Notatki z kursu Testowanie oprogramowania

Małgorzata Dymek

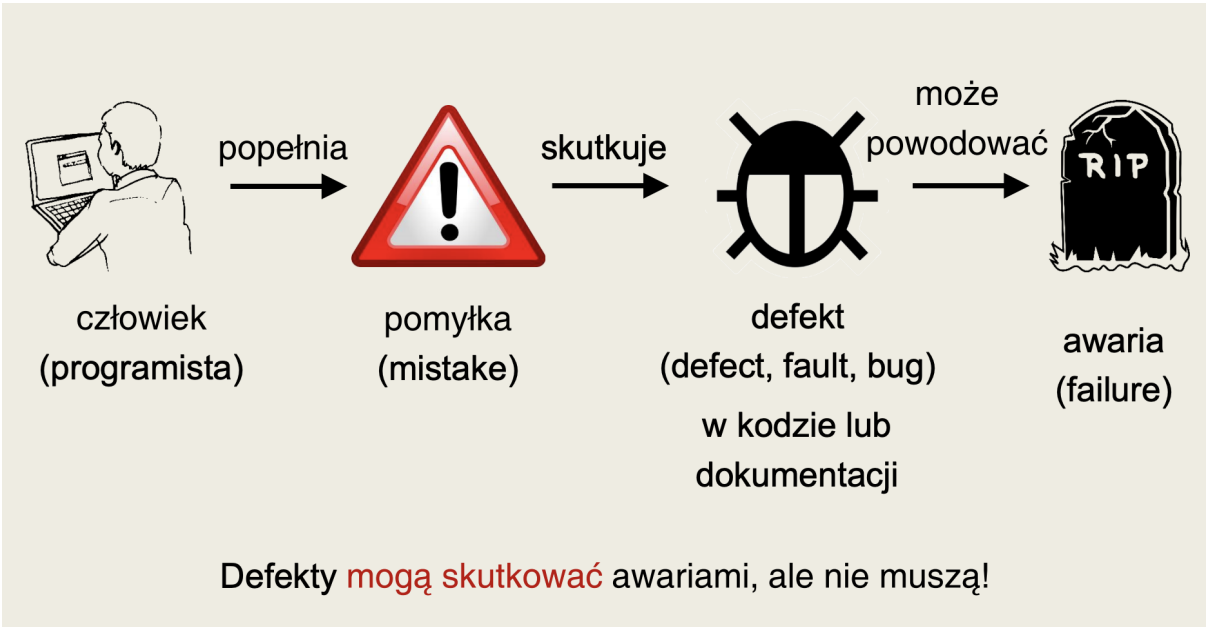
2019/20, semestr zimowy

1 Wprowadzenie.

1.1 Definicje.

Definicja testowania jest niejednoznaczna:

- Testowanie to wykonywanie oprogramowania z intencją wykrywania tkwiących w nim błędów.
- Testowanie to krytyczne sprawdzanie, obserwacja i ewaluacja jakości oprogramowania.
- Testowanie to proces analizowania fragmentu oprogramowania w celu wykrycia różnic pomiędzy istniejącymi a pożądanymi warunkami (czyli defektów) oraz w celu oceny cech tego fragmentu oprogramowania [IEEE].



Pomyłka - człowiek robi coś źle.

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

Błąd (error) – 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.

Walidacja (validation) – ewaluacja oprogramowania w końcowej fazie procesu budowy, dokonywana w celu potwierdzenia zgodności z założonymi celami użycia (**are we building the right thing?**).

Weryfikacja (verification) – ewaluacja we wczesnych fazach, sprawdzająca, czy produkt danej fazy spełnia wymagania (zwykle techniczne) ustalone podczas poprzedniej fazy (**are we building the thing right?**).

1.1.1 Debugowanie.

Testowanie to nie debugowanie.

Testowanie znajduje awarie.

Debugowanie, na podstawie informacji o awarii:

- **lokalizuje miejsce usterki** powodującej tę awarię
- usuwa (**naprawia**) usterkę

Testowanie sprawdza, czy usterka została poprawnie usunięta.

1.2 Rola testowania.

- dobrze zaprojektowany, zdany test redukuje poziom ryzyka
- testowanie zwiększa przekonanie o jakości jeśli znajduje mało defektów lub nie znajduje ich w ogóle
- **jakość systemu** wzrasta gdy defekty są naprawiane
- wymagania kontraktowe i prawne, standardy przemysłowe
- jedna z czynności **QA** - Quality Assurance.

