9126 atrybutów jakości produktów definiuje zużycie zasobów jako atrybut podrzędny efektywności. W testach związanych ze zużyciem zasobów sprawdza się wykorzystanie zasobów systemowych (np. pamięci, miejsca na dysku, przepustowości sieci, połączeń) w odniesieniu do zdefiniowanych wcześniej wartości porównawczych. Porównanie odbywa się przy normalnym obciążeniu i w sytuacjach przeciążenia, na przykład przy dużej liczbie transakcji i dużej ilości danych, w celu stwierdzenia, czy nie występuje nienaturalny wzrost wykorzystania zasobów.

Na przykład, wykorzystanie pamięci jest istotną kwestią sprawdzaną w testach wydajnościowych systemów wbudowanych czasu rzeczywistego. Jeśli przekroczy ono dozwoloną wartość, system może nie mieć wystarczającej pamięci do wykonania operacji w określonym czasie, czego efektem może być spowolnienie działania lub nawet załamanie systemu.

Do weryfikacji zużycia zasobów i wykrywania wąskich gardeł wydajności można także zastosować analizę dynamiczną (patrz punkt 3.3.4).

4.7 Testowanie pielęgnowalności

Czas poświęcany na utrzymanie oprogramowania jest często dłuższy niż czas jego wytworzenia. Celem tzw. testowania pielęgnacyjnego jest przetestowanie zmian we wdrożonym systemie lub wpływu zmienionego środowiska na wdrożony system. Z kolei, aby zapewnić jak największą efektywność procesu utrzymania, wykonuje się testowanie pielęgnowalności, które pozwala zmierzyć stopień łatwości analizowania kodu, wprowadzania w nim zmian i jego testowania.
Wśród celów związanych z pielęgnowalnością, jakie chcą uzyskać interesariusze (np. właściciele lub operatorzy oprogramowania), znajdują się:

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

Testy pielęgnowalności powinny zostać uwzględnione w strategii testów i/lub podejściu do testowania, 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 pielęgnowalności (patrz powyżej) w cyklu życia oprogramowania przeważają nad kosztami wykonania testów pielęgnowalności i wprowadzenia niezbędnych zmian.
- Czynniki ryzyka związane z niską pielęgnowalnością oprogramowania (np. długie czasy reakcji na defekty zgłaszane przez użytkowników i/lub klientów) uzasadniają przeprowadzenie testów tego rodzaju.

Technikami stosowanymi w trakcie testowania pielęgnowalności są m.in. analiza statyczna i przeglądy, zgodnie z opisem przedstawionym w punktach 3.2 i 5.2. Testowanie pielęgnowalności należy rozpocząć od razu po

udostępnieniu dokumentów projektowych i kontynuować je w trakcie prac związanych z implementacją kodu. Pielęgnowalność to cecha związana z kodem i dokumentacją poszczególnych modułów, dlatego można ją ocenić na wczesnym etapie cyklu wytwórczego, bez konieczności oczekiwania na udostępnienie kompletnego, działającego systemu.

Dynamiczne testowanie pielęgnowalności skupia się na udokumentowanych procedurach utrzymania poszczególnych aplikacji (np. opisujących wykonywanie aktualizacji oprogramowania). Wybrane scenariusze pielęgnacyjne są wykorzystywane jako przypadki testowe w celu sprawdzenia, czy udokumentowane procedury pozwalają uzyskać wymagane poziomy usług. Taka forma testowania ma szczególne znaczenie w sytuacji, gdy używana infrastruktura jest złożona, a procedury wsparcia obejmują wiele działów lub organizacji. Testowanie tego rodzaju można wykonać w ramach produkcyjnych testów akceptacyjnych. [Web-1]

4.7.1 Analizowalność, modyfikowalność, stabilność i testowalność

Pielęgnowalność systemu może być również wyrażona jako pracochłonność wymagana do zdiagnozowania przyczyn problemów wykrytych w systemie (zdolność do analizy), implementacji zmian w kodzie (modyfikowalność) oraz przetestowania zmodyfikowanego systemu (testowalność). Stabilność opisuje sposób reakcji systemu na wprowadzanie zmian. W systemach o niskiej stabilności po wprowadzeniu zmian występuje duża liczba problemów wtórnych (tzw. efekt fali) [ISO9126] [Web-1].
Pracochłonność związana z pielęgnowaniem systemu zależy od różnych czynników, w szczególności przyjętej metodyki projektowania oprogramowania (np. zorientowanej obiektowo) i zastosowanych standardów kodowania.

Należy podkreślić, że „stabilność” w tym kontekście nie powinna być mylona z pojęciami „odporność” i „tolerowanie usterek”, opisanymi w punkcie 4.4.2.

4.8 Testowanie przenaszalności

Testy przenaszalności dotyczą łatwości przenoszenia oprogramowania do docelowego środowiska, zarówno po raz pierwszy, jak i z istniejącego środowiska. Testowanie przenaszalności obejmuje testy instalowalności, koegzystencji/kompatybilności i zastępowalności. Testowanie można zacząć od poszczególnych modułów (m.in. weryfikując zastępowalność danego komponentu, np. przejście z jednego systemu zarządzania bazą danych na inny) i zwiększać jego zakres w miarę udostępniania kolejnych fragmentów kodu. Instalowalność czasami można przetestować dopiero wówczas, gdy działają wszystkie komponenty produktu. Przenaszalność powinna zostać uwzględniona w projekcie i wbudowana w produkt, dlatego należy ją wziąć pod uwagę we wczesnych fazach projektowania i tworzenia architektury. Przeglądy projektu i architektury stanowią szczególnie skuteczną metodę identyfikowania wymagań dotyczących przenaszalności i potencjalnych problemów (np. zależności od konkretnego systemu operacyjnego).

4.8.1 Testowanie instalowalności

Testowanie instalowalności przeprowadza się na oprogramowaniu z wykorzystaniem procedur używanych do jego zainstalowania w docelowym środowisku. Może to, na przykład, obejmować oprogramowanie przeznaczone do zainstalowania systemu operacyjnego na procesorze albo kreator instalacji używany do zainstalowania produktu na stacji roboczej.

Typowymi celami testowania instalowalności są:

- Sprawdzenie, czy oprogramowanie może zostać pomyślnie zainstalowane po zastosowaniu instrukcji podanych w podręczniku instalacji (łącznie z uruchomieniem ewentualnych skryptów instalacyjnych) lub za pomocą kreatora instalacji. Obejmuje to również sprawdzenie opcji instalacji dla różnych

konfiguracji programowo/sprzętowych i różnych stopni instalacji (np. instalacji początkowej i aktualizacji).

- Sprawdzenie, czy błędy występujące podczas instalacji (np. błąd ładowania konkretnej biblioteki DLL) są obsługiwane poprawnie przez oprogramowanie instalacyjne i czy system nie jest pozostawiany w stanie niezdefiniowanym (np. z oprogramowaniem częściowo zainstalowanym albo w niepoprawnej konfiguracji).
- Przetestowanie możliwości wykonania częściowej instalacji lub deinstalacji.
- Przetestowanie, czy kreator instalacji jest w stanie zidentyfikować niepoprawne platformy sprzętowe i konfiguracje systemu operacyjnego.
- Sprawdzenie, czy proces instalacji może zostać zakończony w podanym czasie lub w ramach założonej liczby kroków.
- Sprawdzenie, czy można pomyślnie przejść na starszą wersję oprogramowania lub je odinstalować.

