INŻYNIERIA OPROGRAMOWANIA

wykład 9:
TESTOWANIE,

NIEZAWODNOŚĆ
TESTY cd ..

dr inż. Leszek Grocholski
Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

Oszacowanie niezawodności

Niekiedy (umowy dot. świadczenia usług - QoS, umowy outsourcingowe) poziom niezawodności (wartość pewnej miary lub miar) jest określany w wymaganiach klienta.

Częściej, jest on jednak wyrażony w terminach jakościowych, co bardzo utrudnia obiektywną weryfikację. **Jednakże informacja o niezawodności jest przydatna również wtedy, gdy klient nie określił jej jednoznacznie w wymaganiach.**

Dlaczego?

- Częstotliwość występowania błędnych wykonań ma duży wpływ na koszt konserwacji oprogramowania (serwis telefoniczny + wizyty u klienta).

- Znajomość niezawodności pozwala oszacować koszt serwisu, liczbę personelu, liczbę zgłoszeń telefonicznych, łączny koszt serwisu.

- Znajomość niezawodności pozwala ocenić i polepszyć procesy wytwarzania pod kątem zminimalizowania łącznego kosztu wynikającego z kosztów wytwarzania, kosztów utrzymania, powodzenia na rynku, reputacji firmy.

Niezawodność oprogramowania

Miary niezawodności:

- Prawdopodobieństwo błędnego wykonania podczas realizacji transakcji. Każde błędne wykonanie powoduje zerwanie całej transakcji. Miarą jest częstotliwość występowania transakcji, które nie powiodły się wskutek błędów.

- Częstotliwość występowania błędnych wykonań: ilość błędów w jednostce czasu. Np. 0.1/h oznacza, że w ciągu godziny ilość spodziewanych błędnych wykonań wynosi 0.1. Miara ta jest stosowana w przypadku systemów, które nie mają charakteru transakcyjnego.

- Średni czas między błędnymi wykonaniami - odwrotność poprzedniej miary.

- Dostępność: prawdopodobieństwo, że w danej chwili system będzie dostępny do użytkowania. Miarę tę można oszacować na podstawie **stosunku czasu, w którym system jest dostępny, do czasu od wystąpienia błędu do powrotu do normalnej sytuacji**. Miara zależy nie tylko od błędnych wykonań, ale także od narzutu błędów na niedostępność systemu (czasu interwencji).

Wzrost niezawodności oprogramowania

Rezultatem wykrycia przyczyn błędów jest ich usunięcie.
Jeżeli przy tym nie wprowadza się nowych błędów, to można mówić o wzroście niezawodności.

Jeżeli wykonywane są testy czysto statystyczne to wzrost niezawodności określa się następującym wzorem (logarytmiczny wzrost niezawodności):

Niezawodność = Niezawodność początkowa $\times$ exp(-C $\times$ liczba testów)

Miarą niezawodności jest częstotliwość występowania błędnych wykonań.
Stała C zależy od konkretnego systemu. Można ją określić na podstawie obserwacji statystycznych niezawodności systemu, stosując np. metodę najmniejszych kwadratów.

Szybszy wzrost niezawodności można osiągnąć jeżeli dane testowe są dobierane nie w pełni losowo, lecz w kolejnych przebiegach testuje się sytuacje, które dotąd nie były testowane.

Wykrywanie błędów
- rodzaje testów wykrywających błędne wykonania, błędy

Dynamiczne testy zorientowane na wykrywanie błędnych wykonań dzieli się na:

- Testy funkcjonalne (*functional tests, black-box tests*), które zakładają znajomość jedynie wymagań wobec testowanej funkcji. System jest traktowany jako czarna skrzynka, która w nieznany sposób realizuje wykonywane funkcje. Testy powinny wykonywać osoby, które nie były zaangażowane w realizację testowanych fragmentów systemu.

- Testy strukturalne (*structural tests, white-box tests, glass-box tests*), które zakładają znajomość sposobu implementacji testowanych funkcji.

Po wykonaniu testu należy znaleźć błąd w kodzie i go poprawić

Testy funkcjonalne

Techniki projektowania testów

Testy można tak zaprojektować, żeby pokrywały akceptowalne i nieakceptowalne klasy równoważności.

Istnieje większe prawdopodobieństwo, że oprogramowania będzie się błędnie zachowywać dla wartości na krawędziach klas równoważności niż w ich środku, więc testowanie tych obszarów najprawdopodobniej wykryje błędy.

