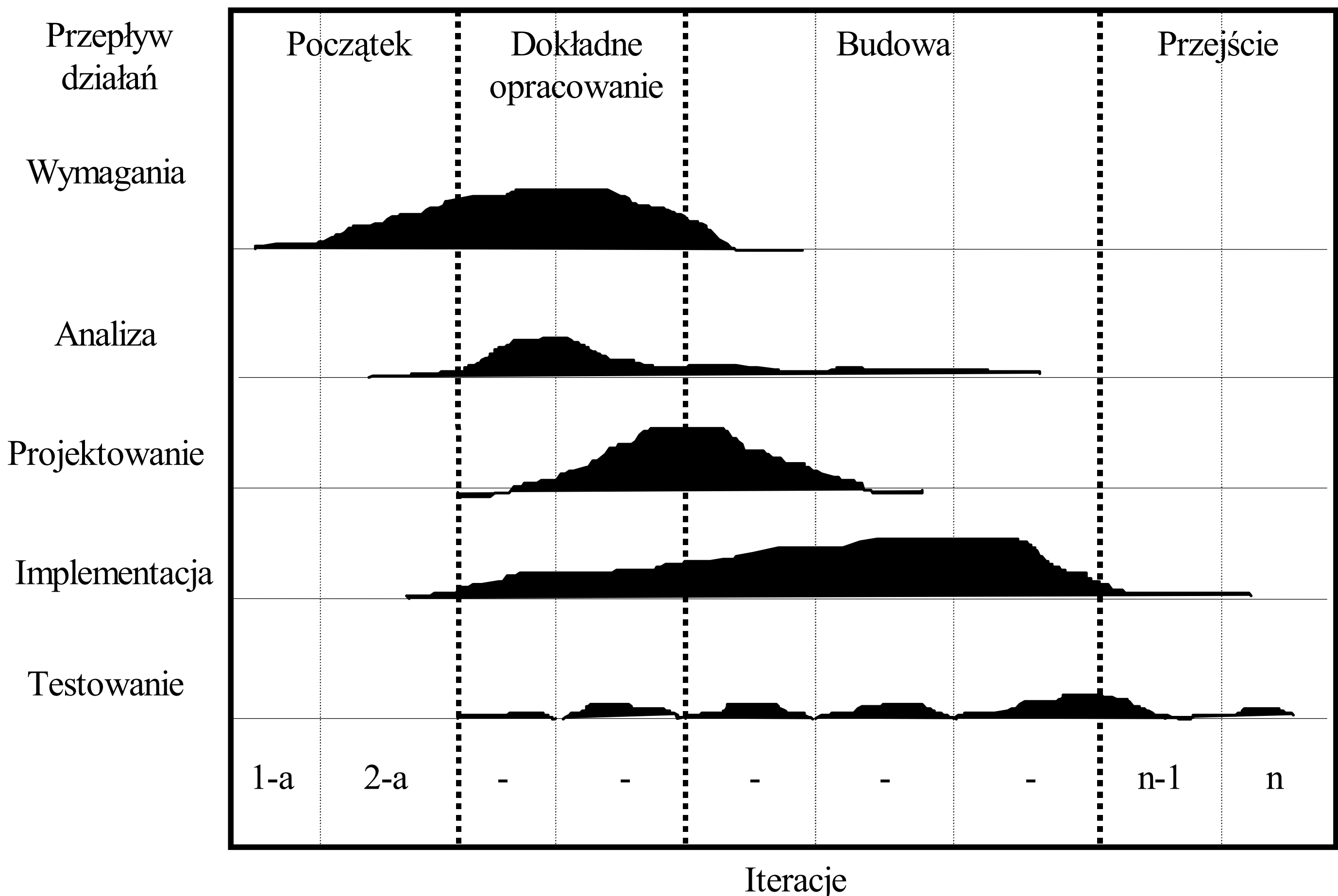
Testowanie programów

6.1. Rola testowania w tworzeniu oprogramowania

Kluczową rolę w powstawaniu oprogramowania stanowi proces usuwania błędów w kolejnych fazach rozwoju oprogramowania na drodze testowania (różne metody testowania dostosowane do stopnia rozwoju oprogramowania).

