

# Notatki z kursu Testowanie oprogramowania

Małgorzata Dymek

2019/20, semestr zimowy

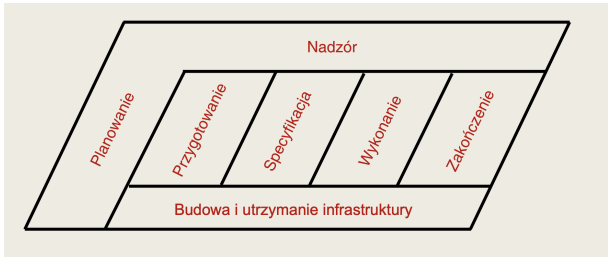
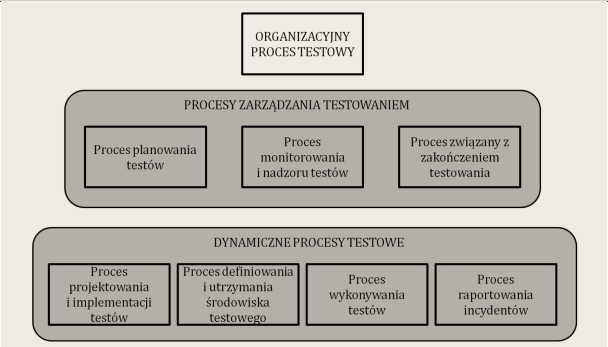
## 1 Wprowadzenie.

### 1.1 Definicje.

TESTOWANIE	
Definicja	Pojęcia
<b>Definicja testowania jest niejednoznaczna:</b> <ul style="list-style-type: none"><li>• Testowanie to <b>wykonywanie oprogramowania z intencją wykrywania</b> tkwiących w nim <b>błędów</b>.</li><li>• Testowanie to krytyczne sprawdzanie, obserwacja i <b>ewaluacja jakości</b> oprogramowania.</li><li>• Testowanie to proces <b>analizowania</b> fragmentu oprogramowania w celu wykrycia różnic pomiędzy istniejącymi a pożądanymi warunkami (czyli defektów) oraz w celu oceny cech tego fragmentu oprogramowania [IEEE].</li></ul>	<b>Pomyłka</b> - człowiek robi coś źle. <b>Defekt (usterka, bug, fault)</b> - statyczny defekt w kodzie (lub dokumentacji), skutek pomyłki człowieka. <b>Błąd</b> – <b>nieprawidłowy stan wewnętrzny</b> programu np. licznik pętli ustawiony na drugim zamiast pierwszym elemencie tablicy. <b>Awaria (failure)</b> – widoczne, nieprawidłowe działanie oprogramowania np. crash systemu, zwrócenie nieprawidłowego wyniku, komunikat o błędzie. <b>Incydent</b> – wydarzenie, które wymaga analizy <b>Suita testowa</b> - ????
<b>7 uniwersalnych zasad testowania.</b>	<b>Normy i standardy związane z testowaniem</b>
<ol style="list-style-type: none"><li>1. Testowanie ujawnia usterki</li><li>2. Testowanie gruntowne jest niewykonalne</li><li>3. Wczesne testowanie</li><li>4. Kumulowanie się błędów</li><li>5. Paradoks pestycydów</li><li>6. Testowanie zależy od kontekstu</li><li>7. Mylne przekonanie o braku błędów</li></ol>	<ul style="list-style-type: none"><li>• <b>IEEE 829</b> – dokumentacja testowa</li><li>• <b>IEEE 1008</b> – standard dla testowania jednostkowego</li><li>• <b>IEEE 1028</b> – standard dla przeglądów i audytów</li><li>• <b>ISO 9126</b> – model jakości (stara)</li><li>• <b>ISO/IEEE 25000</b> – model jakości (nowa)</li><li>• <b>ISO/IEEE 29119</b> – Software Testing Standard</li></ul>

Ewaluacja	
na początku	na końcu
<b>Walidacja</b> - dokonywana w celu potwierdzenia zgodności z założonymi celami użycia.	<b>Weryfikacja</b> - sprawdzająca, czy produkt danej fazy spełnia wymagania (zwykle techniczne) ustalone podczas poprzedniej fazy.
are we building the right thing?	are we building the thing right?
Testowanie to nie debugowanie	
<b>Testowanie</b>	<b>Debugowanie</b>
<ul style="list-style-type: none"> <li>znajduje awarie</li> <li>sprawdza, czy usterka została poprawnie usunięta</li> </ul>	Na podstawie informacji o awarii: <ul style="list-style-type: none"> <li><b>lokalizuje miejsce usterki</b> powodującej tę awarię</li> <li><b>usuwa (naprawia)</b> usterkę</li> </ul>

## 2 Testowanie w cyklu życia.

Modele procesu testowego	
<b>TMAP lifecycle model</b>	<b>ISO 29119</b>
 <p>Proces jest <b>generyczny</b>, 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	
<b>Podstawa testów</b>	wymagania na moduły, projekt szczegółowy, kod
<b>Typowe obiekty</b>	moduły, programy, funkcje, klasy, procedury

TESTY INTEGRACYJNE	
<b>Podstawa testów</b>	projekt systemu, architektura, przypadki użycia
<b>Typowe obiekty</b>	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 - <b>co system robi.</b>	Нефункционална charakterystyka jakościowa wyrażalna ilościowo, np. niezawodność czy użyteczność - <b>jak system działa.</b>