Minimum i maksimum klasy równoważności to jej wartości brzegowe. Wartość brzegowa poprawnego przedziału jest nazywana poprawną wartością brzegową, a wartość brzegowa niepoprawnego przedziału –niepoprawną wartością brzegową.

Testy można zaprojektować tak, żeby pokrywały zarówno poprawne jak i niepoprawne wartości brzegowe. Podczas projektowania testów tworzy się przypadek testowy dla każdej wartości brzegowej.

Testy funkcjonalne

Pełne przetestowanie rzeczywistego systemu jest praktycznie niemożliwe z powodu ogromnej liczby kombinacji danych wejściowych i stanów. Nawet dla stosunkowo małych programów ta liczba kombinacji jest tak ogromna, że pełne testowanie wszystkich przypadków musiałoby rozciągnąć się na miliardy lat.

Zwykle zakłada się, że jeżeli dana funkcja działa poprawnie dla **kilku** danych wejściowych, to działa także poprawnie dla **całej klasy** danych wejściowych. Jest to wnioskowanie czysto heurystyczne. Fakt poprawnego działania w kilku przebiegach nie gwarantuje zazwyczaj, ze błędne wykonanie nie pojawi się dla innych danych z tej samej klasy.

Podział danych wejściowych na klasy odbywa się na podstawie opisu wymagań, np.

Rachunek o wartości do 1000 zł może być zatwierdzony przez kierownika.
Rachunek o wartości powyżej 1000 zł musi być zatwierdzony przez prezesa.

Takie wymaganie sugeruje podział danych wejściowych na dwie klasy w zależności od wysokości rachunku. Jednak przetestowanie tylko dwóch wartości, np. 500 i 1500 jest zazwyczaj niewystarczające. Konieczne jest także przetestowanie wartości **granicznych**, np. 0, dokładnie 1000 oraz maksymalnej wyobrażalnej wartości.

Kombinacja elementarnych warunków

Z poprzedniego przykładu widać, że testy tylko dla jednej danej wejściowej muszą być przeprowadzone dla pięciu wartości: np. 0, 500, 1000, 1500, max.
Jeżeli takich danych jest wiele, to mamy do czynienia z kombinatoryczną eksplozję przypadków testowych.

Dzieląc dane wejściowe na klasy należy więc brać pod uwagę rozmaite kombinacje elementarnych warunków. Np. do wymienionego warunku dołączony jest następujący:

> *Kierownik może zatwierdzić miesięcznie rachunek o łącznej wartości do 10 000 zł. Każdy rachunek przekraczający tę wartość musi być zatwierdzony przez prezesa.*

Wśród danych wyjściowych można teraz wyróżnić następujące klasy:

- rachunek do 1000 zł nie przekraczający łącznego limitu 10 000 zł.
- rachunek do 1000 zł przekraczający łączny limit 10 000 zł.
- rachunek powyżej 1000 zł nie przekraczający łącznego limitu 10 000 zł.
- rachunek powyżej 1000 zł przekraczający łączny limit 10 000 zł.

Uwzględnienie przypadków granicznych powoduje dalsze rozmnożenie przypadków testowych: (0, 500, 1000, 1500, max) $\times$ (<10000, 10000, >10000)

Eksplozja kombinacji danych testowych

W praktyce przetestowanie wszystkich kombinacji danych wejściowych (nawet zredukowanych do “typowych” i “granicznych”) jest najczęściej niemożliwe. Konieczny jest wybór tych kombinacji.

Ogólne zalecenia takiego wyboru są następujące:

- Możliwość wykonania funkcji jest ważniejsza niż jakość jej wykonania. Brak możliwości wykonania funkcji jest poważniejszym błędem niż np. niezbyt poprawne wyświetlenie jej rezultatów na ekranie.

- Funkcje systemu znajdujące się w poprzedniej wersji są istotniejsze niż nowo wprowadzone. Użytkownicy, którzy posługiwali się dana funkcją w poprzedniej wersji systemu będą bardzo niezadowoleni jeżeli w nowej wersji ta funkcja przestanie działać.

- Typowe sytuacje są ważniejsze niż wyjątki lub sytuacje skrajne. Błąd w funkcji wykonywanej często lub dla danych typowych jest znacznie bardziej istotny niż błąd w funkcji wykonywanej rzadko dla nietypowych danych.

