

Notatki z kursu Testowanie oprogramowania

Małgorzata Dymek

2019/20, semestr zimowy

1 Wprowadzenie.

1.1 Definicje.

Pomyłka - człowiek robi coś źle.

Defekt (usterka, bug, fault) - statyczny defekt w kodzie (lub dokumentacji), skutek pomyłki człowieka.

Błąd – nieprawidłowy stan wewnętrzny programu np. licznik pętli ustawiony na drugim zamiast pierwszym elemencie tablicy.

Awaria (failure) – widoczne, nieprawidłowe działanie oprogramowania np. crash systemu, zwrócenie nieprawidłowego wyniku, komunikat o błędzie.

Incydent – wydarzenie, które wymaga analizy

Suita testowa - ????

Paradoks pestycydów - jeśli ciągle uruchamiamy te same testy to tracą one zdolność do znajdowania nowych defektów.

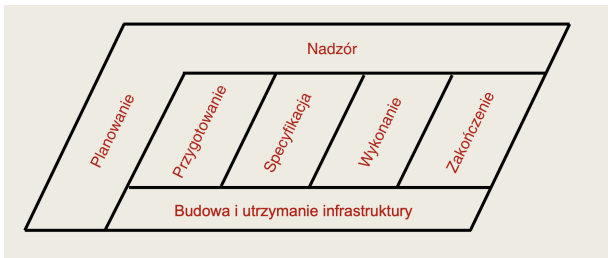
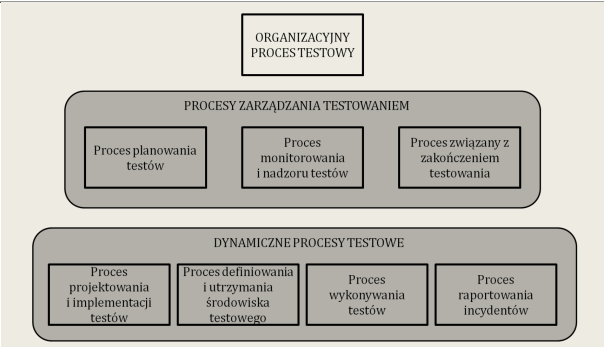
Ewaluacja - na początku i na końcu

- **Walidacja** - dokonywana w celu potwierdzenia zgodności z założonymi celami użycia. (*are we building the right thing?*)
- **Weryfikacja** - sprawdzająca, czy produkt danej fazy spełnia wymagania (zwykle techniczne) ustalone podczas poprzedniej fazy. (*are we building the thing right?*)

Testowanie to nie debugowanie

- **Testowanie** znajduje awarie, sprawdza czy usterki zostały usunięte.
- **Debugowanie** lokalizuje miejsce usterki powodującej awarię i ją usuwa

2 Testowanie w cyklu życia.

Modele procesu testowego	
TMAP lifecycle model	ISO 29119
 <p>Proces jest generyczny, tzn. może być stosowany do wszystkich poziomów i typów testów. Każda faza dzieli się na określone czynności.</p>	

2.1 Poziomy testów.

Poziom testów określa **sposób** testowania ze względu na **postać** testowanego obiektu w kontekście cyklu życia (co testujemy?).

TESTY JEDNOSTKOWE	
Podstawa testów	wymagania na moduły, projekt szczegółowy, kod
Typowe obiekty	moduły, programy, funkcje, klasy, procedury

TESTY INTEGRACYJNE	
Podstawa testów	projekt systemu, architektura, przypadki użycia
Typowe obiekty	interfejsy, podsystemy, konfiguracje systemów

Strategie testów integracyjnych:

- **top-down**: testujemy moduły "w dół", w kolejności w jakiej się wywołują
- **bottom-up**: testujemy moduły od ostatniego wywoływanego
- **funkcjonalne**: testujemy wywołania wewnątrz funkcjonalności
- **sekwencja przeprowadzania transakcji**
- **big-bang**: wszystko integrowane i testowane naraz

TESTY SYSTEMOWE	
Podstawa testów	wymagania na system, przypadki użycia, specyfikacja funkcjonalna, raporty analizy ryzyka
Typowe obiekty	system, podręczniki użytkownika i operatora, konfiguracja systemu
TESTY AKCEPTACYJNE	
Podstawa testów	wymagania użytkownika, wymagania na system, przypadki użycia, procesy biznesowe, raporty analizy ryzyka
Typowe obiekty	Procesy biznesowe w pełni zintegrowanego systemu, procesy operacyjne i utrzymania systemu, procedury, raporty, dane konfiguracyjne

Typowe formy testów akceptacyjnych:

- testy akceptacyjne użytkownika (UAT) – sprawdzenie gotowości do użycia
- testy operacyjne (OAT) – akceptacja przez administratora systemu (testy backupu, przywracania systemu, zarządzania użytkownikami, utrzymania, migracji danych, bezpieczeństwa itp.)
- testy akceptacyjne wymagane kontraktem/regulacjami
- testy alfa, beta (polowe)
 - alfa: przeprowadzane u producenta, ale nie przez zespół deweloperski
 - beta: przeprowadzane u klienta przez klienta/potencjalnego użytkownika

2.2 Typy testów.

Typ testów to zbiór czynności testowych właściwych dla weryfikacji systemu w oparciu o konkretny powód lub cel testów (**jak testujemy?**).

Testowanie funkcjonalne	Testowanie нефункционалне
Funkcja wykonywana przez oprogramowanie - co system robi.	Niefunkcjonalna charakterystyka jakościowa wyrażalna ilościowo, np. niezawodność czy użyteczność - jak system działa.

Testowanie strukturalne	Retesty i testy regresji
<ul style="list-style-type: none"> • oparte na strukturze • zwykle wykonywane po testach czarnoskrzynkowych, aby sprawdzić stopień przetestowania i wyrazić go ilościowo (pokrycie) 	<p>Związany ze zmianą, tzn. potwierdzenie usunięcia defektów (retesty) oraz poszukiwanie niezamierzonych zmian (regresja) - związane ze zmianami</p> <ul style="list-style-type: none"> • regresja = zjawisko pogarszania się jakości systemu na skutek wprowadzanych w nim zmian • retest = przetestowanie naprawionego fragmentu systemu

2.3 Statyczne techniki testowania

Ręczne sprawdzanie (**przeglądy**) i automatyczna analiza (**analiza statyczna**) kodu lub dokumentacji bez uruchamiania kodu, ale zwykle z użyciem narzędzi!

PRZEGLĄDY	
Definicja	Korzyści z przeglądów
<ul style="list-style-type: none"> • Sposoby ręcznego testowania oprogramowania (np. kodu, dokumentacji) • Pozwalają wykryć defekty wcześnie w cyklu życia (np. w wymaganiach), • Aktywność manualna, ale może być wsparta narzędziami 	<ul style="list-style-type: none"> • wczesne wykrycie i naprawa defektów • doskonalenie jakości tworzonego kodu • redukcja kosztu i czasu testów • mniej defektów (w późniejszych fazach) • ulepszenie komunikacji

Rodzaje przeglądów		
Typ przeglądu	Charakterystyka	Cel
nieformalny	brak formalnego procesu, może przybrać formę programowania w parach lub nieformalnej rozmowy	tani sposób na osiągnięcie niewielkich korzyści
przejrzenie	prowadzone przez autora; opcjonalne przygotowanie przed spotkaniem, opcjonalny raport z przeglądu	uczenie się, zrozumienie, znajdowanie usterek
techniczny	przeszkolony moderator, przygotowanie przed spotkaniem, zdefiniowany proces postępowania	podjęcie decyzji, ocena alternatyw, szukanie usterek, rozw. probl. technicznych
inspekcja	przeszkolony moderator, wyróżnione role i metryki, formalny proces, przygotowanie przed spotkaniem, formalny proces kontroli napraw	wyszukiwanie usterek

INSPEKCJE	
Proces	Role
<ul style="list-style-type: none"> • Planowanie • Rozpoczęcie • Przygotowanie indywidualne • Kontrola/ocena/zapis wyników • Poprawki • Zakończenie 	<ul style="list-style-type: none"> • kierownik • moderator • autor • przeglądający • protokolant

2.4 Projektowanie testów.

Warunek testowy (test condition) – **element** lub zdarzenie które może być **sprawdzone** przez jeden lub więcej **przypadków testowych** (np. funkcja, transakcja, atrybut jakościowy, element strukturalny).