1.3 Cele testowania.

- znajdowanie defektów (np. testy jednostkowe)
- uzyskanie pewności co do poziomu jakości (np. testy akceptacyjne)
- dostarczenie informacji do podjęcia decyzii (np. ocena jakości systemu)
- zapobieganie pojawiania się defektów (np. projektowanie testów we wczesnych fazach życia)

Decyzja o zakresie i ilości testów zależy od **ryzyka** (ryzyko techniczne, biznesowe i safety risk) oraz **ograniczeń projektowych** (czas, budżet).

1.4 7 uniwersalnych zasad testowania.

1. Testowanie ujawnia usterki
2. Testowanie gruntowne jest niewykonalne
3. Wczesne testowanie
4. Kumulowanie się błędów
5. Paradoks pestycydów
6. Testowanie zależy od kontekstu
7. Mylne przekonanie o braku błędów

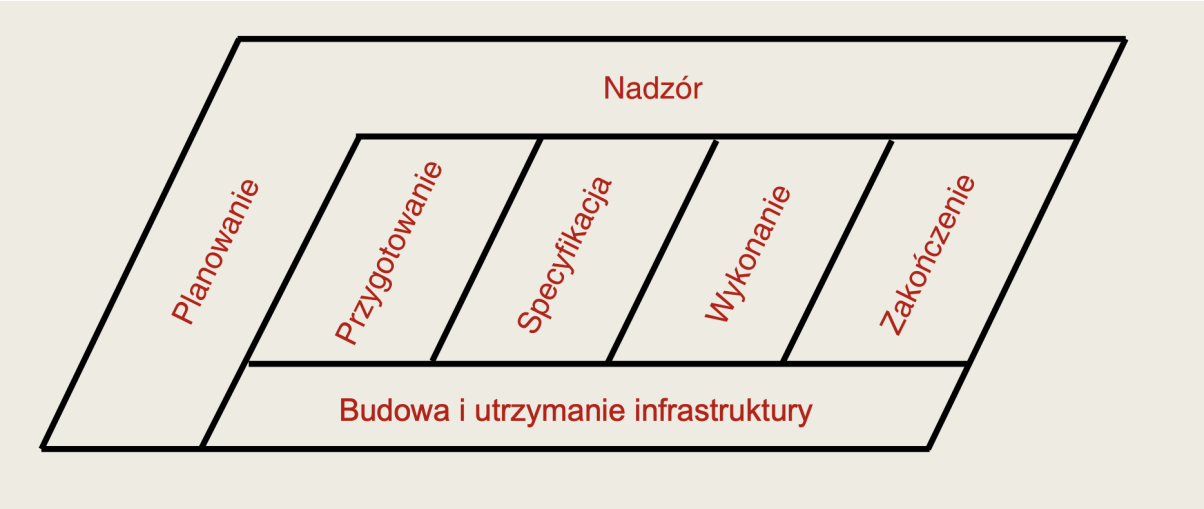
1.5 Normy i standardy związanie z testowaniem

- **IEEE 829** – dokumentacja testowa
- **IEEE 1008** – standard dla testowania jednostkowego
- **IEEE 1028** – standard dla przeglądów i audytów
- **ISO 9126** – model jakości (stara)
- **ISO/IEEE 25000** – model jakości (nowa)
- **ISO/IEEE 29119** – Software Testing Standard