Inne techniki testów funkcjonalnych

Stosowane są również inne techniki testów funkcjonalnych:

- **Testowanie w oparciu o tablicę decyzyjną**

Tablice decyzyjne są bardzo popularne w: bankach, firmach ubezpieczeniowych, telekomunikacyjnych, czy np. Wizardach.

Tablice często dotyczą postępowania umożliwiającego stosowanie reguł (biznesowych) tzn ograniczeń specyficzne dla danej organizacji, zdefiniowane dla całego jej obszaru funkcjonowania.

- Testowanie przejść między stanami

System może różnie odpowiadać w zależności od aktualnych warunków oraz od historii (od stanu). W takim przypadku zachowanie systemu można opisać diagramem przejść stanów (automatem skończonym).

- Testowanie przypadków użycia

Testy strukturalne

W przypadku testów strukturalnych, dane wejściowe dobiera się na podstawie analizy struktury programu realizującego testowane funkcje.
Kryteria doboru danych testowych są następujące:

- Kryterium pokrycia wszystkich instrukcji. Zgodnie z tym kryterium dane wejściowe należy dobierać tak, aby każda instrukcja została wykonana co najmniej raz. Spełnienie tego kryterium zwykle wymaga niewielkiej liczby testów. To kryterium może być jednak bardzo nieskuteczne.

```
if x > 0 then begin ... end; y := ln( x);
```

Dla x >0 wykonane będą wszystkie instrukcje, ale dla x <= 0 program jest błędny.

- **Kryterium pokrycia instrukcji warunkowych**. Dane wejściowe należy dobierać tak, aby każdy elementarny warunek instrukcji warunkowej został co najmniej raz spełniony i co najmniej raz nie spełniony. Testy należy wykonać także dla każdej wartości granicznej takiego warunku. Zastosowanie tego warunku pozwoli wykryć błąd z poprzedniego przykładu, gdyż zmusi do testu dla x = 0 oraz x < 0.

Istnieje szereg innych kryteriów prowadzących do bardziej wymagających testów.

Testowanie programów zawierających pętle

Kryteria doboru danych wejściowych mogą opierać się o następujące zalecenia:

- Należy dobrać dane wejściowe tak, aby nie została wykonana żadna iteracja pętli, lub, jeżeli to niemożliwe, została wykonana minimalna liczba iteracji.

- Należy dobrać dane wejściowe tak, aby została wykonana maksymalna liczba iteracji.

- Należy dobrać dane wejściowe tak, aby została wykonana przeciętna liczba iteracji.

Programy uruchamiające

debuggers

Mogą być przydatne dla wewnętrznego testowania jak i dla testowania przez osoby zewnętrzne. Zakładają testowanie na zasadzie białej skrzynki (znajomość kodu).

Podstawowe funkcje debuggerów:

- Wyświetlenie stanu zmiennych programu i interakcja z testującym z użyciem symboli kodu źródłowego.

- Wykonywanie programów krok po kroku, z różną granulowością instrukcji

- Ustanowienie punktów kontrolnych w programie (zatrzymujących wykonanie)

- Ustanowienie obserwatorów wartości zmiennych

- Zarządzanie plikiem źródłowym podczas testowania i ewentualna poprawa wykrytych błędów w tym pliku.

- Tworzenie dziennika testowania, umożliwiającego powtórzenie testowego przebiegu.

Analizatory pokrycia kodu *coverage analysers*

Analizatory pokrycia kodu są to programy umożliwiające ustalenie obszarów kodu źródłowego, które były wykonane w danym przebiegu testowania. Umożliwiają wykrycie martwego kodu, kodu uruchamianego przy bardzo specyficznych danych wejściowych oraz (niekiedy) kodu wykonywanego bardzo często (co może być przyczyną wąskiego gardła w programie).

Funkcje bardziej zaawansowanych analizatorów przykrycia kodu:

- Zsumowanie danych z kilku przebiegów (dla różnych kombinacji danych wejściowych) np. dla łatwiejszego wykrycia martwego kodu.

- Wyświetlenie grafów sterowania, dzięki czemu można łatwiej monitorować przebieg programu

- Wyprowadzenie informacji o pokryciu, umożliwiające poddanie pokrytego kodu dalszym testom.

- Operowanie w środowisku rozwoju oprogramowania.