Element pokrycia (coverage item) – element lub zdarzenie używane jako **podstawa dla pokrycia testu** (np. przejście w maszynie stanów, instrukcja).

Przypadek testowy - **generyczna struktura**: warunki początkowe, działanie i wynik = wejście, przetwarzanie, wyjście.

3 Czarnoskrzynkowe techniki projektowania testów.

to procedury wywodzenia/wybierania przypadków testowych.

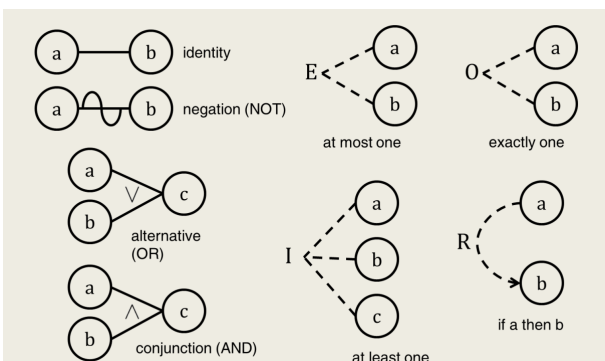
Technika	Element pokrycia
Podział na klasy równoważności	Klasa równoważności
Analiza wartości brzegowych	Wartości brzegowe
Drzewo klasyfikacji	Liść / kombinacja liści
Graf przyczynowo-skutkowy	Kombinacja przyczyn
Tablica decyzyjna	Kombinacja warunków
Testowanie maszyny stanowej	Przejście / sekwencja przejść
Graf przepływu sterowania	Ścieżka
CRUD	Cykl życia danej

Hipoteza błędu - każda technika projektowania testów zaprojektowana jest do wykrywania określonego typu awarii.

3.1 Techniki projektowania testów

oparte na specyfikacji	oparte na strukturze	oparte na doświadczeniu
<ul style="list-style-type: none"> • klasy równoważności • wartości brzegowe • tablice decyzyjne • grafy P-S • maszyna stanowa • drzewa klasyfikacji • przypadki użycia • testowanie losowe 	<ul style="list-style-type: none"> • pokrycie instrukcji • pokrycie decyzji • pokrycie warunków • pokrycie MC/DC • pokrycie przepływu danych • pokrycie pętli • pokrycie ścieżek • testowanie mutacyjne 	<ul style="list-style-type: none"> • testy eksploracyjne • zgadywanie błędów • ataki usterkowe • testy z listą kontrolną • testowanie ad hoc

Tablice decyzyjne	
Tablice decyzyjne <ul style="list-style-type: none"> • testowanie kombinacji warunków • pozwala na systematyczne sprawdzanie wszystkich kombinacji • ułatwia wykrywanie <ul style="list-style-type: none"> – brakującej specyfikacji – błędnej (sprzecznej) specyfikacji 	Minimalizacja tablicy - jeśli wszystkie kombinacje pewnych warunków dają te same akcje, możemy je scalać.

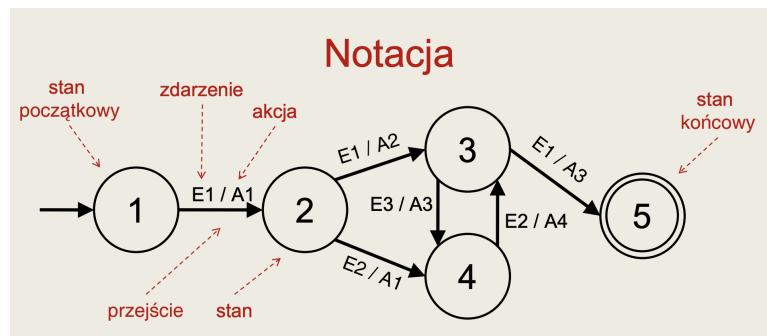
Grafy przyczynowo-skutkowe	
<ul style="list-style-type: none"> • stosowane w tych samych sytuacjach co tablice decyzyjne • graficzna reprezentacja zależności między przyczynami a skutkami • prosta transformacja na tablice decyzyjne 	

3.1.1 Model maszyny stanowej

Testowanie maszyny stanowej

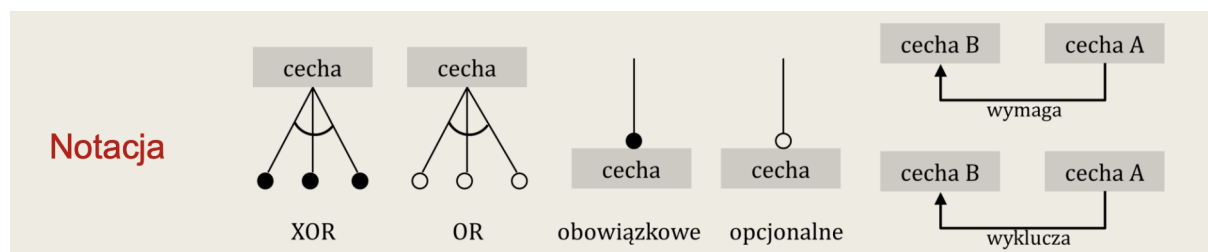
- reprezentacja możliwych stanów systemu i przejść między nimi
- metoda opisu dynamiki systemu

n-switch coverage - pokrycie przejść o **n** stanach pomiędzy stanem początkowym i końcowym. Żeby zidentyfikować pokrycie n-switchy zidentyfikuj wszystkie (n-1)-switche i ich rozszerzenia.



3.1.2 Drzewa klasyfikacji

- szczególna wersja metody **Category-Partition**
- graficzna reprezentacja systemu jako:
 - zestawu **cech**
 - ich **wartości**
 - ewentualnych **związków** między wartościami cech
- odmiana metody: **model cech** (ang. feature model)
- wykorzystywany jako model w podejściu SPL (Software Product Lines)
- wyprowadzanie testów wykorzystuje zwykle jedno z podejść kombinacyjnych



3.1.3 Testowanie kombinatoryczne

- stosowane gdy chcemy testować **kombinacje klas równoważności** różnych podziałów
- metody kombinatoryczne pozwalają na redukcję liczby testów

Pełne pokrycie kombinatoryczne = każda kombinacja klas wszystkich podziałów.

1-wise, Each Choice = każda klasa z każdego podziału ma być przetestowana przynajmniej raz. Liczba kombinacji = max ilości klas.

2-wise, Pairwise = każda para klas z dowolnych dwóch podziałów musi wystąpić przynajmniej raz. Minimalna liczba kombinacji jest NP-zupełna.

3.1.4 Testowanie losowe

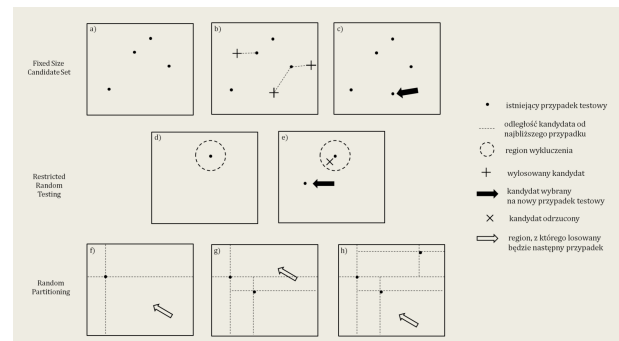
- wymaga możliwości **losowego wyboru elementu dziedziny**

- może być przeprowadzane manualnie, ale **zwykle automatyczne**
- można stosować, gdy trudno modelować dziedzinę wejściową

Automatyzacja testowania losowego.

Pełna automatyzacja testowania losowego jest możliwa gdy:

- można **automatycznie losować dane wejściowe**
- można **automatycznie określać oczekiwane wyjście** lub automatycznie porównywać wyjście ze specyfikacją
 - istnieje wyrocznia
 - interesuje nas tylko crash
 - łatwa weryfikacja wyjścia (sortowanie)
 - łatwo wygenerować wejście z wyjścia (np. pierwiastek/potęga)



3.1.5 Testowanie oparte na use-case'ach

- przypadek użycia opisuje **interakcję użytkownika z systemem**
- zwykle **wysokopoziomowy**, w postaci przepływu „end-to-end”
- testowanie oparte na przypadkach użycia **sprawdza poprawność działania systemu dla przepływów**: głównego i alternatywnych

Generowanie przypadków użycia:

1. Dla każdego przypadku użycia wygeneruj **pełny zbiór scenariuszy**.
2. Dla każdego scenariusza zidentyfikuj **przynajmniej 1 przypadek testowy** i warunki umożliwiające jego wykonanie.
3. Dla każdego przypadku testowego zidentyfikuj **dane**, dla których można przeprowadzić test.