2 Testowanie w cyklu życia.

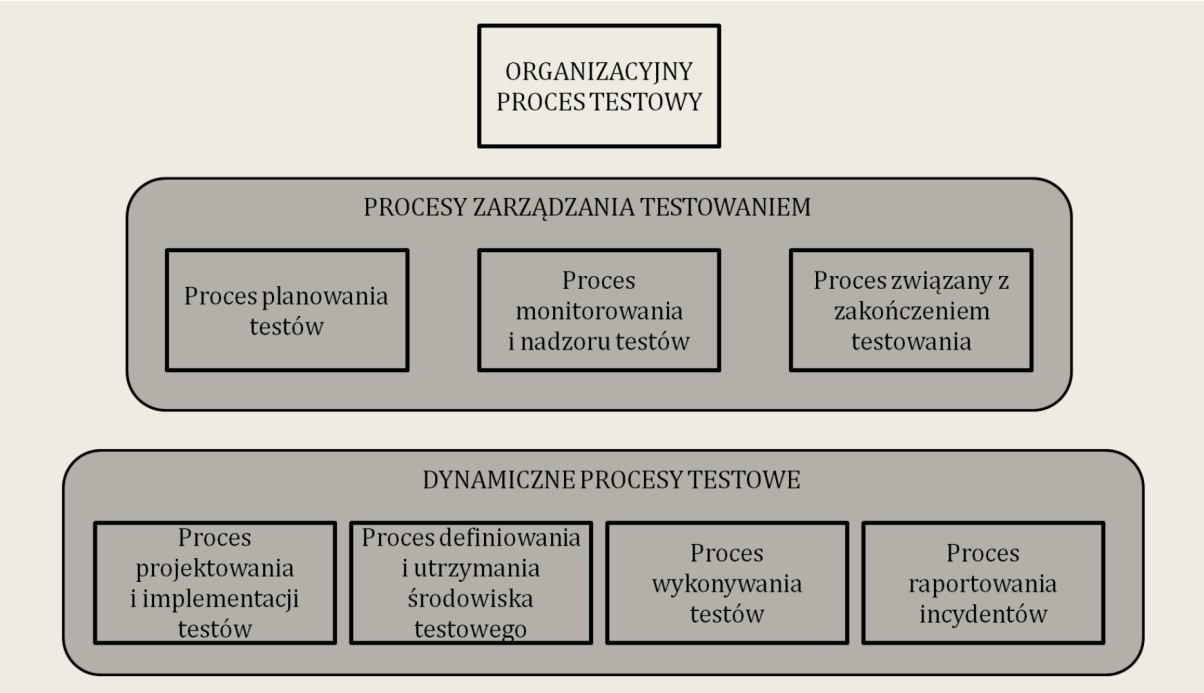
2.1 Modele procesu testowego

2.1.1 TMAP lifecycle model



Proces jest **generyczny**, tzn. może być stosowany do wszystkich poziomów i typów testów. Każda faza dzieli się na określone czynności.

2.1.2 ISO 29119



2.2 Miejsce testowania w modelu cyklu życia.

Testowanie musi wpasować się w jakiś model.

Cechy dobrego testowania.

- każdej czynności wytwórczej odpowiada czynność związana z testowaniem
- każdy poziom testowania ma zdefiniowane cele
- analiza i projektowanie testów dla danego poziomu powinny rozpoczynać się już podczas odpowiadającej im fazy wytwarzania
- testerzy powinni uczestniczyć w przeglądach już od wczesnych wersji dokumentacji tworzonej podczas wytwarzania

Poziomy a typy testów.
Poziomy testów są **ortogonalne** do typów testów, tzn.:

- na danym poziomie testów można wykonywać dowolne typy testów
- dany typ testów może być wykorzystany na dowolnym poziomie

Oczywiście niektóre typy i poziomy są silniej ze sobą powiązane, np.:

- testy jednostkowe to zazwyczaj testy oparte na strukturze
- na etapie testów akceptacyjnych raczej stosuje się testy funkcjonalne i niefunkcjonalne
- testy funkcjonalne zazwyczaj oparte są na specyfikacji (black-box)
- testy integracyjne zazwyczaj dotyczą funkcjonalności

2.3 Poziomy testów.

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

2.3.1 Testy jednostkowe.

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

- inne nazwy: testowanie modułowe, unit testing
- defekty szukane w izolacji od reszty systemu
- wykorzystanie zaślepek i sterowników
- wykorzystanie frameworków do testów jednostkowych i debugowania
- przeprowadzane przez deweloperów – autorów testowanego kodu
- usuwanie defektów bezpośrednio po znalezieniu
- brak formalnego procesu zarządzania defektami

TTD - Test Driven Developement.

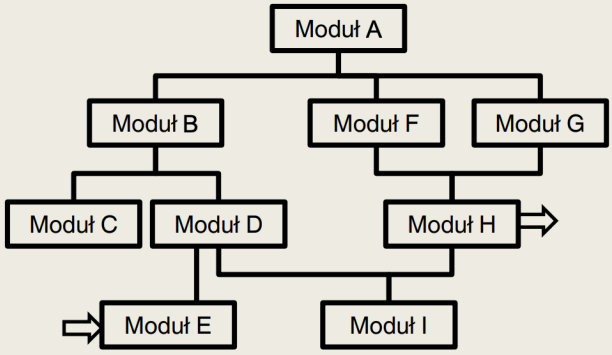
- **Napisanie testu** - sprawdza, że programista rozumie zachowanie nowego kodu
- **Uruchomienie testu**, który jest nie zdany(testuje uprząż testową i sam test; pokazuje, że wystąpi awaria, gdy kod będzie błędny)
- **Napisanie kodu** w minimalnej ilości, wystarczającej do zdania testu; jeśli konieczne, dokonanie refaktoryzacji

2.3.2 Testy integracyjne.

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

- testuje interfejsy i interakcje
- im większy zakres integracji, tym trudniej izolować defekty
- rozumienie architektury testowanych modułów/systemów
- może odbywać się na wielu poziomach (np. integracja systemów)

