Sprawdzanie poprawności oprogramowania
Weryfikacja i walidacja.
Testy statyczne i dynamiczne.
Pomiary poprawności oprogramowania.
Testowanie automatyczne.
Systemy odporne na błędy

Wykładowca
Dr inż. Zofia Kruczkiewicz

Literatura

1. D. Harel, Rzecz o istocie informatyki. Algorytmika, WNT 1992
2. I. Sommerville, Inżynieria oprogramowania, s. Klasyka informatyki, WNT 2003
3. Roger S. Pressman, Praktyczne podejście do oprogramowania, s. Inżynieria oprogramowania, WNT, 2004
4. J. Górski i inni, Inżynieria oprogramowania w projekcie informatycznym, MIKOM 1999
5. Stephen H. Kan, Metryki i modele w inżynierii jakości oprogramowania, MIKOM, 2006
6. K. Frączkowski, Zarządzanie projektem informatycznym. Projekty w środowisku wirtualnym. Czynniki sukcesu i niepowodzeń projektów, Oficyna Wydawnicza Politechniki Wrocławskiej
7. J.M.Brady, Informatyka teoretyczna w ujęciu programistycznym, WNT 1983
8. Mourad Badri, Fadel Toure, Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes, Journal of Software Engineering and Applications, 2012, 5, 513-526
9. M. Bruntink, A. van Deursen, An Empirical Study into Class Testability, Journal of Systems and Software , 01/2006
10. A. H. Watson,T. J. McCabe, Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metrics, http://www.mccabe.com/pdf/mccabe-nist235r.pdf

Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne
7. Plan testowania błędów zakresu produktu
8. Ocena niezawodności programu – testowalność
9. Ocena wykrywalności błędów
10. Koszt wykrywalności błędów

Struktura wykładu

1. **Wstęp**

Wstęp

- **Problem stopu programu**
- **Rola testowania w tworzeniu oprogramowania**
- **Podstawowe definicje**

Problem stopu programu [1]

1. Przykład programu, **który *zatrzymuje się* dla liczb naturalnych nieparzystych, natomiast nie zatrzymuje się dla liczb parzystych**.

dopóki X≠1 dopóty wykonuj
X ← X-2
zatrzymaj się

2. Przykład programu, **który się *zawsze zatrzymuje* dla dowolnych liczb naturalnych, ale *nie można tego formalnie udowodnić*.**
   Oznacza to brak możliwości pełnej automatyzacji testowania.

dopóki X ≠ 1, dopóty wykonuj
jeśli X jest parzyste,
wykonuj X ←X/2
w przeciwnym przypadku (X nieparzyste)
wykonaj X←3*X +1
zatrzymaj się

Np. dla x=1005 liczba pętli: 67

Rola testowania w tworzeniu oprogramowania [1-4]

1. **Testowanie** – kluczowa rola w powstawaniu oprogramowania
   - proces usuwania błędów w kolejnych fazach rozwoju oprogramowania

2. **Różne metody testowania** dostosowane do stopnia rozwoju oprogramowania.

3. W inżynierii oprogramowania poszukuje się **związku między strukturą programu, a**:
   - możliwością powstawania pewnych błędów
   - trudnością ich wykrywania na drodze testowania.

Podstawowe definicje

- ***Atestowanie, walidacja*** (validation) - testowanie zgodności produktu z rzeczywistymi potrzebami użytkownika (**czy zbudowano poprawny produkt**).
- ***Weryfikacja*** (verification) - testowanie zgodności produktu 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 produkcie, **która może, ale nie musi**, prowadzić do niewłaściwego działania.
- ***Błędne wykonanie*** - **uszkodzenie** (failure) to niepoprawne działanie produktu w trakcie jego pracy na skutek błędów.
  - Takie same błędne wykonanie może pochodzić od różnych błędów.

Struktura wykładu

1. **Wstęp**
2. **Klasyfikacje błędów i testów**

Klasyfikacje błędów i testów

- **Klasyfikacja błędów**
- **Klasyfikacja testów ze względu na:**
  - **cel**
  - **technikę wykonania**
  - **zakres**
  - **technikę projektowania testu**

Klasyfikacja błędów

- **błędy wymagań i analizy**: złe sformułowanie problemu, zaniedbanie istotnych parametrów, niewłaściwy algorytm,
- **błędy projektowania:** błędna interpretacja wymagań, błędy logiczne
- **błędy programowe:**
  - **błędy opracowania** szczegółowej struktury programu: zła interpretacja wymagań dla programu, niepełność struktury programu, nie uwzględnienie przypadków szczególnych, niedostateczne dopracowanie błędów, zlekceważenie warunków czasowych
  - **błędy kodowania:**
    - **syntaktyczne**, zazwyczaj rozpoznawane przez kompilator,
    - **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),
  - **błędy kompilacji i konsolidacji:** błędy kompilatora, błędy w zakresach nazw itp.)

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

1.3. testy odporności

– zachowanie systemu pod wpływem braku zasobów (zanik zasilania, awarie sprzętu)

– podaniu niepoprawnych danych i poleceń

1.4. testy wydajności – czas działania funkcji

1.5. testy skalowalności – zachowanie programu pod wpływem dużej liczby przetwarzanych danych, dużej liczby użytkowników itp. – **zbadanie wydajności i niezawodności**

1.6. testy funkcjonalności interfejsu graficznego użytkownika - funkcjonalność formularzy, ergonomia

1.7. testy regresji

Celem testów regresyjnych jest sprawdzenie, że program działa po modyfikacji, usunięciu błędów lub po dodaniu nowej funkcjonalności.

Wykonanie testów regresyjnych opiera się na powtórzeniu dotychczasowego zestawu testów, które wcześniej kończyły się poprawnie.

Klasyfikacja testów cd

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:

2.1.1. testy funkcjonalne: Program traktowany jest jak „czarna skrzynka”. Znane są jedynie wymagania wobec testowanych funkcji programu. Testuje się program w wybranych podzakresach danych, traktując je jako klasy danych wejściowych – testy dla każdej klasy przeprowadza się jedynie dla pewnych wybranych danych w kilku przebiegach, a wnioskuje się o działaniu programu dla całej klasy danych.

2.1.2. testy niefunkcjonalne

odporności
wydajności
skalowalności
funkcjonalności interfejsu graficznego użytkownika

2.1.3. testy strukturalne (metaprogramy): Struktura programu jest znana. Dane wejściowe należy dobrać tak, aby każda instrukcja programu była przynajmniej raz wykonana, oraz tak, aby każda instrukcja warunkowa i pętle były przynajmniej raz wykonane i raz nie wykonane (***kryterium pokrycia instrukcji warunkowych***).

2.2. testy statyczne:

inspekcje: wymagań, struktury produktu,
udowadnianie poprawności programu (np. logika Hoare**),**
testowanie symboliczne (testowanie oparte na strukturze programu i analizowaniu stanu danych w wyniku wykonania programu dla różnych przebiegów sterowania programem (wykonanie lub nie wykonanie instrukcji warunkowych i pętli podczas przejścia przez program)

Testy dynamiczne i statyczne mogą służyć do wykrywania różnych błędów.

Klasyfikacja testów cd

2. Ze względu na technikę wykonania cd:

2.3. testy automatyczne

Realizowane przez programy narzędziowe – automatycznie uruchamiany jest testowany fragment programu i wynik jego działania jest porównywany z wynikami wzorcowymi:

- obiektywna ocena wyniku testowania
- możliwość odtworzenia testu po poprawie kodu
- możliwość podania wielu danych testowych
- muszą być zaprojektowane, utrzymywane i interpretowane przez człowieka

2.4. testy ręczne:

Realizowane przez człowieka są ważne wtedy, gdy program testujący ma trudności z przetwarzaniem i interpretowaniem pewnych informacji, które są naturalne dla człowieka (np **testy statyczne**)

Mogą być obarczone błędami wynikającymi z czynnika ludzkiego (zmęczenie, tendencja do popełniania omyłek, złożone obliczenia)

Klasyfikacja testów cd

3. Ze względu na zakres: wyróżnia się następujące testy (związane z cyklem życia produktu):

3.1. testy jednostkowe – testy pojedynczych elementów programu (funkcji z modułu, metod klasy) – porównanie wyniku z wynikiem wzorcowym (pozytywnym i negatywnym)

3.2. testy integracyjne (testowanie zbioru klas jako komponentów w celu wykrycia:

- Niekompatybilności ich interfejsów,
- Niezgodności cyklu życia,
- Niezgodna interpretacja wymienianych danych