W inżynierii oprogramowania poszukuje się związku między strukturą programu a możliwością powstawania pewnych błędów oraz trudnością ich wykrywania na drodze testowania.



Rys.1. Przepływ działań w iteracyjno-rozwojowym procesie powstawiania oprogramowania obiektowego – udział testowania

6.2. Definicje

Atestowanie (validation) - testowanie zgodności systemu z rzeczywistymi potrzebami użytkownika (czy zbudowano poprawny produkt).

Weryfikacja (verification) - testowanie zgodności systemu z wymaganiami zdefiniowanymi w fazie określania wymagań (czy zbudowano produkt poprawnie w kolejnych fazach życia)

Błąd (fault, error, defect) jest niepoprawną konstrukcją znajdującą się w oprogramowaniu, która może prowadzić do niewłaściwego działania.

Błędne wykonanie - **uszkodzenie** (failure) to niepoprawne działanie systemu w trakcie jego pracy na skutek błędów. Takie same błędne wykonanie może pochodzić od różnych błędów. Jednak błędy nie muszą powodować błędnego wykonania programu!

Klasyfikacja testów:

1. ze względu na cel:
   1.1. testy wykrywające błędy
   1.2. testy statystyczne, określające przyczyny najczęstszych błędnych wykonań oraz ocena niezawodności systemu
2. ze względu na technikę wykonania:
   2.1. testy dynamiczne polegające na wykonaniu fragmentu lub całego programu i porównaniu wyników jego działania z wynikami poprawnymi. Możliwe jest wykonywanie „metaprogramów" wykonanych w różnych fazach powstawania oprogramowania: testy funkcjonalne i strukturalne (metaprogramy)
   2.2. testy statyczne, czyli inspekcje struktury produktu, *udowadnianie poprawności programu*

Testy dynamiczne i statyczne *nie są komplementarne*, tzn. mogą służyć do wykrywania różnych błędów.
Kolejność wykonania testów w procesie powstawiania oprogramowania jest zależna od przyjętej metody testowania i tworzenia oprogramowania.

W końcowej fazie wyróżnia się następujące testy:

- testy modułów
- testy systemu
- testy akceptacji (testy alfa i beta).

Klasyfikacja błędów

1) błędy wymagań i analizy: złe sformułowanie problemu, zaniedbanie istotnych parametrów, niewłaściwy algorytm,
2) błędy projektowania: błędna interpretacja wymagań, błędy logiczne
3) błędy programowe:
   3.1) błędy opracowania szczegółowej struktury programu: zła interpretacja wymagań dla programu, niepełność struktury programu, nie uwzględnienie przypadków szczególnych, niedostateczne dopracowanie błędów, zlekceważenie warunków czasowych
   3.2) błędy kodowania:
      3.2.1) syntaktyczne, zazwyczaj rozpoznawane przez kompilator,
      3.2.2) błędy merytoryczne (nieprawidłowe korzystanie z indeksów i wskaźników, zły przydział pamięci, pominięcie inicjalizacji zmiennych, pomieszanie parametrów funkcji, błąd w pętlach, zamiana wyników decyzji w instrukcjach warunkowych, błędy deklaracji typów i wymiarów danych, błędy zakresów wartości danych,
   3.3) błędy kompilacji i konsolidacji: błędy kompilatora, błędy w zakresach nazw itp.

6.3. Proces testowania symbolicznego

Testowanie i usuwanie błędów prowadzą do osiągnięcia dużej niezawodności programów. Jednak nawet przy dużych nakładach na testowanie można przeanalizować zaledwie małą część wszystkich możliwych kombinacji danych wejściowych wielkiego systemu oprogramowania.