Po instalacji wykonuje się zwykle testowanie funkcjonalności w celu wykrycia ewentualnych usterek powstałych podczas instalacji (np. niepoprawnych konfiguracji i braku dostępności funkcji). Zwykle równolegle z testowaniem instalowalności prowadzone jest testowanie użyteczności, na przykład w celu sprawdzenia, czy użytkownicy otrzymują podczas instalacji zrozumiałe instrukcje, odpowiedzi i komunikaty o błędach.

4.8.2 Testowanie koegzystencji/kompatybilności

Systemy, które nie są ze sobą powiązane, nazywane są kompatybilnymi, jeśli mogą zostać uruchomione w tym samym środowisku (np. na tym samym sprzęcie) bez wzajemnego wpływu na działanie (np. bez konfliktów dotyczących zasobów). Testy kompatybilności należy wykonać wtedy, gdy nowe lub zaktualizowane oprogramowanie ma zostać wdrożone w środowiskach, które zawierają już pewne zainstalowane aplikacje.

Problemy w tym obszarze mogą wystąpić wówczas, gdy aplikacja jest testowana w środowisku, w którym jest jedyną zainstalowaną aplikacją (i w związku z tym braku zgodności nie da się wykryć), a następnie zostanie przeniesiona do innego środowiska (np. produkcyjnego), w którym funkcjonują także inne aplikacje.

Typowymi celami testowania kompatybilności są:

- Ocena możliwego negatywnego wpływu na funkcjonalność w przypadku ładowania aplikacji w tym samym środowisku (np. konflikt użycia zasobów, gdy na serwerze jest uruchomionych wiele aplikacji).
- Ocena wpływu instalacji poprawek i aktualizacji systemu operacyjnego na działanie aplikacji.

Kwestie związane ze zgodnością należy uwzględnić podczas planowania docelowego środowiska produkcyjnego, jednak same testy zwykle wykonuje się dopiero po pomyślnym zakończeniu testowania systemowego i testowania akceptacyjnego przez użytkowników.

4.8.3 Testowanie zdolności adaptacyjnych

Testowanie zdolności adaptacyjnej sprawdza, czy dana aplikacja może funkcjonować poprawnie we wszystkich założonych środowiskach docelowych (sprzęt, oprogramowanie, warstwa pośrednia, system operacyjny itd.). Zatem, system zdolny do adaptacji to system otwarty, który jest w stanie dostosować swoje działanie do zmian wprowadzonych w środowisku lub w poszczególnych częściach samego systemu. W ramach specyfikacji testów zdolności adaptacyjnej należy zidentyfikować, skonfigurować i udostępnić zespołowi testowemu odpowiednie kombinacje środowisk docelowych. Następnie takie środowiska są testowane za pomocą wybranych funkcjonalnych przypadków testowych sprawdzających różnego rodzaju komponenty obecne w środowisku.

Zdolność adaptacyjna może odnosić się do możliwości przeniesienia oprogramowania na różne określone środowiska w wyniku wykonania zdefiniowanej procedury. Testy mogą służyć do weryfikacji tej procedury.

Testy zdolności adaptacyjnej można wykonywać łącznie z testami instalowalności. Zwykle potem wykonuje się testy funkcjonalne, których celem jest wykrycie ewentualnych usterek wprowadzonych podczas dostosowywania oprogramowania do innego środowiska.

4.8.4 Testowanie zastępowalności

Testowanie zastępowalności koncentruje się na możliwości zastąpienia modułów programowych w systemie przez inne komponenty. Może być szczególnie przydatne w systemach, w których do obsługi pewnych komponentów stosowane jest komercyjne oprogramowanie „z półki".

Testy zastępowalności mogą być wykonywane równolegle z testami integracji podstawowej funkcjonalności systemu, jeśli więcej niż jeden wymienny moduł jest już dostępny. Zastępowalność może być oceniana w ramach przeglądu technicznego lub inspekcji na poziomie architektury i projektu, gdy szczególną wagę przywiązuje się do jasnej definicji interfejsów do potencjalnie wymiennych modułów.

5. Przeglądy — 165 minut

Słowa kluczowe
antywzorzec

Cele nauczania dotyczące przeglądów

5.1 Wprowadzenie

TTA 5.1.1 (K2) Kandydat potrafi wyjaśnić dlaczego przygotowanie do przeglądu jest istotne w przypadku technicznego analityka testowego.

5.2 Korzystanie z list kontrolnych podczas przeglądów

TTA 5.2.1 (K4) Kandydat potrafi przeanalizować projekt architektury i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.
TTA 5.2.2 (K4) Kandydat potrafi przeanalizować fragment kodu lub pseudokodu i zidentyfikować problemy zgodnie z listą kontrolną podaną w sylabusie.

5.1 Wprowadzenie

Techniczny analityk testowy musi aktywnie uczestniczyć w procesie przeglądu, przedstawiając swój specyficzny punkt widzenia na poruszane zagadnienia. Powinien zostać przeszkolony w zakresie przeglądów formalnych, tak aby rozumiał swoją rolę w dowolnym procesie przeglądu technicznego. Wszyscy uczestnicy przeglądu technicznego muszą dążyć do osiągnięcia korzyści płynących z dobrze przeprowadzonego przeglądu. Pełny opis przeglądów technicznych, łącznie z wieloma listami kontrolnymi, można znaleźć w pracy [Wiegers02]. Techniczny analityk testowy zwykle bierze udział w przeglądach technicznych i inspekcjach. W ich trakcie jest w stanie przedstawić perspektywę operacyjną (dotyczącą zachowania systemu), która mogła zostać pominięta przez programistów. Ponadto techniczny analityk testowy odgrywa ważną rolę w definiowaniu, realizowaniu i utrzymaniu list kontrolnych przeglądów oraz informacji o wagach defektów.

Niezależnie od typu dokonywanego przeglądu, techniczny analityk testowy musi zaplanować odpowiednio dużo czasu na przygotowanie. Obejmuje to czas potrzebny na przejrzenie produktu, na sprawdzenie powiązanych dokumentów pomocniczych pod kątem spójności oraz na ustalenie, czego może brakować w danym produkcie. Bez odpowiedniego przygotowania przegląd może zostać sprowadzony wyłącznie do prostych prac redakcyjnych. Dobrze przeprowadzony przegląd obejmuje zrozumienie tego, co zostało napisane, ustalenie, czego brakuje, i sprawdzenie, czy dany produkt jest spójny z innymi produktami (już opracowanymi albo dopiero przygotowywanymi). Na przykład, podczas przeglądu planu testów na poziomie integracji techniczny analityk testowy musi również uwzględnić integrowane elementy. Czy są one gotowe do integracji? Czy istnieją jakieś zależności, które powinny zostać udokumentowane? Czy są dostępne dane do testowania punktów integracji? Przegląd nie ogranicza się do danego produktu podlegającego przeglądowi, ale należy w nim również uwzględnić interakcje tego produktu z innymi elementami systemu.
Często się zdarza, że autorzy produktu poddawanego przeglądowi czują się krytykowani. Wszystkie komentarze technicznego analityka testowego w ramach przeglądu powinny być wygłaszane z perspektywy współpracy z autorem w celu uzyskania jak najlepszej jakości produktu. Umożliwi to konstruktywne formułowanie uwag, które będą się odnosić do produktu, a nie do autora. Na przykład, w przypadku niejednoznacznego stwierdzenia lepiej powiedzieć „Nie rozumiem, co należy przetestować, żeby sprawdzić, że to wymaganie zostało poprawnie zaimplementowane. Czy możesz mi pomóc to zrozumieć?” niż „To wymaganie jest

niejednoznaczne i nikt go nie zrozumie".
Zadaniem technicznego analityka testowego w ramach przeglądu jest sprawdzenie, czy informacje zawarte w produkcie wystarczą do wykonania testów. Jeżeli informacji brakuje lub są one niejasne, stanowi to prawdopodobnie defekt, który powinien zostać usunięty przez autora. Konstruktywne, a nie krytyczne podejście ułatwia adresatowi przyjęcie komentarzy i zwiększa produktywność spotkania.