Klasyfikacja testów cd

3.3. testy systemu – testy zintegrowanych komponentów w środowisku zbliżonym do docelowego w celu sprawdzenia:

- Niezgodności interfejsów komponentów
- Błędów logicznych w łączeniu komponentów
- Błędów synchronizacji w systemach:
  - czasu rzeczywistego,
  - opartych na przekazywaniu komunikatów
  - brak odświeżania danych we współdzielonych interfejsach
  - różne cykle życia komponentów

3.4. testy akceptacji (testy alfa i beta) – test z punktu widzenia potrzeb klienta

Kolejność wykonania testów w procesie powstawiania oprogramowania jest zależna od przyjętej metody testowania i tworzenia oprogramowania

Klasyfikacja testów cd

4. Ze względu na technikę projektowania testu

4.1. Metoda „białej skrzynki” – oparta na strukturze logicznej testowanego metody, funkcji itd czyli fragmentu oprogramowania

4.2. Metoda „czarnej skrzynki” – oparta na wymaganiach funkcjonalnych oprogramowania

Zofia Kruczkiewicz – Wyklad_INP002017_2

Struktura wykładu

1. **Wstęp**
2. **Klasyfikacje błędów i testów**
3. **Problemy testowania i lokalizacji błędów**

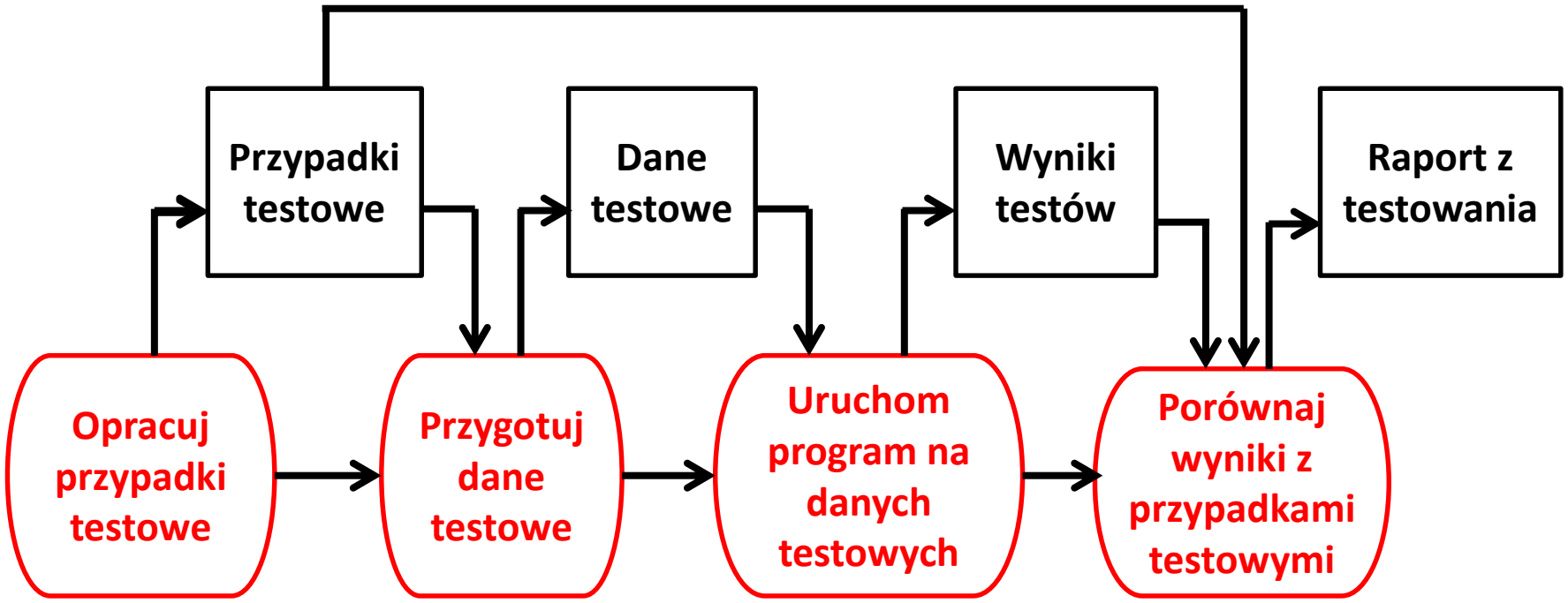
Problemy testowania i lokalizacji błędów

1) **trudność w określeniu możliwie najmniejszej liczby zachowań programu**, wynikającego z pewnego zbioru danych, które należy sprawdzić i uogólnić indukcyjnie uzyskane wyniki
2) **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)
3) **wykrywanie i lokalizacja błędów** – jest skorelowana z jakością oprogramowania, określoną metrykami złożoności modułowej i międzymodułowej kodu programu (wykład 11, [8], [9], [10]):
   - **Fan out,**
   - **LCOM3,**
   - **LOC na klasę,**
   - **NOF (liczba pól),**
   - **NOM (liczba metod),**
   - **RFC,**
   - **WMC,**
   - **McCabe (liczba cyklomatyczna)**

Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów [2]

Proces testowania błędów



Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu [4, 1]

Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu

Faza cyklu życia	Cel weryfikacji	Techniki
Specyfikacja wymagań funkcjonalnych i niefunkcjonalnych	realizowalność, sensowność kompletność, spójność, poprawność	inspekcje wymagań
Tworzenie modelu analizy	poprawność, kompletność, spójność, zgodność z wymaganiami użytkownika i systemu	inspekcje specyfikacji modelu, **symulacje,** **testy funkcjonalne w metajęzyku**
Projektowanie	zgodność z modelem analizy	inspekcje specyfikacji projektu, dowody poprawności, **symulacje,** **testy funkcjonalne w metajęzyku**
Programowanie (kod)	struktura programu komentarze	inspekcje specyfikacji kodu dowody poprawności testy symboliczne **symulacje,** **testy strukturalne** **testy funkcjonalne – metaprogramy**

Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu cd

Faza cyklu życia		Cel weryfikacji	Techniki
Testowanie kodu ze względu na zakres	**Testowanie jednostkowe**	funkcje jednostki (klasy, modułu)	**testy strukturalne**
	Testowanie integracyjne	połączenia klas lub/i modułów = komponenty (kompatybilność interfejsów) sterowanie (cykl życia) przepływ danych (poprawna interpretacja)	testy statyczne (symboliczne) **testy strukturalne ,** **testy funkcjonalne - metaprogramy** **testy regresji**
	Testowanie systemowe	funkcjonalność zintegrowanych części zgodna z wymaganiami funkcjonalnymi i niefunkcjonalnymi	**testy funkcjonalne – metaprogramy** **testy regresji**
	Testowanie akceptacyjne	funkcjonalność zgodna z wymaganiami funkcjonalnymi	**testy funkcjonalne - metaprogramy** **testy regresji**
Konserwacja		Poprawki	**testy funkcjonalne - metaprogramy** **testy regresji**

Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne [7]

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

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

Przykład 2: Testowanie błędnej wersji programu do znajdowania pierwiastka kwadratowego ans z *p*, gdy przedział $0 \leq p < 1$ z dokładnością do *err*, gdzie $0 \leq err < 1$: $p^{1/2} - err \leq ans \leq p^{1/2} + err$

```
#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<1 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);
}
```

Sekwencje programu	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 >=0.25	err < 1	0.5	0.5	2*p - 0.5 >= 0	4*p - 3
B true, exit	**0.25<=p<1**	**0.5<=err<1**	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 >=0.25	err < 1	0.5	0.5	2*p – 0.5 >= 0	4*p - 3
B false ?	p >=0.25	err < 0.5	0.5	0.5	2*p – 0.5 >= 0	4*p – 3
C false !	0.25<=p<1	err < 0.5	0.25	0.5	4*p – 4.25 < 0	8*p - 6
B true exit	**0.25<=p<1**	**0.25<= err<0.5**	0.25	**0.5**	4*p – 4.25 < 0	8*p - 6

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

- **ans = 0.5**
- **$p^{1/2}$ - err ≤ ans ≤ $p^{1/2}$ + err –zgodnie z założeniem**
- **p = err = 0.995, $p^{1/2} \approx 0.997$ – dane wynikające z programu**
  - **$p^{1/2}$ – err = 0.997 – 0.995 = 0.002**
  - **$p^{1/2}$ + err = 0.997 + 0.997 = 1.994**

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

- **ans = 0.5**
- **ans < $p^{1/2}$ - err** (powinno być **ans ≥ $p^{1/2}$ - err**) **niezgodnie z założeniem**
  p = 0.995, err = 0.49 , $p^{1/2} \approx 0.997$ – dane wynikające z programu
  $p^{1/2}$ – err = 0.997 – 0.49 = 0.507

Wniosek

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

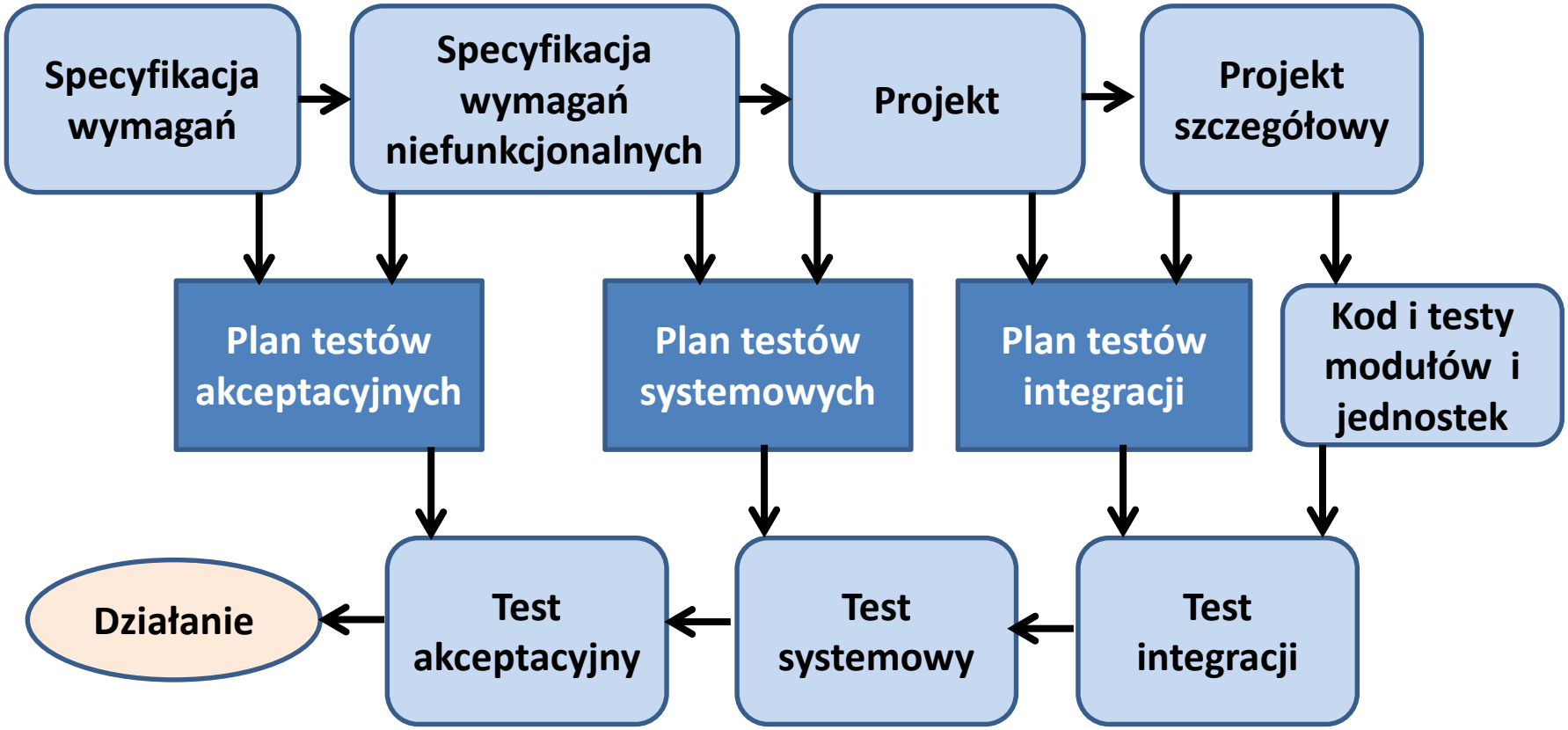
Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne
7. Plan testowania błędów zakresu produktu

Plan testowania błędów zakresu produktu [2]

- testy jednostkowe
- testy integracyjne
- testy systemu
- testy akceptacji (testy alfa i beta)

Plan testowania zakresu w procesie tworzenia oprogramowania

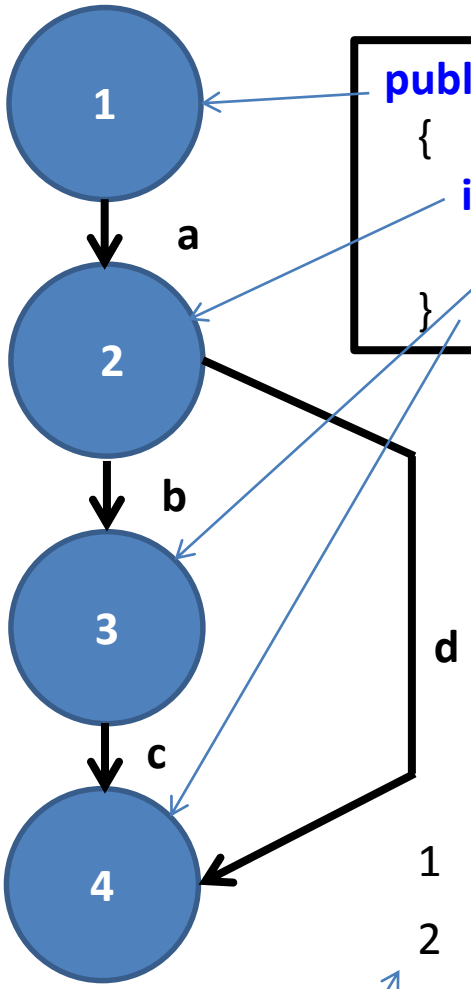


Testy jednostkowe - automatyczne [3]

- Test jednostkowy
  - Projekt testu
  - Realizacja testu
- Test jednostkowy z użyciem obiektów typu Mock
- Adnotacje
- Metody wspomagające testowanie jednostkowe
- Przykład środowiska wspierającego tworzenie i realizację testów jednostkowych

Graf przepływu, macierz grafu, macierz połączeń

dla metody addTytul_ksiazki i projekt testu jednostkowego – pokrycie wierzchołków testami




```
public void addTytul_ksiazki(Tytul_ksiazki tytul_ksiazki)
  {
    if (! mTytul_ksiazki.contains(tytul_ksiazki))
        mTytul_ksiazki.add(tytul_ksiazki);
  }
```

Macierz grafu

	1	2	3	4
1		a		
2			b	d
3				c
4				

Macierz połączeń

	1	2	3	4	
1		1			1-1=0
2			1	1	2-1=1
3				1	1-1=0
4					

Liczba MCabe (cyklomatyczna)
MC = 1+1 =2
wyznacza liczbę testów:
Test1: 1-2-3-4
Test2: 1-2-4

Test jednostkowy (test dynamiczny strukturalny), metoda „białej skrzynki”

```
@Test
public void testAddtytul_ksiazki() {
    System.out.println("addtytul_ksiazki");
    String[] dane_tytul = {"tytul1", "Jan", "Kowalski", "12345", "W1"};
    Tytul_ksiazki val = Tytul(dane_tytul);
    Fasada instance = new Fasada();
    instance.addtytul_ksiazki(val);
    Tytul_ksiazki result = instance.getTytuly_ksiazek().get(0);
    assertEquals(val, result);
    instance.addtytul_ksiazki(val);
    int result1 = instance.getTytuly_ksiazek().size();
    assertTrue("Blad", 1 == result1);
    // TODO review the generated test code and remove the default call t
    e is a prototype.");
}
```

a, b

a, d

Sprawdzenie, czy po próbie dodania tego samego tytułu liczba tytułów nie zmieni się

Porównanie dodanego tytułu pobranego z aplikacji z tytułem wzorcowym

Testy jednostkowe - rola obiektów typu Mock

http://www.oracle.com/technetwork/articles/entarch/mock-shortcomings-082129.html


```
@Before
public void setUp() {
      mockEmployeeDAO = createMock(EmployeeDAO.class);
      employeeBO = new EmployeeBO(mockEmployeeDAO);
      employee = new Employee("Alex", "CA", "US");
}

@Test
public void shouldAddNewEmployee() {
      mockEmployeeDAO.insert(employee);
      replay(mockEmployeeDAO);
                  //powtórzenie stanu obiektu typu Mock: mockEmployeeDAO
      employeeBO.addNewEmployee(employee);
     verify(mockEmployeeDAO);
}
                  //porównanie komunikacji między EmployeeB0 i EmployeeDAO
                  // symulowanym przez mockEmployeeDAO
```

Adnotacje określające sposób i moment testowania

http://www.vogella.com/articles/JUnit/article.html

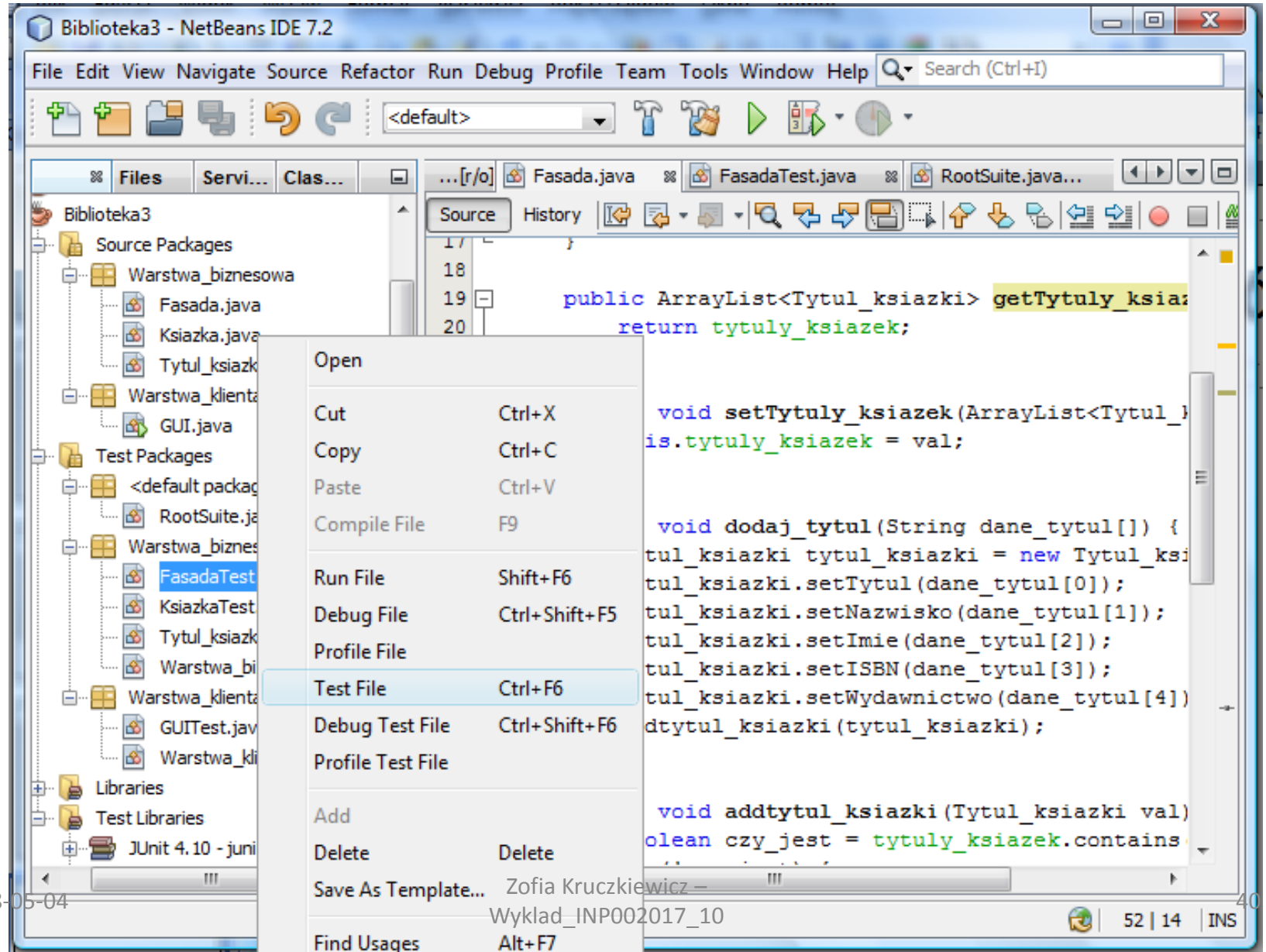
Annotation	Description
@Test public void method().	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

Metody wspomagające ocenę wyniku testu

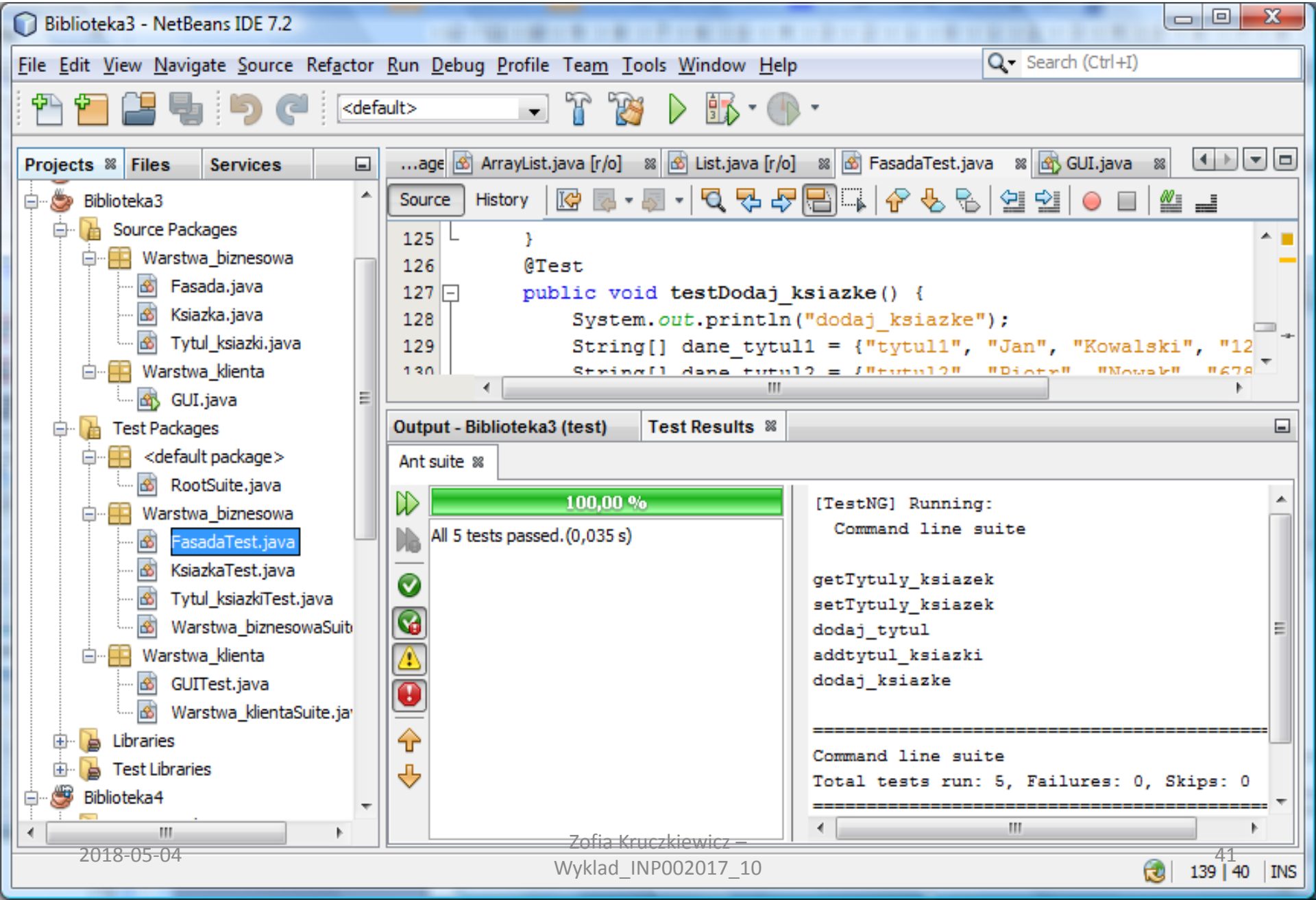
http://www.vogella.com/articles/JUnit/article.html

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message],boolean condition)	Checks that the boolean condition is true.
assertsEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertsEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

Środowisko testowania - uruchomienie testów jednostkowych



Środowisko testowania - wynik testowania

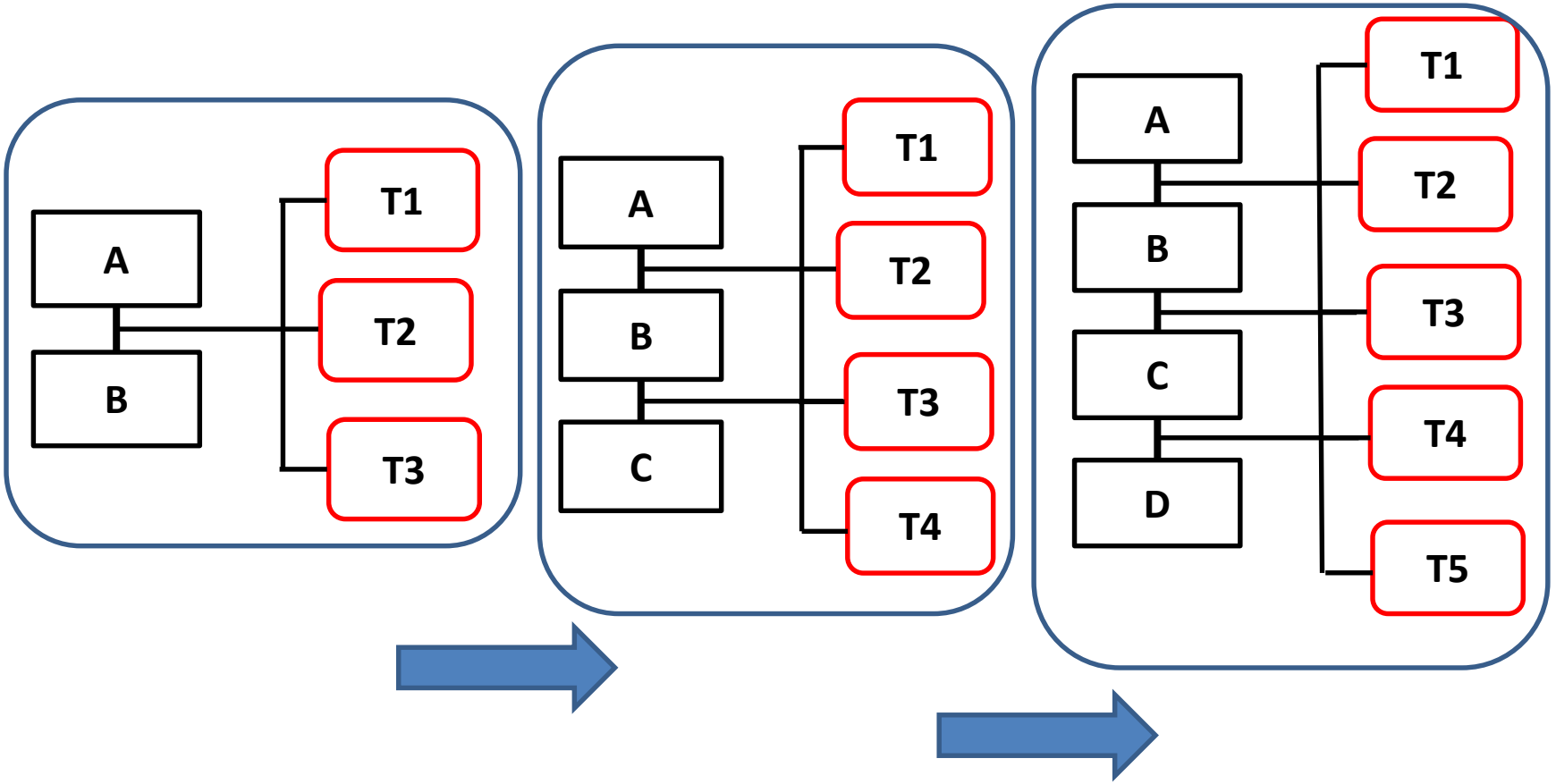


Testy integracyjne [2]

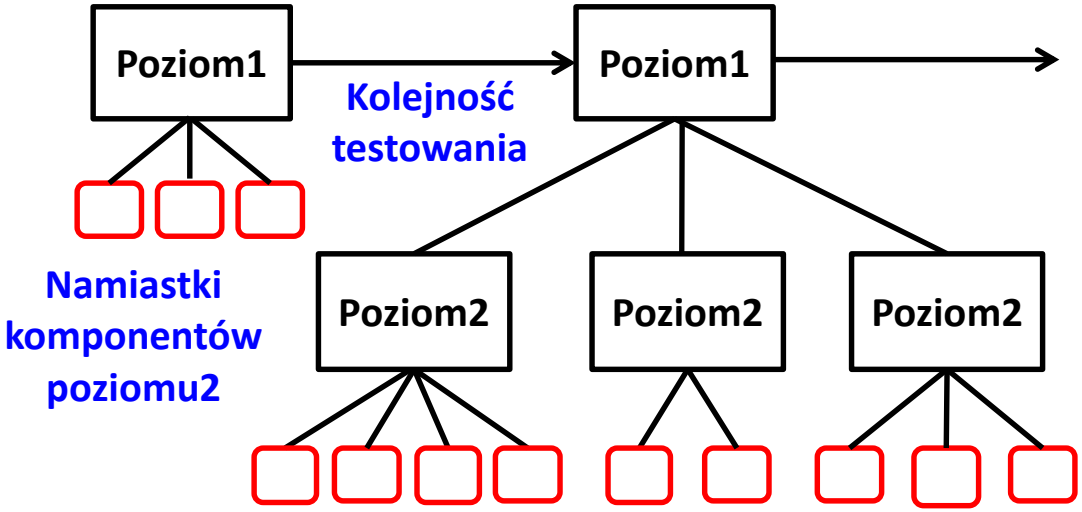
1.Koncepcja testowania

2. Testowanie zstępujące i wstępujące

Koncepcja testowania integracyjnego

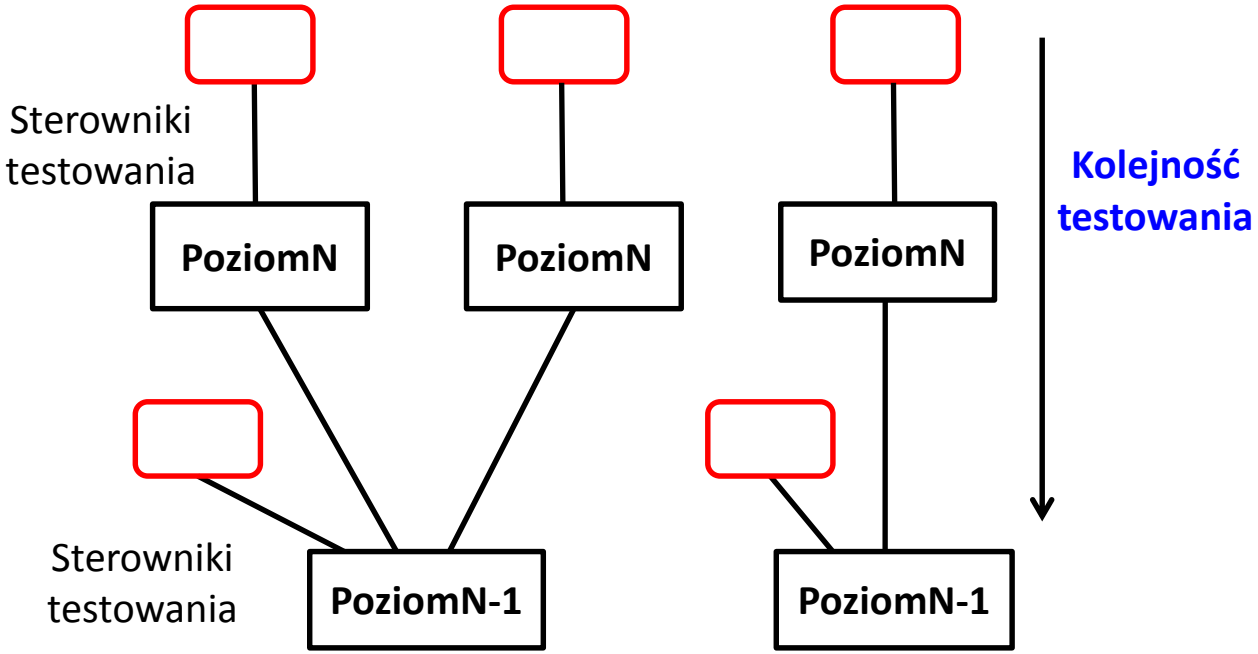


Testowanie integracyjne



Zstępujące testowanie integracyjne

Testowanie od komponentów wysokiego poziomu do niższego



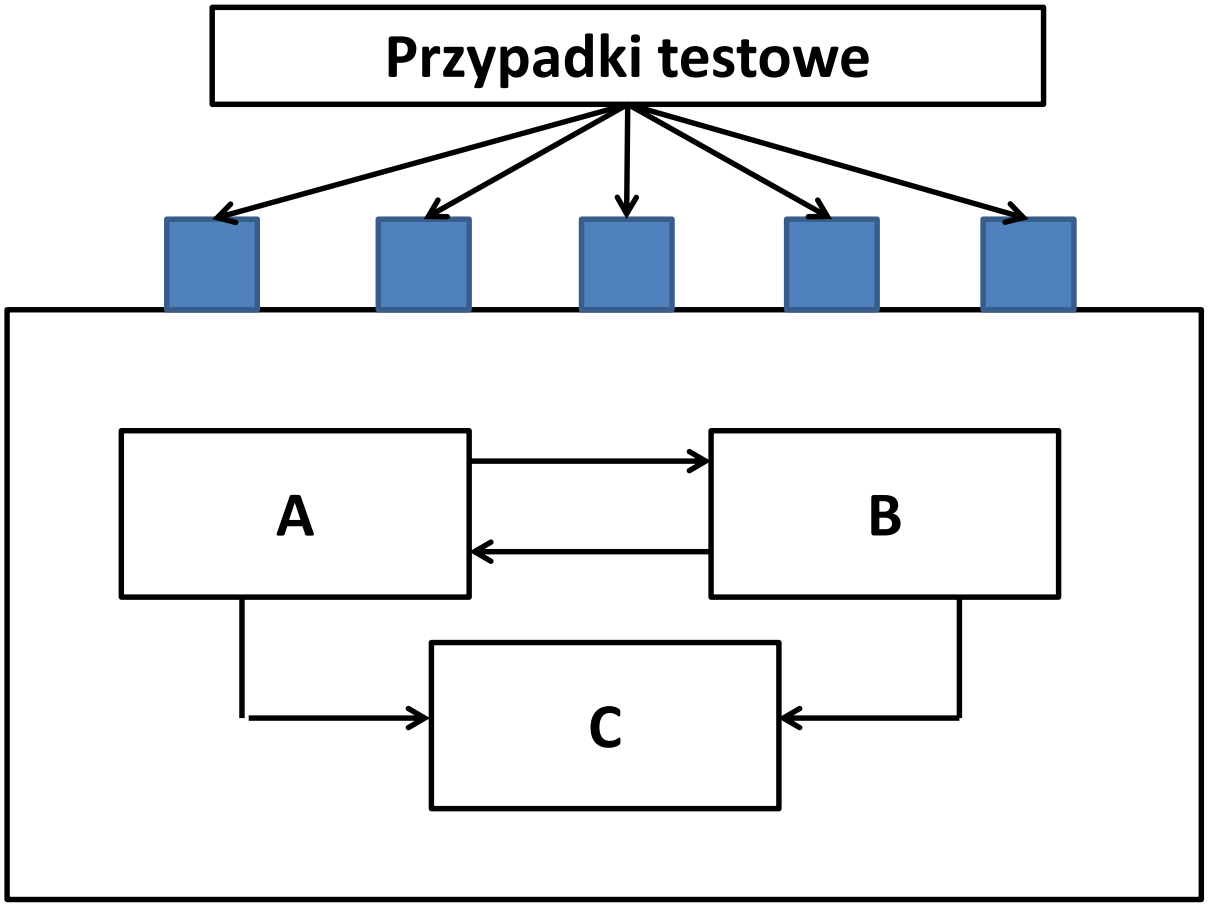
Wstępujące testowanie integracyjne

Integrowanie wyników testowania od komponentów najniższego poziomu do najwyższego poziomu

Testowanie systemowe [2]

- Koncepcja testowania

Testowanie systemowe

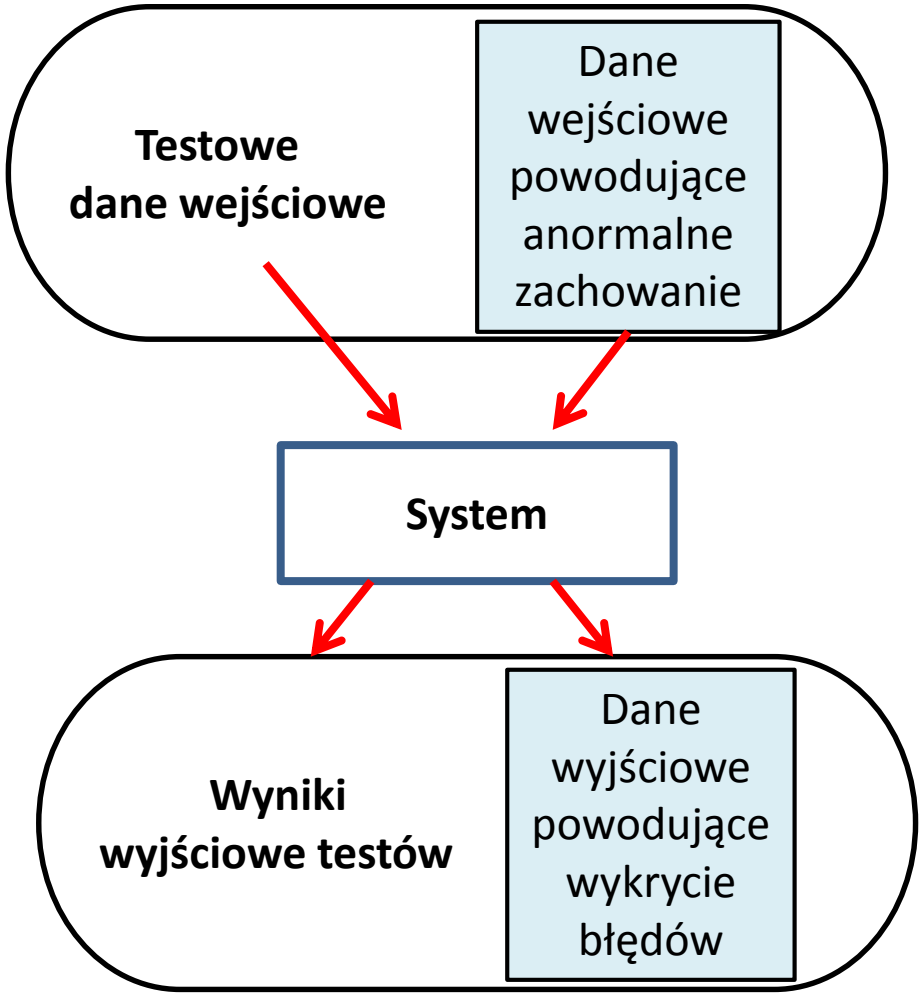


Komponenty:
A, B, C

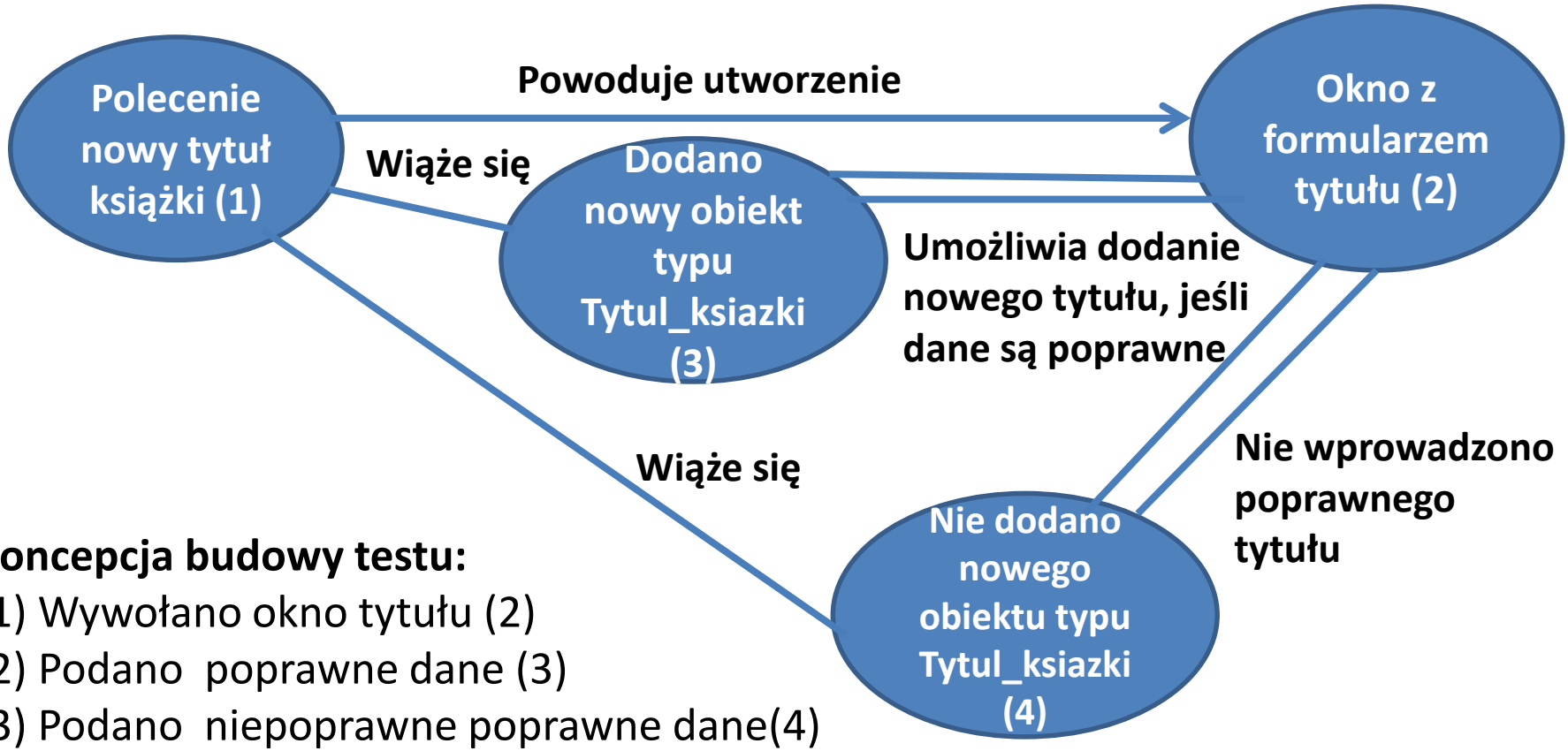
Testowanie funkcjonalne, metoda „czarnej skrzynki” [2, 3]

- Testowanie ”czarnej skrzynki”
- Metody testowania funkcjonalnego - oparte na grafach
- Metody testowania funkcjonalnego – oparte na klasach równoważności
- Metody testowania funkcjonalnego - oparte na analizie wartości brzegowych
- Metody testowania funkcjonalnego - oparte na testowaniu porównawczym
- Metody testowania funkcjonalnego - oparte na metodzie tablic ortogonalnych

Testowanie ”czarnej skrzynki”



Metody testowania funkcjonalnego oparte na grafach – przykład



Koncepcja budowy testu:

(1) Wywołano okno tytułu (2)

(2) Podano poprawne dane (3)

(3) Podano niepoprawne poprawne dane(4)

Pokrycie wierzchołków testami: sprawdzenie, czy uwzględniono wszystkie potrzebne wierzchołki

Pokrycie krawędzi grafów testami: **należy sprawdzić własności krawędzi (zwrotność, przechodniość itd)**

Metody testowania funkcjonalnego - oparte na grafach cd

- **Modelowanie przepływu transakcji**
  - Wierzchołki odpowiadają krokom procesu przetwarzania transakcji
  - Krawędzie odpowiadają logicznym połączeniom między tymi krokami
- **Modelowanie skończonej liczby stanów**
  - Wierzchołki odpowiadają stanom rozpoznawanym przez użytkowania np. wypełnianymi formularzami
  - Krawędzie odpowiadają logicznym połączeniom między tymi stanami
- **Modelowanie przepływu danych**
  - Wierzchołki odpowiadają obiektom danych
  - Krawędzie odpowiadają przekształcaniu obiektów danych na inne obiekty danych
- **Modelowanie zachowania systemu w czasie**
  - Wierzchołki odpowiadają obiektom w programie
  - Krawędzie odpowiadają sekwencyjnym przejściom pomiędzy nimi. Wagi krawędzi odpowiadają czasowi poszczególnych przejść

Metody testowania funkcjonalnego – oparte na klasach równoważności

Dzielenie na klasy równoważności:

- Jeśli dane wejściowe są opisane przedziałem lub konkretną liczbą , to w testach używa się **jedną poprawną i dwie niepoprawne wartości**
- Jeżeli dane wejściowe są opisane zbiorem lub warunkiem logicznym, to w testach używa się **jedną poprawną i jedną niepoprawną wartość**

Metody testowania funkcjonalnego – oparte na analizie wartości brzegowych

Analiza wartości brzegowych

- Jeśli dane wejściowe są opisane przedziałem a, b, to należy testować **wartości a i b oraz wartości nieco większe i nieco mniejsze niż a i b**
- Jeśli dane wejściowe są opisane zbiorem liczb, to testy należy wykonać dla **wartości największej i najmniejszej oraz wartości nieco większych i nieco mniejszych niż te wartości**.
- Te same zasady należy zastosować dla danych wyjściowych
- Należy sprawdzić działanie programu wypełniającego całą przestrzeń danych przy ograniczonych strukturach danych w programie

Metody testowania funkcjonalnego - testowanie porównawcze

Analiza porównawcza

- Stosuje się w przypadku konieczności wykonania kilku różnych rozwiązań o tej samej funkcjonalności np. w celu poprawy niezawodności oprogramowania.
- Testy przeprowadza się jednocześnie wykorzystując te same dane wejściowe.
- Nie można wyeliminować błędów wynikających z błędnej specyfikacji programu.

Metody testowania funkcjonalnego - metoda tablic ortogonalnych

Jeżeli liczba kombinacji danych wejściowych jest niewielka, ale za duża, aby testować wszystkie kombinacje, wtedy **w przypadku braku zależności między błędami** można wykrywać tzw. **błędy jednomodalne**

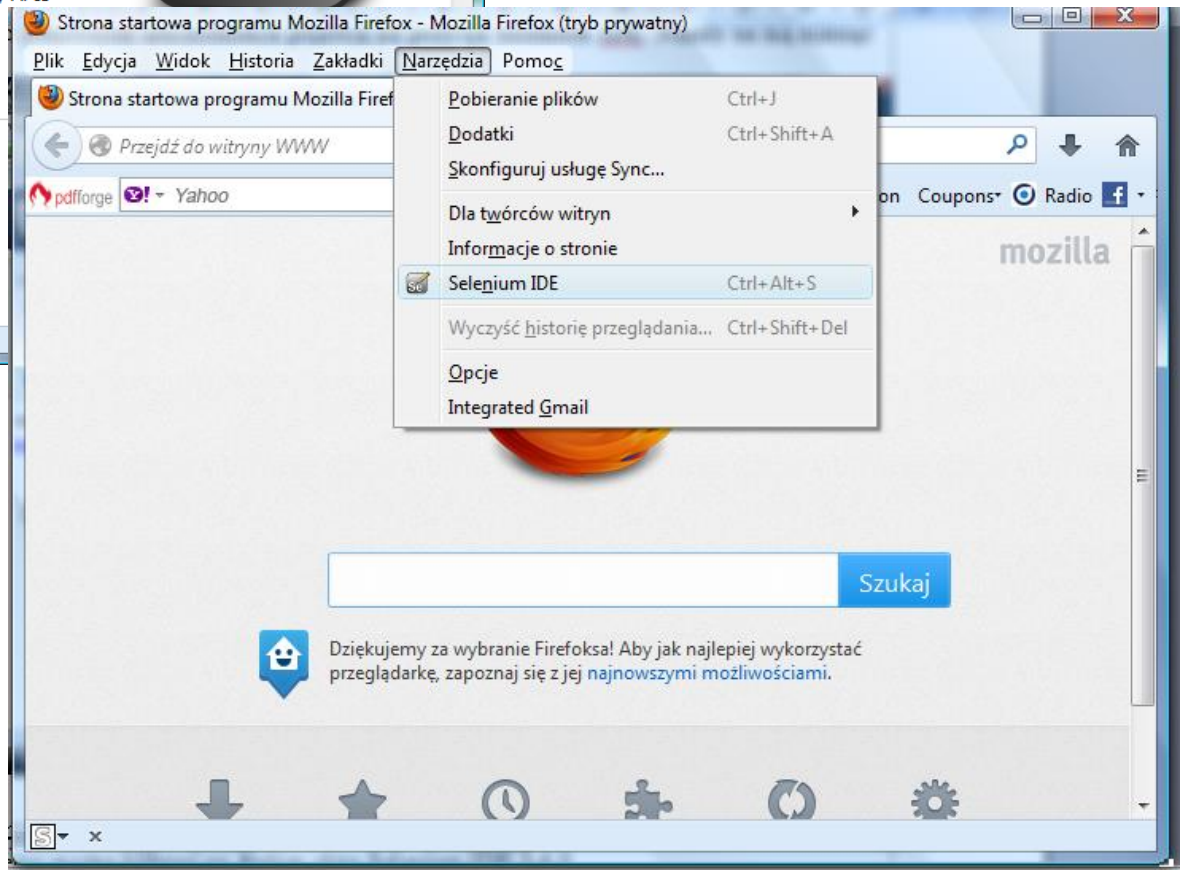
Test	Parametry			
	P1	P2	P3	P4
1	**1**	1	1	1
2	**1**	2	2	2
3	**1**	3	3	3
4	**2**	1	2	3
5	**2**	2	3	1
6	**2**	3	1	2
7	**3**	1	3	2
8	**3**	2	1	3
9	**3**	3	2	1