3.1.6 Testowanie CRUD

- CRUD = **Create, Read, Update, Delete**
- metoda testowania **cyklu życia danych** (encji)
- cykl życia reprezentowany przy pomocy tzw. **macierzy CRUD**
 - wiersze = funkcje, kolumny = encje, przecięcie = operacja
 - dla każdej funkcji sprawdzamy które encje są wykorzystywane
 - a następnie, które akcje (C, R, U, D) są na nich przeprowadzane
- dwa rodzaje testów
 - **sprawdzanie kompletności (statyczny)**
 - * sprawdzenie, czy dla każdej encji występują wszystkie 4 operacje
 - * brak jakiegś akcji niekoniecznie oznacza błąd w systemie, ale powód tego braku powinien zostać wyjaśniony
 - **sprawdzanie spójności (dynamiczny)**
 - * sprawdza **integrację różnych funkcji**
 - * przypadki testowe konstruujemy zadając cały cykl życia encji:

- * każdy przypadek testowy zaczyna się od C i przechodzi do wszystkich możliwych U, kończąc na D; jeśli jest więcej możliwości C i D, tworzy się dodatkowe przypadki
- * po każdej akcji (C, U, D) występuje jedno lub kilka R – to sprawdza, czy encja została poprawnie przetworzona i jest użyteczna dla innych funkcji
- * dla każdej encji, wszystkie wystąpienia akcji (C, R, U i D) we wszystkich funkcjach powinny zostać pokryte przez przypadki testowe

4 Białoskrzynkowe techniki projektowania testów.

Testowanie oparte na **strukturze** oprogramowania lub systemu, np.:

Poziom testów	Przykład struktury
testy modułowe	kod: instrukcje, decyzje, rozgałęzienia, ścieżki
testy integracyjne	drzewo wywołań
testy systemowe	struktura menu, proces biznesowy, struktura strony www

Przykładowe metody **wizualizacji** struktury:

- **graf przepływu sterowania** (CFG, Control Flow Graph) lub danych
- **model procesu biznesowego** w BPML
- **diagram aktywności** (czynności) w UML

Elementy pokrycia = **elementy modelu** (krawędzie, decyzje, ścieżki itp.)

Pokrycie (dla metod białoskrzynkowych) **sprawdza jakość testów czarnoskrzynkowych**; następnie techniki białoskrzynkowe dostarczają testów w celu zwiększenia tego pokrycia.

4.1 Pokrycia grafowe

POKRYCIA

Syntaktyczne	"teoretyczne", wszystkie ścieżki grafu ignorując logikę poszczególnych węzłów.
Semantyczne	uwzględniające tę logikę.

Graf przepływu sterowania	Graf przepływu danych
<ul style="list-style-type: none"> • CFG - Control Flow Graph • graficzna reprezentacja przepływu sterowania <ul style="list-style-type: none"> – wierzchołki = bloki bazowe – krawędzie = przejścia między blokami • blok bazowy = sekwencja instrukcji taka, że jeśli wykona się pierwsza z nich, wszystkie pozostałe w obrębie bloku wykonają się również 	<ul style="list-style-type: none"> • oparty na grafie przepływu sterowania • zawiera informacje o operacjach na zmiennych • możliwe operacje: <ul style="list-style-type: none"> – d = definition, definicja – u = use, użycie – k = kill, zabicie <p>Dla każdego węzła B definiujemy zbiory $d(B)$, $u(B)$, $k(B)$ zawierające zmienne definiowane, używane lub zabijane w danym węźle.</p>

4.1.1 Pokrycie instrukcyjne

- elementy pokrycia: **instrukcje** kodu
- kryterium pokrycia instrukcyjnego: każda linia kodu jest wykonana przynajmniej raz w jakimś teście
- osiągnięcie 100% pokrycia jest często **nieosiągalne**

- **problem:** jak definiować linię kodu? (linia fizyczna? wykonywalna?)
- **problem:** czy rozważać pojedyncze instrukcje, czy bloki bazowe?

4.1.2 Pokrycie krawędziowe (branch testing)

- inna nazwa: **pokrycie przejść** między instrukcjami
- wymagane jest przejście po każdej **krawędzi CFG**, czyli testowany jest każdy **przepływ sterowania**
- Przy założeniu, że CFG ma co najmniej 1 krawędź, **pokrycie krawędziowe subsumuje pokrycie instrukcyjne, nie na odwrót.**

4.1.3 Pełne pokrycie ścieżek

- możliwe, gdy **nie ma pętli** lub **liczba iteracji** wszystkich pętli jest **ograniczona** z góry
- najczęstszy przypadek: **diagramy przepływu procesów**
- nawet dla kodu bez pętli liczba wszystkich ścieżek może **rosnąć wykładniczo** względem punktów decyzyjnych
- **złożoność cyklomatyczna CFG** - liczba ścieżek liniowo niezależnych.
- **algorytm wyznaczania ścieżek liniowo niezależnych**
 - wychodzimy od prostej ścieżki
 - jedna decyzja zmieniana w porównaniu z poprzednią ścieżką
 - po zmianie, będąc w węźle już kiedyś odwiedzionym, kontynuujemy wędrówkę wzdłuż tamtej ścieżki, (staramy się zmienić tak mało, jak to możliwe, w porównaniu z poprzednimi ścieżkami)
 - zwykle pętle iterujemy max. raz

4.1.4 Pokrycie przepływu danych

Wierzchołki: p, n ; Zmienne: v .

Ścieżka $p = (p_1, \dots, p_n)$ jest **def-czysta** ze względu na zmienną v , jeśli v nie jest definiowana w żadnym węźle ścieżki poza p_1 .

Ścieżka $p = (p_1, \dots, p_n)$ jest **du-ścieżką** ze względu na zmienną v , jeśli:

1. jest def-czysta ze względu na v
2. $v \in \text{def}(p_1)$
3. $v \in \text{use}(p_n)$

$\text{du}(n, v)$ - zbiór wszystkich du-ścieżek ze względu na v rozpoczynających się w n

$\text{du}(n, m, v)$ - zbiór wszystkich du-ścieżek ze względu na v rozpoczynających się w n i kończących w m

Kryteria pokrycia przepływu danych:

- **All-defs:** $\forall n : v \in \text{def}(n)$ zbiór testów zawiera przynajmniej jedną du-ścieżkę ze zbioru $\text{du}(n, v)$
- **All-uses:** $\forall (n, m) : v \in \text{def}(n)$ and $v \in \text{use}(m)$ zbiór testów zawiera przynajmniej jedną du-ścieżkę ze zbioru $\text{du}(n, m, v)$
- **All-du-paths:** $\forall n \forall v : v \in \text{def}(n)$ zbiór testów zawiera wszystkie du-ścieżki ze zbioru $\text{du}(n, v)$

4.1.5 Pokrycie pętli

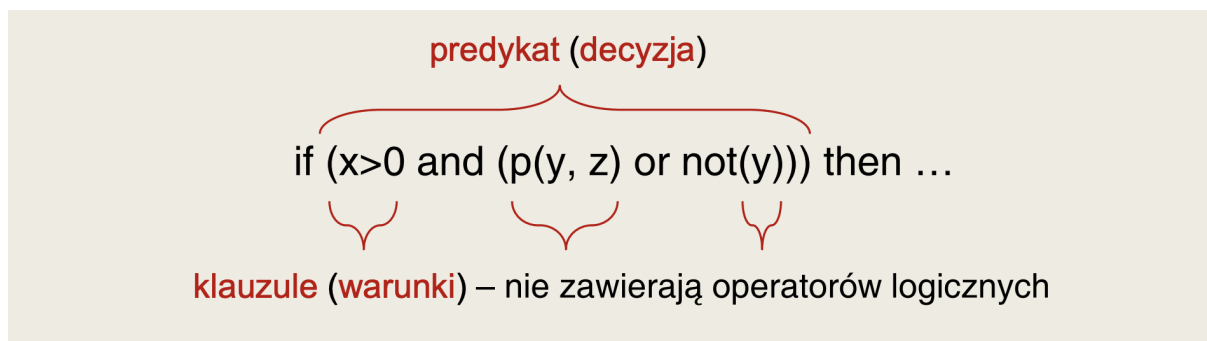
Co testujemy w przypadku pętli?

