ZWIĘKSZANIE POPRAWNOŚCI OPROGRAMOWANIA

Plan prezentacji:

- Poprawność oprogramowania
- Weryfikacja i testowanie.
- Rodzaje testów.
- Testowania względem specyfikacji
- Testowanie względem kodu.
- Ocena poprawności oprogramowania.

Poprawność oprogramowania

→ określa, czy oprogramowanie wypełnia postawione przed nim zadania tzn. czy spełnia wymagania wyspecyfikowane przez zleceniodawcę i czy jest wolne od błędów.

Podstawowe definicje:

Błąd – niepoprawna konstrukcja w programie, mogąca prowadzić do niewłaściwego działania

Błędne wykonanie – niepoprawne działanie programu w trakcie jego pracy, zaobserwowane złe działanie

Testowanie – proces sprawdzania/oceny czy produkt lub jego część jest „dobra”

Błąd może prowadzić do różnych **błędnych wykonań** ale nie musi.
Może się zdarzyć, że program posiada błędy ale działa poprawnie (na razie).

Motywacje prac nad poprawnością oprogramowania:

1. Rosnące oczekiwania klientów wynikające z wysokiej niezawodności sprzętu.
2. Potencjalnie duże koszty błędnych wykonań, wysokie straty finansowe wynikające z błędnego działania oprogramowania, czasem nawet zagrożenie dla życia.
3. Nieprzewidywalność działania oraz trudności usuwania błędów w oprogramowaniu.

Możliwe strategie tworzenia niezawodnego oprogramowania:

1. tworzenie systemów pozbawionych błędów (unikanie błędów)
   a) przed faktem → stosowanie technik minimalizujących liczbę błędów
   b) po fakcie → metody wykrywania i poprawiania błędów (usuwanie błędów)
2. tworzenie systemów odpornych na błędy (tolerancja błędów)

STRATEGIA UNIKANIA BŁĘDÓW
(minimalizacja liczby błędów „przed faktem”)

Pełne uniknięcie błędów w oprogramowaniu nie jest możliwe
ale można zmniejszyć prawdopodobieństwo wystąpienia błędu poprzez:

1. Unikanie niebezpiecznych technik:
   - instrukcji „goto” prowadzącej do programów, których działanie jest trudne do przewidzenia i do zrozumienia,
   - wskaźników i arytmetyki wskaźników: techniki dająca możliwość dowolnej penetracji całej pamięci operacyjnej i dowolnych nieoczekiwanych zmian w tej pamięci,
   - obliczeń równoległe prowadzących do złożonych zależności czasowych i hazardów czasowych (pogoni),
   - przerwań - techniki wprowadzającej również rodzaj równoległości,
   - dynamicznej alokacji pamięci, która bez zapewnienia automatycznego mechanizmu odzyskiwania nieużytków powoduje “wyciekanie pamięci”,
   - niejawnego przekazywania parametrów powodującego nieoczekiwane efekty uboczne funkcji.
2. Stosowanie zasady ograniczonego dostępu dom danych (zmienne lokalne, hermetyzacja),
3. Zastosowanie języków z kontrolą typów i kompilatorów sprawdzających zgodność typów podczas wywoływania funkcji i obliczania wyrażeń.
4. Dokładne specyfikowanie interfejsów pomiędzy funkcjami i modułami oprogramowania
5. Uważne rozpatrywanie przypadków szczególnych (pętle z zerową ilością obiegów, wartości zerowe i niezainicjowane zmienne, puste zbiory, itd.)
6. Wykorzystanie gotowych komponentów (np. gotowych bibliotek procedur lub klas)
7. Minimalizacja różnic pomiędzy modelem pojęciowym i modelem implementacyjnym

Zasada ograniczonego dostępu

Programista nie powinien mieć żadnej możliwości operowania na tej części oprogramowania lub zasobów komputera, która nie dotyczą tego, co aktualnie robi.

Dostęp do czegokolwiek powinien być ograniczony tylko do tego, co jest niezbędne.

STRATEGIA WYKRYWANIA I USUWANIA BŁĘDÓW
(minimalizacja liczby błędów „po fakcie”)

Testowanie względem klienta, względem specyfikacji i względem kodu

w**eryfikacja** – sprawdzanie „zgodności” oprogramowania z założeniami zdefiniowanymi w fazie określania wymagań (ze specyfikacją)

walidacja – (atestowanie) sprawdzanie „jakości” oprogramowania
- czy oprogramowanie jest wolne od błędów
- czy jest zgodne z rzeczywistymi potrzebami użytkownika.

Weryfikacja

- Przeglądy techniczne oraz inspekcje oprogramowania.
- Sprawdzanie czy wymagania na oprogramowanie są zgodne z wymaganiami użytkownika.
- Sprawdzanie czy komponenty projektu są zgodne z wymaganiami na oprogramowanie.
- Testowanie jednostek oprogramowania (funkcji, klas, modułów).
- Testowanie integracji oprogramowania, testowanie systemu.
- Testowanie akceptacji systemu przez użytkowników.

Główne cele testowania:

- wykrycie i usunięcie błędów w systemie,
- ocena niezawodności systemu

Typowe fazy testowania systemu

1. Testy modułów - wykonywane już w fazie implementacji bezpośrednio po zakończeniu realizacji poszczególnych modułów.
2. Testy systemu – po połączeniu poszczególne moduły testowane są poszczególne podsystemy oraz system jako całość
3. Testy akceptacji - w przypadku oprogramowania realizowanego na zamówienie system przekazywany jest do przetestowania przyszłemu użytkownikowi. Testy takie nazywa się wtedy **testami alfa**.

   W przypadku oprogramowania sprzedawanego komercyjnie testy takie polegają na nieodpłatnym przekazaniu pewnej liczby kopii systemu grupie użytkowników. Testy takie nazywa się **testami beta**.

RODZAJE TESTÓW

Testy oprogramowania można poklasyfikować z różnych punktów widzenia.

Z punktu widzenia „celu" testowania:

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

Z punktu widzenia techniki wykonywania testów:

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

Z kolei testy dynamiczne można podzielić na:

- **Testy funkcjonalne** - zakładające znajomość jedynie wymagań wobec testowanej funkcji. System jest traktowany jako czarna skrzynka, która w nieznany sposób realizuje wykonywane funkcje.
- **Testy strukturalne** - zakładające znajomość sposobu implementacji testowanych funkcji.

TESTOWANIE DYNAMICZNE

Testy funkcjonalne

Zazwyczaj pełne przetestowanie rzeczywistego systemu jest praktycznie niemożliwe (z powodu dużej liczby kombinacji danych wejściowych i stanów).

Wówczas można założyć, że jeżeli dany program działa poprawnie dla kilku danych wejściowych, to działa także poprawnie dla całej klasy danych wejściowych (wnioskowanie heurystyczne).

Takie rozumowanie nie zawsze jest słuszne → fakt poprawnego działania dla kilku wybranych danych nie gwarantuje, że dla innych nie pojawi się błędne wykonanie.

W testach funkcjonalnych:

- system traktowany jako czarna skrzynka, która w nieznany sposób realizuje wymagane funkcje
- dane wejściowe dzielone są na klasy, w których działanie powinno być podobne i następnie sprawdzamy działanie funkcji dla takich wyróżnionych przypadków.

Testowanie strukturalne

Tego rodzaju testy wykonują osoby znające sposób implementacji. Dane do testów dobiera się w specjalny sposób oparty o znajomości programu:

- żeby pokryć wszystkie instrukcje programu,
- żeby pokryć działanie instrukcji warunkowych: aby każdy warunek raz był spełniony i raz niespełniony,
- żeby pokryć pętle: dane są dobierane taki sposób aby uwzględnić przypadek jednokrotnego i wielkokrotnego wykonania pętli, oraz przypadek gdy pętla nie wykona się ani jeden raz.

Testowanie statystyczne

Dane wejściowe dobierane są w taki sposób aby system został przetestowany w typowych sytuacjach:

- konstrukcja losowa danych wejściowych zgodnie z rozkładem prawdopodobieństwa wystąpienia tych danych;
- określenie jakie powinny wyjść wyniki na tych danych,
- uruchomienie programu i porównanie otrzymanych wyników z oczekiwanymi.

TESTOWANIE STATYCZNE

Testy statyczne - analiza kodu przez programistów, bez uruchomienia programu (są efektywniejsze od testów strukturalnych)

Można wyróżnić następujące techniki testów statycznych:

- metody formalne - dowody poprawności (bardzo trudne, dla programów o obecnej skali i złożoności najczęściej nie nadają się do zastosowania)
- metody nieformalne:
  - symboliczne śledzenie przebiegu programu (wykonywanie programu "w myśli" przez analizujące osoby)
  - wyszukiwanie typowych błędów.

Testy nieformalne są niedocenione, chociaż bardzo efektywne w praktyce.

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.

Sposób postępowania po ujawnieniu się błędnego wykonania:

- Jeżeli program ma czytelną i logiczną strukturę, powinno być możliwe ustalenie, która jego składowa źle działa.
- Staramy się „osaczyć błąd" - znaleźć możliwie mały fragment programu, który już wykazuje nieprawidłowe działanie.
- Możemy śledzić działanie programu i wartości zmiennych (za pomocą wydruków próbnych albo debuggera).
- Porównujemy ustalony przez nas pożądany sposób działania, z faktycznym przebiegiem sterowania i zawartością pamięci.

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

TOLERANCJA BŁĘDÓW
(tworzenie systemów odpornych na błędy)

Żadna technika nie gwarantuje uzyskania programu w pełni bezbłędnego. Tolerancja błędów oznacza, że program działa poprawnie, a przynajmniej sensownie także wtedy, kiedy zawiera błędy.

Tolerancja błędów oznacza wykonanie przez program następujących zadań:

- Wykrycia błędu.
- Awaryjnego wyjścia z błędu, tj. poprawnego zakończenie pracy modułu, w którym wystąpił błąd.
- Ewentualnej naprawy błędu, tj. zmiany programu tak, aby zlikwidować wykryty błąd.

Istotne jest podanie dokładnej diagnostyki błędu, z dokładnością do linii kodu źródłowego, w której nastąpił błąd.

Sposoby automatycznego wykrywania błędów:

- sprawdzanie warunków poprawności danych (tzw. asercje).
  Sposób polega na wprowadzaniu dodatkowych warunków na wartości wyliczanych danych, które są sprawdzane przez dodatkowe fragmenty kodu.
- porównywanie wyników różnych wersji modułu (opracowanych przez różnych programistów).

POMIARY POPRAWNOŚCI OPROGRAMOWANIA

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).

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.

Przyjmijmy, że:

B - liczba błędów w programie (przed posianiem nowych)
N - oznacza liczbę posianych błędów
M - oznacza liczbę wszystkich wykrytych błędów
X - oznacza liczbę posianych błędów, które zostały wykryte

Wówczas prawdziwa powinna być proporcja:

$$\frac{M-X}{B}=\frac{X}{N}$$

stąd szacowana liczba błędów przed wykonaniem testów:

$$B=(M-X)\frac{N}{X}$$

a szacunkowa liczba błędów pozostałych po usunięciu wykrytych:

$$E=(M-X)\frac{N}{X}-(M-X)=(M-X)\left(\frac{N}{X}-1\right)$$

Uwaga: te oszacowania mogą być nieścisłe, jeżeli posiane błędy nie będą podobne do oryginalnych błędów występujących w programie (przed posianiem).

Technika ta pozwala również na przetestowanie skuteczności metod testowania.
Zbyt mała wartość X/N oznacza konieczność poprawy tych metod.

Miary niezawodności oprogramowania:

- prawdopodobieństwo błędnego wykonania podczas realizacji transakcji (każde błędne wykonanie pojedynczej operacji powoduje zerwanie całej transakcji)
- częstotliwość występowania błędnych wykonań (liczba błędów/godz) stosowana dla 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 (zależy nie tylko od błędnych wykonań, ale także od narzutu błędów na niedostępność systemu)

Analiza niezawodności – drzewa błędów

Dla każdego zagadnienia można określić potencjalne niebezpieczeństwa związane z użytkowaniem systemu, a dla każdego z nich zdefiniować potencjalne przyczyny umożliwiające wystąpienie takiego przypadku.

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