Uwagi dotyczące poprawy testowania:

- należy unikać struktur typu *goto*
- należy dostosować odpowiednio struktury danych do wykonywanych algorytmów, i na odwrót (Wirth)
- należy tworzyć programy łatwo modyfikowalne ze względu na struktury danych
- ograniczyć powiązania między modułami programu
- inicjować zmienne, ograniczyć zmienne globalne, rozważnie operować wskaźnikami, przydziałem pamięci, indeksami tablic (C,C++!)
- należy zabezpieczyć program przed przekroczeniem zakresu wartości danych i przed pomyłkami przy ich wprowadzaniu (nie należy zakładać, że ten obowiązek powinien spoczywać na „użytkowniku” np. zapobieganie dzielenia przez zero)

Problemy testowania:

- trudność w określeniu możliwie najmniejszej liczby zachowań programu, wynikającego z pewnego zbioru danych, które należy sprawdzić i i uogólnić indukcyjnie uzyskane wyniki
- w podejściu statystycznym istnieje tendencja do ułatwiania postępowania i opierania się na często niezbyt dobrze uzasadnionych założeniach (losowy rozkład danych, wzajemna niezależność czynników badanych procesów, operowanie średnią lub wariancją)

Rozwiązanie dużej liczby rozpatrywanych danych można zastąpić metodą wykonywania symbolicznego, opartej na:

- symbole bądź wyrażenia algebraiczne używane są jako wartości zmiennych. Instrukcje podstawienia podstawiają za zmienne wyrażenia algebraiczne
- wybór gałęzi przy instrukcji warunku wprowadza ograniczenia dla symboli
- wykonywanie symboliczne dotyczy całych, często nieskończenie wielkich zbiorów instrukcji, co ogranicza wykorzystania szczególnych atrybutów wartości, które może przybrać symbol.

Przykład 6.1 Przykład symbolicznego wykonania programu sprowadzony do odpowiedniego testowania warunków bez analizowania wartości zmiennych

```
#include "stdio.h"
void main ()
{ float x,y,z;
  if (scanf("%f%f",&x,&y)==2)  // zabezpieczenie przed niewłaściwą formą danych x i y
   {z=2*x + y;
   if (z==0) x=1;              //zabezpieczenie przed niewłaściwą wartością danych
   else x=1/z+1/y;}}
```

Przykład 6.2 [7]

Testowanie błędnej wersji programu do znajdowania pierwiastka kwadratowego z *p*, gdy przedziału $0 \le p < 1$ z dokładnością do *err*, gdzie $0 \le err < 1$.

```
#include "stdio.h"
float pierwiastek_kw(float p, float err)
{ float d=1, ans=0, tt=0, c=2*p;
  //wylicz pierwiastek kwadratowy z p, 0<=p<11 z dokładnością do err, 0 <= err < 1
  if (c >= 2) return 0;         //punkt rozgałęzienia A, p<1 ?
  do
  { if (d<=err) return ans;     //punkt rozgałęzienia B
    d=0.5 * d;
    tt=c-(d+2*ans);
    if (tt>=0)                  //punkt rozgałęzienia C
       {ans=ans+d;              //ten i kolejny wiersz powinny być zamienione
        c=2*(c-(2*ans+d));}
    else c=2*c;
   } while (1);
}
void main ()
{float ans,p,err;
printf("Podaj liczbe i dokladnosc: ");
if (scanf("%f%f",&p,&err)==2)
 {ans=pierwiastek_kw(p, err);
 printf("pierwiastek kw z dokl %f z liczby %f=%f\n",err,p,ans);}
}
```

Wybrano trzy punkty rozgałęzienia: A, B, C i rozważono następujący ciąg wyborów miejscach:

sekwencje wykonywane przez program	p	Err	d	ans	tt	c
A false	p < 1		1	0	0	2*p < 2
B false	p < 1	err < 1	d > err	0	0	2*p < 2
C true ?	p < 1	err < 1	0.5	0.5	2*p - 0.5 >= 0	4*p - 3
B true, exit	p < 1	err >=0.5	0.5	0.5	2*p - 0.5 >= 0	4*p - 3
A false ?	p < 1		1	0	0	2*p < 2
B false ?	p < 1	err < 1	d > err	0	0	2*p < 2
C true ?	p < 1	err < 1	0.5	0.5	2*p – 0.5 >= 0	4*p - 3
B false ?	p < 1	err < 0.5	0.5	0.5	2*p – 0.5 >= 0	4*p – 3
C false !	p < 1	err < 0.5	0.25	0.5	4*p – 4.25 < 0	8*p - 6
B true exit	p <1	err>=0.25	0.25	0.5	4*p – 4.25 < 0	8*p - 6