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

## 2.3 Statyczne techniki testowania

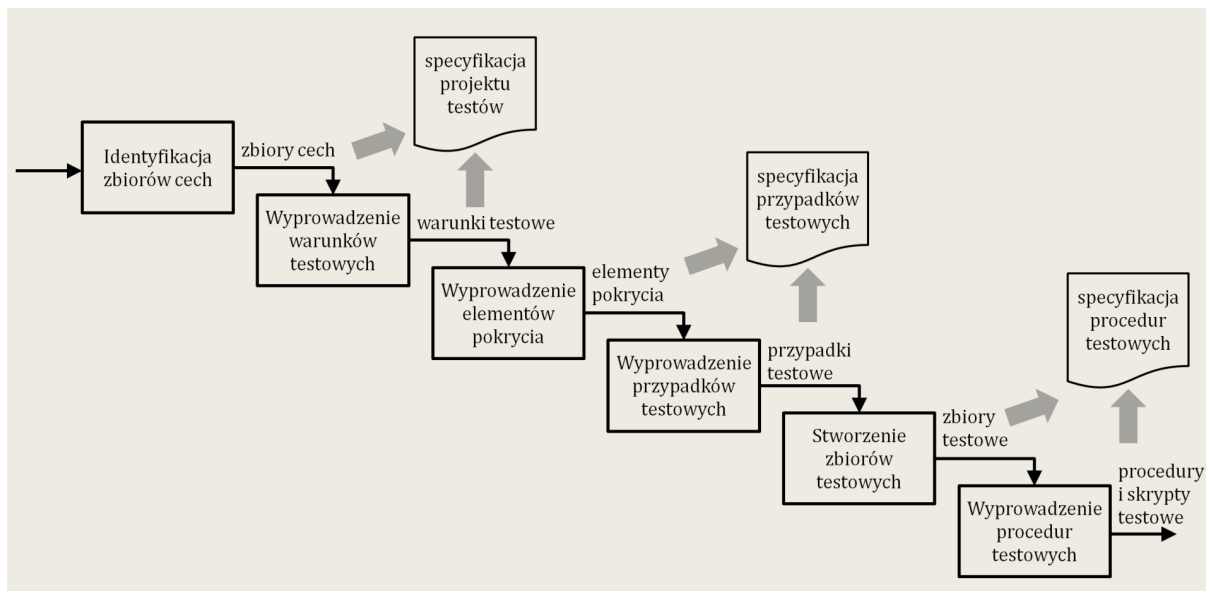
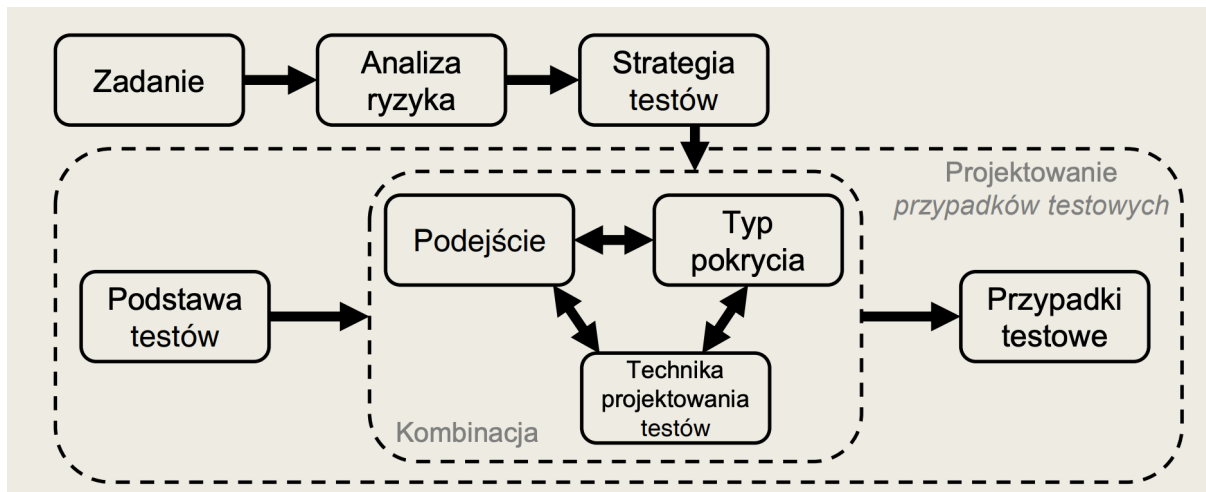
Ręczne sprawdzanie (**przeglądy**) i automatyczna analiza (**analiza statyczna**) kodu lub dokumentacji bez uruchamiania kodu, ale zwykle z użyciem narzędzi!

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

Rodzaje przeglądów		
Typ przeglądu	Charakterystyka	Cel
<b>nieformalny</b>	brak formalnego procesu, może przybrać formę programowania w parach lub nieformalnej rozmowy	tani sposób na osiągnięcie niewielkich korzyści
<b>przejrzenie</b>	przewodzone przez autora; opcjonalne przygotowanie przed spotkaniem, opcjonalny raport z przeglądu	uczenie się, zrozumienie, znajdowanie usterek
<b>techniczny</b>	przeszkolony moderator, przygotowanie przed spotkaniem, zdefiniowany proces postępowania	podjęcie decyzji, ocena alternatyw, szukanie usterek, rozw. probl. technicznych
<b>inspekcja</b>	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"><li>• Planowanie</li><li>• Rozpoczęcie</li><li>• Przygotowanie indywidualne</li><li>• Kontrola/ocena/zapis wyników</li><li>• Poprawki</li><li>• Zakończenie</li></ul>	<ul style="list-style-type: none"><li>• kierownik</li><li>• moderator</li><li>• autor</li><li>• przeglądający</li><li>• protokolant</li></ul>

## 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 kodu).

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

## 3 Czarnoskrzynkowe techniki projektowania testów.

to procedury wywodzenia/wybierania przypadków testowych.