5.2 Korzystanie z list kontrolnych podczas przeglądów

Podczas przeglądów używa się list kontrolnych przypominających uczestnikom o konieczności zweryfikowania konkretnych elementów. Pomagają one również w wyeliminowaniu z przeglądu czynnika osobistego: „używamy tej samej listy we wszystkich przeglądach, nie tylko w odniesieniu do twojego produktu". Listy kontrolne mogą być ogólne, do zastosowania we wszelkiego rodzaju przeglądach, lub skoncentrowane na określonych atrybutach jakości albo obszarach. Na przykład, lista przeznaczona do przeglądów dokumentacji wymagań może zawierać takie punkty, jak sprawdzenie odpowiedniego użycia pojęć „będzie" i „powinien", weryfikacja formatowania oraz inne podobne elementy dotyczące zgodności z szablonem. Specjalne listy kontrolne mogą koncentrować się na zagadnieniach związanych z bezpieczeństwem lub wydajnością. Najbardziej przydatne są listy kontrolne sukcesywnie tworzone przez konkretne jednostki organizacyjne, ponieważ uwzględniają:

- charakter produktu,
- lokalne środowisko wytwórcze:
  - o personel,
  - o narzędzia,
  - o priorytety,
- historię poprzednich sukcesów i defektów,
- specyficzne problemy (np. dotyczące wydajności i bezpieczeństwa).

Listy powinny zostać dostosowane do potrzeb organizacji, a być może także do potrzeb konkretnego projektu. Listy kontrolne zaprezentowane w niniejszym rozdziale należy traktować jedynie jako przykłady.
W niektórych organizacjach rozszerza się podstawowe znaczenie pojęcia lista kontrolna i uwzględnia tzw. antywzorce, czyli często popełniane błędy, niewłaściwe techniki i inne nieefektywne procedury. Pojęcie to pochodzi od popularnej koncepcji wzorców projektowych, stanowiących przeznaczone do wielokrotnego użytku, sprawdzone w praktyce rozwiązania często występujących problemów [Gamma94]. Antywzorzec oznacza zatem często popełniany błąd, powstający nierzadko w wyniku doraźnego rozwiązania jakiegoś problemu.

Należy pamiętać, że jeżeli wymaganie nie jest testowalne, czyli jest zdefiniowane w taki sposób, że techniczny analityk testowy nie może zaprojektować sposobu jego przetestowania, to w takim wymaganiu tkwi defekt. Na przykład wymaganie „Oprogramowanie powinno działać szybko" nie jest testowalne. W jaki sposób techniczny analityk testowy ma stwierdzić, że oprogramowanie działa szybko? Gdyby z kolei wymaganie zawierało stwierdzenie „Maksymalny czas odpowiedzi oprogramowania musi wynosić trzy sekundy przy konkretnym obciążeniu", to testowalność takiego wymagania będzie znacznie większa, jeśli zdefiniuje się znaczenie określenia „przy konkretnym obciążeniu" (np. podając liczbę jednocześnie pracujących użytkowników i wykonywane przez nich działania). Jest to też wymaganie bardzo ogólne, ponieważ w przypadku nietrywialnej aplikacji może na jego podstawie powstać wiele odrębnych przypadków testowych. Prześledzenie związków tego wymagania z przypadkami testowymi ma również kluczowe znaczenie, ponieważ w przypadku zmiany wymagania należałoby dokonać przeglądu i odpowiednich modyfikacji wszystkich przypadków testowych.

5.2.1 Przeglądy architektury

Architektura oprogramowania określa fundamentalne zasady organizacji systemu, opisujące jego komponenty, ich wzajemne relacje i relacje ze środowiskiem oraz zasady projektowania i rozwoju systemu. [ANSI/IEEE Std 1471-2000], [Bass03].
Listy kontrolne stosowane w trakcie przeglądów architektury mogą na przykład zawierać weryfikację poprawnej implementacji następujących elementów (lista pochodzi ze strony [Web-3]):

- zestawianie połączeń — skrócenie narzutu czasowego związanego z nawiązywaniem połączeń z bazą danych poprzez utworzenie współużytkowanej puli połączeń;
- równoważenie obciążenia — równomierne rozłożenie obciążenia między poszczególne zasoby;
- przetwarzanie rozproszone;
- buforowanie — używanie lokalnej kopii danych w celu skrócenia czasu dostępu;
- inicjowanie z opóźnieniem (tzw. leniwe inicjowanie);
- współbieżność transakcji;
- odseparowanie procesów przetwarzania transakcyjnego na bieżąco (OLTP) od analitycznego przetwarzania na bieżąco (OLAP);
- replikacja danych.

Więcej szczegółów (niezwiązanych z przygotowaniem do egzaminu certyfikacyjnego) można znaleźć na stronie [Web-4], która odwołuje się do pracy zawierającej analizę 117 list kontrolnych pochodzących z 24 źródeł. Omówiono tam różne kategorie elementów list kontrolnych i podano przykłady elementów dobrze sformułowanych elementów oraz takich, których należy unikać.

5.2.2 Przeglądy kodu

Listy kontrolne przeglądów kodu są z założenia bardzo szczegółowe i, podobnie jak w przypadku przeglądów architektury, sprawdzają się najlepiej, jeśli dotyczą konkretnego języka programowania, projektu i przedsiębiorstwa. Uwzględnienie antywzorców dotyczących kodu może okazać się pomocne, szczególnie w przypadku mniej doświadczonych programistów.

Na listach kontrolnych używanych w trakcie przeglądów kodu można uwzględnić sześć poniższych elementów (propozycja na podstawie [Web-5]):
1. Struktura

- Czy kod w pełny i prawidłowy sposób implementuje projekt?
- Czy kod jest zgodny z odpowiednimi standardami kodowania?
- Czy kod ma poprawną strukturę, jednolity styl i jednolite formatowanie?
- Czy istnieją niewywoływane lub niepotrzebne procedury i inne fragmenty nieosiągalnego kodu?
- Czy w kodzie pozostały zaślepki i procedury testowe?
- Czy jakieś fragmenty kodu mogą zostać zastąpione wywołaniami do zewnętrznych komponentów wielokrotnego użytku lub funkcji z biblioteki?
- Czy istnieją bloki powtarzającego się kodu, które można zastąpić pojedynczą procedurą?
- Czy wykorzystanie pamięci jest efektywne?
- Czy jako stałe używane są wartości symboliczne, a nie „magiczne liczby” lub stałe łańcuchowe?