Programy porównujące

comparators

Są to narzędzia programistyczne umożliwiające porównanie dwóch programów, plików lub zbiorów danych celem wykrycia cech wspólnych i różnic. Często są niezbędne do porównania wyników testów z wynikami oczekiwanymi. Programy porównujące przekazują w czytelnej postaci różnice pomiędzy aktualnymi i oczekiwanymi danymi wyjściowymi.

Ekranowe programy porównujące mogą być bardzo użyteczne dla testowania oprogramowania interakcyjnego. Są niezastąpionym środkiem dla testowania programów z graficznym interfejsem użytkownika.

Inne narzędzia wspomagające testowanie:

Duża różnorodność narzędzi stosowanych w różnych fazach rozwoju oprogramowania. Np. wspomaganie do planowania testów, automatyczne zarządzania danymi wyjściowymi, automatyczna generacja raportów z testów, generowanie statystyk jakośc i niezawodności, wspomaganie powtarzalności testów, itd.

Testy statyczne

Polegają na analizie kodu bez uruchomienia programu. Techniki są następujące:
- metody nieformalne - dowody poprawności
- metody nieformalne

Dowody poprawności nie są praktycznie możliwe dla rzeczywistych programów. Nie istnieją wyłącznie w ideach teoretyków informatyki. Stosowanie ich dla programów o obecnej skali i złożoności jest trudne.
(Ale dowody poprawności stosuje się np.. w systemach krytycznych)

Statyczne metody nieformalne polegają na analizie kodu przez programistów.
- czyli inspekcje
Dwa niewykluczające się podejścia:
- śledzenie przebiegu programu (wykonywanie programu "w myśli" przez analizujące osoby)
- wyszukiwanie typowych błędów

Testy nieformalne są niedocenione, chociaż bardzo efektywne w praktyce.
Uwaga:
Na ogół testy funkcjonalne są bardziej skuteczne niż testy strukturalne.

Typowe błędy wykrywane statycznie

- **niezainicjowane** zmienne
- porównania na równość liczb **zmiennoprzecinkowych**
- **Indeksy** wykraczające poza tablice
- błędne operacje na **wskaźnikach**
- błędy w warunkach **instrukcji warunkowych**
- niekończące się **pętle**
- błędy popełnione dla **wartości granicznych** (np. > zamiast >=)
- błędne użycie lub pominięcie **nawiasów** w złożonych wyrażeniach
- nieuwzględnienie **błędnych danych**

Postępowanie podczas statycznych testów nieformalnych:

Programista, który dokonał implementacji danego modułu w nieformalny sposób analizuje jego kod.

Kod uznany przez programistę za poprawny jest analizowany przez doświadczonego programistę. Jeżeli znajdzie on pewną liczbę błędów, moduł jest zwracany programiście do poprawy.

Szczególnie istotne moduły są analizowane przez grupę osób.

Ocena liczby błędów – koszty konserwacji

Błędy w oprogramowaniu niekoniecznie są bezpośrednio powiązane z jego zawodnością. Oszacowanie liczby błędów ma jednak znaczenie dla producenta oprogramowania, gdyż ma wpływ na koszty konserwacji oprogramowania. Szczególnie istotne dla firm sprzedających oprogramowanie pojedynczym lub nielicznym użytkownikom (relatywnie duży koszt usunięcia błędu).

Dane umożliwiające szacowanie kosztów konserwacji dot. usuwania błędów:

- Szacunkowa liczba błędów w programie

- Średni procent błędów zgłaszanych przez użytkownika systemu, na podstawie danych z poprzednich przedsięwzięć.

- Średni koszt usunięcia błędu na podstawie danych z poprzednich przedsięwzięć.

Technika “posiewania błędów”

Polega na tym, że do programu celowo wprowadza się pewną liczbę błędów podobnych do tych, które występują w programie. Wykryciem tych błędów zajmuje się inna grupa programistów niż ta, która dokonała “posiania” błędów.

Załóżmy, że:

N oznacza liczbę posianych błędów
M oznacza liczbę wszystkich wykrytych błędów
X oznacza liczbę posianych błędów, które zostały wykryte

Wyniki mogą być mocno chybione, jeżeli “posiane” błędy nie będą podobne do rzeczywistych błędów występujących w programie.
Technika ta pozwala również na przetestowanie skuteczności metod testowania.
Zbyt mała wartość X/N oznacza konieczność poprawy tych metod.

