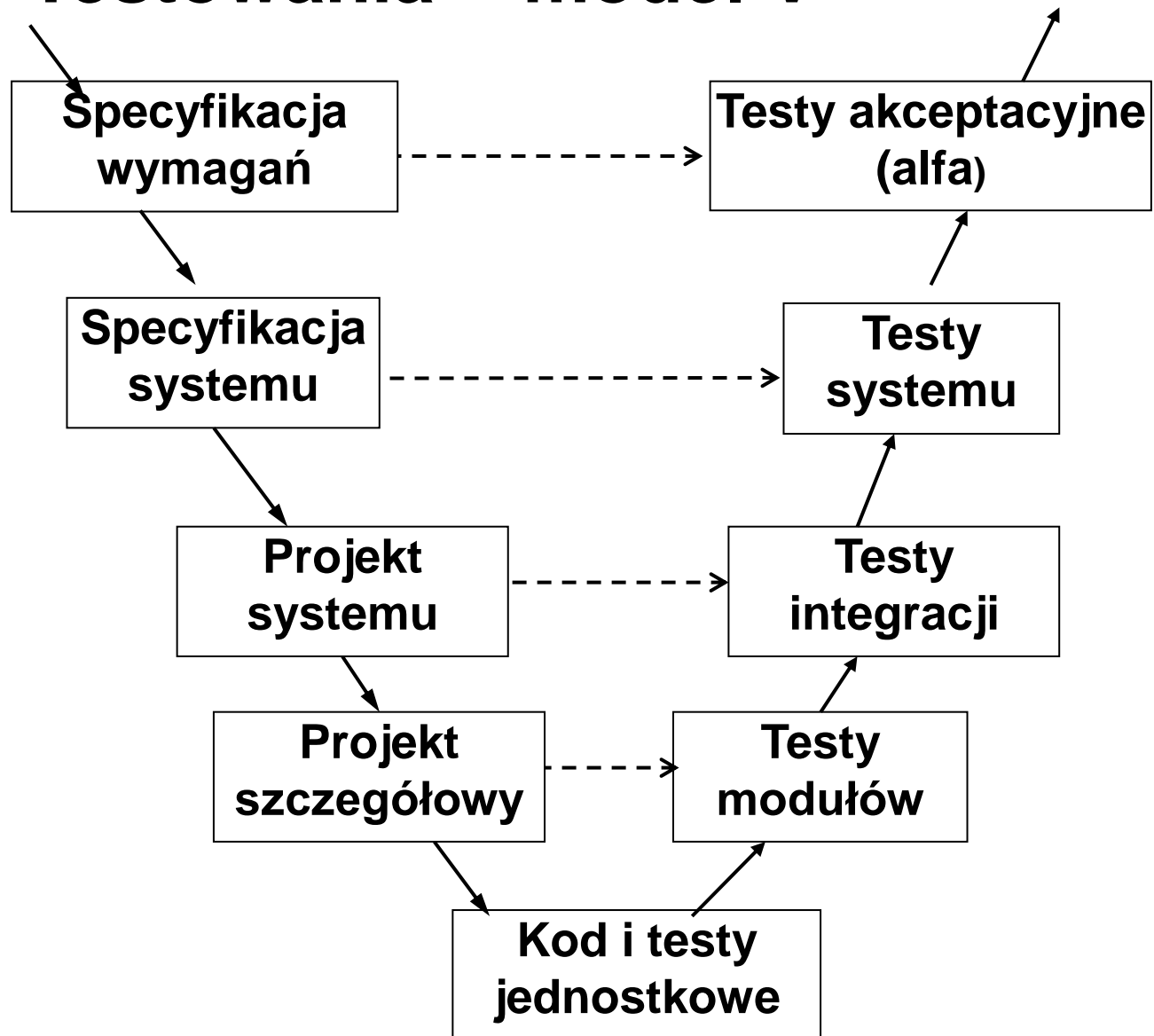


TESTOWANIE UKIERUNKOWANE NA WYSZUKIWANIE DEFECTÓW W PROGRAMIE (DEFECT TESTING)