Po sekwencji <*A false, B false, C true, B true*> mamy

$0.5=ans \le p^{1/2} + err$ i $0.5=ans \ge p^{1/2} - err$, p=err=0.995; $p^{1/2} \approx 0.997$

Po sekwencji <*A false, B false, C true, B false, C false, B true*> mamy jednak

$0.5=ans < p^{1/2} - err$ dla p=0.995, err=0.49 .

Program nie przeszedł pomyślnie testu, jednak nie znaleziono przyczyny błędu.

6.4. Opis techniki indukcji kontynuacyjnej

Indukcja kontynuacyjna rozszerza wykonywanie symboliczne do metody dowodzenia formalnego.

Przykład 6.3 [7]

```
wejście: I := 1; R := 1;
pętla:  if I< N then
             begin I := I + 1; R := R*I ;  goto pętla
           end
wyjście:
```

Należy zdefiniować asercję L : x = wyrażenie, która stwierdza, że istnieje pewien stan programu powstający wtedy, gdy sterowanie znajdzie się w punkcie oznaczonym etykietą L, w którym wartością symboliczną identyfikatora x jest wyrażenie algebraiczne *wyrażenie.* Należy udowodnić, że powyższy program oblicza wartość silni, czyli:

Jeśli $n \geq 0$, zaś na wejściu $N = n$, to na wyjściu $N = n$, $I = n$, $R = n!$.

Dowód:

Założenie $N = n$

pętla : $N = n$. $I = 1$, $R = 1$ dla $i = 1$ udowodniono

Należy udowodnić, że dla $1 \leq i \leq n$

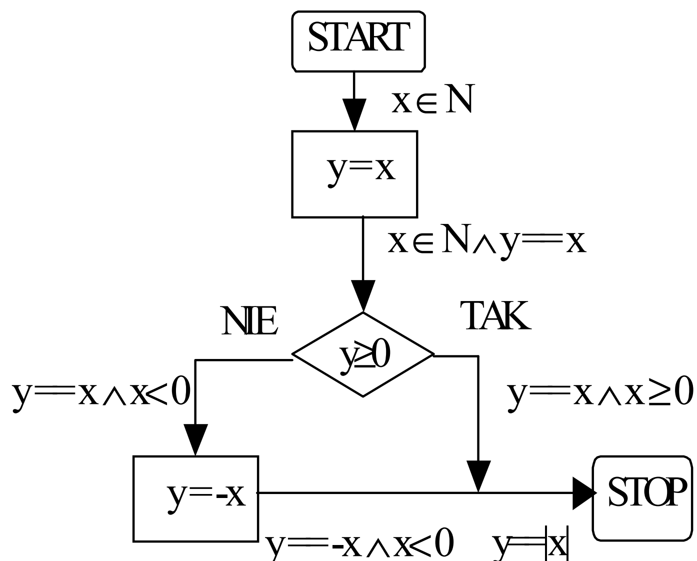
pętla: $N = n$, $I = i$, $r = i!$

pętla : $N = n$, $I = i+1$, $R = i! * (i+1)$ dla $i < n$ dla $i=i+1$

wyjście: $N = n$, $I = n$, $R = n!$

6.5. Przykłady testów

Przykład 6.4



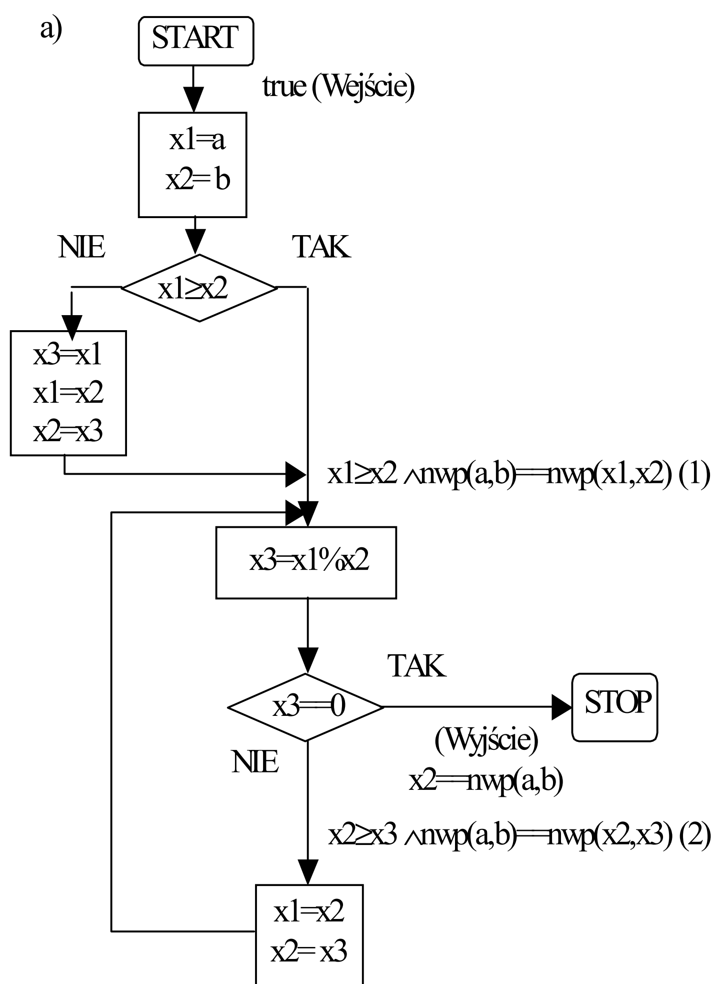
Sekwencja	x	y
A true exit	x≥0	y=x≥0
A false exit	x < 0	y = -x >0

Stąd y = |x| dla 0 < x i x ≥ 0

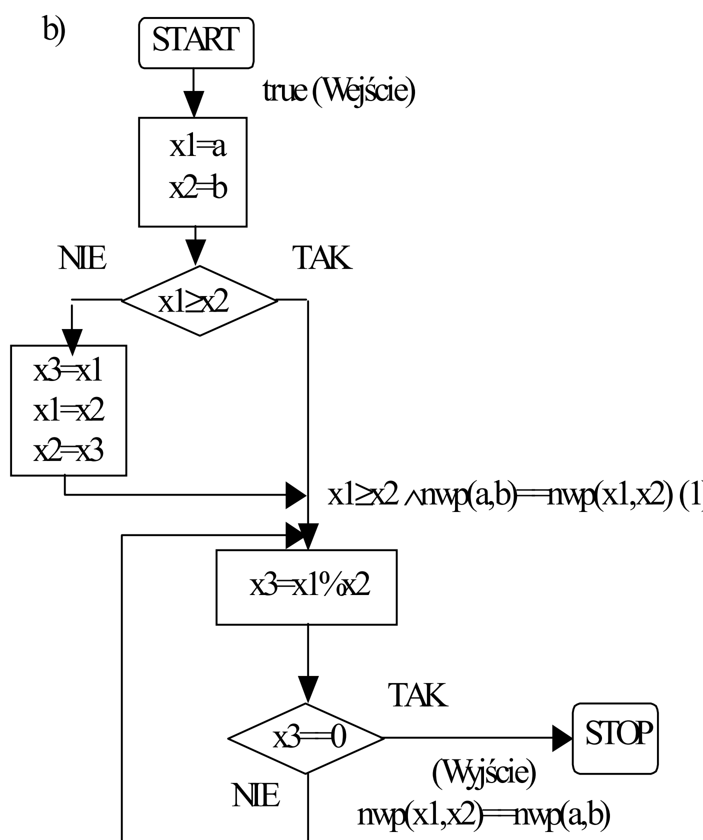
Test indukcji kontynuacyjnej
Tak: x ≥ 0, y == x ≥ 0
Nie: x <0, y = =-x > 0
Stop : y == |x|
Dowód testu pomyślny