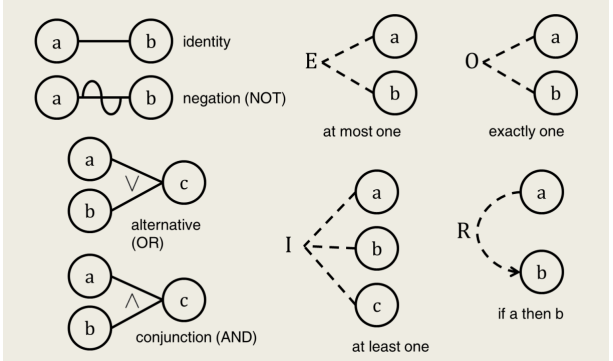
Technika	Element pokrycia
Podział na klasy równoważności	Klasa równoważności
Analiza wartości brzegowych	Wartości brzegowe
Drzewo klasyfikacji	Liść / kombinacja liści
Graf przyczynowo-skutkowy	Kombinacja przyczyn
Tablica decyzyjna	Kombinacja warunków
Testowanie maszyny stanowej	Przejście / sekwencja 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"> <li>• klasy równoważności</li> <li>• wartości brzegowe</li> <li>• tablice decyzyjne</li> <li>• grafy P-S</li> <li>• maszyna stanowa</li> <li>• drzewa klasyfikacji</li> <li>• przypadki użycia</li> <li>• testowanie losowe</li> </ul>	<ul style="list-style-type: none"> <li>• pokrycie instrukcji</li> <li>• pokrycie decyzji</li> <li>• pokrycie warunków</li> <li>• pokrycie MC/DC</li> <li>• pokrycie przepływu danych</li> <li>• pokrycie pętli</li> <li>• pokrycie ścieżek</li> <li>• testowanie mutacyjne</li> </ul>	<ul style="list-style-type: none"> <li>• testy eksploracyjne</li> <li>• zgadywanie błędów</li> <li>• ataki usterkowe</li> <li>• testy z listą kontrolną</li> <li>• testowanie ad hoc</li> </ul>

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

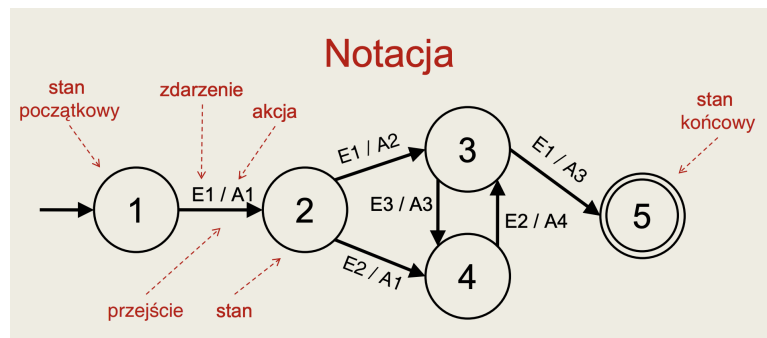
Grafy przyczynowo-skutkowe	
<ul style="list-style-type: none"> <li>• stosowane w tych samych sytuacjach co tablice decyzyjne</li> <li>• graficzna reprezentacja zależności między przyczynami a skutkami</li> <li>• prosta transformacja na tablice decyzyjne</li> </ul>	

### 3.1.1 Model maszyny stanowej

#### Testowanie maszyny stanowej

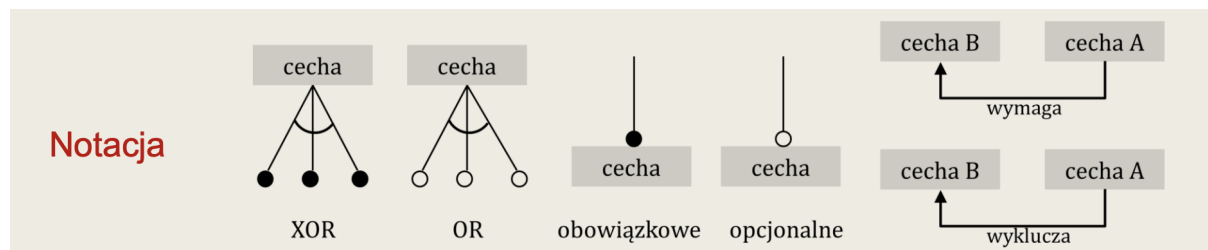
- reprezentacja możliwych stanów systemu i przejść między nimi
- metoda opisu dynamiki systemu

**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

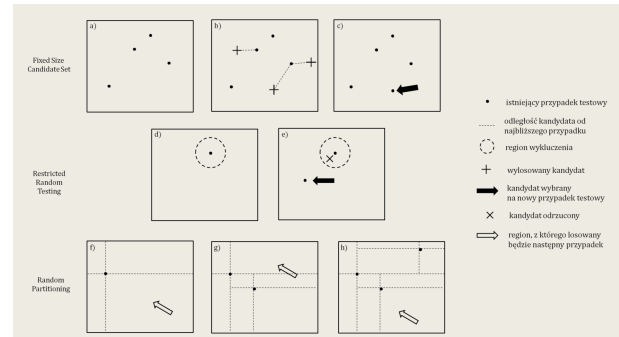
**Cechy**

- 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 (np. zewnętrzny, redundantny system)
  - interesuje nas tylko to, czy wykonanie zakończy się crashem
  - natura wyjścia sprawia łatwość weryfikacji (np. 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 jakiegokolwiek działania 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

W technikach białoskrzynkowych przypadki testowe projektuje się tak, by pokrywały określone 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"> <li>• <b>CFG</b> - Control Flow Graph</li> <li>• graficzna reprezentacja przepływu sterowania (graf skierowany) <ul style="list-style-type: none"> <li>– wierzchołki = bloki bazowe</li> <li>– krawędzie = przejścia między blokami</li> </ul> </li> <li>• blok bazowy = sekwencja instrukcji taka, że jeśli wykona się pierwsza z nich, wszystkie pozostałe w obrębie bloku wykonają się również</li> </ul>	<ul style="list-style-type: none"> <li>• oparty na grafie przepływ sterowania</li> <li>• zawiera informacje o operacjach na zmiennych</li> <li>• możliwe operacje: <ul style="list-style-type: none"> <li>– <b>d</b> (definition, definicja) – miejsce definiowania zmiennej</li> <li>– <b>u</b> (use, użycie) – miejsce użycia zmiennej</li> <li>– <b>k</b> (kill, zabicie) – miejsce usunięcia zmiennej z pamięci</li> </ul> </li> </ul> <p>Dla każdego węzła <math>B</math> definiujemy zbiory <math>d(B)</math>, <math>u(B)</math>, <math>k(B)</math> zawierające zmienne definiowane, używane lub zabijane w danym węźle.</p>