Dr hab. inż. Ilona Bluemke

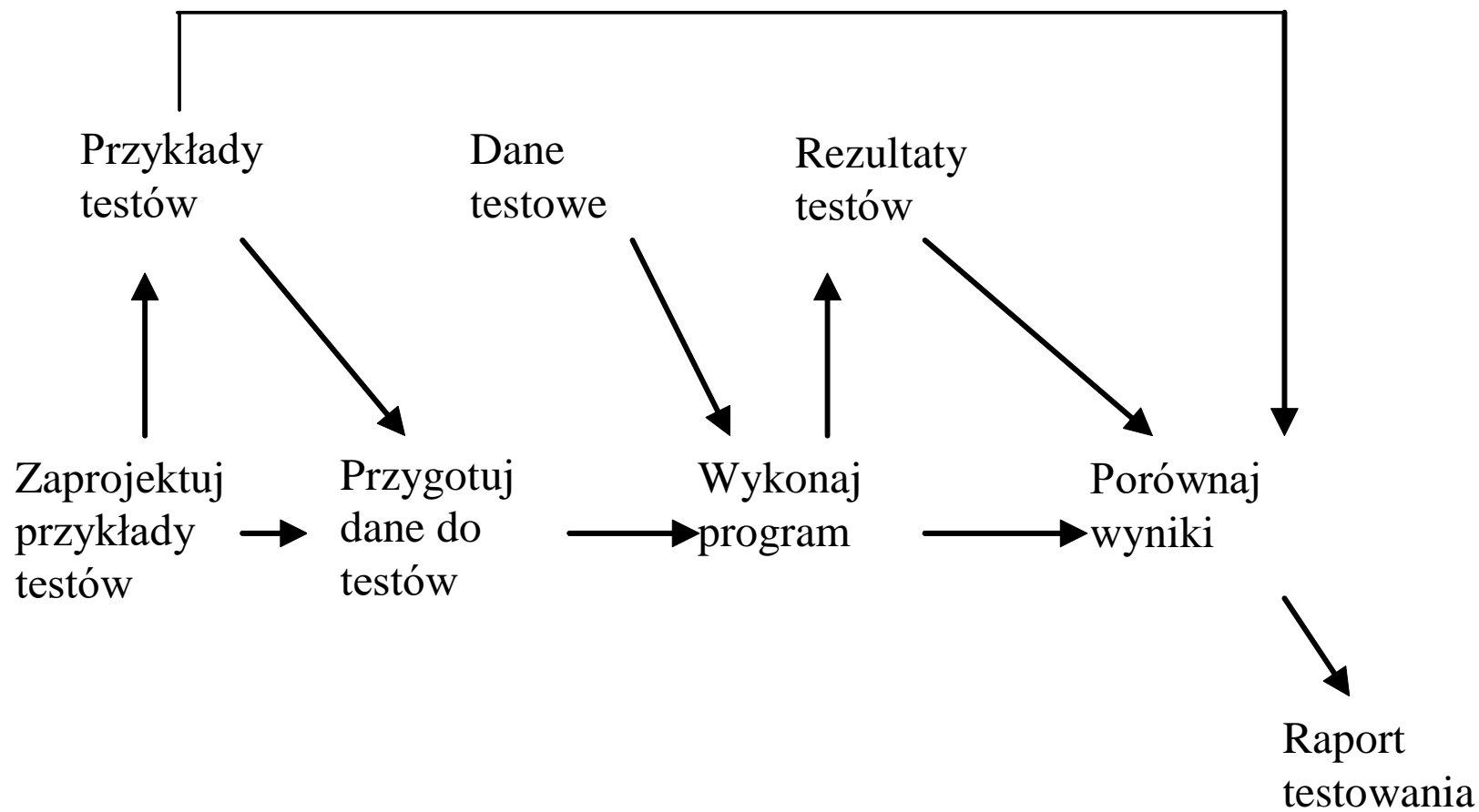
Fazy Testowania – model V



Cel testowania

- Celem „**defect testing**” jest ujawnienie defektów systemu.
- Celem **testowania walidacyjnego** jest pokazanie, że system spełnia specyfikację (testy akceptacyjne).
- **Dobry test** (defect test) to taki, który wykrywa błąd oprogramowania. Jeśli testy nie wykryły błędów nie znaczy to, że program jest poprawny, lecz że wykonano testy, które nie wykryły defektów.

Model testowania



Test case –przypadek testowy

- określenie funkcji testu,
- specyfikacja WE
- specyfikacja WY

Test case'y należy zaprojektować

- Testowanie **wyczerpujące** (exhaustive)

Wykonanie każdej instrukcji w programie, przejście przez każdą możliwą ścieżkę – **praktycznie niemożliwe**.

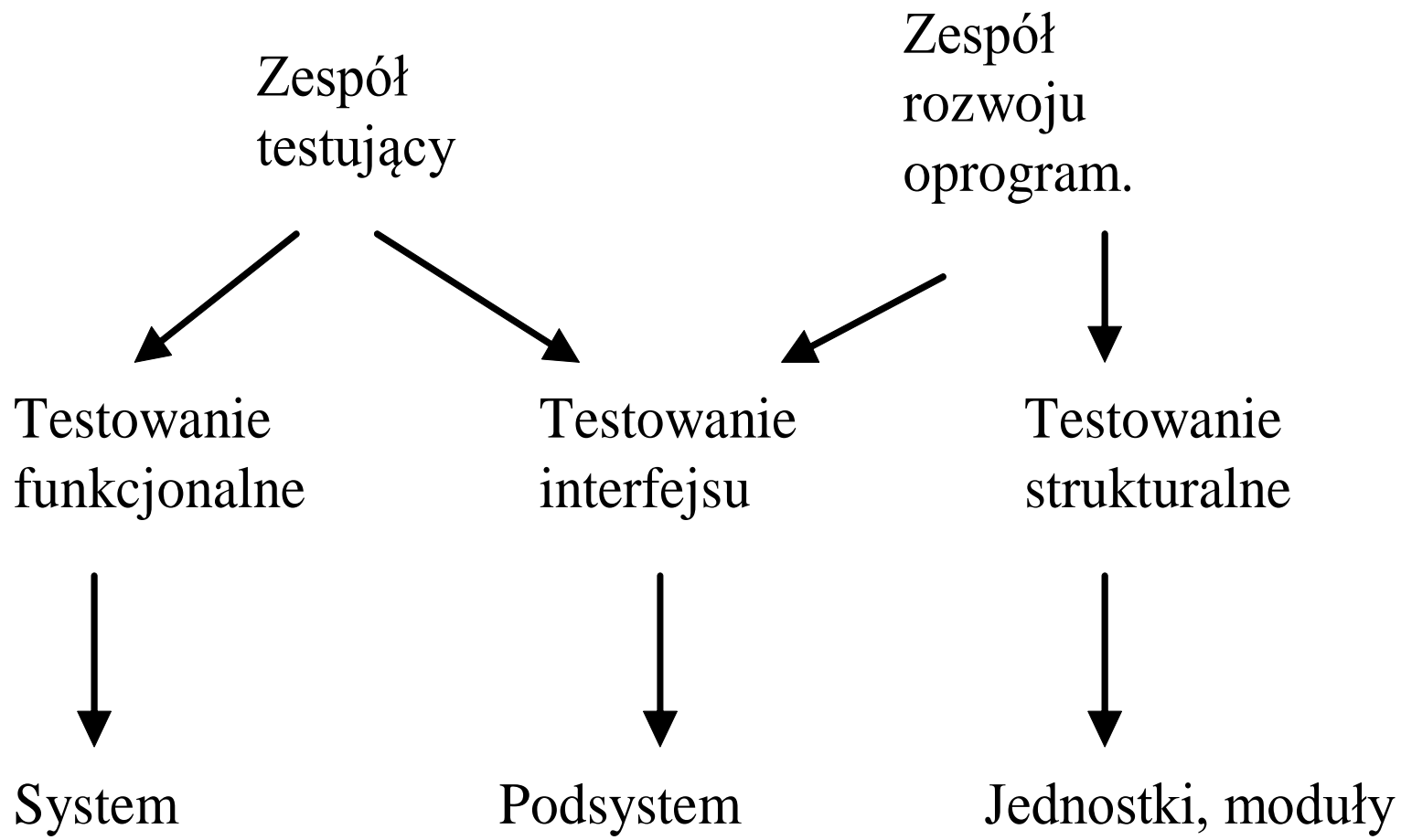
- Testowanie powinno się opierać o podzbiór możliwych przypadków.

Wskazówki – Petchenik

- Testowanie **możliwości systemu** jest ważniejsze od testowania jego komponentów. Przypadki powinny być tak dobrane by zidentyfikować błędy wstrzymujące, uniemożliwiające pracę użytkownika np. utraty danych
- Testowanie **starych możliwości** jest ważniejsze niż testowanie nowych, użytkownicy pracują wg starych nawyków.
- Testowanie **typowych sytuacji** jest ważniejsze niż sytuacji brzegowych

Podejścia do „defect testing”

- **Funkcjonalne (black box)** – testy wyprowadzone ze specyfikacji
- **Strukturalne (white box)** – testy wyprowadzone na podstawie znajomości struktury programu
- **Testowanie interfejsów** - testy wyprowadzone ze specyfikacji i na podstawie znajomości wewnętrznych interfejsów



Efektywność testowania

liczba wykrytych błędów / jedn.czasu

Basili & Selby (1987) eksperyment porównujący efektywność testowania „czarnych skrzynek” i testowania strukturalnego.