- problemy w **inicjalizowaniu** pętli
- problemy z niezainicjalizowanymi zmiennymi
- problemy z **powtarzaniem instrukcji** zawartych w pętli
- kwestie związane z **wydajnością/zasobami**

Problemy z testowaniem pętli

- istnienie pętli = **nieskończona liczba ścieżek** do przetestowania
- pętle zagnieżdżone = jeszcze większy problem

4.2 Pokrycia logiczne

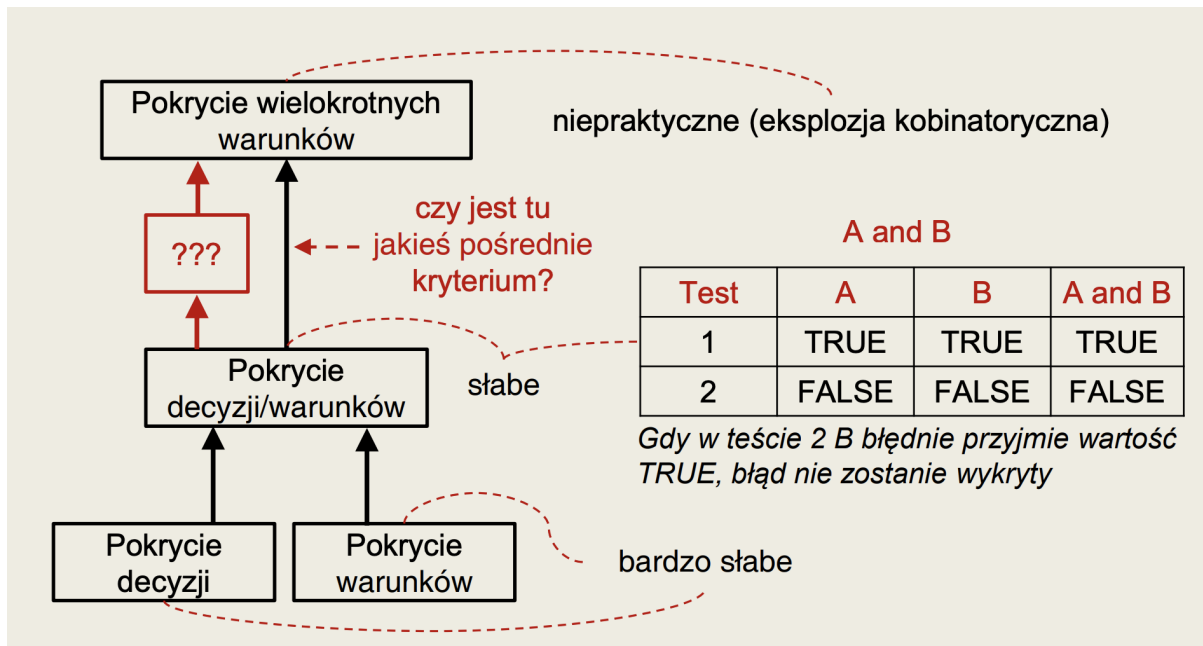


Problemy związane z pokryciem logicznym

- **Problem sterowania** - jakie wartości x , y , z zadać, aby program wykonał się dokładnie taką ścieżką, jaką chcemy?
- **Problem zwarcia** - leniwa ewaluacja formuł logicznych przez kompilator.

4.2.1 Podstawowe kryteria

- **Testowanie decyzji** rozważa decyzję jako niepodzielną całość.
 - każda **decyzja** musi **przynajmniej raz** przyjąć wartość TRUE i przynajmniej raz FALSE.
 - praktycznie tożsame z **pokryciem krawędzi**.
- **Testowanie warunków** rozważa sposób ewaluacji decyzji.
 - **każdy warunek** musi **przynajmniej raz** przyjąć wartość TRUE i przynajmniej raz wartość FALSE
- **Testowanie wielokrotnych warunków** testuje **wszystkie możliwe kombinacje** wartości logicznych **warunków** tworzących decyzję.
 - wada: liczba testów jest wykładnicza względem liczby różnych warunków: dla N warunków musi być 2^N testów
 - jeśli zachodzi **zwarcie**, liczba rzeczywistych testów może być zazwyczaj zredukowana.



4.2.2 Kryterium MC/DC

- słabsze niż wielokrotne warunki, ale silniejsze od warunków/decyzji
- dla N warunków wymaga zazwyczaj $N+1$ testów (a więc liniowo)
- wymaga dostarczenia takich testów, by każdy warunek pokazywał niezależnie swój wpływ na zmianę warunku logicznego decyzji, tzn. dla każdego warunku W w decyzji D muszą istnieć 2 testy:
 - W jest TRUE w jednym z nich i FALSE w drugim
 - D jest TRUE w jednym z nich i FALSE w drugim
 - wartości logiczne pozostałych warunków w tych dwóch testach nie zmieniają się
- warunek W dla tych dwóch testów nazywany jest **klauzulą aktywną**, a pozostałe warunki – **klauzulami pobocznymi**
- Jak uczynić klazulę aktywną?
 - niech D zawiera warunek A ; chcemy, by A była klazulą aktywną
 - aby znaleźć wartości pozostałych warunków tak, by zmiana A wpływała na zmianę D , obliczamy $D[A=TRUE] \text{ xor } D[A=FALSE]$
 - wszystkie wartości logiczne dla pozostałych warunków, które spełniają powyższą formułę są dobrymi kandydatami

Krok 1 – klauzule aktywne

(A and B) or C	1	0
A	1 • •	0 • •
B	• 1 •	• 0 •
C	• • 1	• • 0

Krok 2 – klauzule poboczne

(A and B) or C	1	0
A	1 1 0	0 1 0
B	1 1 0	1 0 0
C	1 0 1	1 0 0

Krok 3 – usunięcie powtórzeń

(A and B) or C	1	0
A	1 1 0	0 1 0
B	1 1 0	1 0 0
C	1 0 1	1 0 0

Wynik: suita testowa

TEST	A	B	C
1	TRUE	TRUE	FALSE
2	FALSE	TRUE	FALSE
3	TRUE	FALSE	FALSE
4	TRUE	FALSE	TRUE

Zalety

- dobre kryterium pośrednie
- subsumuje pokrycie warunków/decyzji, jednocześnie wymagając tylko liniowej względem liczby klauzul liczby testów
- dobra w znajdowaniu defektów takich jak:
 - brakujący warunek, który powinien być obecny
 - AND błędnie zaimplementowany jako OR i vice versa
 - błędnie zaimplementowany operator relacyjny, np. < zamiast >
- idea metody: wykryje błąd, gdy nastąpi błędne wartościowanie jednego (dowolnego!) z warunków decyzji

Wady

- problematyczne, jeśli występują tzw. **termy powiązane**
 - może się nie dać spełnić warunku MC/DC
 - np. dla $D = (A \text{ or } B) \text{ and } (\text{not } A)$ termy A i (not A) są powiązane
 - żadna wartość B nie pozwala A być klauzulą aktywną
- możliwe rozwiązania:
 - wymagać kryterium MC/DC jedynie dla termów niepowiązanych
 - analizować każdą decyzję zawierającą termy powiązane przypadek po przypadku
- **problem zwarcia** może uniemożliwić osiągnięcie odpowiedniego pokrycia MC/DC
- **bardziej skomplikowane** niż słabsze kryteria
 - na szczęście można **znajdować testy automatycznie!**

5 Pozostałe techniki testowania.

5.1 Testowanie eksploracyjne

- Każde podejście do testów jest w jakimś stopniu eksploracyjne.
- Efektywność techniki zależy od stopnia posiadanych umiejętności.
- W eksploracji można używać **formalnych technik** projektowania testów!

Idea:

1. Pobieźna **eksploracja** systemu jako całości
2. **Sprawdzenie** modułów
3. Szczegółowa **analiza** dwóch rzeczy w jednym z modułów
4. Pogłębiona **analiza zagadnienia**

Forma działania: przez określony **czas**, **tester** wchodzi w interakcję z **produktem**, aby wypełnić testerską **misję** i **zaraportować** wyniki.

Przeprowadzane w **sesjach**, często z użyciem **karty testów** (test charter):

- np "zidentyfikuj i sprawdź wszystkie stwierdzenia z podręcznika użytkownika"
- **nie jest bardzo precyzyjna** – adresatem jest **tester dobrze znający system** i środowisko
- daje **swobodę** i **nie narzuca** konkretnych **rozwiązań** i podejść – wiele zależy od samego testera.

Kiedy używać testowania eksploracyjnego?

Zwykle zawsze, gdy nie wiadomo jaki ma być następny test.

- gdy trzeba dać szybką informację zwrotną o nowym produkcie/cesze
- gdy trzeba szybko nauczyć się nowego produktu
- gdy użyto planowego testowania i szukamy różnorodności w testach
- gdy chcemy znaleźć najważniejszy defekt w najkrótszym czasie
- gdy chcemy sprawdzić pracę innego testera przez niezależną analizę
- gdy chcemy zbadać i wyizolować konkretny defekt
- gdy chcemy określić poziom konkretnego ryzyka w celu oceny zastosowania planowanego testowania w tym obszarze

5.2 Pomniejsze techniki

Zgadywanie błędów	Testowanie oparte na listach kontrolnych
<ul style="list-style-type: none"> • czasem utożsamiane z testowaniem ad-hoc • technika niebazująca na żadnym systematycznym podejściu czy technice • opiera się na doświadczeniu testera w testowaniu 	<ul style="list-style-type: none"> • tester korzysta z wysokopoziomowej listy elementów do zanalizowania, sprawdzenia lub zapamiętania • modelowa lista kontrolna może dotyczyć różnych aspektów, np.: <ul style="list-style-type: none"> – charakterystyk jakościowych – standardów GUI – kluczowych operacji – standardów kodowania
Ataki usterkowe	Techniki oparte na modelach defektów
<ul style="list-style-type: none"> • idea: atak na oprogramowanie przez jego interfejs: użytkowy (GUI, API) lub systemowy (system plikowy, interfejs bazodanowy, interfejs OS) 	<ul style="list-style-type: none"> • taksonomia defektów to system (hierarchicznych) kategorii stworzony jako pomoc w klasyfikowaniu defektów

5.3 Testowanie mutacyjne

- metoda **oparta na składni**
- można ją zaklasyfikować jako **białoskrzynkową**
- uważana za jedną z **najmocniejszych** technik testowania
- bezpośrednio **testuje testy**, nie program
- **mutant** = zmodyfikowany, kompilowalny kod
- idea: **każdy mutant powinien zostać zabity** przez przynajmniej 1 test
- **pokrycie mutacyjne** = $\frac{\text{\#zabitych mutantów}}{\text{\#mutantów}}$

Zalety

- **wysoce zautomatyzowany** proces, wsparcie narzędziowe
- **tani** we wdrożeniu
- jedna z **najefektywniejszych** technik testowania

Wady

- **wymaga wielu zasobów, czasochłonny**
- efektywność zależy od doboru operatorów mutowacyjnych
- problem **mutantów równoważnych** = mających ten sam efekt, np $! = i <$ w warunku pętli

5.4 Analiza statyczna

Cechy analizy statycznej

- **wczesne wykrywanie** defektów
- identyfikacja defektów trudnych do wykrycia w testach
- **wykrywanie zależności i niespójności** w modelach oprogramowania
- zwiększenie **pielęgnowalności** kodu
- **zapobieganie defektom**

Typowe defekty wykrywane przez analizę statyczną

- odwołania do niezainicjalizowanej zmiennej
- niespójne interfejsy,
- niewykorzystane zmienne,
- martwy kod,
- brakująca lub błędna logika,
- zbyt skomplikowane instrukcje,
- naruszenie standardów kodowania,
- słabe punkty zabezpieczeń,
- naruszenie reguł modelowania, itp.

5.4.1 Techniki analizy statycznej

- **Analiza złożoności**

Złożoność cyklomatyczna (CC, McCabe cyclomatic complexity) to miara stopnia **skomplikowania struktury kodu**.

Metody (równoważne) obliczenia CC:

1. $CC = \max$ liczba ścieżek liniowo niezależnych
2. $CC = E - N + 2$
3. $CC = \text{liczba zamkniętych obszarów CFG} + 1$
4. $CC = \text{liczba decyzji w CFG} + 1$ (switch z $n > 2$ liczony jako $n-1$ decyzji)

Generalnie, im mniejsza wartość CC, tym lepiej.

- **Parsowanie kodu** - analiza błędów poprawnych syntaktycznie.

- **Analiza przepływu danych**, np:

- przypisanie nieprawidłowej wartości do zmiennej (typy)
- użycie zmiennej przed jej zdefiniowaniem
- użycie usuniętej uprzednio zmiennej
- redefiniowanie zmiennej przed jej użyciem

- **Graf wywołań**

- **Graf skierowany**, w którym:

- * **wierzchołki** = jednostki oprogramowania (np. moduły, funkcje itp.)
- * **krawędzie** = komunikacja między jednostkami (np. wywołanie)

- **Złożoność** grafu wywołań: 1 za ifa, $n-1$ za switch, 2 za pętle.

- Służy do:
 - * określenia kolejności testów integracyjnych
 - * dolnego oszacowania na liczbę testów integracyjnych
- W testach integracyjnych chcemy określić ich kolejność tak, aby:
 - * nie budować zbyt wielu namiastek/sterowników
 - * gdy nastąpi awaria, łatwo będzie zidentyfikować miejsce defektu
 - * będziemy testować rzeczywiste interakcje (np. faktyczne wywołania)

5.5 Analiza dynamiczna

- wykorzystywana do **wykrywania awarii**, gdzie symptomy mogą nie być natychmiastowo widoczne
- analiza dynamiczna **działa na uruchomionym programie/systemie**

5.5.1 Przykładowe techniki analizy dynamicznej

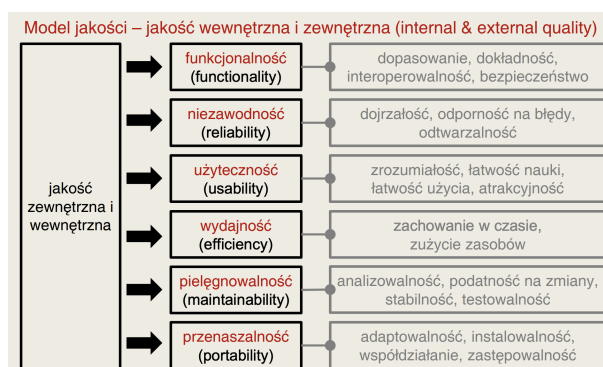
- wykrywanie wycieków pamięci
 - symptomy: stopniowe pogarszanie się wydajności aplikacji
- wykrywanie dzikich wskaźników
- analiza wydajności
 - narzędzia mogą pomóc wykryć **wąskie gardła** wydajności
 - np. informacja o **ilości wywołań** modułu podczas wykonania
 - często wywoływane moduły to kandydaci do usprawnienia wydajności
 - połączona informacja o dynamicznym zachowaniu systemu i o grafach wywołań pozwala testerowi zidentyfikować moduły mogące być kandydatami na szczegółowe testowanie
- analiza zachowania sieci
- analiza aplikacji przy użyciu profilera

6 Testowanie нефункционалне.

6.1 Jakość oprogramowania

Jakości jest pojęciem **niejednoznacznym i wielowymiarowym**.

6.1.1 Model jakości ISO 9126



6.1.2 Model jakości ISO/IEEE 25000

6.1.3 Niezawodność

Niezawodność - zdolność oprogramowania do **bezbłędnego działania** przez określony czas lub przez określoną liczbę operacji. Testy niezawodności wykorzystują **profile operacyjne**.

Cechy niezawodności:

- **dojrzałość** (zdolność do bezawaryjnego działania przy występowaniu usterek)
- **tolerancja na błędy** (np. obsługa wyjątków)
- **odtwarzalność** (zdolność działania po awarii)

6.1.4 Bezpieczeństwo

- Bezpieczeństwo to zbiór atrybutów oprogramowania umożliwiający **ochronę przed nieautoryzowanym dostępem** do programu i danych.
- **OWASP (Open Web Application Security Project)** - obszerna baza zasobów dot. bezpieczeństwa webowego, np.: baza ataków, książek i innych materiałów o bezpieczeństwie.
- **SAMM (Software Assurance Maturity Model)** - model pozwalający organizacji zdefiniować i wdrożyć strategię dla zapewnienia bezpieczeństwa, dopasowaną do określonych ryzyk.

6.1.5 Użyteczność

- Użyteczność to **zdolność systemu do bycia zrozumiałym**, łatwym do nauczenia i użycia.
- Testowanie **skupia się na użytkownikach**.
- **Efekty** testowania obserwowane na **prawdziwych**, końcowych **użytkownikach**, a nie testerach.

Podcharakterystyki

- Zrozumiałość
- Łatwość nauki
- Łatwość obsługi
- Atrakcyjność

Metryki

- **efektywność** = np. stopień osiągnięcia przez użytkownika celów
- **wydajność** = ilość zasobów zużytych
- **satysfakcja** klienta z użytkowania produktu

TESTOWANIE UŻYTECZNOŚCI

Proces	
Formatywne testowanie użyteczności - przeprowadzane iteracyjnie w kolejnych fazach projektowania i prototypowania.	Całościowe testowanie użyteczności - przeprowadzane po implementacji .
Rodzaje	
Formalne testowanie użyteczności - często wymaga uprzedniego przygotowania rzeczywistych lub reprezentatywnych użytkowników (dostarczenie scenariuszy zadań, instrukcji).	Nieformalne testowanie użyteczności - pozwala użytkownikowi eksperymentować z oprogramowaniem, aby obserwatorzy ocenili jak trudna dla użytkownika jest praca z systemem.
Walidacja	
powinna być przeprowadzona w warunkach tak bliskich rzeczywistym warunkom użytkowania oprogramowania, jak to tylko możliwe, np. w laboratorium użyteczności .	
Wytyczne (guidelines)	
są stosowane aby uzyskać spójne podejście do wykrywania i raportowania defektów użyteczności na wszystkich etapach cyklu życia.	

Specyfikacja	
<ul style="list-style-type: none"> • Inspekcja, ewaluacja, lub przegląd <ul style="list-style-type: none"> – Wymagania i specyfikacja. – Heurystyczna ocena projektu GUI • Weryfikacja i walidacja bieżącej implementacji <ul style="list-style-type: none"> – Weryfikacja przy pomocy przypadków testowych cech użyteczności zdefiniowanych w wymaganiach – Walidacja przez scenariusze testowe 	<ul style="list-style-type: none"> • Dynamiczna interakcja z prototypami <ul style="list-style-type: none"> – Praca z prototypami i pomoc deweloperom w ich rozwijaniu • Ankiety i kwestionariusze <ul style="list-style-type: none"> – Zbieranie obserwacji i informacji zwrotnej – SUMI (Software Usability Measurement Inventory), WAMMI (Website Analysis and Measurement Inventory) – standaryzowane benchmarki

6.1.6 Wydajność

- **Zdolność** oprogramowania do **zapewnienia odpowiedniej efektywności** w działaniu, relatywnie do ilości zużytych zasobów
- Najczęstsza przyczyna błędów wydajności: **błędy projektowe**.

Rodzaje testów wydajności	Metryki dla wydajności
<ul style="list-style-type: none"> • testowanie obciążenia • testowanie warunków skrajnych • testowanie skalowalności • testowanie wykorzystania zasobów • testowanie wytrzymałościowe • testowanie skokowe • testowanie niezawodności • testowanie z aktywnym tłem • testowanie punktu krytycznego 	<ul style="list-style-type: none"> • % wykorzystania procesora w kluczowym czasie • dostępna pamięć (RAM, wirtualna) • top N aktywnych procesów • liczba przełączeń kontekstów na sekundę • długość kolejek (procesor, dysk) w danym czasie • czas podróży pakietu • czas prezentacji danych klientowi

6.1.7 Pielęgnowalność

- Pielęgnowalność to **łatwość modyfikowania oprogramowania**.
- Oprogramowanie się nie zużywa, ale staje się **przestarzałe**.
- **Testowanie pielęgnowalności**
 - Zwykle nie przy pomocy skryptów testowych
 - Większość defektów jest **niewidoczna dla testowania dynamicznego**
 - Najlepiej sprawdzają się **techniki statyczne**
- **Podcharakterystyki pielęgnowalności**
 - **Analizowalność** - łatwość diagnozowania problemów (problem: spaghetti code)
 - **Modyfikowalność** - zdolność do wprowadzania zmian (problem: zły standard kodowania)
 - **Stabilność** - zdolność do unikania niespodziewanych efektów na skutek zmian (problem: brak kohezji)
 - **Testowalność** - łatwość walidacji po wprowadzonej zmianie (problem: zła dokumentacja)

6.1.8 Przenaszalność

Przenaszalność to **łatwość, z jaką oprogramowanie może być przeniesione** między środowiskami.

- **Adaptowalność** – zdolność do adaptacji w innym środowisku bez podejmowania akcji innych niż przewidziane w tym celu.
- **Zastępowalność**
- **Instalowalność**
- **Koegzystencja** – zdolność do współistnienia z innym, niezależnym oprogramowaniem dzielącym środowisko – np. „przywłaszczenie” standardowego skrótu klawiszowego przez inny program.

7 Automatyzacja testowania.

Efekt próbnika (probe effect) - niezamierzony wpływ na zachowanie systemu spowodowany pomiarami tego systemu. Narzędzie które wpływa na wynik testu nazywamy **inwazyjnym**.

Zalecenia dotyczące architektury systemu automatyzacji:

- **pojedyncza odpowiedzialność** komponentów; każdy odpowiada za pojedynczy obszar
- **rozszerzalność** - komponenty rozszerzalne, ale nie modyfikowalne
- **zastępowalność**
- **rozdzielność**
- **abstrakcyjność zależności** - komponenty zależą od abstrakcji

7.1 Techniki automatyzacji testów

Technika	Zalety	Wady
nagraj i odtwórz	stosowalne na poziomie GUI lub API, łatwe do konfiguracji i użycia, nie wymaga znajomości języków	trudne w utrzymaniu, problemy gdy potrzeba czasu na odpowiedź systemu
skrypty linearne	brak żmudnych i kosztownych przygotowań, znajomość programowania niekonieczna gdy skrypt tworzony automatycznie	koszt automatyzacji liniowy ze względu na liczbę skryptów; trudne i kosztowne w utrzymaniu
skrypty zorganizowane	redukcja kosztów utrzymania, zmniejszenie kosztu automatyzacji nowych testów	zwiększone koszty początkowe tworzenia reużywalnych skryptów; wymaga umiejętności programowania
data-driven testing	niski koszt dodania testu; nie wymaga znajomości programowania; tanie w utrzymaniu	ograniczona możliwość przeprowadzania testów negatywnych
keyword-driven testing	tanie w utrzymaniu; swoboda w tworzeniu testów	kosztowna implementacja słów kluczowych; trudność w doborze właściwych słów

7.2 Testowanie oparte na modelu (MBT)

- **Podstawowa idea:** ulepszyć jakość i efektywność projektu i implementacji testów przez:
 - projekt wyczerpującego modelu MBT, zwykle z użyciem narzędzi,
 - model jako specyfikacja projektu testów, automatyczna generacja przypadków testowych,
- **Rodzaje modeli:** strukturalne, behawioralne, danych (UML).

Efektywność

- modelowanie ułatwia **komunikację** z interesariuszami
- **zrozumienie**
- **łatwiejsze zaangażowanie** interesariuszy
- **łatwa identyfikacja „problematycznych” części systemu**
- **wczesna generacja i analiza przypadków testowych** - możliwe przed stworzeniem systemu

Wydajność

- **wczesne unikanie defektów** - weryfikacja wymagań
- **możliwe reużycie artefaktów MBT**
- **automatyzacja** - np. generacja testaliów
- **adaptacja do zmian** - różne suity testów mogą być generowane z tego samego modelu
- **redukcja kosztów przy zmianie wymagań** - „single point of maintenance”

Kryteria wyboru testów	
Oparte na pokryciu	Inne
<ul style="list-style-type: none">• Wymagania połączone z modelem - elementy modelu są połączone z wybranymi wymaganiami. Pełne pokrycie wymagań odpowiada zestawowi testów całkowicie pokrywających wybrany zbiór wymagań.• Pokrycie Elementów modelu MBT - bazuje na wewnętrznej strukturze modelu.• Oparte na danych - związane są z takimi technikami projektowania testów jak:<ul style="list-style-type: none">– podział na klasy równoważności– analiza wartości brzegowych– testy kombinatoryczne (np. pair-wise)	<ul style="list-style-type: none">• Losowe przechodzenie przez model, wszystkie przejścia są równo prawdopodobne. W podejściu stochastycznym wybór oparty o rozkład prawdopodobieństwa. Model reprezentuje profil użycia (profil operacyjny).• Oparte na scenariuszu/wzorcu - scenariuszem może być use case lub scenariusz użycia; wzorzec = częściowo zdefiniowany scenariusz, który można zastosować do modelu MBT aby wyprowadzić jeden lub wiele testów.• Sterowane projektem - podejście oparte na dodatkowej informacji projektowej.

8 Zarządzanie testowaniem.

- Zarządzanie strategicznie,
- Zarządzanie operacyjne,
- Zarządzanie zespołem testowym.

8.1 Testowanie oparte na ryzyku.

Ryzyko – możliwość (prawdopodobieństwo) wystąpienia negatywnego lub niepożądanego zdarzenia.

Poziom ryzyka – ważność określona przez prawdopodobieństwo i wpływ; **ilościowy** lub **jakościowy**.

Zarządzanie ryzykiem - w podejściu opartym na ryzyku ryzyko jest podstawową bazą testów.

Identyfikacja ryzyka		
proces identyfikacji możliwych do wystąpienia ryzyk. Techniki identyfikacji ryzyka:		
<ul style="list-style-type: none">• burza mózgów• listy kontrolne• historia awarii	<ul style="list-style-type: none">• wywiady eksperckie• szablony ryzyk• niezależna ocena	<ul style="list-style-type: none">• doświadczenie z poprzednich projektów

<p style="text-align: center;">Analiza ryzyka</p> <p>proces oceny zidentyfikowanych ryzyk w celu oszacowania ich wpływu oraz prawdopodobieństwa wystąpienia,</p>	
<p>Wyjście: lista ryzyk</p> <ul style="list-style-type: none"> • przypisane poziomy/priorytety, • określenie zakresu testowania, • określenie metod zapobiegania ryzykom 	<p>Szacowanie ilościowe analizy ryzyka to określenie kosztu materializacji ryzyka. Często trudne lub nie możliwe – wtedy stosuje się podejście jakościowe.</p>
<p style="text-align: center;">Łagodzenie ryzyka</p> <p>proces implementacji planów mających zapobiegać ryzyku. Metody łagodzenia ryzyka:</p>	
<ul style="list-style-type: none"> • łagodzenie ryzyka przez podjęcie czynności prewencyjnych, • plany awaryjne mające na celu redukcję siły oddziaływania ryzyka. 	<ul style="list-style-type: none"> • transfer ryzyka, czyli przeniesienie ryzyka na stronę trzecią (np. ubezpieczyciela), • zignorowanie i zaakceptowanie ryzyka.
<p>Podczas fazy łagodzenia ryzyka następuje priorytetyzacja testów wg poziomu ryzyka. Poziom ryzyka wpływa na zakres testowania.</p>	
<p style="text-align: center;">Kontrola (monitorowanie) ryzyka</p> <p>ciągła obserwacja aktualnego stanu systemu Przykładowe metryki zbierane w ramach procesu:</p>	
<ul style="list-style-type: none"> • % pokrytych wymagań, funkcjonalności • % testów zdanych/nie zdanych • poziom zminimalizowanego i rezydualnego ryzyka 	<ul style="list-style-type: none"> • ryzyka w podziale na: <ul style="list-style-type: none"> – zminimalizowane (testy przeszły) – "w trakcie" (wykryto problemy) – niepokryte (nie uruchomiono testów)

8.1.1 Analiza ryzyka

Metody analizy ryzyka	
<p>FMEA (Failure Mode and Effect Analysis)</p> <ul style="list-style-type: none"> • Podejście do systematycznej identyfikacji możliwych awarii systemu • Awarie są priorytetyzowane wg: <ul style="list-style-type: none"> – konsekwencji ich wystąpień – prawdopodobieństwa (częstości) wystąpień – łatwości ich wykrycia • Cel FMEA: podjęcie akcji w celu eliminacji lub redukcji awarii, poczynając od najpoważniejszych 	<p>FTA (Fault Tree Analysis)</p> <ul style="list-style-type: none"> • Inaczej analiza drzewa usterek • Metoda analizy niezawodności, bezpieczeństwa i utrzymywalności • Dedukcyjna procedura używana w celu określenia różnych kombinacji awarii software'u, hardware'u i błędów ludzkich, które mogą wywołać niepożądane efekty na poziomie systemowym • Podejście top-down (od ogółu – awarii systemowej, do szczegółu – pojedynczych błędów na niskich poziomach). Podstawowe elementy to bramki logiczne.
<p>QFD (Quality Function Deployment)</p> <ul style="list-style-type: none"> • proces/narzędzia przekształcania wymagań użytkownika na specyfikacje systemowe 	<p>PRisMa (Product Risk Management)</p> <ul style="list-style-type: none"> • prawdopodobieństwo i wpływ traktowane osobno, nie łącznie

POZIOM RYZYKA = PRAWDOPODOBIENSTWO * WPŁYW (biznesowy).

8.2 Biznesowa wartość testowania.

Model CoQ (Cost of (Poor) Quality) - model opisujący **koszty** związane z dostarczeniem produktu lub usługi o **niskiej jakości**. Rodzaje kosztów

Rodzaj kosztów	Powód ich ponoszenia
koszty zapobiegania	aby unikać defektów; dot. wymagań, planowania i zapewniania jakości, szkoleń (np. szkolenie deweloperów, aby pisany kod był lepszej jakości)
koszty wykrywania	aby wykrywać defekty; ponoszone nawet, gdy nie wykryjemy żadnych defektów (np. wydatki na analizę, projektowanie, implementację, niektóre koszty wykonania testów)
koszty wewnętrznego błędu	z powodu wykrycia awarii (np. pozostałe koszty wykonania testów, koszty re-testów, koszt naprawy defektu przez programistę)
koszty zewnętrznego błędu	z powodu niewykrycia awarii (np. koszty wsparcia technicznego, helpdesku, naprawy defektów polowych, kary umowne)

8.3 Planowanie testów.

8.3.1 Szacowanie

Rola szacowania - określenie **pracochłonności** i **kosztochłonności** projektu.

Do szacowania można użyć analizy punktów testowych (Test Point Analysis) - TMap.

8.3.2 Dokumentacja

Cele dokumentowania	Rodzaje dokumentów
<ul style="list-style-type: none">• opisanie obowiązujących w organizacji lub projekcie reguł, standardów, celów do osiągnięcia, stanowiących punkt odniesienia• akceptowanie/zezwalanie/potwierdzanie wykonania określonych czynności• utrwalenie danych• pomoc w monitorowaniu i kontroli procesów• pomoc w opanowaniu złożoności projektów• platforma komunikacji	<ul style="list-style-type: none">• Polityka testów – wysokopoziomowy opis zasad, podejść i głównych zadań organizacji.• Strategia testów – wysokopoziomowy opis poziomów testów do wykonania oraz testów w ramach tych poziomów.• Plan testów – opis zakresu, metod, zasobów i harmonogramu czynności testowych.• Dziennik wykonania testów – zapis czynności testowych.

8.3.3 Metryki

Na podstawie metryk możemy:

- **mierzyć postęp** procesu testowego
- **obliczyć zwrot** z inwestycji
- **ocenić** i porównać różne możliwe **podejścia**
- **ocenić** i kontrolować **wydajność** procesu
- **ocenić** i kontrolować **poprawę** procesu
- zbudować system „**wczesnego ostrzegania**”
- zbudować **modele predykcyjne**
- **porównywać** proces z procesami konkurencji

Wymiary postępu testowania.

- Ryzyka, defekty, testy i pokrycie – **metryki ilościowe**.
- Pewność – najbardziej subiektywna, często **jakościowa**; pomiar za pomocą ankiet, wywiadów itd.

Rodzaje metryk

- **Metryki projektu** - opisują postęp w kierunku uprzednio zdefiniowanych celów lub kryteriów wyjścia, np. odsetek zdanych testów,
- **Metryki produktu** - opisują atrybuty wytwarzanego oprogramowania, np. gęstość defektów, złożoność cyklomatyczna,
- **Metryki procesu** - mierzą zdolność procesu wytwórczego lub testowego, np. odsetek defektów znalezionych/usuniętych w danej fazie projektu.

$$MTTF = \frac{\sum_{i=1}^N OK_i}{N}$$

$$MTTR = \frac{\sum_{i=1}^N R_i}{N}$$

$$MTBF = MTTF + MTTR$$

MTTF = Mean Time To Failure

MTTR = Mean Time To Repair

MTBF = Mean Time Between Failures

Przykłady metryk

- Metryki **ryzyka produktowego** - pokrycie ryzyk
- Metryki **defektów** - MTTF, MTBF, analiza napraw, zgłoszonych defektów
- Metryki **przypadków testowych**
- Metryki **pokrycia** - stopień pokrycia wymagań, kodu, klas równoważności itd.
- Metryki **pewności** - stabilność, niezawodność, ocena klienta

8.4 Zarządzanie incydentami

8.4.1 IEEE 1044.

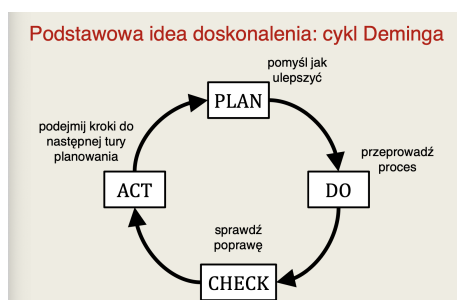
Cykl życia defektu wg IEEE 1044.

1. **Rozpoznanie (recognition)** – zaobserwowanie anomalii wskazującej na potencjalny defekt
2. **Badanie (investigation)** incyduentu - wykrywa powiązane problemy i proponuje rozwiązania
3. **Działanie (action)** – rozwiązanie defektu, akcje zapobiegawcze, testy regresji i testy potwierdzające
4. **Dyspozycja (disposition)** – zbieranie dalszych informacji i przeniesienie defektu w stan końcowy

Krok	Czynności		
	rejestruj...	klasyfikuj...	identyfikuj wpływ...
Rozpoznanie	wspomagające informacje	na podstawie ważnych atrybutów	na podstawie postrzeganego wpływu
Badanie	zaktualizuj i dodaj dodatkowe informacje	zaktualizuj i dodaj klasyfikację na podst. ważnych atrybutów	aktualizuj na podstawie badania
Działanie	dodaj dane oparte o podjęte działanie	dodaj dane oparte o podjęte działanie	aktualizuj na podstawie działania
Dyspozycja	dodaj dane bazujące na dyspozycji	na podstawie dyspozycji	aktualizuj na podstawie dyspozycji

Atrybuty defektu wg IEEE 1044	Klasyfikacje wpływu defektu wg IEEE 1044
<p>zgłoszenie defektu pozwalające na podjęcie działania jest:</p> <ul style="list-style-type: none"> • kompletne - nie brakuje żadnych ważnych szczegółów, • zwięzłe - nie zawiera nieistotnych informacji, • precyzyjne - nie wprowadza czytelnika w błąd, • obiektywne - bazuje na faktach, nie atakuje nikogo. 	<ul style="list-style-type: none"> • dotkliwość (severity) • priorytet • wartość dla klienta • sukces misji (mission safety) • harmonogram projektu • koszt projektu • ryzyko projektu • jakość projektu • kwestie społeczne

8.4.2 Główne modele doskonalenia



- Test Maturity Model (TMM)
- Test Process Improvement (TPI, TPI Next)
- Critical Testing Processes (CTP)
- Systematic Test and Evaluation Process (STEP)
- Test Organization Maturity (TOM)
- Test Improvement Model (TIM)
- Software Quality Rank (SQR)
- TMap

Modele referencyjne

- **procesu** - (jednowymiarowa) **ocena dojrzałości procesu**, wskazują **kolejność usprawnień**
- **zawartości** - **opisują** ważne procesy software'owe i co powinno się z nimi robić, ale **nie szeregują** zadań w żadnej kolejności

9 Jakość oprogramowania.

	Zapewnianie jakości (QA)	Testowanie (QC)
Cel	Ulepszyć proces produkcji i testowania oprogramowania, aby defekty się nie pojawiały	Identyfikacja defektów po wyprodukowaniu produktu, a przed wydaniem do klienta
Jak?	Wdrożenie systemu zarządzania jakością; okresowe audyty	Znajdowanie i eliminowanie problemów z jakością aby wymagania klienta były spełnione
Co?	Zapobieganie problemom przez planowe i systematyczne działania	Wykorzystanie technik testowania do identyfikacji defektów
Odpowiedzialność	Wszyscy są odpowiedzialni	Zwykle zespół testerski
Przykład	Weryfikacja	Walidacja, testowanie
Techniki	Statistical Process Control	Statistical Quality Control
Jako narzędzie	QA to narzędzia zarządcze	QC to narzędzie korekcyjne

Przykłady aktywności QA.

- **definiowanie i implementacja procesów**
 - metodyka wytwarzania oprogramowania
 - zarządzanie projektami
 - zarządzanie wymaganiami, konfiguracją
 - pomiary oprogramowania, szacowanie
 - projektowanie oprogramowania
 - proces testowy
- **identyfikacja słabych punktów** w procesach i ciągła poprawa procesów
- **przeprowadzanie audytów**
- **szkolenia**

9.0.1 Metryki oprogramowania

GQM - Goal-Question-Metric

- **Cel** - co chce osiągnąć klient?
- **Pytania** - scharakteryzowanie metody osiągnięcia celu
- **Metryki** - ilościowe odpowiedzi.

Miary wolumenowe kodu

- **LOC (Lines of Code)**
 - prosta, niezawodna, wspierana narzędziami metoda
 - problem: dokładny pomiar dopiero po implementacji
 - problem: nie mierzy tego, co program robi
- **FP (Function Points)**
 - mierzy tzw. punkty funkcyjne
 - zaleta: można mierzyć na etapie projektu
 - zaleta: pomiar niezależny od języka programowania – wada: bardziej skomplikowana metoda liczenia

Metryki złożoności strukturalnej

- **rozmiar** (LOC, FP)
- **złożoność cyklomatyczna**
- **złożoność Halsteada** (# operatorów i operandów)
- **przepływ informacji** (z i do modułu)
- **złożoność systemu** (pomiar pielęgnowalności)
- metryki strukturalne dla OOP
 - **WMC** (Weighted Methods defined per Class)
 - **DIT** (Depth of Inheritance Tree)
 - **NOC** (Number Of Children)
 - **CBO** (Coupling Between Objects)
 - **RFC** (Response For a Class)
 - **LCOM** (Lack Of Cohesion)

Metryki złożoności konceptualnej dotyczą trudności w zrozumieniu wymagań/kodu/itp.

Metryki złożoności obliczeniowej dotyczą złożoności obliczeń programu w trakcie jego wykonania.

Modele statyczne defektów

- Model fazowy.
- Modele zmian w kodzie.
- **Model Rayleigha** - opisuje rozkład znajdowania defektów w czasie.

Posiew defektów (fault seeding) – metoda sztucznego wprowadzania defektów do kodu w celu sprawdzenia efektywności istniejących testów; np **testowanie mutacyjne**.

Wstrzykiwanie defektów (fault injection) – metoda sztucznego wprowadzania defektów/wywoływania awarii, w celu sprawdzenia jak program radzi sobie z nietypowymi, błędnymi, „dziwnymi” sytuacjami.

Analiza mutacyjna – wykorzystanie testowania mutacyjnego jako modelu predykcyjnego defektów.