Strategie testów integracyjnych



H – zwraca wynik
E – pobiera wejście
B+C+D+E – jedna funkcjonalność
główna transakcja: wejście-wyjście

połączenie oznacza, że moduł położony wyżej wywołuje niższy

top-down: A-BFG > B-CD > F-H, G-H > D-EI, H-I
bottom-up: D-EI > H-I > B-CD > F-H, G-H > A-BFG
funkcjonalne: B-CD, D-E > …
sekwencja przeprowadzania transakcji: E-D-B-A-G-H > …
big-bang: wszystko integrowane i testowane naraz (nie polecane)

2.3.3 Testy systemowe.

Podstawa testów	wymagania na system, przypadki użycia, specyfikacja funkcjonalna, raporty analizy ryzyka
Typowe obiekty	system, podręczniki użytkownika i operatora, konfiguracja systemu

- sprawdza zachowanie systemu jako całości
- zakres testu określony w planie testów
- środowisko testowe podobne do produkcyjnego – minimalizacja ryzyka awarii zależnych od środowiska
- często konieczność testowania z niekompletnymi wymaganiami
- zwykle przeprowadzane przez niezależny zespół testerski

2.3.4 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

- cel: zyskanie zaufania do systemu
- znajdowanie defektów nie jest głównym celem
- często przeprowadzane przez klienta lub użytkownika
- ocenia gotowość systemu, ale niekoniecznie ostatni etap testów

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

2.4 Typy testów.

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

- **Testowanie funkcjonalne** - funkcja wykonywana przez oprogramowanie - **co system robi**.
- **Testowanie niefunkcjonalne** - niefunkcjonalna charakterystyka jakościowa, np. niezawodność czy użyteczność - **jak system działa**.
  - przykłady: testy wydajności, obciążenia, użyteczności, utrzymania, niezawodności, przenaszalności, bezpieczeństwa
  - mierzą charakterystyki jakościowe systemu
  - wyrażalne ilościowo (np. czas odpowiedzi w testach wydajności)
  - mogą odnosić się do modelu jakości, na przykład:
    * normy: ISO 9126 – Software Product Quality, ISO 25000
    * model zabezpieczeń (np. wg OWASP)
    * model użyteczności (np. heurystyki Nielsena)
    * profil operacyjny (dla testów wydajności)
- **Testowanie strukturalne** - struktura lub architektura systemu
  - oparte na strukturze
    * np. kod, graf przepływu sterowania, struktura menu, model procesu biznesowego
  - zwykle wykonywane po testach czarnoskrzynkowych, aby sprawdzić stopień przetestowania i wyrazić go ilościowo (pokrycie)
  - pokrycie = stopień w jakim dana struktura została przetestowana
    * wyrażane w % pokrytych elementów
  - uzyskanie 100% może być nieosiągalne
- **Retesty i testy regresji** - związany ze zmianą, tzn. potwierdzenie usunięcia defektów (retesty) oraz poszukiwanie niezamierzonych zmian (regresja) - **związane ze zmianami**
  - **regresja** = zjawisko pogarszania się jakości systemu na skutek wprowadzanych w nim zmian
  - testy regresji to testy niezmienionych fragmentów programu po dokonaniu zmiany w programie; są naturalnym kandydatem do automatyzacji
  - **retest** = przetestowanie naprawionego fragmentu systemu

2.5 Statyczne techniki testowania

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

2.5.1 Przeglądy.

- Sposoby ręcznego testowania oprogramowania (np. kodu, dokumentacji)
- Mogą być wykonane przed fazą testów dynamicznych
- Pozwalają wykryć defekty wcześnie w cyklu życia (np. w wymaganiach), przez co usunięcie tych defektów jest tanie
- Aktywność manualna, ale może być wsparta narzędziami
- Idea: sprawdzić produkt i skomentować go
- Przeglądom może podlegać wszystko (specyfikacja wymagań, projekt, kod, plan testów, specyfikacja testów, przypadki testowe, skrypty testowe, podręczniki użytkownika, strony www itd.)

Korzyści z przeglądów:

- 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
nieformalny	brak formalnego procesu, może przybrać formę programowania w parach lub nief
przejrzenie	prowadzone przez autora; opcjonalne przygotowanie przed spotkaniem, opcjonalny
techniczny	przeszkolony moderator, przygotowanie przed spotkaniem, zdefiniowany proce
inspekcja	przeszkolony moderator, wyróżnione role i metryki, formalny proces, przygotowanie przed spotkani

