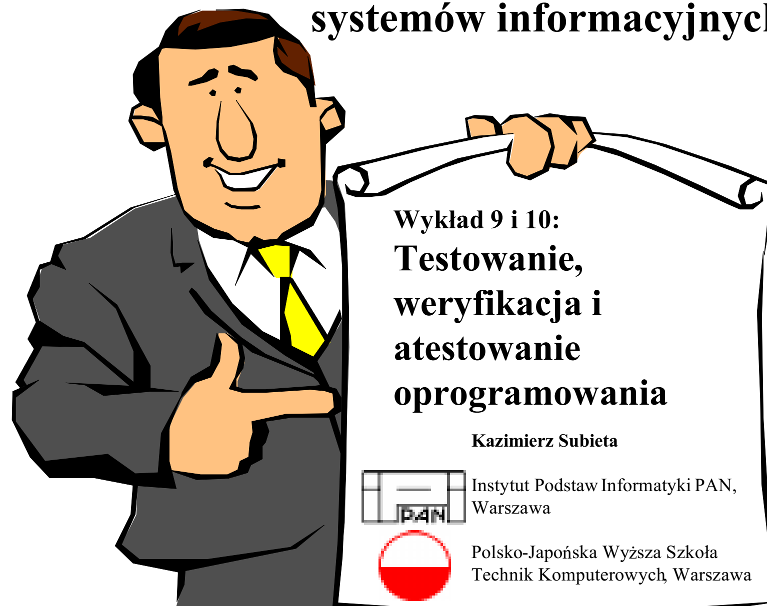
Wytwarzanie, integracja i testowanie systemów informacyjnych



Wykład 9 i 10:

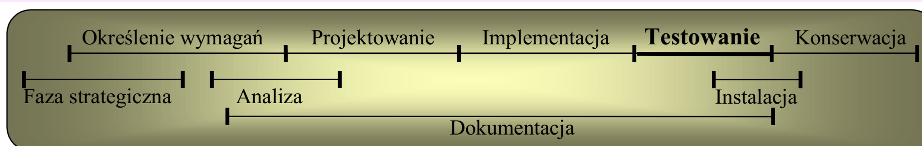
Testowanie, weryfikacja i atestowanie oprogramowania

Kazimierz Subieta

Instytut Podstaw Informatyki PAN, Warszawa

Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Warszawa

Faza testowania



Rozróżnia się następujące terminy:

Atestowanie (*validation*) - testowanie zgodności systemu z rzeczywistymi potrzebami użytkownika.

Weryfikacja (*verification*) - testowanie zgodności systemu z wymaganiami zdefiniowanymi w fazie określenia wymagań.

Dwa główne cele testowania:

- wykrycie i usunięcie błędów w systemie
- ocena niezawodności systemu

Błąd i błędne wykonanie

Błąd (*failure, error*) - niepoprawna konstrukcja znajdująca się w programie, która może doprowadzić do niewłaściwego działania.

Błędne wykonanie (*failure*) - niepoprawne działanie systemu w trakcie jego pracy.

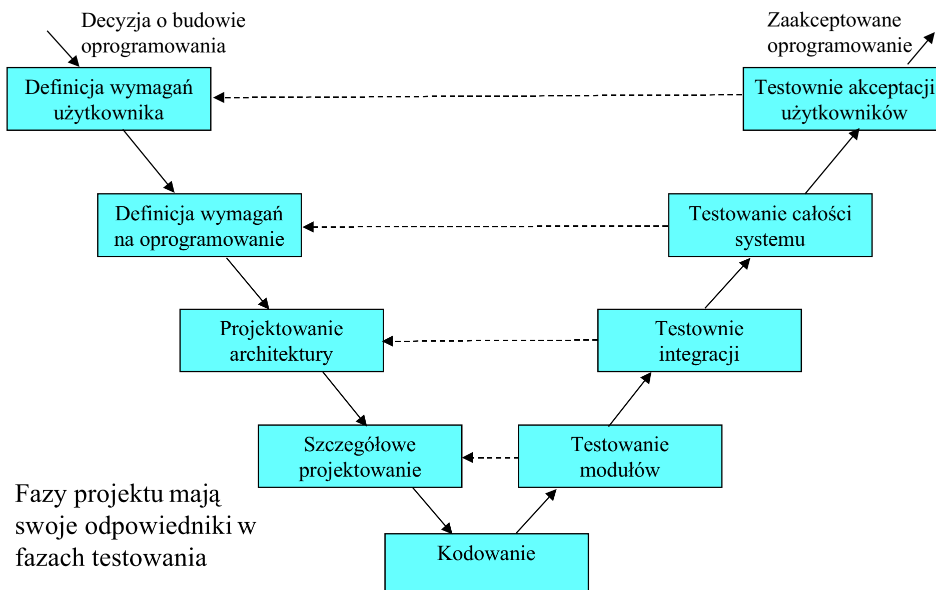
Błąd może prowadzić do różnych błędnych wykonań.
To samo błędne wykonanie może być spowodowane różnymi błędami.