#### 4.1.1 Pokrycie instrukcyjne

##### Cechy:

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

##### Problemy praktyczne:

- jak definiować linię kodu? (linia fizyczna? wykonywalna?)
- czy rozważać pojedyncze instrukcje, czy bloki bazowe?
  - to wpływa na metryki pokrycia

#### 4.1.2 Pokrycie krawędziowe (branch testing)

- inna nazwa: **pokrycie przejść** między instrukcjami
- z punktu widzenia struktury, wymagane jest przejście po każdej **krawędzi CFG**, czyli testowany jest każdy **przepływ sterowania**
- **suita testowa spełniająca 100% pokrycia instrukcji nie musi spełniać 100% przejść między instrukcjami!**
- 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 (rozsądną wartością)
- najczęstszy przypadek: **diagramy przepływu procesów**
- nawet dla kodu bez pętli liczba wszystkich ścieżek może **rosnąć wykładniczo** względem punktów decyzyjnych
- **zbiór ścieżek liniowo niezależnych** to odpowiednik **bazy przestrzeni wektorowej**.
- **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.

#### 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  $\text{du}(n, v)$

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  $m$

#### 4.1.5 Pokrycie pętli

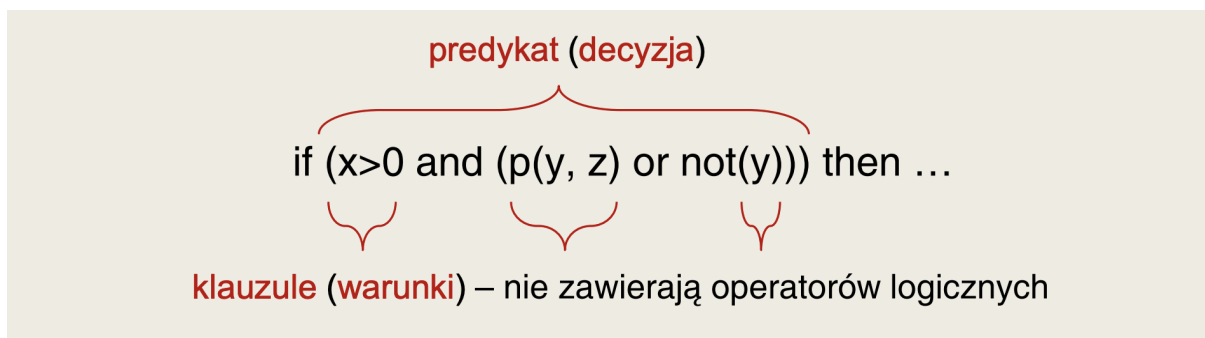
##### Co testujemy w przypadku pętli?

- problemy w **inicjalizowaniu** pętli (np. „błąd o jeden” w iteratorze)
- problemy z niezainicjalizowanymi zmiennymi (poprzez jednokrotne wykonanie pętli)
- 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
- defekty w pętlach są często **trudne do wykrycia**
- jak testować pętle? istnieją różne kryteria i podejścia

#### 4.2 Pokrycia logiczne



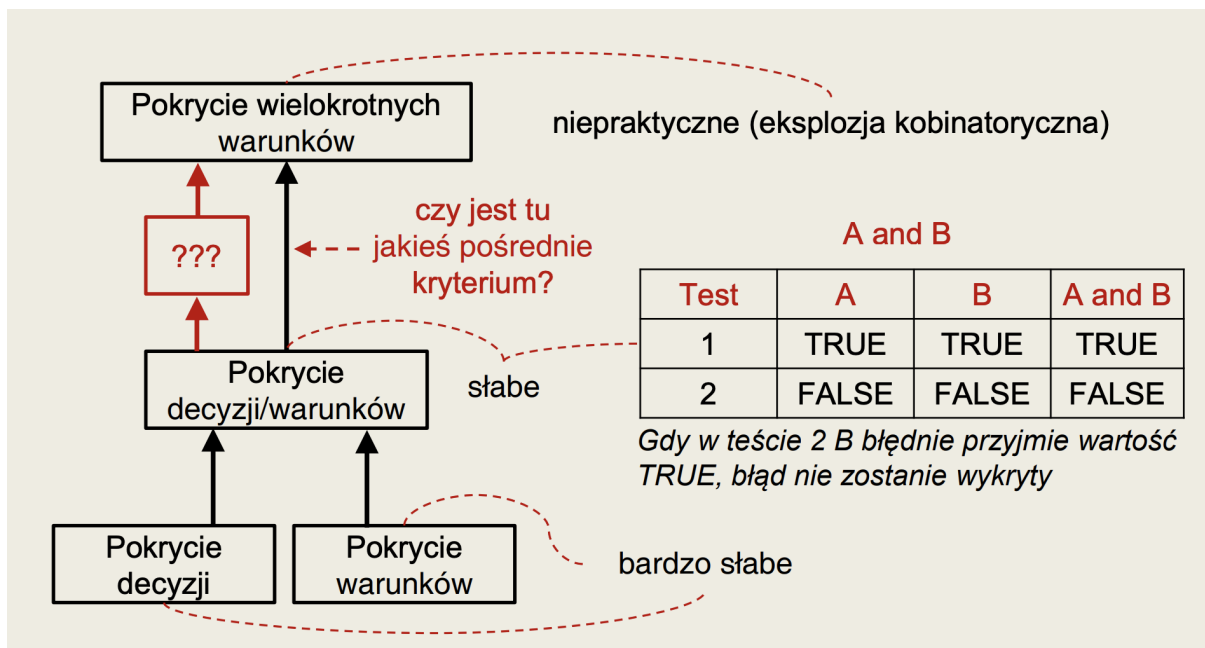
##### Problemy związane z pokryciem logicznym

- **Problem sterowania** - jakie wartości  $x, y, z$  zadać, aby program wykonał się dokładnie taką ścieżką, jaką chcemy? Istnieją metody pomagające w tym, np. wykonanie symboliczne kodu (np. KLEE).
- **Problem zwarcia** - zwarcie to optymalizacja kompilatora pozwalająca szybciej ewaluować formuły logiczne (leniwa ewaluacja).