2.5.2 Inspekcja

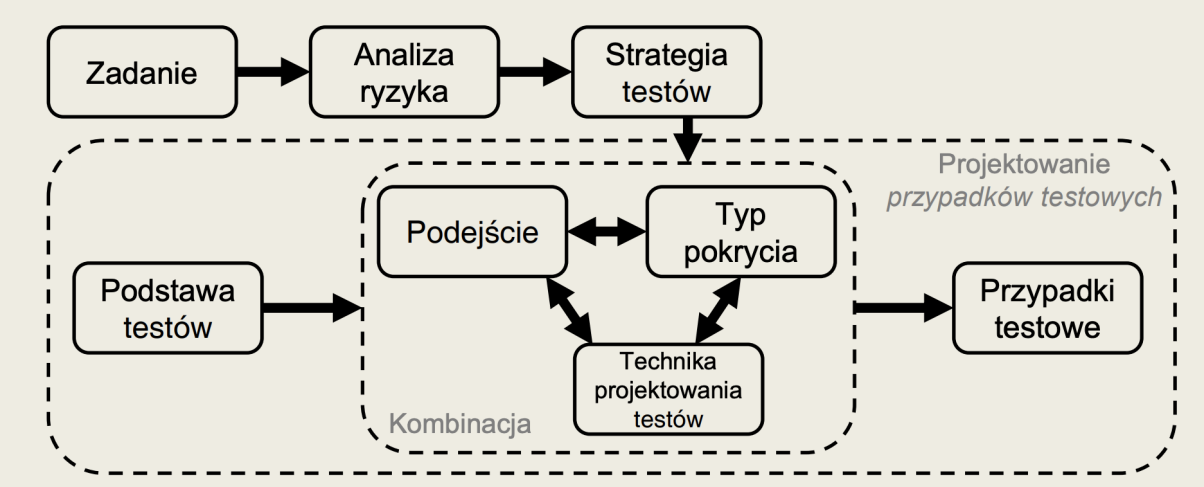
Role:

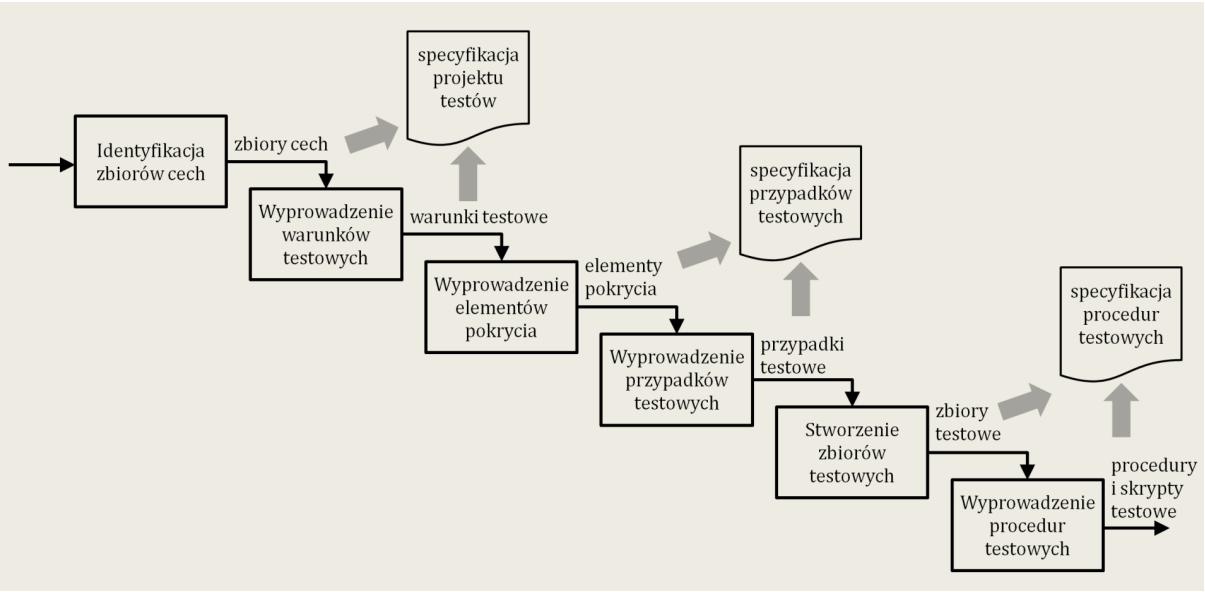
- kierownik
- moderator
- autor
- przeglądający
- protokolant

Proces:

- Planowanie
- Rozpoczęcie
- Przygotowanie indywidualne
- Kontrola/ocena/zapis wyników
- Poprawki
- Zakończenie