- Czy istnieją moduły o zbyt wysokim stopniu złożoności, które należy zrestrukturyzować lub podzielić na wiele modułów?

2. Dokumentacja
   - Czy kod jest udokumentowany w jasny i prawidłowy sposób, a styl komentarzy umożliwia proste wprowadzanie zmian?
   - Czy wszystkie komentarze mają treść zgodną z kodem?
   - Czy dokumentacja jest zgodna z obowiązującymi standardami?

3. Zmienne
   - Czy wszystkie zmienne są właściwie zdefiniowane i mają znaczące, spójne i zrozumiałe nazwy?
   - Czy istnieją zbyteczne lub nieużywane zmienne?

4. Operacje arytmetyczne
   - Czy w kodzie unika się porównywania liczb zmiennoprzecinkowych?
   - Czy w kodzie w systematyczny sposób unika się błędów zaokrągleń?
   - Czy w kodzie nie pojawiają się operacje dodawania i odejmowania liczb o różniących się znacznie rzędach wielkości?
   - Czy sprawdzane są dzielniki (czy są różne od zera i nie mają zaburzonych wartości)?

5. Pętle i gałęzie
   - Czy wszystkie pętle, gałęzie kodu i konstrukcje logiczne są pełne, poprawne i prawidłowo zagnieżdżone?
   - Czy w łańcuchach instrukcji IF-ELSEIF najczęstsze przypadki są sprawdzane na początku?
   - Czy w bloku instrukcji IF-ELSEIF lub CASE sprawdzane są wszystkie przypadki i czy są uwzględnione klauzule ELSE lub DEFAULT?
   - Czy każda instrukcja CASE ma klauzulę DEFAULT?
   - Czy warunki zakończenia pętli są oczywiste i zawsze osiągalne?
   - Czy indeksy (w tym indeksy tablic) są poprawnie zainicjowane tuż przed wejściem do pętli?
   - Czy niektóre z instrukcji znajdujących się w pętlach można umieścić poza pętlami?
   - Czy kod w pętli stara się nie manipulować zmienną sterującą ani korzystać z niej w momencie wyjścia z pętli?

6. Programowanie defensywne
   - Czy sprawdzane jest przekroczenie wartości granicznych w tablicy, rekordzie lub pliku przez indeksy, wskaźniki i indeksy tabel?
   - Czy sprawdzana jest poprawność i kompletność zaimportowanych danych i argumentów wejściowych?
   - Czy wszystkie zmienne wyjściowe mają przypisane wartości?
   - Czy w każdej instrukcji używany jest właściwy element danych?
   - Czy każdy przydział pamięci jest zwalniany?
   - Czy w przypadku dostępu do urządzeń zewnętrznych stosowane są limity czasu lub mechanizmy

obsługi błędów?

- Czy przed próbą dostępu do pliku kod sprawdza, czy plik istnieje?
- Czy w momencie zakończenia programu wszystkie pliki i urządzenia pozostają w prawidłowym stanie?

Dalsze przykłady list kontrolnych używanych w przeglądach kodu na różnych poziomach testowania można znaleźć w serwisie [Web-6].

6. Narzędzia testowe i automatyzacja testów — 195 minut

Słowa kluczowe

testowanie sterowane danymi, narzędzie do debagowania, narzędzie do posiewu usterek, narzędzie do testowania hiperłączy, testowanie oparte o słowa kluczowe, narzędzie do testów wydajnościowych, narzędzie nagrywająco-odtwarzające, analizator statyczny, narzędzie do wykonywania testu, narzędzie do zarządzania testami

Cele nauczania dotyczące narzędzi testowych i automatyzacji testów

6.1 Integracja i wymiana informacji między narzędziami testowymi

TTA-6.1.1 (K2) Kandydat potrafi omówić aspekty techniczne, które należy uwzględnić w przypadku używania wielu narzędzi.

6.2 Definiowanie projektu automatyzacji testów

TTA-6.2.1 (K2) Kandydat potrafi omówić czynności wykonywane przez technicznego analityka testowego podczas konfigurowania projektu automatyzacji testów.
TTA-6.2.2 (K2) Kandydat potrafi omówić różnice między automatyzacją sterowaną danymi i automatyzacją opartą o słowa kluczowe.
TTA-6.2.3 (K2) Kandydat potrafi omówić często występujące problemy techniczne, z powodu których w projektach automatyzacji nie udaje się uzyskać zaplanowanego zwrotu z inwestycji,
TTA-6.2.4 (K3) Kandydat potrafi utwórz tabelę słów kluczowych na podstawie danego procesu biznesowego.

6.3 Kategorie narzędzi testowych

TTA-6.3.1 (K2) Kandydat potrafi omówić zastosowanie narzędzi do posiewu usterek i wstrzykiwania błędów.
TTA-6.3.2 (K2) Kandydat potrafi omówić główne cechy narzędzi do testów wydajnościowych i narzędzi monitorujących oraz zagadnienia dotyczące ich wdrażania.
TTA-6.3.3 (K2) Kandydat potrafi przedstawić ogólne zastosowania narzędzi do testowania stron internetowych.
TTA-6.3.4 (K2) Kandydat potrafi omówić sposoby wspierania przez narzędzia koncepcji testowania opartego na modelu.
TTA-6.3.5 (K2) Kandydat potrafi omówić zastosowanie narzędzi używanych do obsługi testowania jednostkowego i procesu budowania wersji.

6.1 Integracja i wymiana informacji między narzędziami testowymi

Za wybór i integrację narzędzi odpowiada kierownik testów, jednak technicznemu analitykowi testowemu może zostać zlecone zweryfikowanie sposobu integracji narzędzia (lub zestawu narzędzi), tak aby można było zapewnić dokładne śledzenie danych pochodzących z różnych obszarów testowania, np. analizy statycznej, automatyzacji wykonywania testów i zarządzania konfiguracją. Ponadto, techniczny analityk testowy (o ile dysponuje umiejętnościami programistycznymi) może również brać udział w tworzeniu kodu przeznaczonego do integracji narzędzi, które nie zawierają gotowych mechanizmów integracji.

W idealnym przypadku w zestawie narzędzi informacje nie powinny być powielane w różnych narzędziach. Przechowywanie skryptów testowych zarówno w bazie zarządzania testami, jak i w systemie zarządzania konfiguracją, jest bardziej czasochłonne, rośnie też wówczas prawdopodobieństwo popełnienia błędu. Lepszym rozwiązaniem jest zastosowanie systemu zarządzania testami zawierającego moduł zarządzania konfiguracją lub możliwego do zintegrowania z narzędziem do zarządzania konfiguracją używanym już w danej organizacji. Dobrze zintegrowane narzędzia do śledzenia defektów i do zarządzania testami pozwalają testerom na

zgłaszanie defektów w trakcie wykonywania przypadków testowych, bez konieczności opuszczania narzędzia do zarządzania testami. Z kolei dobrze zintegrowane narzędzia do analizy statycznej powinny być w stanie zgłaszać wykryte incydenty i ostrzeżenia bezpośrednio do systemu zarządzania defektami, choć opcja ta powinna być konfigurowalna ze względu na możliwość wygenerowania dużej liczby ostrzeżeń.
Zakup narzędzi testowych od jednego dostawcy nie oznacza automatycznie, że narzędzia te będą współpracować ze sobą we właściwy sposób. Preferowanym podejściem do integracji narzędzi jest podejście skoncentrowane na danych. Dane muszą być wymieniane bez interwencji użytkownika, we właściwym czasie, z gwarancją poprawności i z możliwością ich odzyskiwania w przypadku wystąpienia błędów. Spójny interfejs użytkownika jest przydatnym elementem, jednak najważniejszymi celami integracji narzędzi powinny być rejestrowanie, przechowywanie, ochrona i prezentacja danych.