#### 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**
  - wymaga przetestowania **wszystkich możliwych kombinacji** 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 klauzulę aktywną?**
  - niech D zawiera warunek A; chcemy, by A była klauzulą 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	<del>1 1 0</del>	1 0 0
C	1 0 1	<del>1 0 0</del>

### 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**
- popularne w środowisku producentów awioniki (standard DO-178C)
- **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** (short-circuiting) 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. Pobieżna **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", "sprawdź GUI pod kątem zgodności ze standardem Windows".
- **nie jest bardzo precyzyjna** – adresatem jest **tester dobrze znający system**, środowisko, słownictwo itp.
- 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"><li>• czasem utożsamiane z <b>testowaniem ad-hoc</b></li><li>• technika niebazująca na żadnym systematycznym podejściu czy technice</li><li>• opiera się na <b>doświadczeniu testera</b> w testowaniu</li></ul>	<ul style="list-style-type: none"><li>• tester korzysta z <b>wysokopoziomowej listy elementów do zanalizowania</b>, sprawdzenia lub zapamiętania</li><li>• modelowa <b>lista kontrolna</b> może dotyczyć różnych aspektów, np.:<ul style="list-style-type: none"><li>– charakterystyk jakościowych</li><li>– standardów GUI</li><li>– kluczowych operacji</li><li>– standardów kodowania</li></ul></li></ul>
Ataki usterkowe	Techniki oparte na modelach defektów
<ul style="list-style-type: none"><li>• idea: <b>atak</b> na oprogramowanie przez jego <b>interfejs</b>: użytkowy (GUI, API) lub systemowy (system plikowy, interfejs bazodanowy, interfejs OS)</li></ul>	<ul style="list-style-type: none"><li>• <b>taksonomia defektów</b> to system (hierarchicznych) kategorii stworzony jako pomoc w klasyfikowaniu defektów</li></ul>

## 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** =  $\frac{\#zabitych\ mutantów}{\#mutantów}$

### Zalety

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

### Wady

- **wymaga wielu zasobów, czasochłonny**
- efektywność zależy od doboru operatorów mutacyjnych
- 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.  $CC = \max \text{liczba ścieżek liniowo niezależnych}$
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)

- **Służy do 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 нефunkcjonalne.

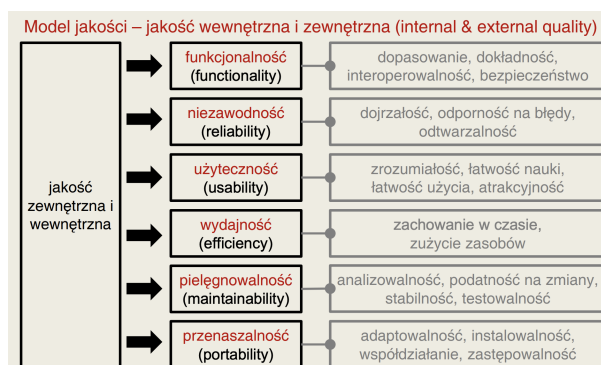
### 6.1 Jakość oprogramowania

Jakości jest pojęciem **niejednoznacznym** i **wielowymiarowym**.

Różne spojrzenia wg Garvina:

- jakość oparta na **produkcie** (np. charakterystyki jakościowe)
- jakość oparta na **użytkowniku** (funkcjonalność, fitness for use)
- jakość oparta na **wytwarzaniu** (jakość „techniczna” procesu)
- jakość oparta na **wartości** (czas vs. wysiłek vs. koszt)
- jakość **transcendentna** (nieoperacyjna, trudno mierzalna, np. „grywalność” gry komputerowej)

#### 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
  - baza książek i innych materiałów o bezpieczeństwie
  - portal społecznościowy
  - projekty związane z bezpieczeństwem
  - dokumentacja, modele
- **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** - jak łatwo zrozumieć co program robi i dlaczego mielibyśmy go używać?
- **Łatwość nauki** - łatwość zrozumienia jak program działa
- **Łatwość obsługi** - czy obsługa programu jest intuicyjna?
- **Atrakcyjność** - czy użytkownik chętnie używa programu?

#### Metryki

Testowanie użyteczności jest ukierunkowane na pomiar:

- **efektywności**
  - np. stopień osiągnięcia przez użytkownika celów, z określoną dokładnością
- **wydajności**
  - ilość zasobów zużytych
- **satysfakcji** klienta z użytkowania produktu

## TESTOWANIE UŻYTECZNOŚCI

Proces	
<b>Formatywne testowanie użyteczności</b> - przeprowadzane <b>iteracyjnie</b> w kolejnych <b>fazach projektowania</b> i prototypowania.	<b>Całościowe testowanie użyteczności</b> - przeprowadzane <b>po implementacji</b> .

Rodzaje	
<b>Formalne testowanie użyteczności</b> - często wymaga uprzedniego <b>przygotowania</b> rzeczywistych lub reprezentatywnych <b>użytkowników</b> (dostarczenie scenariuszy zadań, instrukcji)	<b>Nieformalne testowanie użyteczności</b> - pozwala użytkownikowi <b>eksperymentować</b> z oprogramowaniem, aby obserwatorzy ocenili jak trudna dla użytkownika jest praca z systemem

Walidacja
powinna być przeprowadzona <b>w warunkach</b> tak <b>bliskich rzeczywistym</b> warunkom użytkowania oprogramowania, jak to tylko możliwe, np. w <b>laboratorium użyteczności</b> .

Wytyczne (guidelines)
są stosowane aby uzyskać <b>spójne podejście</b> do wykrywania i raportowania defektów użyteczności <b>na wszystkich etapach</b> cyklu życia.