2.6 Projektowanie testów.

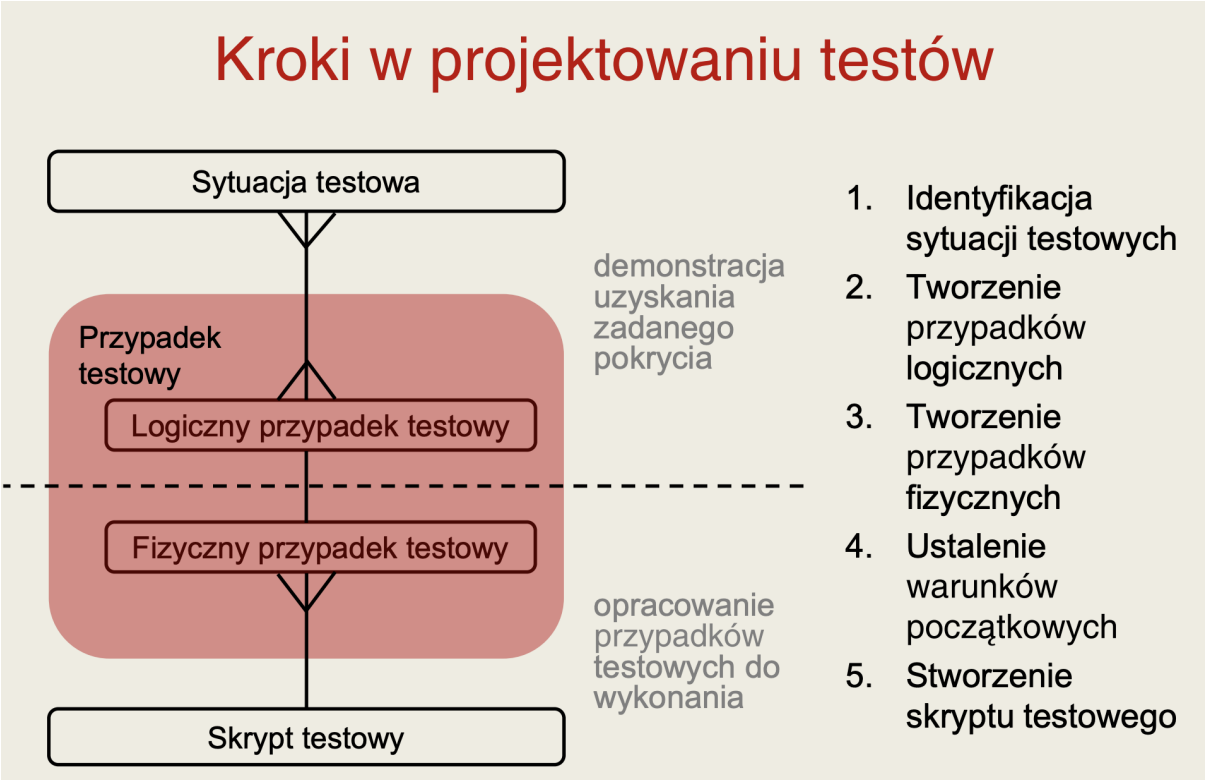




Warunek testowy (test condition) – element lub zdarzenie które może być sprawdzone przez jeden lub więcej przypadków testowych (np. funkcja, transakcja, atrybut jakościowy, element strukturalny).

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.

- systematyzują podejście do testowania
- efektywne w znajdowaniu możliwych awarii
- mogą uchronić przed redundantnymi testami
- powtarzalne
- oparte na modelach (np. działania oprogramowania)
- dostarczają elementy pokrycia

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.

Pokrycie - stopień przetestowania elementów pokrycia dla zadanego warunku testowego, mierzone procentowo (0-100%), dotyczy określonego kryterium.

Zbiór testów spełnia kryterium pokrycia, jeśli pokrycie dla tych testów wynosi 100%

Kryterium K1 **subsumuje** kryterium K2, jeśli każdy zestaw testów spełniający K1 spełnia również K2.

3.1 Techniki projektowania testów

oparte na specyfikacji	**oparte na strukturze**	**oparte na doświadczeniu**
specyfikacja używa modeli, formalnych lub nieformalnych	testy tworzone na podstawie informacji o strukturze programu	testy tworzone na podstawie wiedzy, doświadczenia i intuicji testerów
przypadki testowe mogą być tworzone systematycznie z tych modeli	można mierzyć stopień pokrycia oprogramowania i dodawać nowe testy by zwiększyć pokrycie	inne źródło informacji to wiedza o możliwych defektach i ich umiejscowieniu
• klasy równoważności • wartości brzegowe • tablice decyzyjne • grafy P-S • maszyna stanowa • drzewa klasyfikacji • przypadki użycia • testowanie losowe	• pokrycie instrukcji • pokrycie decyzji • pokrycie warunków • pokrycie MC/DC • pokrycie przepływu danych • pokrycie pętli • pokrycie ścieżek • testowanie mutacyjne	• t. eksploracyjne • zgadywanie błędów • ataki usterkowe • t. z listą kontrolną • testowanie ad hoc