Organizacja powinna ocenić koszty automatyzacji wymiany informacji i zestawić je z ryzykiem utraty informacji lub braku synchronizacji wynikającym z niezbędnych działań wykonywanych ręcznie. Integracja może być zadaniem kosztownym i trudnym, dlatego należy uwzględnić ten problem w ogólnej strategii zastosowania narzędzi.

Niektóre zintegrowane środowiska programistyczne mogą uprościć proces integracji między narzędziami, które są przygotowane do działania w takich środowiskach. Pozwala to ujednolicić wygląd i zachowanie narzędzi korzystających ze wspólnej platformy. Podobieństwo interfejsu użytkownika nie gwarantuje jednak bezproblemowej wymiany informacji między poszczególnymi komponentami. Może się okazać, że do przeprowadzenia integracji niezbędne jest opracowanie odpowiedniego kodu.

6.2 Definiowanie projektu automatyzacji testów

Narzędzia testowe, a zwłaszcza narzędzia do automatyzacji testów, muszą mieć odpowiednią architekturę i zostać bardzo dobrze zaprojektowane, aby ich zastosowanie było opłacalne. Efektem wdrożenia strategii automatyzacji testów bez opracowania właściwej architektury będzie zestaw narzędzi kosztowny w utrzymaniu i niewystarczający do realizacji zamierzonych celów, a uzyskanie zakładanego zwrotu z inwestycji okaże się niemożliwe.

Projekt automatyzacji testów należy traktować jak projekt programistyczny. Oznacza to konieczność opracowania dokumentów architektury i projektów szczegółowych, wykonania przeglądów projektu i kodu, przetestowania komponentów i przetestowania integracji modułów, a także przeprowadzenia końcowych testów systemowych. Zastosowanie niestabilnego lub niepoprawnego kodu automatyzacji testów może niepotrzebnie wydłużyć lub skomplikować proces testowania.

Techniczny analityk testowy wykonuje wiele zadań związanych z automatyzacją testów. Należą do nich:

- Ustalenie osób odpowiedzialnych za wykonanie testów;
- Wybór odpowiedniego narzędzi dla organizacji, określenie harmonogramu, kwalifikacji zespołu i wymagań dotyczących pielęgnowalności (może to w szczególności oznaczać decyzję o opracowaniu własnego narzędzia zamiast dokonywania zakupu);
- Zdefiniowanie wymagań dotyczących interfejsów między narzędziem do automatyzacji i innymi systemami, np. narzędziem do zarządzania testami i narzędziem do zarządzania defektami;
- Wybór podejścia do automatyzacji, tj. opartego o słowa kluczowe lub sterowanego danymi (patrz punkt 6.2.1);
- Określenie kosztów wdrożenia (w tym szkoleń) wspólnie z kierownikiem testów;

- Określenie harmonogramu projektu automatyzacji i zaplanowanie czasu na utrzymanie systemu;
- Przeszkolenie analityków testowych i analityków biznesowych w zakresie korzystania z automatyzacji i sposobów dostarczania danych;
- Określenie sposobu wykonywania testów automatycznych;
- Określenie sposobów łączenia rezultatów testów automatycznych z rezultatami testów manualnych.

Działania te i wynikające z nich decyzje mają wpływ na skalowalność i pielęgnowalność rozwiązania do automatyzacji. Należy poświęcić wystarczającą ilość czasu na przeanalizowanie dostępnych opcji, narzędzi i technologii oraz zrozumienie planów organizacji na przyszłość. Niektóre z tych działań wymagają dokładniejszej analizy, szczególnie w trakcie podejmowania decyzji. Zostało to bardziej szczegółowo opisane w kolejnych punktach.

6.2.1 Wybór podejścia do automatyzacji

Automatyzacja testów nie ogranicza się do testowania za pośrednictwem graficznego interfejsu użytkownika. Istnieją narzędzia do automatyzacji testów na poziomie interfejsu API, interfejsu wiersza poleceń i innych punktów styku z testowanym oprogramowaniem. Jedną z pierwszych decyzji, jakie musi podjąć techniczny analityk testowy, jest wybór interfejsu, z którego najlepiej będzie skorzystać w trakcie automatyzacji testów.

Jednym z problemów związanych z testowaniem za pośrednictwem graficznego interfejsu użytkownika jest możliwość wprowadzania w nim zmian w miarę rozwoju oprogramowania. Może to znacznie zwiększyć nakład pracy związany z utrzymaniem kodu do automatyzacji testów, w zależności od sposobu zaprojektowania tego kodu. Na przykład, automatyczne przypadki testowe (często nazywane skryptami testowymi) utworzone za pomocą funkcji nagrywania i odtwarzania mogą po zmianie interfejsu nie działać zgodnie z wymaganiami. Dzieje się tak dlatego, że w nagranym skrypcie zapisane są interakcje z obiektami graficznymi wykonane przez testera po ręcznym uruchomieniu oprogramowania, natomiast 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, a powstały skrypt jest następnie modyfikowany w celu zwiększenia pielęgnowalności kodu (np. poprzez zastąpienie fragmentów skryptu wywołaniami funkcji wielokrotnego użytku).

Dane używane w poszczególnych testach mogą być zależne od testowanego oprogramowania, chociaż wykonywane kroki są w zasadzie identyczne (np. podczas testowania obsługi błędów w polu wejściowym za pomocą wprowadzania wielu niepoprawnych wartości i weryfikacji błędu związanego z każdą wartością). Opracowywanie i utrzymywanie odrębnych skryptów testów automatycznych dla każdej testowanej wartości jest nieefektywne. Często stosowanym rozwiązaniem technicznym tego problemu jest przeniesienie danych ze skryptów do zewnętrznej składnicy, na przykład arkusza kalkulacyjnego lub bazy danych. Tworzy się funkcje uzyskujące dostęp do konkretnych danych dla każdego wykonania skryptu testowego, dzięki czemu jeden skrypt może obsłużyć zestaw danych testowych zawierający wartości wejściowe i oczekiwane wartości wynikowe (np. wartości wyświetlane w polu tekstowym lub komunikaty o błędach). Takie podejście nazywamy testowaniem sterowanym danymi. Opracowuje się w nim skrypt testowy, który przetwarza dostarczone dane, a także jarzmo testowe i infrastrukturę niezbędną do uruchomienia skryptu lub zestawu skryptów. Faktyczne dane przechowywane w arkuszu lub bazie danych są tworzone przez analityków testowych, którzy znają funkcje biznesowe realizowane przez oprogramowanie. Taki podział pracy pozwala osobom odpowiedzialnym za opracowanie skryptów testowych (np. technicznym analitykom testowym) skoncentrować się na

implementacji inteligentnych skryptów automatyzacji, podczas gdy właścicielem samego testu pozostaje analityk testowy. W większości przypadków za uruchomienie skryptów testowych po zaimplementowaniu i przetestowaniu automatyzacji odpowiada właśnie analityk testowy.