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

#### 6.1.6 Wydajność

- **Zdolność** oprogramowania do **zapewnienia odpowiedniej efektywności** w działaniu, relatywnie do ilości zużytych zasobów
- Systemy o znaczeniu krytycznym muszą dostarczać swoje funkcje w ściśle określonym czasie
- Dla systemów czasu rzeczywistego (i innych) ważne jest **zużycie zasobów**.
- Najczęstsza przyczyna błędów wydajności: **błędy projektowe** (a więc trudne do usunięcia w późniejszych fazach testów).

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

### 6.1.7 Pielęgowalność

- Pielęgowalność to **łatwość modyfikowania oprogramowania**.
- Oprogramowanie się nie zużywa, ale staje się **przestarzałe**.
- **Testowanie pielęgowalności**
  - Zwykle nie przy pomocy skryptów testowych
  - Większość defektów jest **niewidoczna dla testowania dynamicznego**
  - Najlepiej sprawdzają się **techniki statyczne**

#### Podcharakterystyki pielęgowalności i problemy z nimi związane:

<ul style="list-style-type: none"><li>• <b>Analizowalność</b> - łatwość diagnozowania problemów<ul style="list-style-type: none"><li>– spaghetti code</li><li>– brak dobrej dokumentacji</li><li>– brak lub złe standardy/zalenia</li><li>– abstrakcja kodu</li></ul></li><li>• <b>Modyfikowalność</b> - zdolność do wprowadzania zmian<ul style="list-style-type: none"><li>– brak kohezji</li><li>– zły styl kodowania</li><li>– kiepska dokumentacja</li></ul></li></ul>	<ul style="list-style-type: none"><li>• <b>Stabilność</b> - zdolność do unikania niespodziewanych efektów na skutek zmian<ul style="list-style-type: none"><li>– związanie</li><li>– brak kohezji</li><li>– jakość wymagań</li></ul></li><li>• <b>Testowalność</b> - łatwość walidacji po wprowadzonej zmianie<ul style="list-style-type: none"><li>– brak lub zła dokumentacja</li><li>– brak komentarzy w kodzie</li><li>– antywzorce projektowe</li><li>– zła konwencja nazewnictwa</li><li>– brak instrumentacji kodu</li></ul></li></ul>
---	---

### 6.1.8 Przenaszalność

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

#### Podcharakterystyki przenaszalności

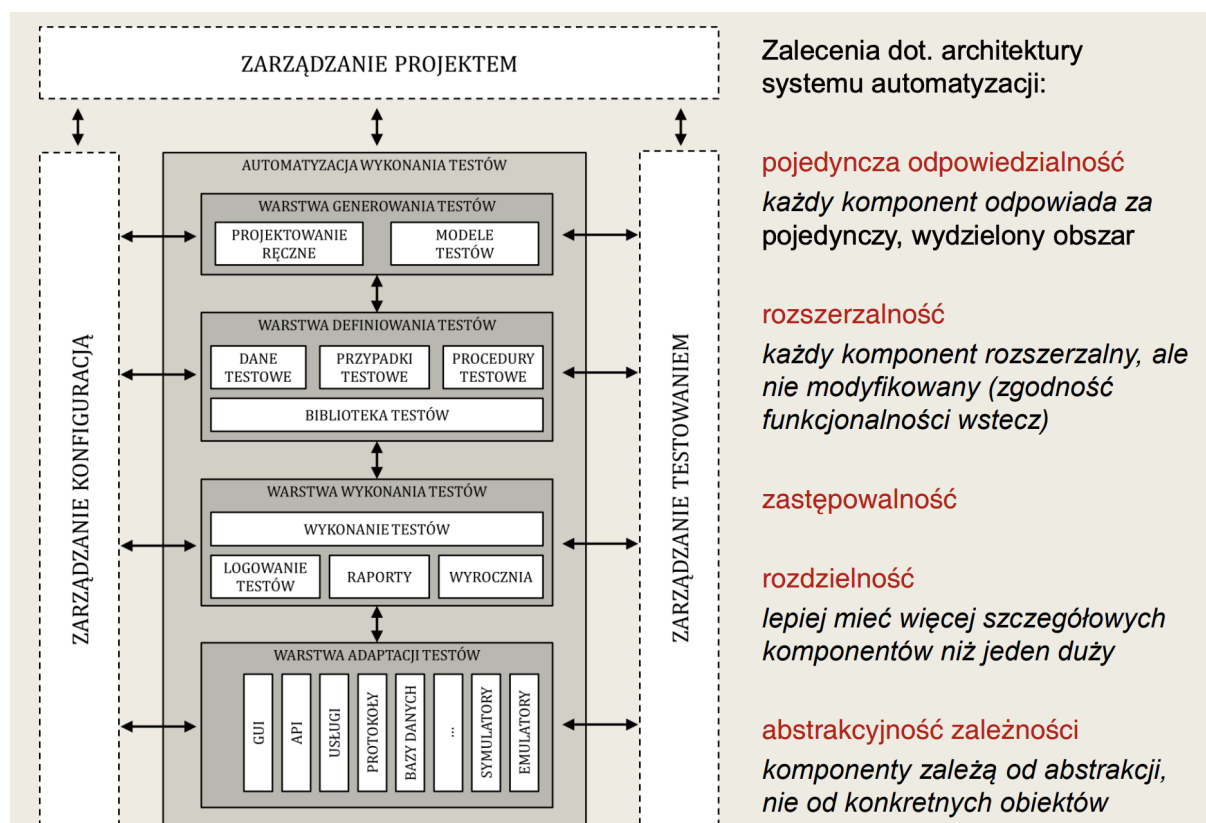
- **Adaptowalność** – zdolność do adaptacji w innym środowisku bez podejmowania akcji innych niż przewidziane w tym celu.
- **Zastępowalność**
- **Instalowalność**
- **Koegzystencja** – zdolność do współistnienia z innym, niezależnym oprogramowaniem dzielącym środowisko – np. „przywłaszczenie” standardowego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

### 7.1 Generyczna architektura automatyzacji testów



## 7.2 Automatyczna generacja danych testowych

- metoda losowa
- generacja z rozkładu prawdopodobieństwa
- generacja z modelu
- generacja na podstawie symbolicznego wykonania kodu
- generacja metodyczna (np. pair-wise)
- generacja na podstawie danych zewnętrznych

## 7.3 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

## 7.4 Testowanie oparte na modelu (MBT)

- **Podstawowa idea:** ulepszyć jakość i efektywność projektu i implementacji testów przez:
  - projekt wyczerpującego modelu MBT, zwykle z użyciem narzędzi,
  - użycie modelu jako specyfikacji projektu testów, pozwalającego na automatyczną generację przypadków testowych z modelu
- **Rodzaje modeli:** strukturalne, behawioralne, danych.

### Efektywność

- modelowanie ułatwia **komunikację** z interesariuszami
- **zrozumienie**
- **łatwiejsze zaangażowanie** interesariuszy modelem
- **łatwa identyfikacja „problematicznych” 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 suitey testów mogą być generowane z tego samego modelu
- **redukcja kosztów przy zmianie wymagań** - MBT pomaga zredukować koszty utrzymania gdy zmieniają się wymagania, bo model MBT dostarcza „single point of maintenance”

Kryteria wyboru testów	
Oparte na pokryciu	Inne
<ul style="list-style-type: none"> <li>• <b>Wymagania połączone z modelem</b> - elementy modelu są połączone z wybranymi wymaganiami. Pełne pokrycie wymagań odpowiada zestawowi testów całkowicie pokrywających wybrany zbiór wymagań.</li> <li>• <b>Elementy modelu MBT</b> - bazuje na <b>wewnętrznej strukturze modelu</b>. Definiuje się elementy pokrycia, a testy powinny je pokrywać.</li> <li>• <b>Oparte na danych</b> - związane są z takimi technikami projektowania testów jak: <ul style="list-style-type: none"> <li>– podział na <b>klasy równoważności</b></li> <li>– analiza <b>wartości</b> brzegowych</li> <li>– testy kombinatoryczne (np. pair-wise)</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• <b>Losowe</b> - polega na <b>losowym przechodzeniu przez model</b>, wszystkie przejścia są równo prawdopodobne. W <b>podejściu stochastycznym</b> wybór oparty o <b>rozkład prawdopodobieństwa</b>. Model reprezentuje profil użycia (tzw. <b>profil operacyjny</b>).</li> <li>• <b>Oparte na scenariuszu/wzorcu</b> - scenariuszem może być use case lub scenariusz użycia; wzorzec = częściowo zdefiniowany scenariusz, który można zastosować do modelu MBT aby wyprowadzić jeden lub wiele testów.</li> <li>• <b>Sterowane projektem</b> - podejście oparte na dodatkowej informacji projektowej, która została dodana do modelu aby wspierać zarządzanie testami i/lub aby osiągnąć specyficzne cele testowe w projekcie.</li> </ul>

## 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 ryzyka określona przez prawdopodobieństwo i wpływ; może być 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"> <li>• burza mózgów</li> <li>• listy kontrolne</li> <li>• historia awarii</li> <li>• wywiady eksperckie</li> </ul>	<ul style="list-style-type: none"> <li>• szablony ryzyk</li> <li>• niezależna ocena</li> <li>• doświadczenie z poprzednich projektów</li> </ul>

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

### 8.1.1 Analiza ryzyka

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

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

## 8.3 Planowanie testów.

### 8.3.1 Szacowanie

Szacowanie pozwala na lepszą **alokację zasobów** i lepszą **organizację** projektu. Jest podstawą do podjęcia decyzji o ograniczeniu zakresu projektu, jeśli estymowany koszt lub czas jest zbyt duży.

Szacowanie jest **trudne**. Problemy wynikają nie tylko z samej natury problemu, lecz także z tzw. **przyczyn politycznych**.

**Rola szacowania**

- określenie **pracochłonności** projektu
- określenie **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"><li>• <b>opisanie</b> obowiązujących w organizacji lub projekcie <b>reguł, standardów, celów</b> do osiągnięcia, stanowiących punkt odniesienia podczas prac projektowych</li><li>• akceptowanie/zezwalanie/potwierdzanie wykonania określonych czynności</li><li>• utrwalenie danych</li><li>• pomoc w monitorowaniu i kontroli procesów</li><li>• pomoc w opanowaniu złożoności projektów</li><li>• mniej lub bardziej formalna <b>platforma komunikacji</b></li></ul>	<ul style="list-style-type: none"><li>• <b>Polityka testów</b> – wysokopoziomowy opis <b>zasad, podejść</b> i głównych <b>zadań</b> organizacji dotyczących testowania.</li><li>• <b>Strategia testów</b> – wysokopoziomowy opis <b>poziomów</b> testów do wykonania oraz testów w ramach tych poziomów dla organizacji lub programu.</li><li>• <b>Plan testów</b> – opis <b>zakresu, metod, zasobów i harmonogramu</b> czynności testowych dla projektu.</li><li>• <b>Dziennik wykonania testów</b> – zapis czynności testowych.</li></ul>

### 8.3.3 Metryki

Na podstawie metryk możemy:

- **mierzyć postęp** procesu testowego
- **obliczyć zwrot** z inwestycji (ROI), tzn. czy proces testowy daje pożądane korzyści
- **ocenić** i porównać różne możliwe **podejścia** do testowania
- **ocenić** i kontrolować **wydajność** procesu testowego
- **ocenić** i kontrolować **poprawę** procesu testowego
- **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, kwestionariuszy



$MTTF$  = Mean Time To Failure - średni czas do awarii

$MTTR$  = Mean Time To Repair - średni czas do naprawy

$MTBF$  = Mean Time Between Failures - średni czas między awariami

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

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

$$MTBF = MTTF + MTTR$$

Wraz ze wzrostem dojrzałości oprogramowania wzrasta MTTF.

#### Przykłady metryk

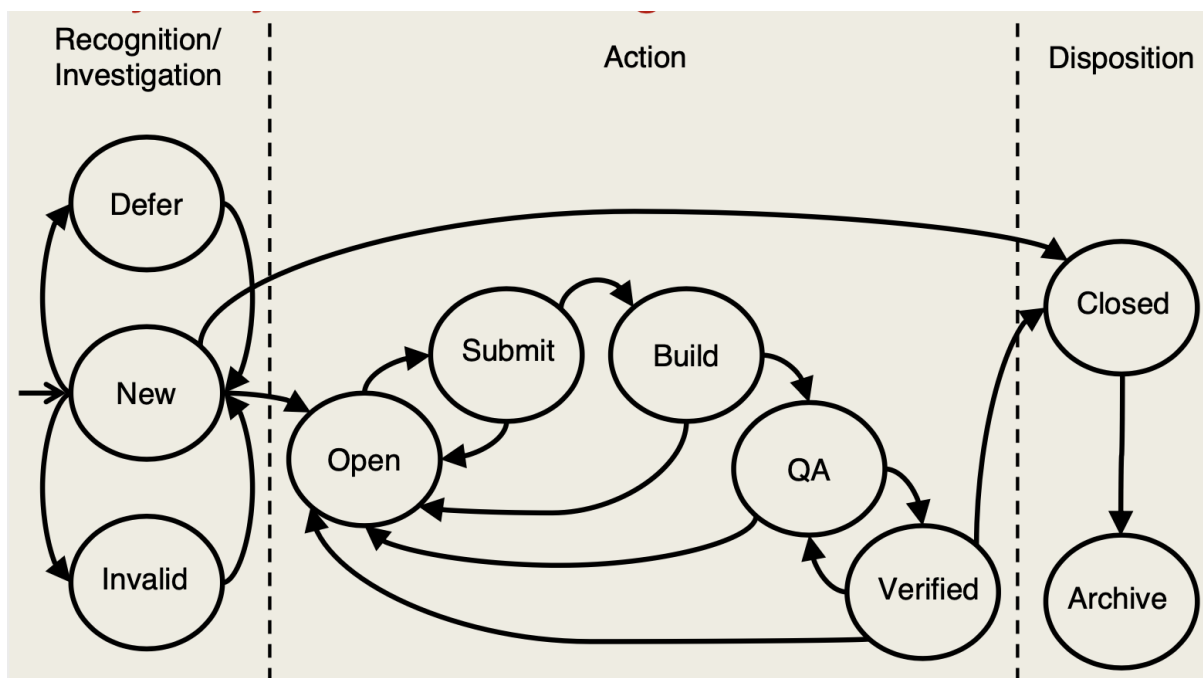
- Metryki **ryzyka produktowego** - pokrycie ryzyk
- Metryki **defektów** - MTTF, MTBF, analiza napraw, zgłoszonych defektów
- Metryki **przypadków testowych**
- Metryki **pokrycia** - stopień pokrycia wymagań, kodu, klas równoważności itd.
- Metryki **pewności** - stabilność, niezawodność, ocena klienta

### 8.4 Zarządzanie incydentami

#### 8.4.1 IEEE 1044.

Cykl życia defektu wg IEEE 1044.

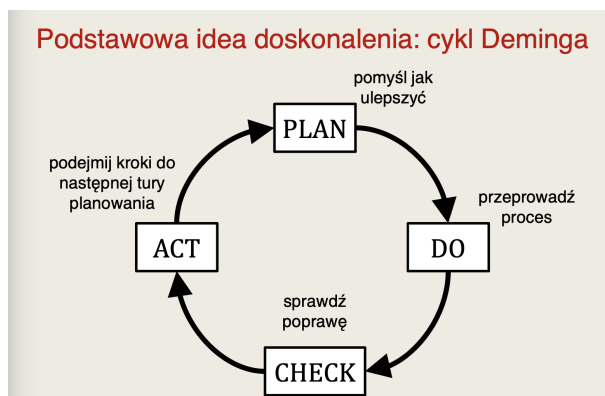
1. **Rozpoznanie (recognition)** – zaobserwowanie anomalii (incydent) wskazującej na potencjalny defekt – może nastąpić w dowolnej fazie cyklu życia oprogramowania
2. **Badanie (investigation)** – badanie incydentu; może wykryć powiązane problemy i zaproponować rozwiązania
3. **Działanie (action)** – możemy chcieć rozwiązać defekt lub podjąć akcje zapobiegania wystąpienia podobnych defektów w przyszłości; po rozwiązaniu muszą nastąpić testy regresji i testy potwierdzające; testy dotychczas blokowane przez defekt mogą zostać wykonane
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"> <li>• <b>kompletne</b> - nie brakuje żadnych ważnych szczegółów,</li> <li>• <b>zwięzłe</b> - nie zawiera nieistotnych informacji,</li> <li>• <b>precyzyjne</b> - nie wprowadza czytelnika w błąd,</li> <li>• <b>obiektywne</b> - bazuje na faktach, nie atakuje nikogo.</li> </ul>	<ul style="list-style-type: none"> <li>• dotkliwość (severity)</li> <li>• priorytet</li> <li>• wartość dla klienta</li> <li>• sukces misji (mission safety)</li> <li>• harmonogram projektu</li> <li>• koszt projektu</li> <li>• ryzyko projektu</li> <li>• jakość projektu</li> <li>• kwestie społeczne</li> </ul>

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

### Przykłady aktywności QA.

- **definiowanie i implementacja procesów**
  - metodyka wytwarzania oprogramowania
  - zarządzanie projektami
  - zarządzanie konfiguracją
  - zarządzanie wymaganiami
  - metody pomiaru 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** (stopień komplikacji kodu) [zalecane < 10]
- **złożoność Halsteada** (# operatorów i operandów); krytykowana!
- **przepływ informacji** (z i do modułu)
- **złożoność systemu** (dla pomiaru pielęgnalności)
- **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.