Proces weryfikacji oprogramowania można określić jako poszukiwanie i usuwanie błędów na podstawie obserwacji błędnych wykonań oraz innych testów.

Aktywności związane z weryfikacją

- Przeglądy techniczne, próbne przejścia
- Sprawdzenie, czy wymagania na oprogramowanie są realizacją wymagań użytkownika (czy wymagania uzytkownika są weryfikowalne)
- Sprawdzenie, czy komponenty projektu są weryfikowalne zgodnie z wymaganiami użytkownika
- Testowanie jednostek oprogramowania (modułów)
- Testowanie modułów po ich integracji
- Testowanie systemu przez jego twórców
- Testowanie akceptacji systemu przez użytkowników
- Audyt (kontrola jakości oprogramowania zgodna z odpowiednią normą)

Związek faz projektu z fazami testowania



Fazy projektu mają swoje odpowiedniki w fazach testowania

Skład zespołu oceniającego oprogramowanie

Dla poważnych projektów ocena oprogramowania nie moze być wykonana w sposób amatorski. Musi być powołany zespół, którego zadaniem będzie zarówno przygotowanie testów jak i ich przeprowadzenie.

Potencjalny skład zespołu oceniającego:

- Kierownik
- Sekretarz
- Członkowie, w tym:
  - użytkownicy
  - kierownik projektu oprogramowania
  - inzynierowie oprogramowania
  - bibliotekarz oprogramowania
  - personel zespołu zapewnienia jakości
  - niezależny personel weryfikacji i atestowania
  - niezależni eksperci nie związani z rozwojem projektu

Zadania kierownika: nominacje na członków zespołu, organizacja przebiegu oceny i spotkań zespołu, rozpowszechnienie dokumentów oceny pomiędzy członków zespołu, organizacja pracy, przewodniczenie posiedzeniom, wydanie końcowego raportu, i być może inne zadania.

Co to jest audyt?

audit

Audytem nazywany jest niezależny przeglad i ocena jakości oprogramowania, która zapewnia zgodność z wymaganiami na oprogramowanie, a także ze specyfikacją, generalnymi założeniami, standardami, procedurami, instrukcjami, kodami oraz kontraktowymi i licencyjnymi wymaganiami.

Aby zapewnić obietywność, audyt powinien byc przeprowadzony przez osoby niezależne od zespołu projektowego.

Audyt powinien być przeprowadzany przezodpowiednią organizację audytu (która aktualnie formuje się w Polsce, Polskie Stowarzyszenie Audytu) lub przez osoby posiadające uprawnienia/licencję audytorów.

Reguły i zasady audytu są określone w normie:
ANSI/IEEE Std 1028-1988

Rodzaje testów

Testy można klasyfikować z różnych punktów widzenia:

- **Wykrywanie błędów**, czyli testy, których głównym celem jest wykrycie jak największej liczby błędów w programie
- **Testy statystyczne**, których celem jest wykrycie przyczyn najczęstszych błędnych wykonań oraz ocena niezawodności systemu.

Z punktu widzenia techniki wykonywania testów można je podzielić na:

- **Testy dynamiczne**, które polegają na wykonywaniu (fragmentów) programu i porównywaniu uzyskanych wyników z wynikami poprawnymi.
- **Testy statyczne**, oparte na analizie kodu

Typowe fazy testowania systemu

- **Testy modułów**: Są one wykonywane już w fazie implementacji bezpośrednio po zakończeniu realizacji poszczególnych modułów
- **Testy systemu**: W tej fazie integrowane są poszczególne moduły i testowane są poszczególne podsystemy oraz system jako całość
- **Testy akceptacji** (*acceptance testing*): W przypadku oprogramowania realizowanego na zamówienie system przekazywany jest do przetestowania przyszłemu użytkownikowi. Testy takie nazywa się wtedy testami **alfa**. W przypadku oprogramowania sprzedawanego rynkowo testy takie polegają na nieodpłatnym przekazaniu pewnej liczby kopii systemu grupie użytkowników. Testy takie nazywa się testami **beta**.

Co podlega testowaniu (1)?

Wydajność systemu i poszczególnych jego funkcji (czy jest satysfakcjonująca).

Interfejsy systemu na zgodność z wymaganiami określonymi przez uzytkowników

Własności operacyjne systemu, np. wymagania logistyczne, organizacyjne, użyteczność/ stopień skomplikowania instrukcji kierowanych do systemu, czytelność ekranów, operacje wymagające zbyt wielu kroków, jakość komunikatów systemu, jakość informacji o błędach, jakość pomocy.

Testy zużycia zasobów: zużycie czasu jednostki centralnej, zużycie pamięci operacyjnej, przestrzeni dyskowej, itd.

Zabezpieczenie systemu: odporność systemu na naruszenia prywatności, tajności, integralności, spójności i dostępności. Testy powinny np. obejmować:

- zabezpieczenie haseł użytkowników
- testy zamykania zasobów przed niepowołanym dostępem
- testy dostęu do plików przez niepowołanych użytkowników
- testy na możliwość zablokowania systemu przez niepowołane osoby
- ....

Co podlega testowaniu (2)?

Przenaszalność oprogramowania: czy oprogramoanie bedzie działać w zróżnicowanym środowisku (np. różnych wersjach Windows 95, NT, Unix), przy różnych wersjach instalacyjnych, rozmiarach zasobów, kartach graficznych, rozdzielczości ekranów, oprogramowaniu wspomagającym (bibliotekach), ...

Niezawodność oprogramowania, zwykle mierzoną średnim czasem pomiędzy błędami.

Odtwarzalność oprogramowania (*maintainability*), mierzoną zwykle średnim czasem reperowania oprogramowania po jego awarii Pomiar powinien uwzględniać średni czas od zgłoszenia awarii do ponownego sprawnego działania.

Bezpieczeństwo oprogramowania: stopień minimalizacji katastrofalnych skutków wynikających z niesprawnego działania. (Przykładem jest wyłaczenie prądu podczas działania w banku i obserwacja, co się w takim przypadku stanie.)

Kompletność i jakość założonych funkcji systemu.

Nie przekraczania ograniczeń, np. na zajmowaną pamięć, obciążenia procesora,...

Co podlega testowaniu (3)?

Modyfikowalność oprogramowania, czyli zdolność jego do zmiany przy zmieniających się założeniach lub wymaganiach

Obciążalność oprogramowania, tj. jego zdolność do poprawnej pracy przy ekstremalnie dużych obciążeniach. Np. maksymalnej liczbie użytkowników, bardzo dużych rozmiarach plików, dużej liczbie danych w bazie danych, ogromnych (maksymalnych) zapisach, bardzo długich liniach danych źródłowych. W tych testach czas nie odgrywa roli, chodzi wyłacznie o to, czy system poradzi sobie z ekstremalnymi rozmiarami danych lub ich komponetów orazz maksymalnymi obciążeniami na jego wejściu.

Skalowalność systemu, tj. spełnienie warunków (m.in. czasowych) przy znacznym wzroście obciążenia.

Akceptowalność systemu, tj. stopień usatysfakcjonowania użytkownków.

Jakość dokumentacji, pomocy, materiałów szkoleniowych, zmniejszenia bariery dla nowicjuszy.

Schemat testów statystycznych

- Losowa konstrukcja danych wejściowych zgodnie z rozkładem prawdopodobieństwa tych danych
- Określenie wyników poprawnego działania systemu na tych danych
- Uruchomienie systemu oraz porównanie wyników jego działaniaz poprawnymi wynikami.

Powyższe czynności powtarzane są cyklicznie.

Stosowanie testów statystycznych wymaga określenia rozkładu prawdopodobieństwa danych wejściowych możliwie bliskiemu rozkładowi, który pojawi się w rzeczywistości. Dokładne przewidzenie takiego rozkładu jest trudne, w związku z czym wnioski wyciągnięte na podstawie takich testów mogą być nietrafne.

Założeniem jest przetestowanie systemu w typowych sytuacjach. Istotne jest także przetestowanie systemu w sytuacjach skrajnych, nietypowych, ale dostatecznie ważnych.

Istotną zaletą testów statystycznych jest możliwość ich automatyzacji, a co za tym idzie, możliwości wykonania dużej ich liczby. Technika ta jest jednak mało efektywna.

Określenie niezawodności oprogramowania

Miary niezawodności:

- **Prawdopodobieństwo błędnego wykonania** podczas realizacji transakcji. Każde błędne wykonanie powoduje zerwanie całej transakcji Miarą jest częstość 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.

Oszacowanie niezawodności

Niekiedy 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.

Wzrost niezawodności oprogramowania

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

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

Niezawodność = Niezawodność początkowa × exp(-C × liczba testów)

Miarą niezawodnośći 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

Dynamiczne testy zorientowane na wykrywanie błędów dzieli się na:

- **Testy funkcjonalne** (*functional tests, black-box tests*), któe 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.

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) × (<10000, 10000, >10000)

Eksplozja kombinacji danych testowych

W praktyce przetestowanie wszystkich kombinacji danych wejściowch (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 dla nietypowych danych

Testy strukturalne

W przypadku testów strukturalnych, dane wejściowe dobiera się na podstawie analizy struktury programu realizujacego 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 nalezy 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, jezeli 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).

Własności programów uruchamiających:

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 źródłówym podczas testowania i ewentualna poprawa wykrytych błędów w tym pliku.

Tworzenie dziennika testowania, umożliwiającego powtórzenie testowego przebiegu.

Analizatory przykrycia kodu

coverage analysers

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 waskiego gardła w programie).

Bardziej zaawansowane analizatory przykrycia kodu umożliwiają:

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

Wyświetlenie grafów sterowania, dzięki czemu mozna łatwiej monitorować przebieg programu

Wyprowadzenie informacji o przykryciu, umożliwiające poddanie przykrytego kodu dalszym testom.

Operowanie w środowisku rozwoju oprogramowania.

Programy porównujące

comparators

Są to narzędzia programistyczne umozliwiające porównanie dwóch programów, plików lub zbiorów danych celem wykrycia cech wspólnychi 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 niezastąionym środkiem dla testowania programów z graficznym interfejsem użytkownika.

Pozostałe narzędzia do testowania:

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 jakości i niezawodności, wspomaganie powtarzalności testów, itd.

Testy statyczne

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

Dowody poprawności nie są praktycznie możliwe dla rzeczywistych programów. Istnieją wyłacznie w urojeniach (pseudo-?) teoretyków informatyki. Stosowanie ich dla programów o obecnej skali i złożoności jest pozbawione sensu.
(Niestety, nadal występuje nacisk środowiska akademickiego na rozwój i nauczanie tych od dawna martwych metod.)

Statyczne metody nieformalne polegają na analizie kodu przez programistów.

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.
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

Strategia 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

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).

Podstawy szacowania kosztu konserwacji związanego z usuwaniem 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

Szacunkowa liczba błędów przed wykonaniem testów: (M - X) * N/X

Szacunkowa liczba błędów po usunięciu wykrytych (“posiane” błędy zostały też usunięte): (M - X) * (N/X - 1)

Szacunki te mogą być mocno chybione, jeżeli “posiane” błędy nie będą podobne do rzeczywistych błędów występujących w programie.
Technika ta pozwala również na przetestownie 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 współpacujące można zastąpić implementacjami szkieletowymi.

- **Testowanie zstępujące**: rozpoczyna się od testowania modułów wyższego poziomu. Moduły niższego poziomu zastępuje się implementacjami szkieletowymi. 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 uzytkowników, liczby transakcji na godzinę. Testy polegają na wymuszeniu obciążenia równego lub większego od maksymalnego.

Testy odporności (*robustness testing*). Celem tych testów jest sprawdzenie dziaania 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. aparatura medyczna, oprogramowanie wpomagające sterowanie samolotem. 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

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

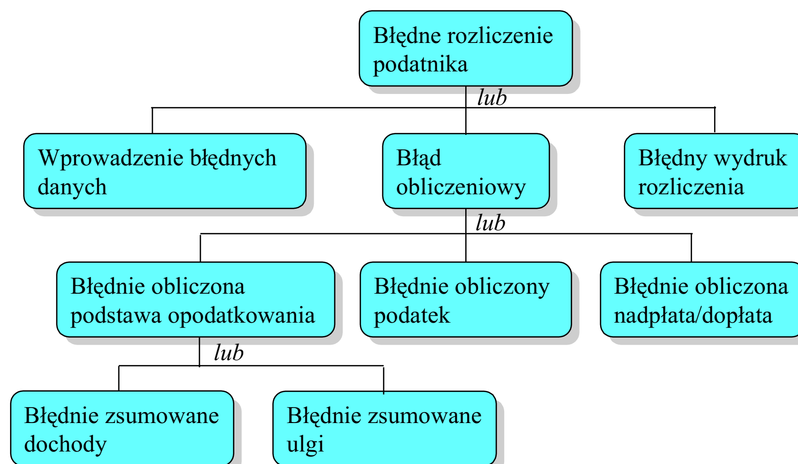
Np. dla programu podatowego mogą wystąpić następujące niebezpieczeństwa:

- 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.
- 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 rezultaty 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.