Przykład 6.5

a)



b)



Przykład 6.5 a) i 6.5.b) dotyczy wyznaczenia największego wspólnego podzielnika wyznaczonego wg algorytmu Euklidesa [7].

Testy programu z przykładu 6.5 a)

Należy wykazać, że program oblicza poprawnie wartość x2 =nwp (a,b)

S oznacza następujący program C++:

```
const int a=2;
const int b=11;
void main()
{ int x1,x2,x3;
  if (a>=b)  {x1=a;x2=b;} //punkt A
  else {x1=b;x2=a;}
  x3=x1%x2;
  while(x3)             // punkt B
  { x1=x2;
    x2=x3;
    x3=x1%x2;}}
```

Sekwencja wykonania	x1	x2	x3
A true	a	b	x1 % x2
B true	x1 = x2	x2=x3	x3 = x1 % x2
B false exit	x1 = x1	x2 = x2=nwp(x1,x2)	x3=x1%x2=0
A false	b	a	x1 % x2
B true	x1 = x2	x2=x3	x3 = x1 % x2
B false exit	x1 = x1	x2 = x2=nwp(x1%x2)	x3=x1%x2=0

Test indukcji kontynuacyjnej
Tak: a ≥ b, x1=a, x2 = b, x3=x1 % x2
pętla: x3>0, x1=x2, x2= x3, x3 = x1 % x2
wyjście: x1=x1, x2=x2, x3 =x1%x2=0, x2=nwp(x1,x2)=nwp(a,b)

Tak: a<b, x1=b, x2 = a, x3=x1 % x2
pętla: x3>0, x1=x2, x2= x3, x3 = x1 % x2
wyjście: x1=x1, x2=x2, x3 =x1%x2=0, x2=nwp(x1,x2)=nwp(a,b)

Testy programu z przykładu 6.5 b)

Należy wykazać, że program oblicza poprawnie wartość x2 =nwp (a,b)

S oznacza następujący program C++:

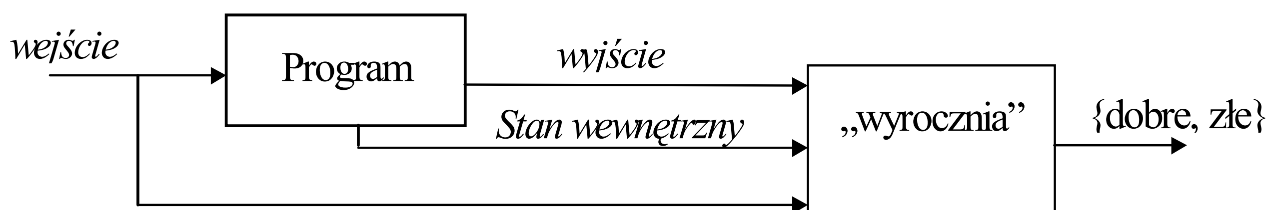
```
const int a=2;
const int b=10;
void main()
{ int x1,x2,x3;
  if (a>=b)  {x1=a;x2=b;}
  else {x1=b;x2=a;}
  x3=x1%x2;
  while(x3)x3=x1%x2;}
```

Sekwencja wykonania	x1	x2	x3
A true	a	b	x1 % x2
B true	x1 = a	x2=b	x3 = x1 % x2
B false exit ?	x1 = a	x2 = b=nwp(a,b)	x3=a%b=0
A false	b	a	x1 % x2
B true	x1 = b	x2=a	x3 = x1 % x2
B false exit ?	x1 = b	x2 = a=nwp(b%a)	x3=b%a=0

Test indukcji kontynuacyjnej
Tak: a ≥ b, x1=a, x2 = b, x3=x1 % x2
pętla: x3>0, x1=x1, x2= x2, x3 = x1 % x2
wyjście: x1=a, x2=b, x3 =a%b=0, x2=nwp(a,b). tylko gdy b=nwp

Tak: a<b, x1=b, x2 = a, x3=x1 % x2
pętla: x3>0, x1=x2, x2= x3, x3 = x1 % x2
wyjście: x1=b, x2=a, x3 =p%a=0, x2=nwp(b,a) tylko, gdy a jest nwp

6.6. Testowalność



Rys. 2. Testowanie i wyznaczanie testowalności [A.Bertolino,L.Strigini:On the Use of Testability Assessment,IEEE TRANSACTION ON SOFTWARE ENGINEERING,vol. 22, no. 2, February 1996]

Wyrocznia jest następującą funkcją:

$$Wyrocznia: D \times R \times (zbiór_wartości_stanów_programu) \rightarrow \{dobry, zły\}$$

gdzie *D* - dziedzina danych wejściowych, *R* - dziedzina danych wyjściowych
zbiór_wartości_stanów_programu–zbiór obserwowanych wartości zmiennych

1) *Testowalność* oprogramowania $Testab_{ABS}$ jest prawdopodobieństwem warunkowym, że wynik testu programu dla wejścia określonego prawdopodobieństwem dystrybucji wejść jest *zły* (określony przez funkcję *wyroczni*), pod warunkiem, że wystąpiły błędy w programie.

$$Testab_{ABS} = P(\text{ zły } | \text{ prawd. dystrybucji wejść, wyrocznia, błędy})$$

2) *Testowalność* oprogramowania $Testab_{HV}$ jest prawdopodobieństwem, że program jest uszkodzony (błędnie wykonany) dla danego prawdopodobieństwa dystrybucji wejść, pod warunkiem że wystąpiły błędy w programie.

$$Testab_{HV} = P(\textit{ uszkodzony } | \textit{ prawd. dystrybucji wejść, błędy})$$

Zależność między definicjami jest następująca:

$$Testab_{ABS} = Testab_{HV} \frac{Pokrycie}{P(uszkodzony \mid \text{bledny stan programu})}$$

gdzie

Pokrycie jest prawdopodobieństwem, że wynik *wyroczni* będzie *zły* dla danego prawdopodobieństwa dystrybucji wejść, gdy wystąpił błędny stan programu.

$$Pokrycie = P(\text{ zły } | \text{ prawd. dystrybucji wejść, bledny stan programu })$$

bledny stan programu: błędy w obserwowanych zmiennych

4) Związek między testowaniem (liczbą przeprowadzonych testów) i niezawodnością

Niezawodność programu jest częstotliwością jego błędnych wykonań. Rośnie ona logarytmicznie w zależności od liczby przeprowadzonych testów.

$$Niezawodność = Niezawodność_poczatkowa * exp(-C * Liczba_testów)$$

6.7. Ocena liczby błędów metodą posiewania błędów

Na podstawie wszystkich znalezionych błędów oraz błędów sztucznie wprowadzonych do programu można oszacować liczbę błędów w programie.

N - liczba wprowadzonych błędów
M - liczba wszystkich wykrytych błędów
X - liczba wprowadzonych błędów, które zostały wykryte

Szacunkowa liczba błędów przez wykonaniem testów:

$$Błędy_{calk} = \frac{(M - X) \bullet N}{X}$$

Liczba błędów po usunięciu wykrytych, w tym wszystkich sztucznie wprowadzonych:

$$Błędy_{Poz} = (M - X) \bullet (\frac{N}{X} - 1)$$

Współczynnik *X/N* opisuje efektywność wykonywanych testów..

Metody posiewania błędów:
- losowe zakłócenia w przypisywaniu danych
- losowe mutacje kodu - zmiany kodu źródłowego zmieniającego sterowanie lub dane w programie
- losowe zakłócenia między interfejsami modułów