Testy systemu

Techniki:

- testowanie wstępujące
- testowanie zstępujące

- Testowanie wstępujące: najpierw testowane są pojedyncze moduły, następnie moduły coraz wyższego poziomu, aż do osiągnięcia poziomu całego systemu. Zastosowanie tej metody nie zawsze jest możliwe, gdyż często moduły są od siebie zależne. Niekiedy moduły współpracujące można zastąpić implementacjami szkieletowymi (atrapy, tzw „mocki”).

- **Testowanie zstępujące**: rozpoczyna się od testowania modułów wyższego poziomu. Moduły niższego poziomu zastępuje się implementacjami szkieletowymi (atrapy,”mocki”).
- Po przetestowaniu modułów wyższego poziomu dołączane są moduły niższego poziomu. Proces ten jest kontynuowany aż do zintegrowania i przetestowania całego systemu.

Testy pod obciążeniem, testy odporności

Testy obciążeniowe (*stress testing*). Celem tych testów jest zbadanie wydajności i niezawodności systemu podczas pracy pod pełnym lub nawet nadmiernym obciążeniem. Dotyczy to szczególnie systemów wielodostępnych i sieciowych. Systemy takie muszą spełniać wymagania dotyczące wydajności, liczby użytkowników, liczby transakcji na godzinę. Testy polegają na wymuszeniu obciążenia równego lub większego od maksymalnego.

Testy odporności (*robustness testing*). Celem tych testów jest sprawdzenie działania w przypadku zajścia niepożądanych zdarzeń, np.

- zaniku zasilania
- awarii sprzętowej
- wprowadzenia niepoprawnych danych
- wydania sekwencji niepoprawnych poleceń

Bezpieczeństwo oprogramowania

Pewnej systemy są krytyczne z punktu widzenia bezpieczeństwa ludzi, np.
Może to być także zagrożenie pośrednie, np. systemy eksperckie w dziedzinie medycyny, systemy informacji o lekach.

Bezpieczeństwo niekoniecznie jest pojęciem tożsamym z niezawodnością.

System zawodny może być bezpieczny, jeżeli skutki błędnych wykonań nie są groźne.

Wymagania wobec systemu mogą być niepełne i nie opisywać zachowania systemu we wszystkich sytuacjach. Dotyczy to zwłaszcza sytuacji wyjątkowych, np. wprowadzenia niepoprawnych danych. Ważne jest, aby system zachował się bezpiecznie także wtedy, gdy właściwy sposób reakcji nie został opisany.

Niebezpieczeństwo może także wynikać z awarii sprzętowych.
Analiza bezpieczeństwa musi uwzględniać oba czynniki.

Analiza bezpieczeństwa

Techniki:
-oparte na doświadczeniu,
-oparte na analizie ryzyka (dużo odmian)

Zaczyna się od określenia potencjalnych niebezpieczeństw związanych z użytkowaniem systemu: możliwości utraty życia, zdrowia, strat materialnych, złamania przepisów prawnych.