Analiza wartości brzegowych

- technika oparta na podziale na klasy równoważności

- działa dla klas, na których zadany jest porządek liniowy
- wartość brzegowa = wartość min/max zadanej klasy równoważności
- można rozważać wartości brzegowe dla wejścia i wyjścia

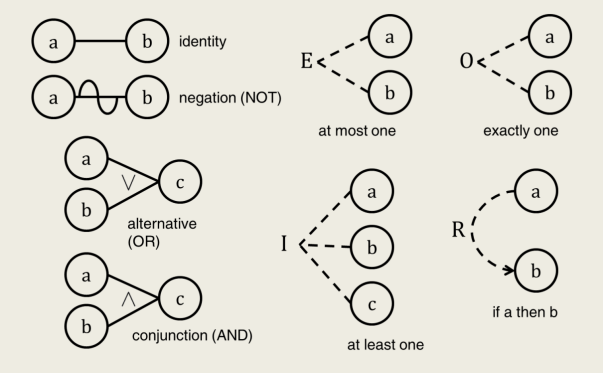
Tablice decyzyjne

- testowanie kombinacji warunków
- dobra metoda do testowania logiki biznesowej
- pozwala na systematyczne sprawdzanie wszystkich kombinacji
- ułatwia wykrywanie
  - brakującej specyfikacji
  - błędnej (sprzecznej) specyfikacji

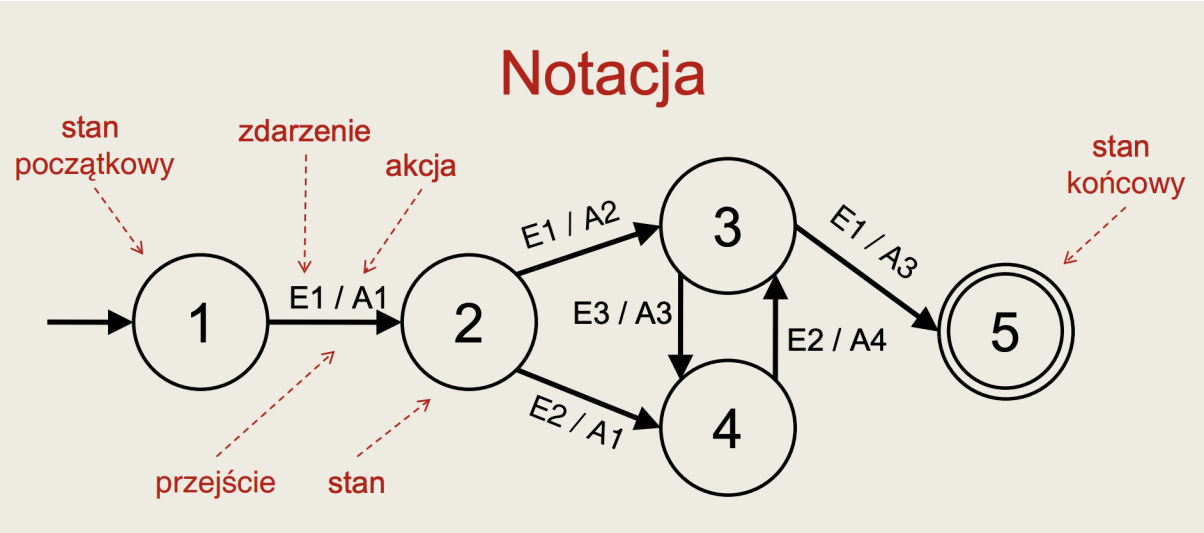
Minimalizacja tablicy - jeśli wszystkie kombinacje pewnych warunków dają te same akcje, możemy je scalić.

Grafy przyczynowo-skutkowe

- 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



Tablica przejść deifniuje akcje i przejścia dla wszystkich kombinacji stan/zdarzenie.