- Testowanie „czarnych skrzynek” okazało się efektywniejsze.

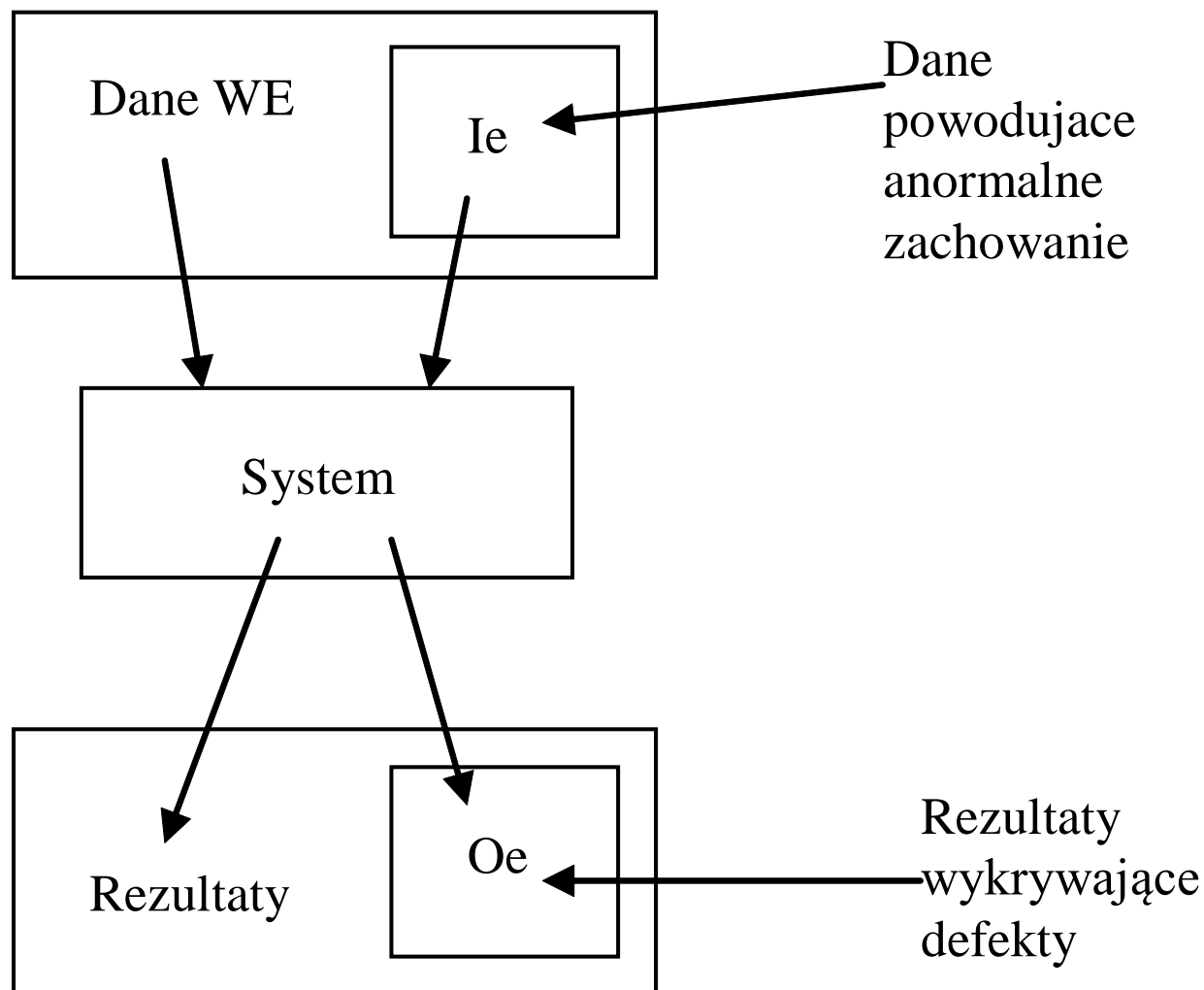
Badano także efektywność inspekcji kodu i testowania

- Statyczne inspekcja kodu okazała się tańsza i b. efektywna. Potwierdziły to eksperymenty Gilb’a & Graham’a 1993 i innych (1996)

Testowanie funkcjonalne

- Polega na wyprowadzeniu testów na podstawie specyfikacji systemu.
- System traktowany jest jako „**czarna skrzynka**”, której zachowanie może być określone na podstawie wejść i odpowiadających im wyjść.

Testowanie funkcjonalne



Podział na klasy równoważności

Equivalence partitioning (Bezier 1990)

Podział danych wejściowych na klasy, grupy o wspólnej charakterystyce. Program zachowuje się podobnie dla wszystkich elementów grupy.

Rezultaty programu też można podzielić na pewne grupy (podziały mogą się nakładać).

Cel - znalezienie takich podziałów

Wskazania:

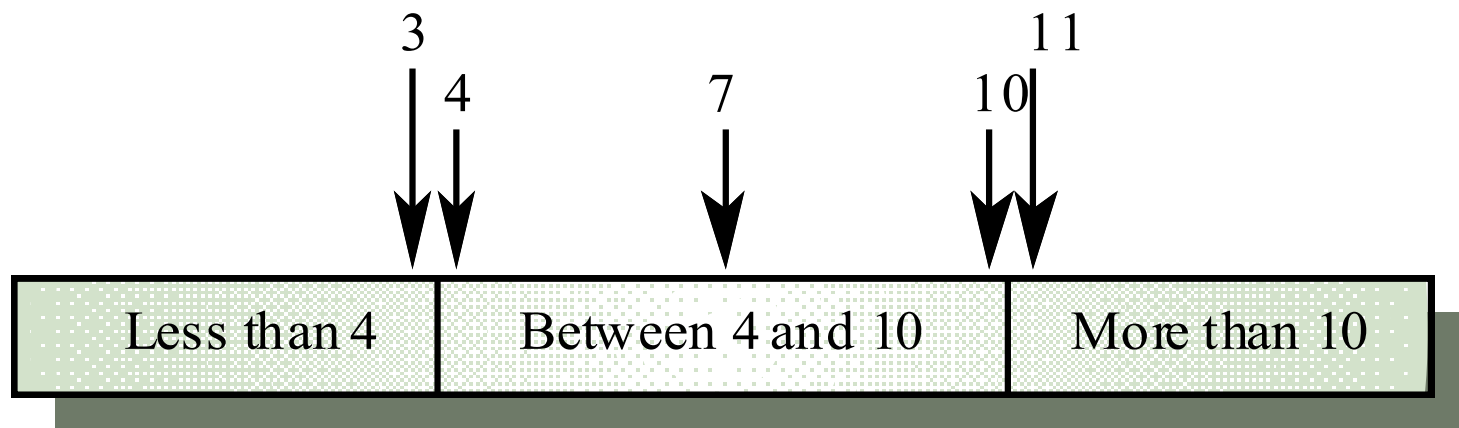
wybierać przykłady testów ze środka (typowe)
i z brzegów (nietypowe) grupy.

Testowanie warunków granicznych dziedziny danych

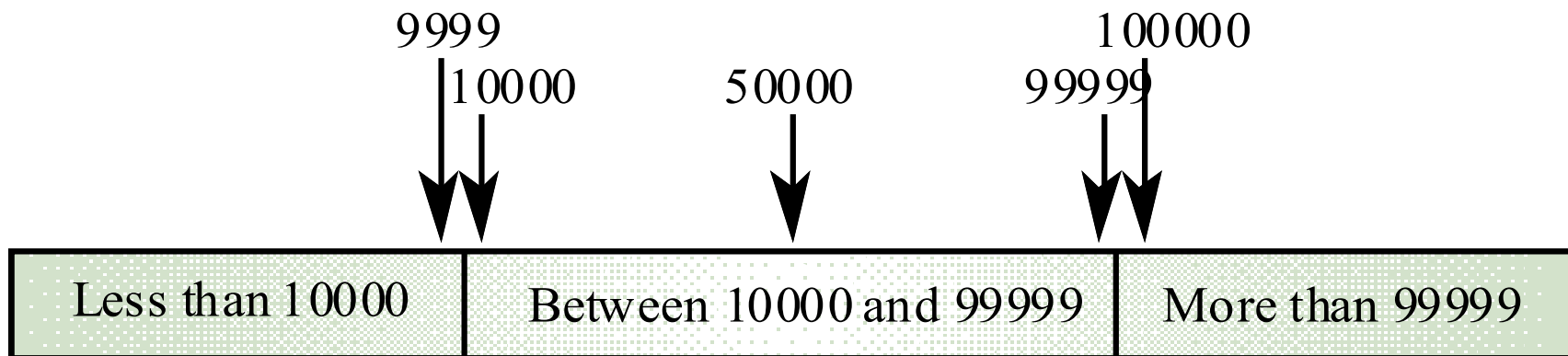
- element pierwszy, środkowy i ostatni,
- zbiór pusty, jedno-elementowy, wielo-elementowy, maksymalny
- element najbliższy i najdalszy,

Wartość domyślna, wartość pusta, spacja, zero, brak danych

Klasy równoważności - przykład



Number of input values



Input values

Przykład specyfikacji

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** LEM_INDEX) ;

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Podziały na bazie specyfikacji funkcji

Grupy podziału ze względu na tablicę *T*:

- tablica pusta (nie zawsze możliwe)
- tablica 1-elementowa,
- tablica wielo-elementowa

Grupy podziału ze względu na klucz *Key*:

- nie ma klucza w tablicy
- jest klucz w tablicy (w tym podgrupy):
 - jest pierwszym elementem tablicy
 - jest ostatnim elementem tablicy
 - jest wewnętrznym elementem tablicy

Testy na bazie specyfikacji funkcji

Przykładowe przypadki testowe wg kombinacji podziałów na grupy

| Input sequence (T) | Key (Key) | Output (Found, L) |
|----------------------------|-----------|-------------------|
| 17 | 17 | true, 1 |
| 17 | 0 | false, ?? |
| 17, 29, 21, 23 | 17 | true, 1 |
| 41, 18, 9, 31, 30, 16, 45 | 45 | true, 7 |
| 17, 18, 21, 23, 29, 41, 38 | 23 | true, 4 |
| 21, 23, 29, 33, 38 | 25 | false, ?? |

Testowanie strukturalne

- Osoba testująca może analizować kod, korzystać ze struktury komponentu do opracowania testu. Znalezienie algorytmu pozwala na znalezienie dalszych podziałów.
- Metody:
 - ☐ Pokrycia kodu
 - ☐ Pokrycia danych

Pokrycie przepływu sterowania

Pokrycie:

- instrukcji, linii kodu (line coverage)
co najmniej jednokrotne wykonanie każdej instrukcji dla danego testu (zestawu testów)
- warunków (rozejść decyzyjnych)
każdy elementarny warunek ma zostać co najmniej raz spełniony i co najmniej raz nie spełniony
- bloków, funkcji
- ścieżek

Wyznaczanie pokrycia

Narzędzia - **analizatory** pokrycia kodu:

- określają pokrycie dla testu
- sumaryczne pokrycie dla zbioru testów
- wskazują niepokryty kod.

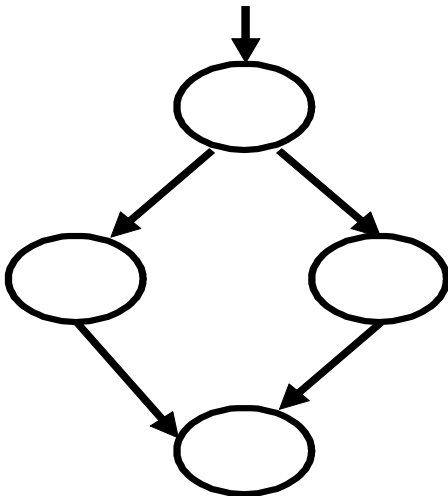
Testowanie **wyczerpujące** (exhaustive)

przejsię przez każdą możliwą ścieżkę wykonania programu – praktycznie niemożliwe.

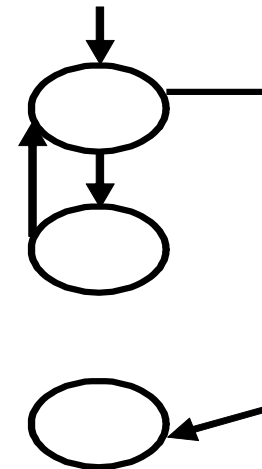
Graf przepływu sterowania

Węzły - decyzje, krawędzie - przepływ sterowania.
Można pominąć ciągi instrukcji sekwencyjnych.

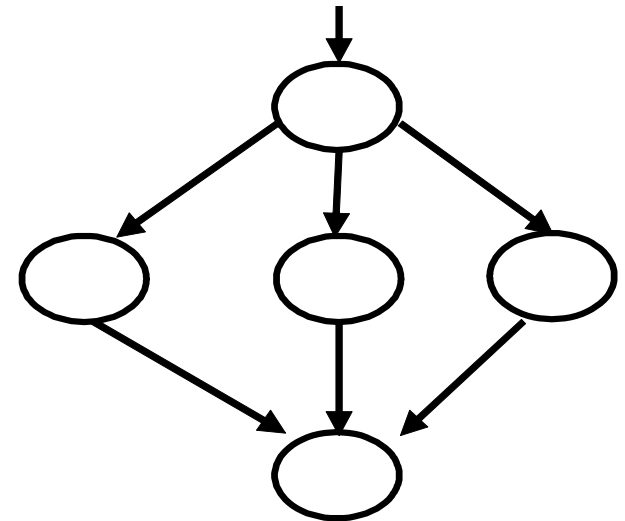
**if – then
– else**



loop – while



case - of



Testowanie ścieżek

Bazowy zbiór niezależnych ścieżek – **minimalny** zbiór ścieżek, których liniowa kombinacja generuje każdą możliwą ścieżkę w grafie. Niezależna ścieżka przechodzi przez co najmniej jedną nową krawędź grafu przepływu sterowania.

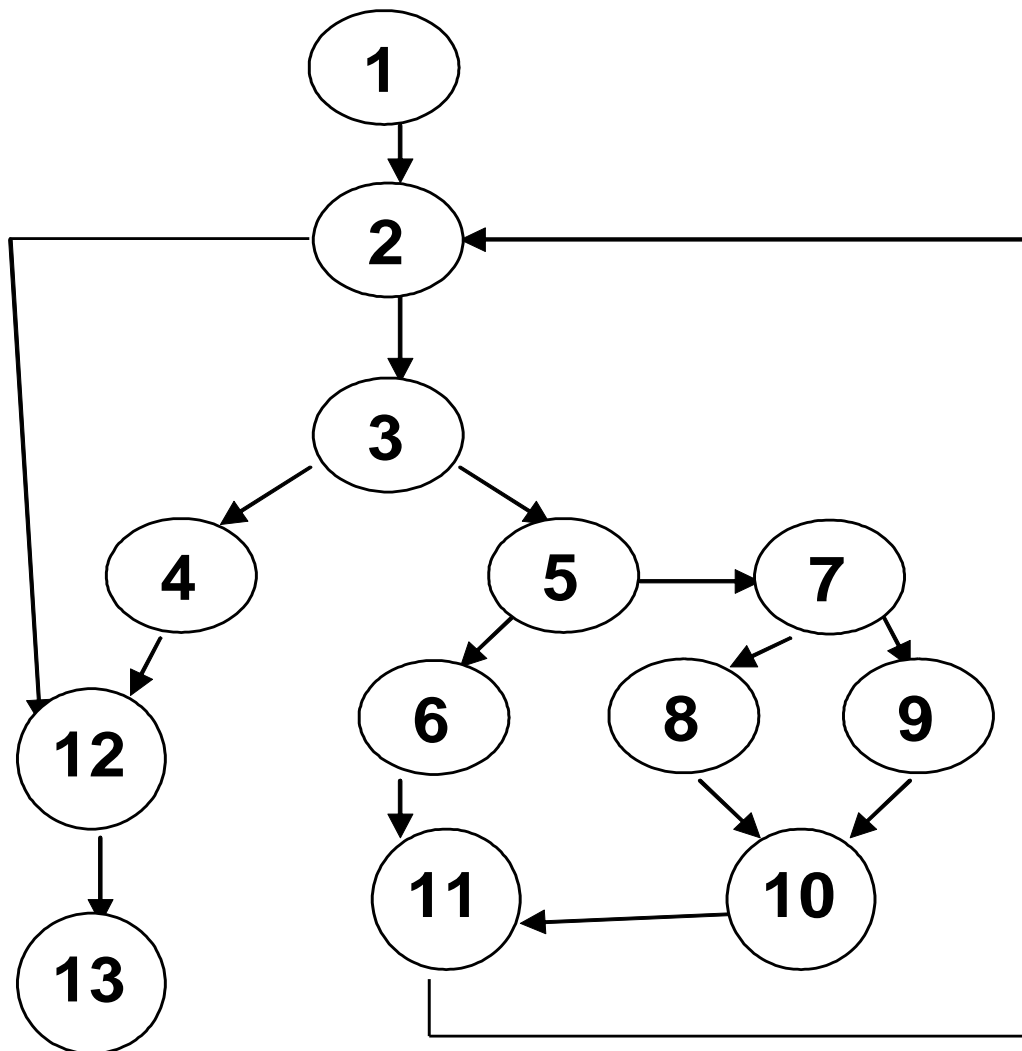
Liczba niezależnych ścieżek - złożoności cyklicznej (cyklomatycznej) Mc Cabe z grafu przepływu sterowania.

$CC(G) = \text{liczba krawędzi} - \text{liczba węzłów} + 2$

Liczba prostych warunków w programie **+1** (bez goto)

Zapewnia **pokrycie** instrukcji i warunków

Testowanie ścieżek - przykład



Np.


1) 1, 2, 12, 13

2) 1, 2, 3, 4, 12, 13

3) 1, 2, 3, 5, 6, 11,
2, 12, 13

4) 1, 2, 3, 5, 7, 8,
10, 11, 2, 12, 13

5) 1, 2, 3, 5, 7, 9,
10, 11, 2, 12, 13



Wykonanie wszystkich niezależnych ścieżek gwarantuje

- wykonanie co najmniej 1 każdej instrukcji
- wykonanie skoku warunkowego dla prawdy i fałszu

Liczba **niezależnych** ścieżek programu

- może być określona po obliczeniu **złożoności cykloatycznej McCabe** (1976) z grafu przepływu sterowania.

$$CC(G) = \text{l.krawędzi} - \text{l.węzłów} + 2$$

Testowanie niezależnych ścieżek

- Po obliczeniu liczby niezależnych ścieżek następnym krokiem jest **opracowanie przykładu testu wykonania każdej ścieżki**.
- Testowanie niezależnych ścieżek **nie testuje wszystkich możliwych kombinacji przejść w programie**. Defekty mogą się pojawić przy określonej kombinacji ścieżek.

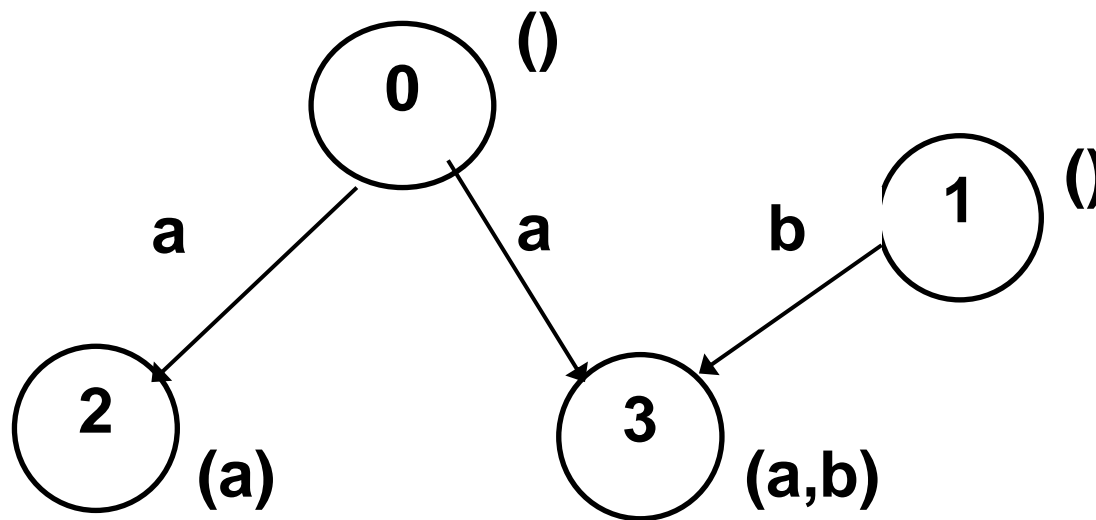
Pokrycie przepływu danych

Oparte na modelu przepływu danych:
sekwencja par węzłów pozostających w relacji
<definicja(X)_i , użycie(X)_j>

Możliwe jest takie wykonanie programu,
że pomiędzy **podstawieniem** wartości
zmiennnej X w i-tym węźle,
a jej **użyciem** w j-tym węźle
nie występuje żadne inne podstawienie zmiennej X.

Pokrycie przepływu danych - przykład

```
0)   int fun (int a)
      { int b;
1)       cin >> b;
2)       while (a) {
3)           b = b - a;
```



$\langle 0,2 \rangle, \langle 1,3 \rangle, \langle 0,3 \rangle$

Kryteria testowania pokryć danych

- wykonanie wszystkich par *<definicja-użycie>* dla wszystkich zmiennych

(all def uses paths coverage)

Np. **<0,2>**, **<1,3>**, **<0,3>**

- dla każdej definicji (przypisania) wykonanie co najmniej jednej pary

Np. **<0,3>**, **<1,3>**

- wykonanie wszystkich par typu:

<definicja, użycie zmiennej w wyrażeniu warunkowym>

(all-c-uses coverage)

Np. **<0,2>**

Testowanie z pokryciem przepływu danych

- Po określeniu zbioru par wybrać najkrótsze ścieżki do ich pokrycia.
- Podać testy dla tych ścieżek, jeśli możliwe do wykonania
- Ewentualnie szukać dłuższych ścieżek
- Lub automatycznie sprawdzać pokrycie par dla różnych testów bez wyznaczania ścieżek

Testowanie obiektowe

Poziomy

- testowanie operacji (funkcjonalne i strukturalne)
- testowanie obiektów
- testowanie zbiorów (gron) obiektów
- testowanie systemu obiektowego V&V – względem wymagań funkcjonalnych i нефunkcjonalnych

Testowanie obiektów

- Testowanie w izolacji wszystkich operacji
- Testowanie ciągów wykonań operacji danej klasy
- Ustawienie i użycie wszystkich atrybutów klasy
- Klasy równoważności operacji
np. inicjalizacja atrybutów, dostęp, modyfikacja
- Użycie obiektu we wszystkich możliwych stanach (ciągi zmian stanów)

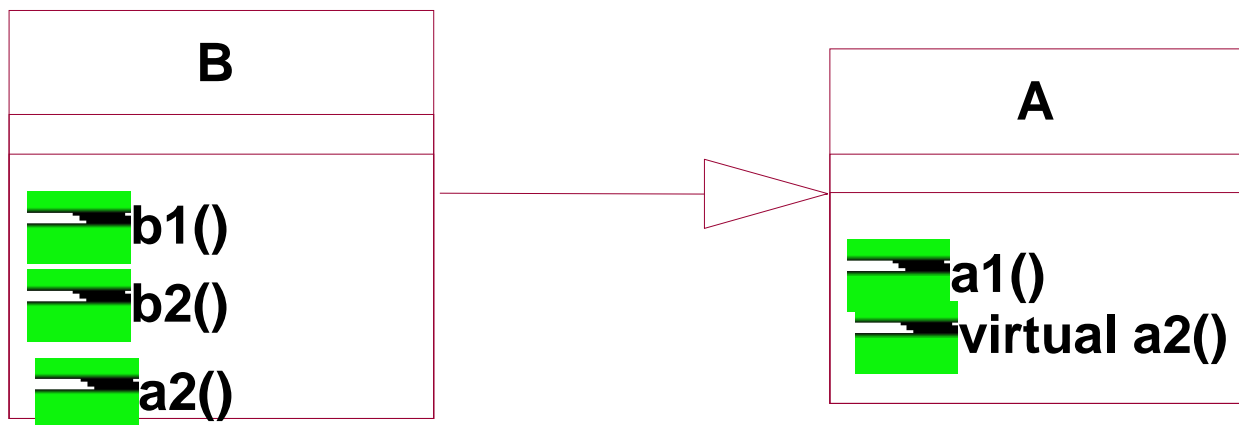
Zbiory obiektów zależnych

■ Testowanie hierarchii klas

- operacje odziedziczone,
- funkcje wirtualne dla obiektów klasy bazowej i potomnych, polimorfizm,
- dziedziczenie wielopoziomowe,
- operacje przy dziedziczeniu wielobazowym

■ Testowanie klas zaprzyjaźnionych

Testowanie klasy B - przykład



- Testowanie użycia atrybutów z klasy B i dostępnych atrybutów klasy A
- Testowanie strukturalne i funkcjonalne metod.
- Testowanie odziedziczonej metody `a1()` w kontekście obiektów klasy B.
- Testowanie metody `a2()` w kontekście obiektów klasy A i klasy B
- Testowanie użycia obiektów klasy B w możliwych stanach.

Integracja obiektów

Integracja obiektów - testowanie wstępujące lub zstępujące zwykle nieadekwatne

- **Testowanie w oparciu o opis użycia systemu**

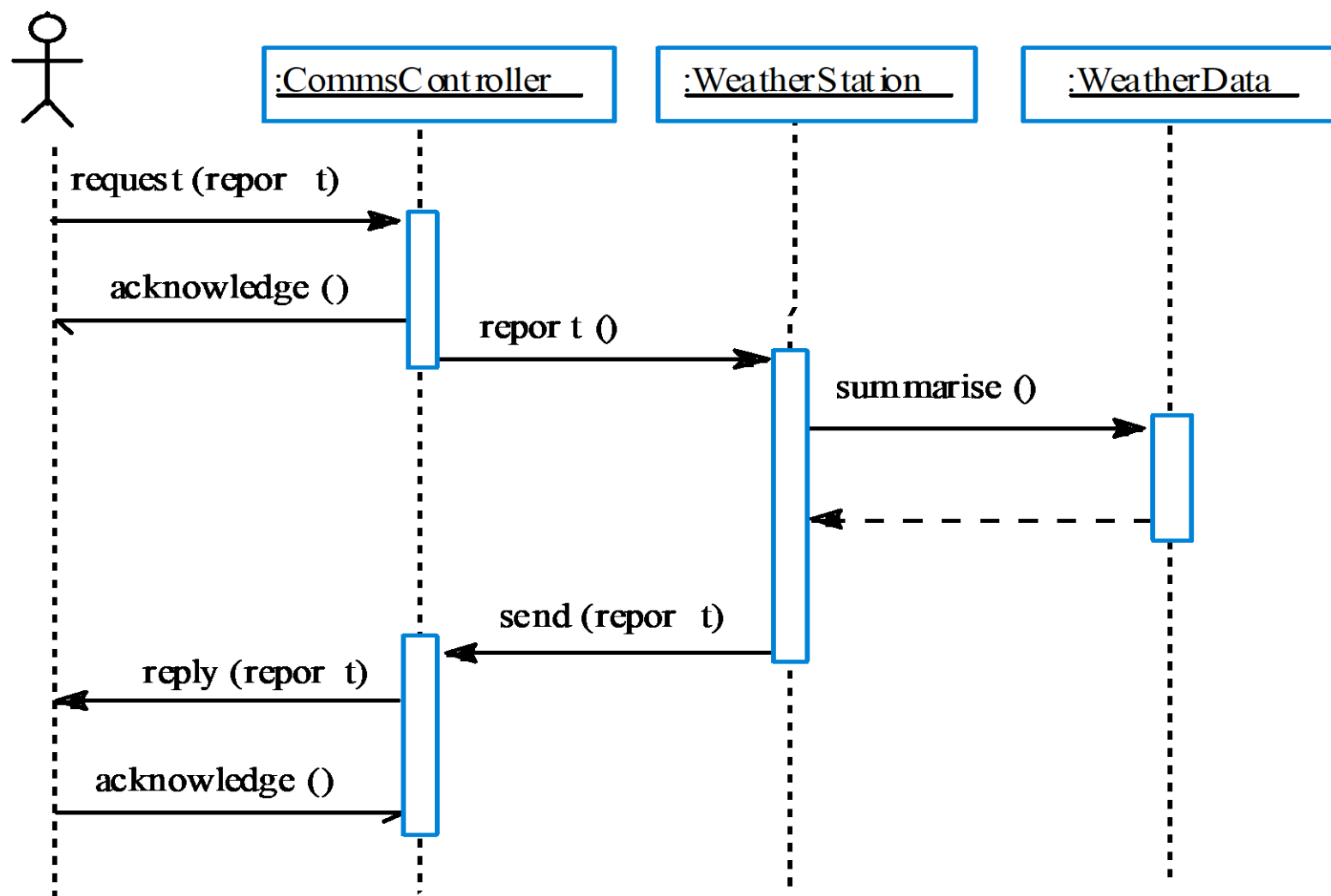
przypadki użycia (use case) i definiujące je diagramy sekwencji lub współpracy

- **Testowanie przepływów (wątków)**

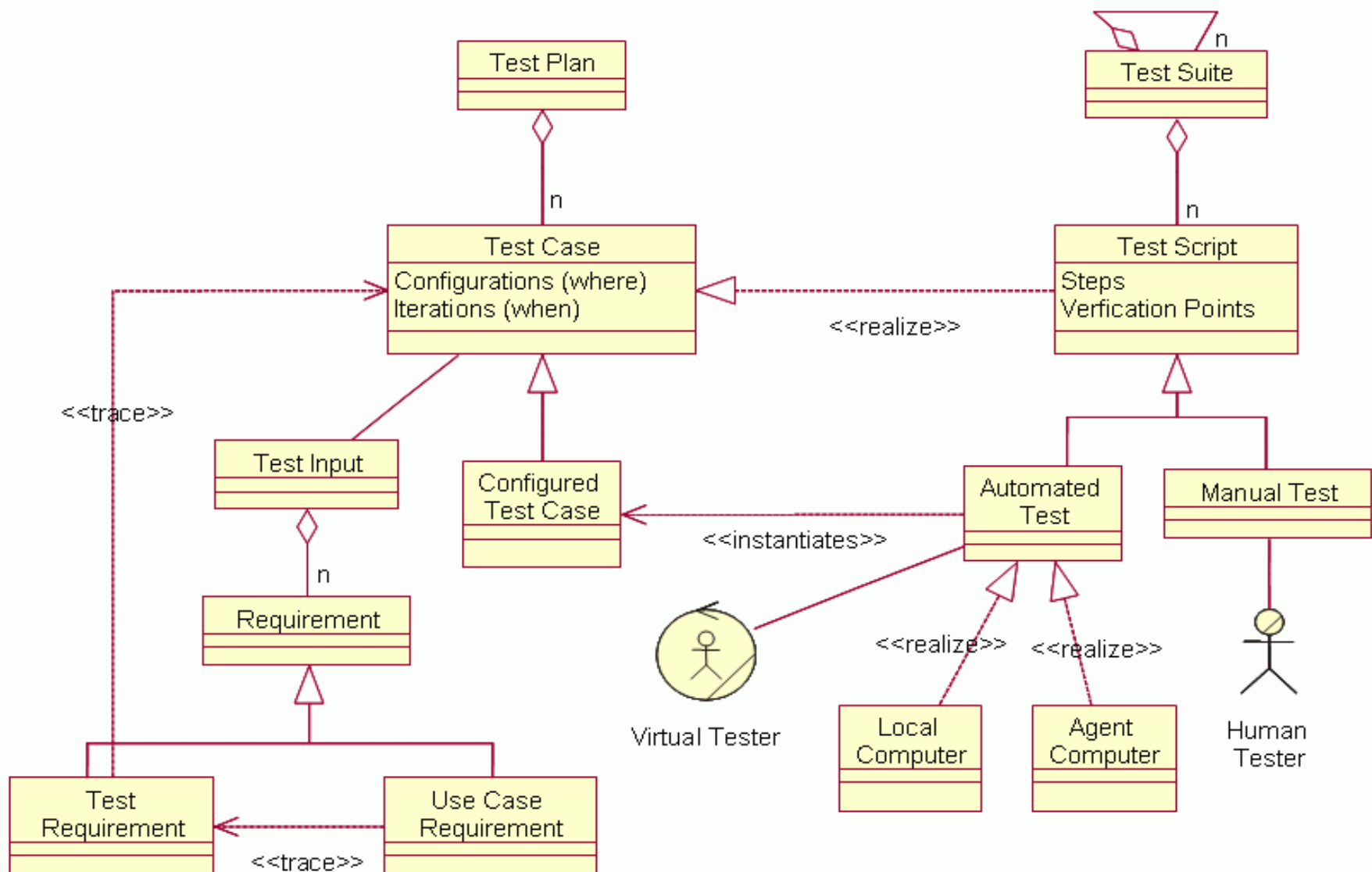
reakcje na zbiory zdarzeń wejściowych

Podstawą testów zbiorów klas i systemu są modele analityczne i projektowe

Generacja testów z diagramów sekwencji



Realizacja testów



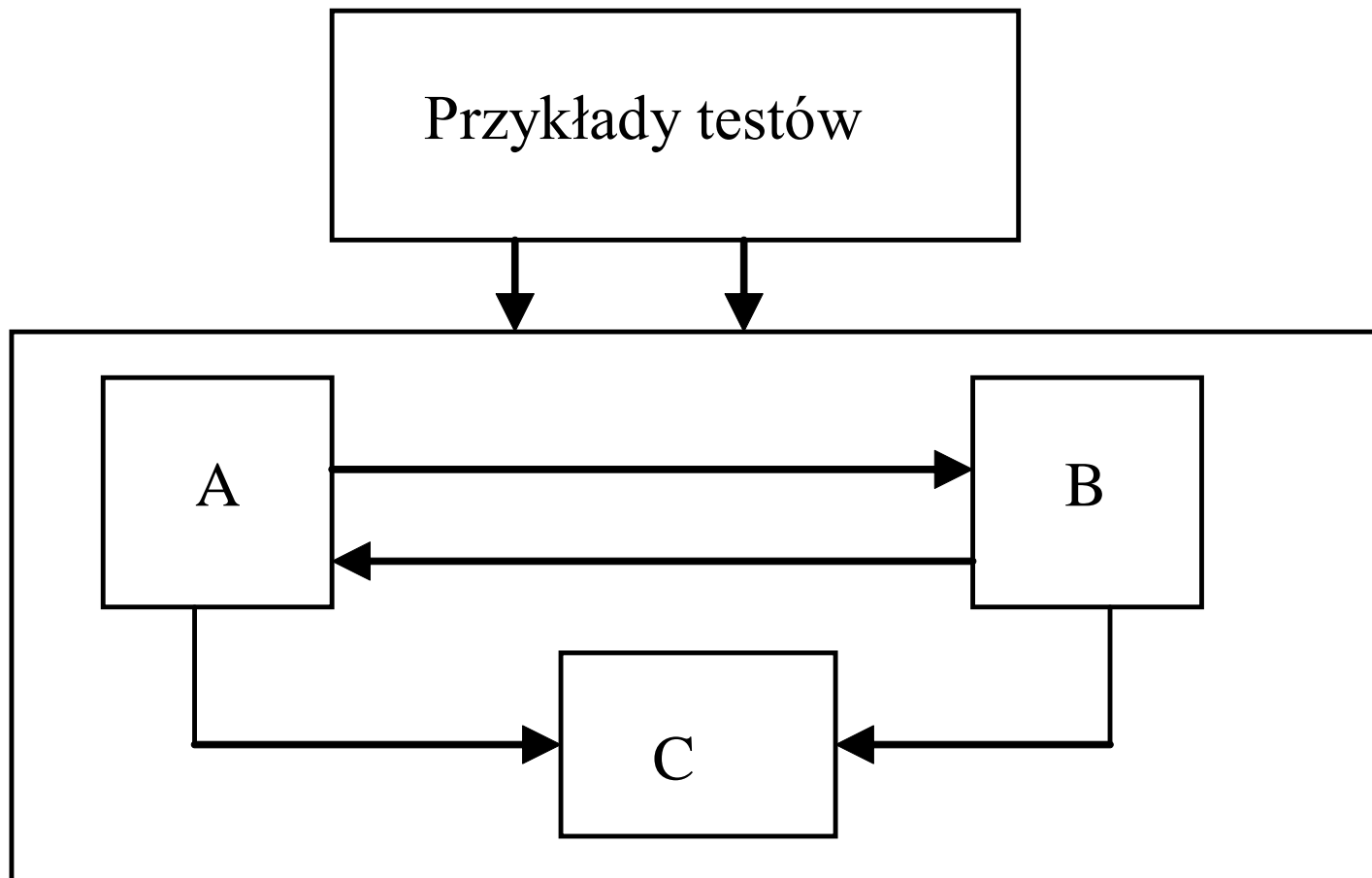
Narzędzia

- Do testowania jednostkowego np. JUnit, NUnit (w TDD test driven development)
- Do generacji skryptów testujących np. z modeli
- Narzędzia przechwytyjąco/odtworzące (testowanie GUI)
- Do organizacji procesu testowania (bazy skryptów testujących, danych wejściowych i wyroczeni)
- Do symulacji obciążenia

Testowanie interfejsów

- Stosowane przy integracji modułów, podsystemów.
- Testowanie ma wykryć błędy w interfejsie lub w założeniach dotyczących interfejsu.
- Ważne dla systemów obiektowych, szczególnie, gdy klasy są „reuse”.

Przykłady testów



Typy interfejsów

- **parametryczne** (przekazywane wskazania na dane, funkcje)
- **wspólna pamięć** (shared memory)
- **proceduralne** (podsystem zawiera zbiór procedur i ich wywołania mogą mieć miejsce z innych podsystemów)
- **przekazywanie komunikatów** (message passing) (podsystem wysyła komunikat żądający usługi innego podsystemu, zwrotny komunikat zawiera rezultat usługi – systemy klient-serwer)

Klasy błędów interfejsów

- **Użycia** – szczególnie częsty w interfejsach parametrycznych (błędny typ parametru, kolejność, liczba parametrów)
- **Błędneho zrozumienia** – komponent wywołujący błędnie zakłada, jakie ma być zachowanie komponentu wywoływanego (np. że wektor ma być uporządkowany a nie jest)
- **Błędy czasowe** – systemy czasu rzeczywistego, komunikacja poprzez wspólną pamięć lub przekazywanie komunikatów np. Producent i konsument danych pracują z różnymi prędkościami. Konsument może dostać „stare” dane. Dobrze zaprojektowany interfejs powinien uniemożliwić takie sytuacje. Testowanie jest trudne, błędy ujawniają się w pewnych, często nieoczekiwanych sytuacjach.

Wskazania:

- Zaprojektuj test, gdzie **wartości parametrów są ekstremalne** w swoich zakresach
- Sprawdź interfejs ze **wskazaniami null**
- Interfejs proceduralny – zaprojektuj test, który spowoduje błąd komponentu
- Systemy z przekazywaniem komunikatów – **testowanie stresujące**. Generowanie większej liczby komunikatów niż zakładano może ujawnić problemy czasowe.
- Komunikacja poprzez **wspólną pamięć** – testy powinny zawierać **różną kolejność dostępu** do pamięci (mogą ujawnić błędy spowodowane założeniem określonej kolejności)