W innym podejściu, nazywanym testowaniem opartym o słowa kluczowe lub o słowa akcji, dodatkowo oddziela się działania, które mają zostać wykonane na dostarczonych danych, od samego skryptu testowego [Buwalda01]. Aby uzyskać taki stopień rozdzielenia, eksperci merytoryczni (np. analitycy testowi) tworzą metajęzyk wysokiego poziomu, który ma raczej charakter opisowy, a nie służy do bezpośredniego uruchamiania kodu. Każda instrukcja tego języka opisuje pełny lub częściowy proces biznesowy z danej dziedziny, który może wymagać przetestowania. Na przykład, wśród słów kluczowych związanych z procesem biznesowym mogą się znaleźć słowa „Zaloguj", „Utwórz Użytkownika" i „Usuń Użytkownika". Słowo kluczowe opisuje działanie wysokiego poziomu wykonywane w dziedzinie aplikacji. Można również zdefiniować działania niższego poziomu opisujące interakcje z interfejsem oprogramowania, takie jak „Kliknij Przycisk", „Wybierz Z Listy" albo „Przejdź Drzewo". Służą one to testowania funkcji graficznego interfejsu użytkownika, które nie odpowiadają bezpośrednio słowom kluczowym używanym w procesie biznesowym.

Po zdefiniowaniu słów kluczowych i używanych danych osoba odpowiedzialna za automatyzację testów (np. techniczny analityk testowy) tłumaczy słowa kluczowe związane z procesem biznesowym i działania niższego poziomu na kod automatyzacji testów. Słowa kluczowe i działania, a także używane w testach dane, można zapisać w arkuszach lub wprowadzić za pomocą konkretnych narzędzi obsługujących automatyzację testów opartą o słowa kluczowe. Środowisko automatyzacji testów implementuje słowa kluczowe w postaci zestawu wykonywalnych funkcji lub skryptów. Narzędzia odczytują przypadki testowe zapisane za pomocą słów kluczowych i wywołują odpowiednie funkcje testowe lub skrypty, które je implementują. Skrypty wykonywalne są zbudowane modułowo, aby łatwo je było odwzorowywać na konkretne słowa kluczowe. Do zaimplementowania takich modułowych skryptów konieczna jest umiejętność programowania.

Rozdzielenie wiedzy na temat logiki biznesowej od zadań programowania niezbędnego do implementacji skryptów automatyzacji pozwala w optymalny sposób wykorzystać kwalifikacje osób biorących udział w testach. Techniczny analityk testowy, będący osobą odpowiedzialną za automatyzację, może skutecznie wykorzystać swoje kompetencje w zakresie programowania i nie musi stać się ekspertem merytorycznym w wielu obszarach biznesowych.
Oddzielenie kodu od danych podlegających modyfikacjom pozwala odizolować proces automatyzacji od wprowadzanych zmian, poprawić ogólną pielęgnowalność kodu i zwiększyć wartość zwrotu z inwestycji w automatyzację.

W ramach każdego projektu automatyzacji testów należy przewidzieć awarie oprogramowania i określić sposób ich obsługi. Jeśli wystąpi awaria, osoba odpowiedzialna za automatyzację musi podjąć decyzję dotyczącą działania oprogramowania. Czy awarię należy zarejestrować i kontynuować testy? Czy testy należy przerwać? Czy można obsłużyć wystąpienie awarii za pomocą konkretnego działania (np. kliknięcia przycisku w oknie dialogowym) albo poprzez dodanie opóźnienia do testu? Nieobsłużone awarie oprogramowania mogą nie tylko spowodować problem w wykonywanym w danym momencie teście, ale wpłynąć także na rezultaty kolejnych testów.

Istotne jest także uwzględnienie stanu, w jakim system znajduje się na początku i na końcu testowania. Może okazać się konieczne przywrócenie systemu do zdefiniowanego stanu po zakończeniu wykonywania testów. Zestaw testów automatycznych da się dzięki temu wielokrotnie uruchamiać bez ręcznej interwencji użytkownika, który musi przywracać system do znanego stanu. Aby to osiągnąć, testy automatyczne powinny, na przykład, usuwać utworzone dane lub zmieniać status rekordów w bazie. Środowisko automatyzacji

powinno zagwarantować, że zakończenie testów odbędzie się w poprawny sposób (np. nastąpi wylogowanie).

6.2.2 Modelowanie procesów biznesowych na potrzeby automatyzacji

Aby wdrożyć podejście do automatyzacji testów oparte o słowa kluczowe, należy przeprowadzić modelowanie procesów biznesowych, które mają zostać przetestowane, w języku słów kluczowych wysokiego poziomu. Język taki powinien być intuicyjny dla użytkowników, którymi prawdopodobnie będą analitycy testowi biorący udział w projekcie.

Słowa kluczowe w ogólności służą do opisu wysokopoziomowych interakcji biznesowych z systemem. Na przykład, słowo kluczowe „Anuluj zamówienie" może oznaczać konieczność sprawdzenia, czy zamówienie istnieje, zweryfikowania praw dostępu osoby, która zgłosiła żądanie anulowania, wyświetlenia zamówienia oraz żądania potwierdzenia anulowania. Przypadki testowe specyfikuje analityk testowy, korzystając z sekwencji słów kluczowych (np. „Zaloguj", „Wybierz Zamówienie", „Anuluj Zamówienie") oraz odpowiednich danych. Poniżej przedstawiono prostą tabelę wartości wejściowych opartą o słowa kluczowe. Umożliwia ona przetestowanie zdolności oprogramowania do dodawania, resetowania i usuwania kont użytkowników:

Słowo kluczowe	**Użytkownik**	**Hasło**	**Rezultat**
Dodaj użytkownika	Użytkownik1	Hasło1	Komunikat: Użytkownik został dodany
Dodaj użytkownika	@Rec34	@Rec35	Komunikat: Użytkownik został dodany
Resetuj hasło	Użytkownik1	Witaj	Komunikat potwierdzający: Hasło zostało zresetowane
Usuń użytkownika	Użytkownik1		Komunikat: Niepoprawny użytkownik lub hasło
Dodaj użytkownika	Użytkownik3	Hasło3	Komunikat: Użytkownik został dodany
Usuń użytkownika	Użytkownik2		Komunikat: Nie znaleziono użytkownika

Skrypt automatyzacji korzystający z tej tabeli wyszukuje wartości, których należy użyć. Na przykład, jeśli dotrze do wiersza zawierającego słowo kluczowe „Usuń użytkownika", wymagana jest tylko nazwa użytkownika. Dodanie nowego użytkownika wiąże się z koniecznością podania nazwy i hasła. Można także odwoływać się do wartości wejściowych pochodzących ze składnicy danych, tak jak w przypadku drugiego słowa kluczowego „Dodaj Użytkownika", przy którym podano odwołanie do danych, a nie same dane. Zapewnia to większą elastyczność dostępu do danych, które mogą ulegać zmianom w miarę wykonywania testów. Dzięki temu podejście sterowane danymi można połączyć z użyciem słów kluczowych.