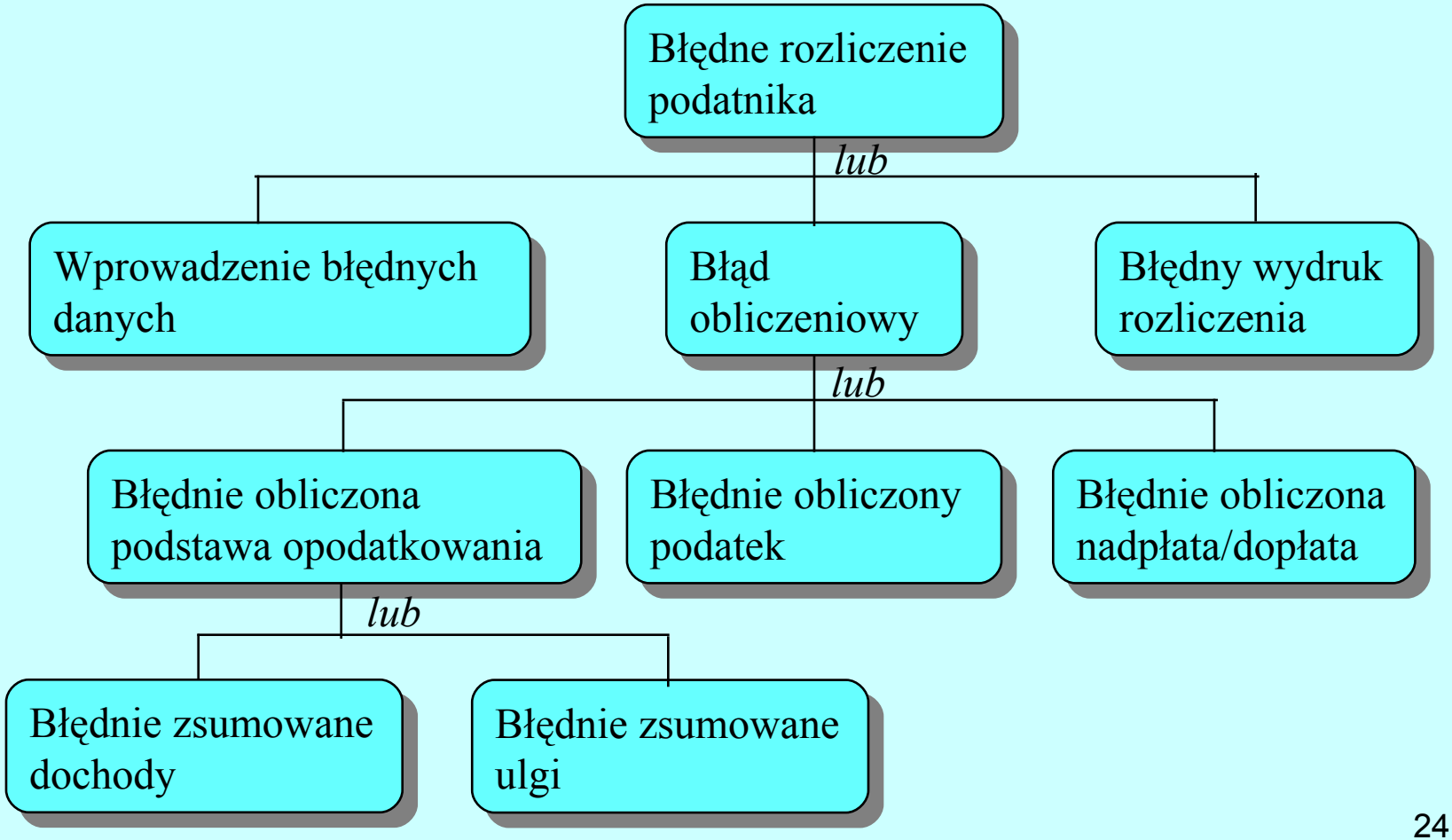
Np. dla programu podatkowego mogą wystąpić następujące niebezpieczeństwa:

- błędne rozliczenie się z urzędem podatkowym
- nie złożenie zeznania podatkowego
- złożenie wielu zeznań dla jednego podatnika

Drzewo błędów

fault tree

Korzeniem drzewa są jest jedna z rozważanych niebezpiecznych sytuacji. Wierzchołkami są sytuacje pośrednie, które mogą prowadzić do sytuacji odpowiadającej wierzchołkowi wyższego poziomu.



Techniki zmniejszania niebezpieczeństwa

- Położenie większego nacisku na unikanie błędów podczas implementacji fragmentów systemu, w których błędy mogą prowadzić do niebezpieczeństw.

- Zlecenie realizacji odpowiedzialnych fragmentów systemu bardziej doświadczonym programistom.

- Zastosowanie techniki programowania N - wersyjnego w przypadku wymienionych fragmentów systemu. Ten sam komponent zaprogramowany przez nizalezne zespoly programistow. Wyniki działania komponentu sa porównywalne.

- Szczególnie dokładne przetestowanie tych fragmentów systemu.

- Wczesne wykrywanie sytuacji, które mogą być przyczyną niebezpieczeństwa i podjęcie odpowiednich, bezpiecznych akcji. Np. ostrzeżenie w pewnej fazie użytkownika o możliwości zajścia błędu (asercje + zrozumiałe komunikaty o niezgodności).

Czynniki sukcesu, rezultaty testowania

Czynniki sukcesu:

- Określenie fragmentów systemu o szczególnych wymaganiach wobec niezawodności.

- Właściwa motywacja osób zaangażowanych w testowanie. Np. stosowanie nagród dla osób testujących za wykrycie szczególnie groźnych błędów, zaangażowanie osób posiadających szczególny talent do wykrywania błędów

Podstawowe wyniki testowania:

- Poprawiony kod, projekt, model i specyfikacja wymagań

- Raport przebiegu testów, zawierający informację o przeprowadzonych testach i ich rezultatach.

- Oszacowanie niezawodności oprogramowania i kosztów konserwacji.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę