Niezawodność

- Błąd – fault, error, bug
- Błędne wykonanie – failure
- Czy niezawodność jest ważna ?
- Miary niezawodności:
  - Prawdopodobieństwo błędnego wykonania podczas realizacji transakcji
  - Częstotliwość występowania błędnych wykonań
  - Średni czas pomiędzy błędnymi wykonaniami
  - Dostępność (procent czasu, w którym system jest dostępny)

Typowa zależność między gęstością błędów a niezawodnością

Gęstość błędów [/1000l]	Średni czas pomiędzy błędnymi wykonaniami
>30	<2 min
20-30	4-15 min
10-20	5-60 min
5-10	1-4h
2-5	4-24h
1-2	24-160h
<1	-

Jak poprawiać niezawodność

- Unikanie błędów
- Stosowanie warunków poprawności
- Odporność na błędy
- Testowanie
- Wykorzystanie gotowych komponentów
- Generowanie kodu

Testowanie

- Atestowanie i weryfikacja
  - Atestowanie – testowanie zgodności z rzeczywistymi potrzebami użytkowników
  - Weryfikacja – testowanie zgodności z wcześniej określonymi wymaganiami
- Cele testowania:
  - wykrycie i usunięcie błędów w systemie - wykrywanie błędów
  - ocena niezawodności systemu – testy statystyczne

Testowanie

- Techniki testowania:
  - testy dynamiczne – uruchamianie programu
  - testy statyczne – analiza kodu
- Etapy testowania
  - Testy jednostkowe – w modelu kaskadowym wykonywane w fazie implementacji
  - Testy integracyjne
  - Testy systemu
  - Testy akceptacyjne

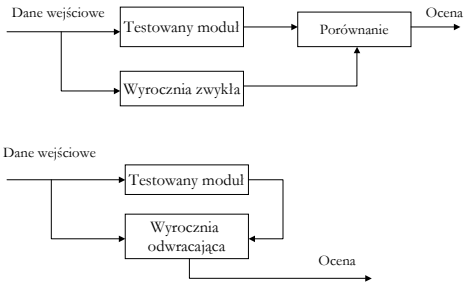
Testy dynamiczne

- Jak dobrać dane testowe (w tym informacje kontrolne)?
- Jak określić czy wykonanie jest poprawne?
- Jak zlokalizować błąd?

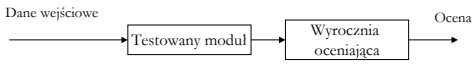
Jak określić czy wykonanie jest poprawne?

- Sytuacje oczywiste, np. zawieszenie się programu,
- Porównanie z poprawnymi wynikami - dane z przeszłości, wyniki wyznaczone ręcznie, porównanie z wynikami innych programów (zastosowanie wyroczni)
- Zastosowanie warunków poprawności - ręczne, zautomatyzowane (zewnętrzny lub wewnętrzny kod)
- Powtarzalność wyników

Wyrocznia zwykła i odwracająca



Wyrocznia oceniająca (stosująca warunki poprawności)



Wykorzystanie sytuacji szczególnych

- Np. funkcja:

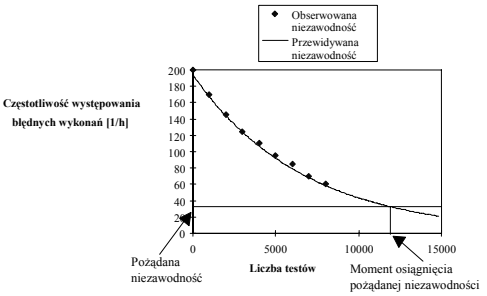
$$\cos(x)\sum_{i=1}^{I}\sin(x)^{I-i+1}x^{i}$$

- Przyjmie wartość 0 dla x = 0, $\Pi/2$, Π...

Testy statystyczne

- Losowy dobór danych testowych z rozkładem prawdopodobieństwa zbliżonym do rzeczywistego
- Możliwość automatyzacji
- Wykrywają przyczyny najczęstszych błędnych wykonań
- Pozwalają na oszacowanie niezawodności oprogramowania !

Poprawa niezawodności w testach statystycznych



Testy nastawione na wykrywanie błędów

- Dane testowe dobierane są systematycznie tak, aby maksymalizować szanse na wykrycie nowych błędów
- Rodzaje:
  - Funkcjonalne
  - Strukturalne

Testy funkcjonalne – testy czarnej skrzynki – testowanie według powinności

- Dane testowe dobierane są na podstawie analizy specyfikacji oprogramowania
- Podział danych wejściowych na klasy – grupy danych, dla których program działa jakościowo tak samo
- Testujemy dla kilku przypadków z każdej klasy
- Testowanie warunków granicznych

Testy funkcjonalne - przykład

- Zamówienia o wartości powyżej 3000 zł są realizowane w trybie przetargu, zamówienia do 3000 zł w trybie zapytania ofertowego
  - Zamówienie o wartości do 3000 zł
  - Zamówienie o wartości dokładnie 3000 zł
  - Zamówienie o wartości powyżej 3000 zł

Testy funkcjonalne – przykład, c.d.

- Sprzęt komputerowy jest zawsze kupowany w trybie przetargu
  - sprzęt komputerowy, wartość < 3000,
  - sprzęt komputerowy, wartość = 3000,
  - sprzęt komputerowy, wartość > 3000,
  - inne zakupy, wartość < 3000,
  - inne zakupy, wartość = 3000,
  - inne zakupy, wartość > 3000.

Testy funkcjonalne przykład

```
/** Posortowana lista różnych elementów
*  Dla każdego i, j >= 0 i < size () i < j => (*this) [i] <
   (*this) [j]
*/
class TOrderedList : public vector <double> {
public:
   /** Sprawdza czy podany element jest na liście */
   bool Exists (double dItem);
};
```


Klasy danych

- Lista jest pusta
- Lista nie jest pusta
  - Podany element jest na liście
  - Podanego elementu nie ma na liście

Testy funkcjonalne - przykład

- Funkcja przyjmuje dane w zakresie 1-100. Dla danych spoza tego zakresu zwraca wyjątek
- Klasy danych:
  - [1-100]
  - <1 i > 100
  - Wartości graniczne
  - 1 i 100

Testy strukturalne – testy białej/szklanej skrzynki – testowanie według implementacji

- Dane testowe dobierane są na podstawie analizy kodu i dokumentacji technicznej

Kryterium pokrycia wszystkich instrukcji programu

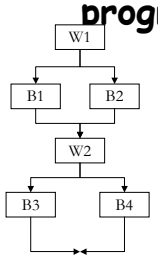
- Dobierz zestaw danych testowych tak, aby po wykonaniu wszystkich testów każda instrukcja programu została przynajmniej raz wykonana
- Kryterium często nie wystarczające:

```
int a;
if (W)
    a = b;
c = a;
```


Kryterium pokrycia wszystkich instrukcji programu - przykład

```
TData TerminPlatnosci()
   if (Faktura.Typ == _MATERIALY) { // W1
      TerminPlatnosci = min (DataAktualna + 14,
      Faktura.TerminPlatnosci); // B1
   }
   else {
      Faktura.Sprzet = true; // B2
   }
   if (Faktura.Wartosc > 1000) { // W2
      Faktura.OsobaAkceptujaca = Dyrektor; // B3
      PrzekazDoAkceptacji (TerminPlatnosci);
   }
   else {
      Faktura.Zaakceptowania = true; // B4
   }
```


Kryterium pokrycia wszystkich instrukcji programu - przykład



- Dwie ścieżki, np.:
  - W1-B1-W2-B3
  - W1-B2-W2-B4
- Zapewniają pokrycie wszystkich instrukcji programu

Kryterium pokrycia wszystkich instrukcji warunkowych

- Każda instrukcja warunkowa musi być przynajmniej raz spełniona i raz nie spełniona

```
int a;
if (W)
    a = b;
c = a;
```

Ta instrukcja musi być raz spełniona i raz nie spełniona

Kryterium pokrycia wszystkich warunków elementarnych

- Każdy warunek elementarny musi być przynajmniej raz spełniony i raz nie spełniony

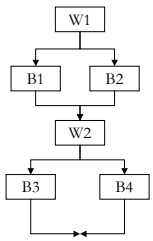
  if ((A > B) && (C == D))

- Każdy warunek elementarny (A > B) i (C == D) musi być przynajmniej raz spełniony i raz nie spełniony

Kryterium pokrycia wszystkich ścieżek w grafie przepływu sterowania programu

- Należy przejść przez testowany fragment programu wszystkimi możliwymi ścieżkami

Kryterium pokrycia wszystkich ścieżek w grafie przepływu sterowania programu



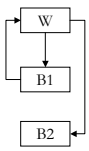
- Niezbędne jest pokrycie czterech ścieżek:
  - W1-B1-W2-B3
  - W1-B2-W2-B4
  - W1-B1-W2-B4
  - W1-B2-W2-B3

Łączenie kryteriów

- Jeżeli dana ścieżka jest definiowana przez warunek złożony, np.
  (A < B) || (A > C)
- to należy przejść tę ścieżkę przy (nie)spełnionym każdym z tych warunków (lub nawet każdej ich kombinacji)

Testowanie pętli

```
while (W)
  B1;
B2
```



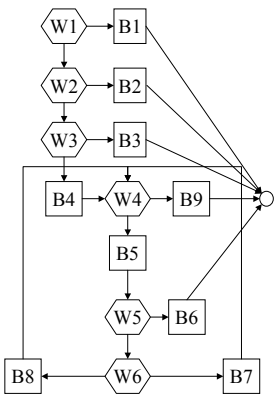
- Możliwe ścieżki:
  - W-B2
  - W-B1-W-B2
  - W-B1-W-B1-W -B2
  - ...
  - W-{B1-W} -B2

Testowanie pętli – wskazówki szczególne

- Dobierz dane tak, aby:
  - pętla została wykonana minimalną liczbę razy
  - pętla została wykonana maksymalną liczbę razy
  - pętla została wykonana przeciętną liczbę razy

Testy strukturalne przykład

```
/** Posortowana lista różnych elementów
*  Dla każdego i, j >= 0 i < size () i < j => (*this) [i] <
   (*this) [j]
*/
class TOrderedList : public vector <double> {
public:
   /** Sprawdza czy podany element jest na liście */
   bool Exists (double dItem);
};
```

Możliwe ścieżki

W1-B1

W1-W2-B2

W1-W2-W3-B3

W1-W2-W3-B4-W4-B9

W1-W2-W3-B4-W4-B5-W5-B6

W1-W2-W3-B4-W4-{B5-W5-W6-B7-W4}-B9

W1-W2-W3-B4-W4-{B5-W5-W6-B8-W4}-B9

W1-W2-W3-B4-W4-{B5-W5-W6-B8-W4| W4-B5-W5-W6-B8-W4}-B5-W5-B6

- W1-B1
  - Pusta lista
- W1-W2-B2
  - Lista niepusta. Podany element jest mniejszy od pierwszego elementu na liście lub większy od ostatniego elementu na liście
- W1-W2-W3-B3
  - Lista niepusta. Podany element jest równy pierwszemu lub ostatniemu elementowi na liście
- W1-W2-W3-B4-W4-B9
  - Lista niepusta o długości 1 lub 2. Podany element jest większy od pierwszego elementu na liście i mniejszy od ostatniego elementu na liście
- W1-W2-W3-B4-W4-B5-W5-B6
  - Lista niepusta o długości >= 3. Podany element jest równy środkowemu elementowi listy

- W1-W2-W3-B4-W4-{B5-W5-W6-B7-W4}-B9
- W1-W2-W3-B4-W4-{B5-W5-W6-B8-W4}-B9
  - Lista niepusta o długości >= 3. Podanego elementu nie ma na liście
- W1-W2-W3-B4-W4-{B5-W5-W6-B8-W4| W4-B5-W5-W6-B8-W4}-B5-W5-B6
  - Lista niepusta o długości >= 4. Podany element jest na liście


```
void main () {
      double a, b, c;
      cout << "Podaj a: ";
      cin >> a;
      cout << "Podaj b: ";
      cin >> b;
      cout << "Podaj c: ";
      cin >> c;
      double delta = b * b - 4 * a * c;
      if (delta < 0)
            cout << "Brak rozwiązań rzeczywistych\n";
      else if (delta == 0) {
            cout << "Rozwiazanie: " << (-b) / (2 * a);
            cout << '\n';
      }
      else {
            cout << "Rozwiązania: ";
            cout << (-b - sqrt (delta)) / (2 * a) << "   ";
            cout << (-b + sqrt (delta)) / (2 * a) << '\n';
      }
}
```

Testy statyczne

- Formalne – dowody poprawności
- Nieformalne – inspekcje kodu (choć to drugie jest pojęciem szerszym)
  - ~150 linii kodu/h
  - Inspekcje kodu są bardzo efektywną formą testowania !

Dwa sposoby analizy programu

- Śledzenie przebiegu programu
- Liniowy przegląd kodu

Lista kontrolna typowych błędy

- Błąd zakresu
- Błędna operacja na zmiennych dynamicznych
- Błąd w wyrażeniu arytmetycznym/logicznym/znakowym
- Błąd w instrukcji warunkowej
- Niezainicjowanie zmiennej
- Niezgodność typów/błędna konwersja typów
- Nieuwzględnienie sytuacji wyjątkowej
- Błąd algorytmiczny
- Błąd techniczny

Rodzaje nieformalnych testów statycznych

- Indywidualne przeglądy kodu (cudzego lub własnego)
- Formalne przeglądy techniczne
- Praca parami

Formalne przeglądy techniczne

- 3-5 osób
- Wstępne przygotowanie 1-2h na osobę
- Czas trwania < 2h
- Uczestnicy:
  - Autor, lider przeglądu, 2-3 recenzentów, sekretarz
- Prezentacja autora
- Decyzja
  - Akceptacja, zwrot do małych poprawek, zwrot do dużych poprawek
- Podpis

Formalne przeglądy techniczne - wskazówki

- Ocena produktu, a nie autora
- Ustalenie i trzymanie się zakresu przeglądu
- Ograniczona dyskusja
- Wskazywanie błędów, a nie koniecznie rozwiązywanie
- Notatki
- Ograniczona liczba uczestników
- Wcześniejsze przygotowanie
- Lista kontrolna
- Zapewnienie zasobów i zaplanowanie w harmonogramie

Testy jednostkowe

- Testy jednostkowe nie powinny być odkładane w czasie
- Zasada „trochę kodu, test, trochę kodu, test"
- Fowler: „Zawsze, kiedy chcesz wyprowadzić na ekran lub zobaczyć pod debuggerem jakąś zmienną lub wyrażenie, napisz prosty test"
  - lub warunek poprawności

Testy poprzedzające kodowanie

- Zasady stosowane np. w programowaniu ekstremalnym
- Wytwarzanie
  - Najpierw test, potem funkcjonalność
- Wprowadzanie poprawek
  - Najpierw test, potem poprawka

Warunki poprawności a warunki testów

- Warunki poprawności muszą być spełnione zawsze niezależnie od danych wejściowych
- Warunki testów są spełnione dla konkretnych danych

Narzędzia testów jednostkowych

- JUnit, CppUnit
- Funkcje
  - Definiowanie testów jednostkowych
  - Współdzielenie kontekstu testowania – zmienne, obiekty
  - Zestawy testów
  - Wykonywanie testów (wzorzec *Template method*)
  - Interfejs użytkownika

Przykład

```
class Complex {
 friend bool operator ==(const Complex& a,
   const Complex& b);
  double real, imaginary;
public:
  Complex( double r, double i );
};
```


Wykorzystanie CppUnit

```
class ComplexNumberTest : public CppUnit::TestCase {
public:
  ComplexNumberTest( std::string name ) : CppUnit::TestCase(
    name ) {}

  void runTest() {
    CPPUNIT_ASSERT( Complex (10, 1) == Complex (10, 1) );
    CPPUNIT_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );
  }
};
```

Testowanie połączone z zarządzaniem konfiguracjami

- Fakt, że moduł/program przeszedł zestaw testów nie oznacza, że przejdzie go w przyszłości !!!
- Testowanie po zbudowaniu systemu
- Zautomatyzowany zestaw testów sprawdzających poprawność aktualnej wersji
- Automatyczne uruchamianie testów po wprowadzeniu zmian w repozytorium projektu

Szacowanie skuteczności testowania i liczby nieznanych błędów

- Posiewanie błędów
- Porównywanie niezależnych testów

Posiewanie błędów

- X - liczba nieznanych błędów w programie
- P – liczba posianych błędów
- W – liczba wykrytych błędów
- WP ≤ W liczba wykrytych posianych błędów

- WP/P – skuteczność testowania
- X = P/WP × (W-WP)

Porównywanie niezależnych testów

- W1, W2 – liczby błędów wykrytych w dwóch niezależnie przeprowadzonych testach
- WP <W1, W2 – liczba wspólnych błędów wykrytych w obu testach
- WP/W2, WP/W1 – skuteczności testowania w testach pierwszym! i drugim!
- X = W2/WP × (W1-WP)
- X = W1/WP × (W2-WP)

Psychologiczne aspekty testowania

- Problem motywacji
- Czy powinno się wprowadzać oddzielny zespół testujący
- Osoby szczególnie predestynowane do testowania

Unikanie błędów

- Kwestia nastawienia psychicznego
- Ile kosztuje błąd?
  - Koszt wykrycia (błędnego wykonania)
  - Koszt lokalizacji
  - Koszt poprawy
  - Koszt dystrybucji do klientów
  - Obsługa zgłoszeń o niepoprawnym działaniu programu
  - Koszt opinii klientów
- Analiza przyczyn błędów

Unikanie niebezpiecznych technik

- Goto, itp.
- Wskaźniki
- Liczby zmiennopozycyjne
- Rekursja
- Przerwania
- Obliczenia równoległe
- Skomplikowane wyrażenia

Zasada ograniczonego dostępu

- Lub zasada *„need to know"* (dostęp do danych ma tylko ten, kto **musi wiedzieć**)
- Dostęp do danych i funkcji powinny mieć tylko te fragmenty kodu, które z nich rzeczywiście korzystają
  - Trudne do pełnego zrealizowania w typowych językach programowania
- Języki obiektowe
  - Pola i metody chronione, prywatne, klasy, funkcje zaprzyjaźnione, pakiety

Stosowanie kompilatorów sprawdzających zgodność typów - wymuszanie niezgodności typów

- Przykład z zajęć:
  - FunctionChange = Distance (...) + Distance (...) - Distance (...) - Distance (...)

int unsigned int

- Wymuszanie niezgodności typów, np. poprzez typy obiektowe

C++ językiem o silnej typizacji?

```
typedef int TLiczbaPracownikow;
typedef int TLiczbaKomputerow;
typedef int TLiczbaDni;

TLiczbaPracownikow LiczbaPracownikow;
TLiczbaKomputerow LiczbaKomputerow;
TLiczbaDni LiczbaDni;

LiczbaKomputerow = LiczbaDni + LiczbaPracownikow;
```

Poprawne

Pisanie czytelnego kodu

```
TData du;
TData da;
TCzas cp;

Cp = da – du;

CzasPracy = DataAktualna – DataUrodzenia;
```

- Rola standardów kodowania

Warunki poprawności

- Warunek poprawności to warunek logiczny, który zawsze musi być spełniony w danym miejscu programu
- Jest to warunek konieczny, ale nie wystarczający poprawności programu

Źródła warunków poprawności

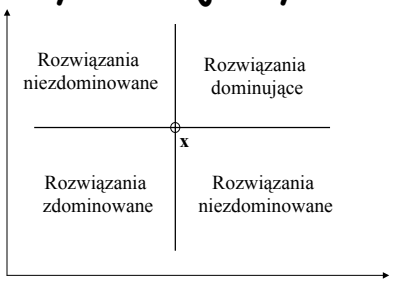
- Ograniczenia dotyczące danych – zmiennych, pól, parametrów
  - Wzrost > 0
  - Płaca minimalna <= Płaca maksymalna
  - Rodzic.DataUrodzenia < Dziecko.DataUrodzenia
  - Dla każdego i, j i < j => A [i] < A [j]
- Niezmienniki algorytmów

Sortowanie tablicy N elementowej

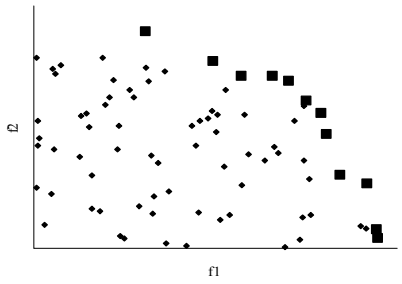
Dla każdego i = 1,...,N-1 {
  Dla każdego j = i + 1,...,N {
    Jeżeli *A* [i] > *A* [j] **to**
      Zamień *A* [i] z *A* [j]
  }
  (A[i] <= A[i + k], k=1,...N-i)
}

Niezmiennik algorytmu

Relacja dominacji (dla maksymalizacji kryteriów)

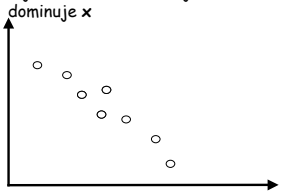


Rozwiązania niezdominowane



Uaktualnianie zbioru rozwiązań Pareto-optymalnych

- Uaktualnianie zbioru *PP* przy wykorzystaniu rozwiązania **x**
  - Wszystkie rozwiązania zdominowane przez **x** są usuwane z *PP*
  - **x** jest dodawane do *PP* jeżeli żadne rozwiązanie w *PP* dominuje **x**




```
bool bZdominowane = false;
bool bDominujace = false;
Dla każdego i dopóki bZdominowane == false
   Jeżeli X dominuje N [i] to
      Usuń N [i]
      bDominujace = true
   W przeciwnym wypadku jeżeli X jest
     zdominowane przez to N [i]
      bZdominowane = true
Jeżeli !bZdominowane to
   Dodaj X do N
!(bZdominowane && bDominujace)
```

Warunki poprawności

- Cechy dobrego warunku poprawności
  - Prostota
  - Ogólność
- Makro assert w C++

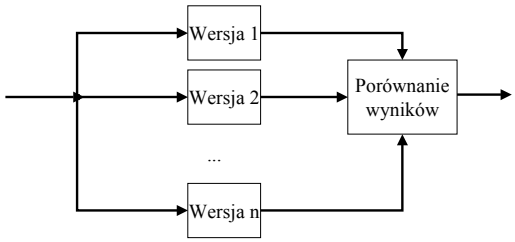
Odporność na błędy (tolerancja błędów)

- Program jest odporny na błędy jeżeli nie prowadzą one do błędnych wykonań
- Etapy:
  - wykrycie błędu
  - wyjście z błędu, tj. zakończenia pracy modułu, w którym wystąpił błąd w poprawny sposób
  - ewentualna naprawa błędu, tj. zmiany programu tak, aby zlikwidować wykryty błąd

Jak wykrywać błędy podczas pracy programu

- Sprawdzanie warunków poprawności danych
- Porównywanie wyników różnych wersji modułu

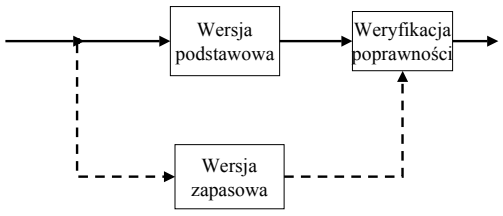
Programowanie n-wersyjne



Programowanie n-wersyjne

- Załóżmy, że prawdopodobieństwo wystąpienia błędnego wykonania w module wynosi 0.01
- Jeżeli są to zdarzenia niezależne to prawdopodobieństwo wystąpienia błędnego wykonania jednocześnie w dwóch wersjach modułu wynosi $0.01 \times 0.01 = 0.0001$
- Prawdopodobieństwo takiego samego błędnego wykonania może być jeszcze dużo mniejsze

Technika zapasowych modułów



Powrót do poprzedniego stanu

- Stosowane, jeżeli korzystamy tylko z jednej wersji modułu i nie możliwe zapewnienie poprawnego wykonania

- Zachowywanie poprawnego stanu
- Zapamiętywanie zmian
- Nadmiarowość danych