Należy zwrócić uwagę na następujące kwestie:

- Im bardziej szczegółowe są słowa kluczowe, tym bardziej szczegółowe scenariusze można uwzględnić, jednak trudniej wówczas zarządzać językiem wysokiego poziomu.
- Umożliwienie analitykom testowym specyfikowania działań niskiego poziomu („Kliknij przycisk", „Wybierz z listy" itp.) pozwala obsłużyć w testach tego rodzaju większą liczbę różnych sytuacji. Jednak ponieważ działania te bezpośrednio wiążą się z graficznym interfejsem użytkownika, testy wymagają wykonania dodatkowych czynności pielęgnacyjnych po wprowadzeniu zmian w systemie.
- Zagregowane słowa kluczowe mogą uprościć projektowanie, jednak także utrudnić utrzymanie środowiska. Na przykład, można zdefiniować sześć różnych słów kluczowych, które razem służą do utworzenia rekordu. Czy należy utworzyć jeszcze jedno słowo kluczowe, które wywołuje po kolei sześć pozostałych, aby uprościć to działanie?
- Niezależnie od czasu poświęconego na analizę, podczas tworzenia języka słów kluczowych może pojawić się konieczność dodania nowych słów kluczowych. Słowo kluczowe posiada dwa znaczenia, tj. logikę biznesową i funkcję automatyzacji, która ją wywołuje. Należy zatem utworzyć proces, który uwzględni oba znaczenia.

Automatyzacja testów oparta na słowach kluczowych może znacznie zmniejszyć koszty utrzymania, ale koszty zaprojektowania takich testów są większe, podobnie jak stopień skomplikowania. Ponadto, poprawne zaprojektowanie testów wymaga więcej czasu i jest niezbędne do uzyskania zakładanego zwrotu z inwestycji.

6.3 Kategorie narzędzi testowych

W tym punkcie przedstawiono informacje na temat narzędzi, których może użyć techniczny analityk testowy, a które nie zostały omówione w sylabusie analityka testowego dla poziomu zaawansowanego [ISTQB_ALTA_SYL] i sylabusie poziomu podstawowego [ISTQB_FL_SYL].

6.3.1 Narzędzia do posiewu usterek i wstrzykiwania błędów

Narzędzia do posiewu błędów są używane głównie na poziomie kodu i tworzą w usystematyzowany sposób usterki pewnego typu (lub ograniczonej liczby typów) w kodzie. Narzędzia rozmyślnie wprowadzają defekty do testowanych elementów w celu oceny jakości zestawów testowych (czyli możliwości wykrycia defektów). Proces wstrzykiwania błędów koncentruje się na testowaniu mechanizmu obsługi usterek wbudowanego w przedmiot testów, polegającym na stworzeniu nieprawidłowych warunków. Narzędzia do wstrzykiwania błędów umyślnie wprowadzają do oprogramowania niepoprawne wartości wejściowe, aby sprawdzić, czy poradzi ono sobie z obsługą usterki.

Oba typy narzędzi wykorzystywane są zwykle przez technicznych analityków testowych, choć mogą ich również używać programiści do testowania nowego kodu.

6.3.2 Narzędzia do testów wydajnościowych

Narzędzia do testów wydajnościowych mają dwie główne funkcje:

- generowanie obciążenia,
- pomiar i analiza odpowiedzi systemu na zadane obciążenie.

Generowanie obciążenia odbywa się poprzez zaimplementowanie zdefiniowanego wcześniej profilu

produkcyjnego (patrz punkt 4.5.4) w postaci skryptu. Skrypt może zostać zarejestrowany wstępnie dla jednego użytkownika (np. z wykorzystaniem narzędzia nagrywająco-odtwarzającego), a później zaimplementowany dla określonego profilu produkcyjnego przy użyciu narzędzia do testów wydajnościowych. Implementacja musi brać pod uwagę zróżnicowanie danych w poszczególnych transakcjach (lub zbiorach transakcji).

Narzędzia do testów wydajnościowych generują obciążenie przez symulowanie dużej liczby użytkowników („wirtualnych") realizujących określone profile produkcyjne i generujących dane wejściowe o ustalonej wielkości. Skrypty testujące wydajność często odtwarzają interakcję użytkowników z systemem na poziomie protokołu komunikacyjnego, a nie przez symulowanie interakcji przy użyciu interfejsu GUI, jak to ma miejsce w przypadku standardowych skryptów automatyzacji. Zwykle pozwala to zmniejszyć liczbę odrębnych „sesji" niezbędnych w trakcie testowania. Niektóre narzędzia do generowania obciążenia sterują działaniem aplikacji również za pomocą interfejsu użytkownika, aby dokładniej zmierzyć czas odpowiedzi systemu w warunkach obciążenia.

Narzędzia do testów wydajnościowych dokonują wielu pomiarów, które można przeanalizować w trakcie wykonywania testów lub po ich zakończeniu. Typowe metryki i raporty to:

- liczba symulowanych użytkowników;
- liczba i typ transakcji generowanych przez symulowanych użytkowników oraz częstotliwość pojawiania się transakcji;
- czasy odpowiedzi na poszczególne żądania transakcji zgłaszane przez użytkowników;
- raporty i wykresy prezentujące obciążenie i czasy reakcji;
- raporty dotyczące wykorzystania zasobów (np. w funkcji czasu, z podaniem wartości minimalnych i maksymalnych).

Głównymi czynnikami, które należy wziąć pod uwagę podczas wdrażania narzędzi do testów wydajnościowych, są:

- sprzęt i przepustowość sieci wymagane do wygenerowania obciążenia;
- kompatybilność narzędzia z protokołami komunikacyjnymi używanymi przez testowany system;
- elastyczność narzędzia pozwalająca na implementowanie różnych profili produkcyjnych;
- wymagane funkcje monitorowania, analizy i raportowania.

Ze względu na dużą pracochłonność wymaganą do wytworzenia narzędzi do testów wydajnościowych nie są one zwykle opracowywane w ramach projektu, tylko nabywane. Może jednak okazać się zasadne opracowanie konkretnego narzędzia do testowania wydajności, jeśli z powodu ograniczeń technicznych nie da się skorzystać z gotowego produktu lub jeśli profil obciążenia i realizowane przez niego funkcje są stosunkowo proste.

6.3.3 Narzędzia do testowania stron internetowych

Dostępnych jest wiele wyspecjalizowanych narzędzi komercyjnych i narzędzi typu *open source*, przeznaczonych do testowania stron internetowych. Na poniższej liście przedstawiono zastosowania niektórych najczęściej używanych narzędzi z tej kategorii:

- narzędzia do testowania hiperłączy służące do skanowania serwisu i wykrywania brakujących i niepoprawnych hiperłączy;
- narzędzia do sprawdzania kodu HTML i XML, weryfikujące zgodność kodu ze standardami dla poszczególnych stron tworzonych w serwisie internetowym;

- symulatory obciążenia służące do testowania reakcji serwera na podłączenie się dużej liczby użytkowników;
- lekkie narzędzia do wykonywania testów automatycznych, współpracujące z różnymi przeglądarkami;
- narzędzia do skanowania serwera, poszukujące niepowiązanych („osieroconych") plików;
- narzędzia do sprawdzania pisowni obsługujące HTML;
- narzędzia do sprawdzania arkuszy CSS;
- narzędzia do sprawdzania przypadków naruszenia norm, np. standardów dostępności (Section 508 w Stanach Zjednoczonych i M/376 w Europie);
- narzędzia do wykrywania różnych rodzajów problemów dotyczących zabezpieczeń.

Dobrym źródłem narzędzi *open source* do testowania stron internetowych jest serwis [Web-7]. Organizacja odpowiedzialna za jego zawartość ustala standardy internetowe i udostępnia różnorodne narzędzia do wyszukiwania niezgodności z tymi standardami.

Niektóre narzędzia zawierające robota indeksującego (ang. *web spider*) mogą również dostarczać informacje na temat wielkości stron i czasu niezbędnego do ich pobrania, a także dostępności poszczególnych stron (i np. występowaniu błędu 404 protokołu HTTP). To przydatne informacje dla programistów, administratorów i testerów.

Analitycy testowi i techniczni analitycy testowi używają takich narzędzi głównie w trakcie testów systemowych.

6.3.4 Narzędzia wspomagające testowanie oparte na modelu

Testowanie oparte na modelu (ang. *model-based testing*, MBT) to technika, w której do opisu pożądanego zachowania systemu sterowanego oprogramowaniem używany jest pewien model formalny, na przykład automat skończony. Komercyjne narzędzia MBT (zob. [Utting 07]) często udostępniają mechanizm umożliwiający użytkownikom „uruchomienie" modelu. Interesujące wątki z uruchomienia modelu można zapisać i wykorzystać jako przypadki testowe. Inne wykonywalne modele, np. sieci Petriego i diagramy stanów, również dopuszczają ten rodzaj testowania. Modeli (i narzędzi) MBT można użyć do wygenerowania dużych zbiorów odrębnych przebiegów wykonania.

Narzędzia tego typu pozwalają ograniczyć bardzo dużą liczbę możliwych ścieżek, które mogą zostać wygenerowane w ramach modelu.

Testowanie przy użyciu tych narzędzi może dostarczyć innej perspektywy na testowane oprogramowanie. Można w ten sposób wykryć defekty, które nie zostały zauważone w trakcie testowania funkcjonalnego.

6.3.5 Narzędzia do testowania komponentów i budowania wersji

Narzędzia do testowania komponentów i budowania wersji są przeznaczone dla programistów, jednak w wielu przypadkach są używane i utrzymywane przez technicznych analityków testowych, zwłaszcza w kontekście projektów stosujących metodyki zwinne.

Narzędzia do testowania komponentów są często związane z językiem używanym do programowania danego modułu. Na przykład, jeśli językiem programowania jest Java, do automatyzacji testów jednostkowych można użyć środowiska JUnit. Z wieloma innymi językami związane są specjalne narzędzia testowe; są one ogólnie nazywane środowiskami xUnit. W środowiskach tych obiekty testowe generowane są dla każdej utworzonej klasy, co pozwala uprościć zadania wykonywane przez programistów podczas automatyzacji testowania

komponentów.

Narzędzia do debagowania obsługują ręczne testowanie komponentów na bardzo niskim poziomie, pozwalając programistom i technicznym analitykom testowym zmieniać wartości zmiennych w czasie wykonywania i przechodzić przez kolejne linie kodu podczas testowania. Narzędzia do debagowania ułatwiają również programistom lokalizowanie i identyfikowanie problemów w kodzie, gdy zespół testowy przekaże informację o występowaniu awarii.

Narzędzia do automatyzacji budowania wersji często dają możliwość automatycznego uruchomienia procesu budowania nowej wersji po zmodyfikowaniu komponentu. Po zakończeniu takiej operacji inne narzędzia automatycznie uruchamiają testy komponentów. Taki poziom automatyzacji procesu budowania jest często spotykany w środowiskach ciągłej integracji.

Poprawnie skonfigurowany zestaw narzędzi tego typu może mieć istotny wpływ na poprawę jakości wersji przekazywanych do testowania. Jeśli zmiany wprowadzone przez programistę spowodują się wprowadzenie błędów regresji w danej wersji, zwykle będzie wiązać się to z niepowodzeniem wykonania niektórych testów automatycznych. Zanim wersja zostanie przekazana do środowiska testowego, programista podejmuje próbę wyjaśnienia przyczyn i rozwiązania problemów.

7. Dokumenty pomocnicze

7.1 Standardy

W odpowiednich rozdziałach niniejszego dokumentu wspomniano następujące standardy:

- [ANSI/IEEE Std 1471-2000] Recommended Practice for Architectural Description of Software-Intensive Systems (rozdział 5)
- IEC-61508 (rozdział 2)
- [ISO25000] ISO/IEC 25000:2005, Software Engineering — Software Product Quality Requirements and Evaluation (SQuaRE) (rozdział 4)
- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering — Software Product Quality (rozdział 4)
- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992 (rozdział 2)

7.2 Dokumenty ISTQB

- [ISTQB_AL_OVIEW] ISTQB: Przegląd poziomu zaawansowanego, wersja 2012
- [ISTQB_ALTA_SYL] ISTQB: Sylabus dla poziomu zaawansowanego — analityk testowy, wersja 2012
- [ISTQB_FL_SYL] ISTQB: Sylabus dla poziomu podstawowego, wersja 2011
- [ISTQB_GLOSSARY] ISTQB: Słownik terminów używanych w testowaniu oprogramowania, wersja 2.2, 2012

7.3 Książki

[Bass03]: Len Bass, Paul Clements, Rick Kazman „Software Architecture in Practice (2nd edition)”, Addison-Wesley 2003] ISBN 0-321-15495-9 (wydanie polskie: „Architektura oprogramowania w praktyce”, Wydanie II, Helion 2011, ISBN 9788324633029)
[Bath08]: Graham Bath, Judy McKay, „The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6
[Beizer90]: Boris Beizer, „Software Testing Techniques Second Edition”, International Thomson Computer Press, 1990, ISBN 1-8503-2880-3
[Beizer95]: Boris Beizer, „Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4
[Buwalda01]: Hans Buwalda, „Integrated Test Design and Automation”, Addison-Wesley Longman, 2001, ISBN 0-201-73725-6
[Copeland03]: Lee Copeland, „A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X
[Gamma94]: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-201-63361-2
[Jorgensen07]: Paul C. Jorgensen, „Software Testing, a Craftsman’s Approach third edition”, CRC press, 2007, ISBN-13:978-0-8493-7475-3
[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; „Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4
[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, „TMap Next for result-driven

testing"; UTN Publishers, 2006, ISBN: 90-72194-79-9
[McCabe76]: Thomas J. McCabe, „A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, grudzień 1976 r. PP 308-320
[NIST96]: Arthur H. Watson and Thomas J. McCabe, „Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, przygotowano w ramach kontraktu NIST numer 43NANB517266, wrzesień 1996 r.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, „The Web-Testing Handbook", STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07]: 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

7.4 Inne dokumenty pomocnicze

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.testingstandards.co.uk
[Web-2] http://www.nist.gov (NIST — National Institute of Standards and Technology)
[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4] http://portal.acm.org/citation.cfm?id=308798
[Web-5] http://www.processimpact.com/pr_goodies.shtml
[Web-6] http://www.ifsq.org
[Web-7] http://www.W3C.org

Rozdział 4: [Web-1], [Web-2]
Rozdział 5: [Web-3], [Web-4], [Web-5], [Web-6]
Rozdział 6: [Web-7]