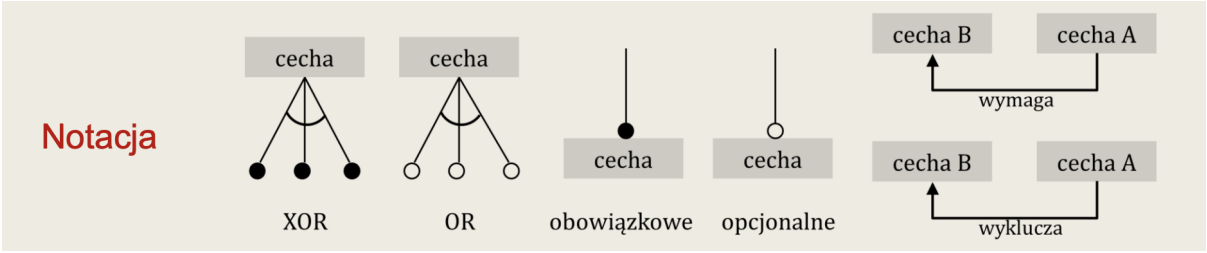
Testowanie maszyny stanowej

- reprezentacja możliwych stanów systemu i przejść między nimi
- metoda opisu dynamiki systemu
- testowanie może sprawdzać np.:
  - czy wszystkie przejścia są poprawne?
  - czy określone sekwencje przejść są poprawne?
  - czy da się wymusić przejścia niepoprawne?
- model maszyny stanowej jest jednym z najpopularniejszych modeli opisu systemu
- UML: state chart diagram
  - znacznie bogatszy niż model opisywany w niniejszym wykładzie!

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

- nie mylić z modelem nauczania maszynowego!
- szczególna wersja metody Category-Partition
- graficzna reprezentacja systemu jako:
  - zestawu cech
  - ich wartości (tworzących dla każdej cechy poprawny podział)
  - 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
- strategia „każdy z każdym” prowadzi do eksplozji kombinatorycznej
- metody kombinatoryczne pozwalają na redukcję liczby testów
- redukcja liczby testów jest nieodłącznie związana z ryzykiem!

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
- zwykle mniej efektywne niż ine metody
- można stosować, gdy trudno modelować dziedzinę wejściową
- testowanie losowe może wykorzystywać
  - rozkłady prawdopodobieństwa uwzględniające np. typowe zachowania użytkowników
  - modele losowości, np. łańcuchy Markowa

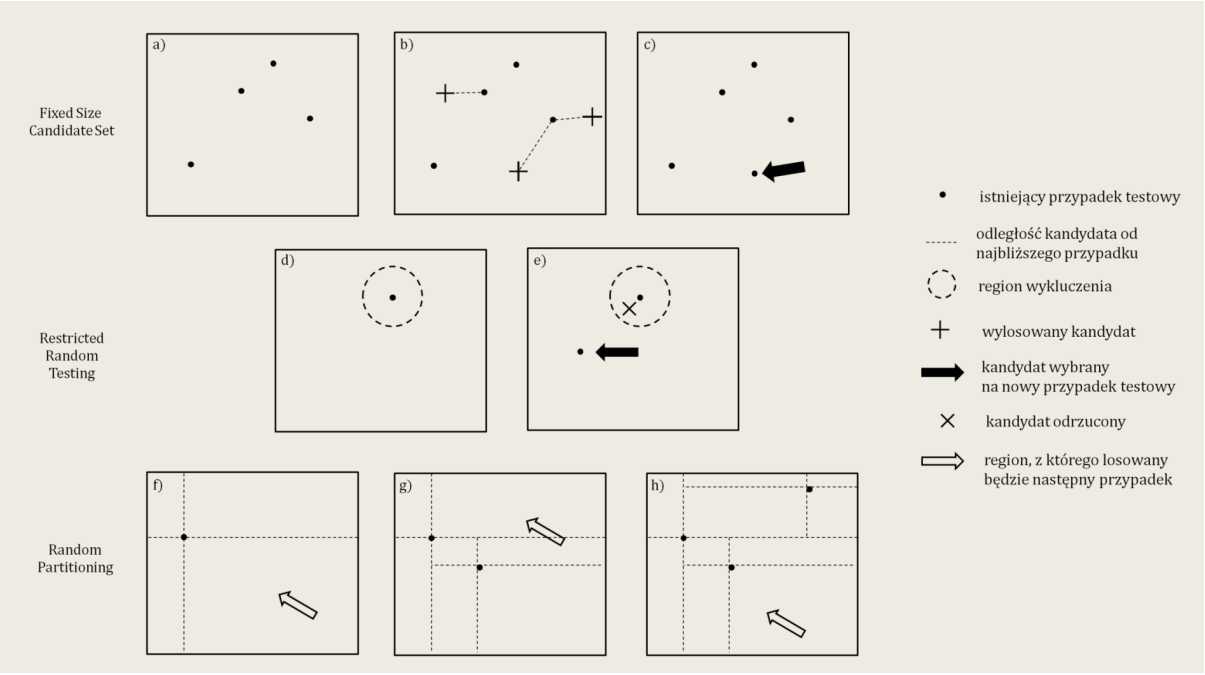
Automatyzacja testowania losowego.

Pełna automatyzacja testowania losowego jest możliwa gdy:

- można automatyczne losować dane wejściowe
- można automatycznie określać oczekiwane wyjście lub automatycznie porównywać wyjście ze specyfikacją

To ostatnie możliwe, gdy:

- 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 jakiejś 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