Zamiast 81 testów (3^4 - trzy wartości przyjmowane przez 4 dane wejściowe) wykonuje się 9 testów ze względu na P1

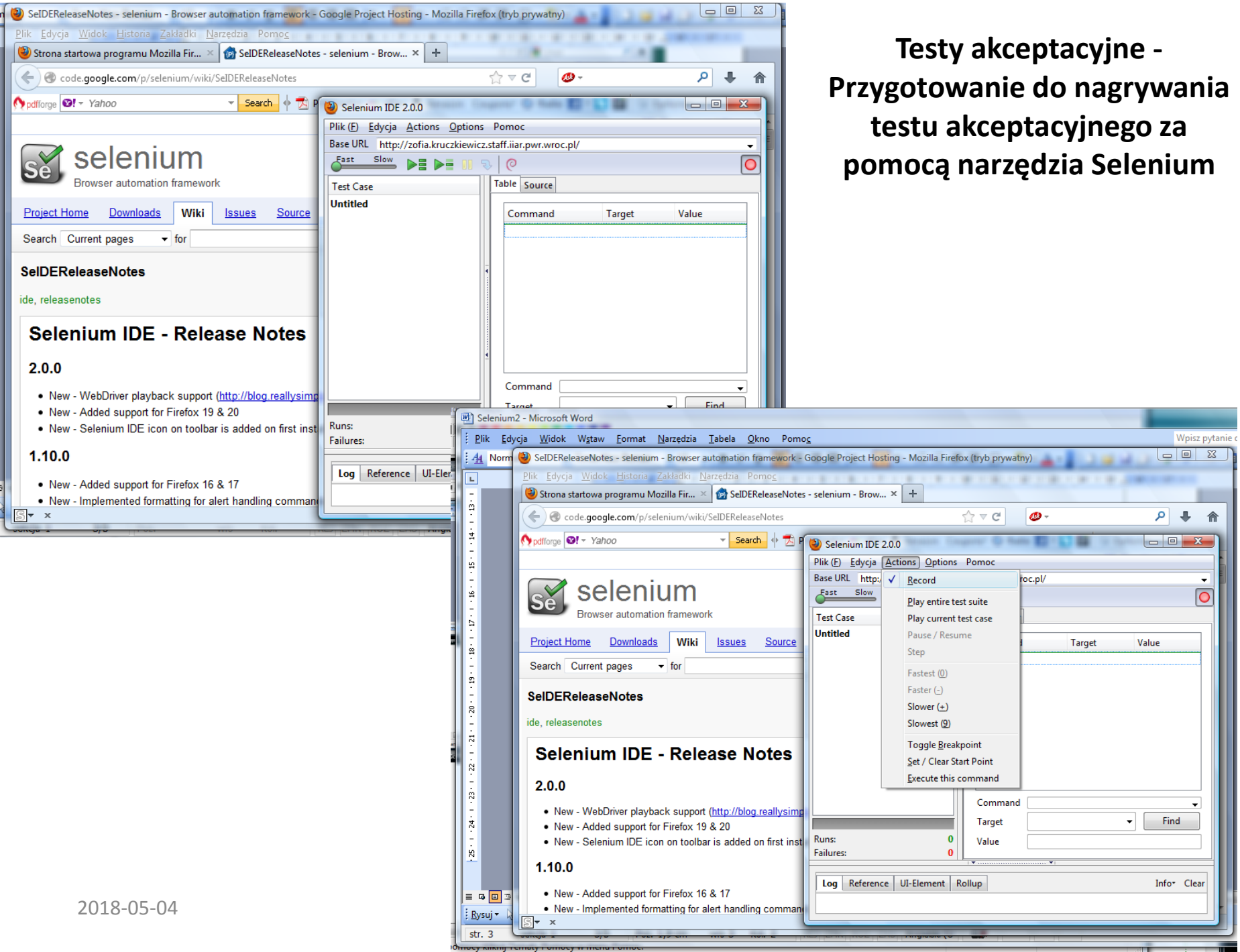
Testy akceptacyjne (funkcjonalne) - przykład

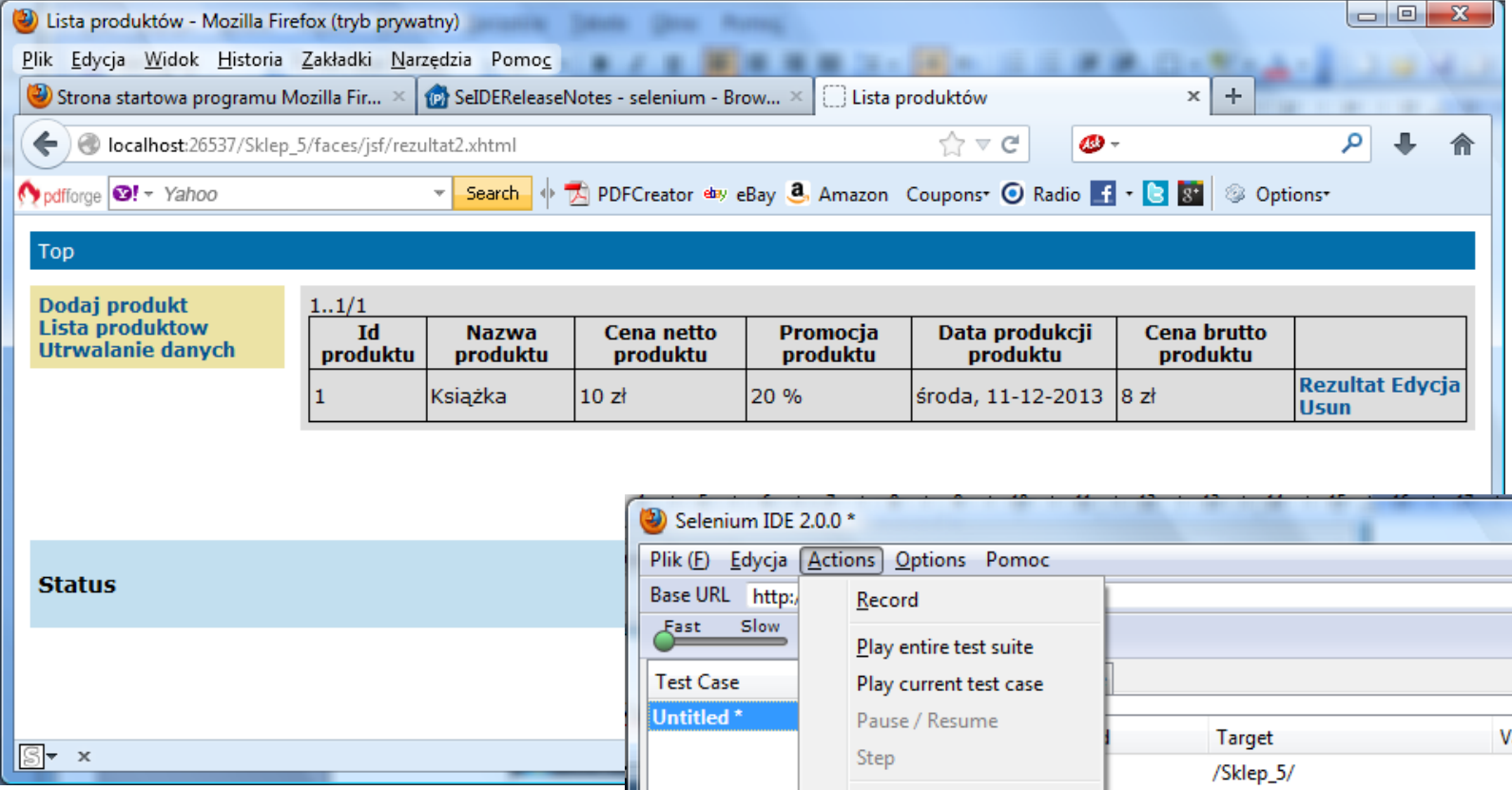
- Instalacja narzędzia do testowania Selenium
- Nagrywanie testu
- Odtwarzanie testu

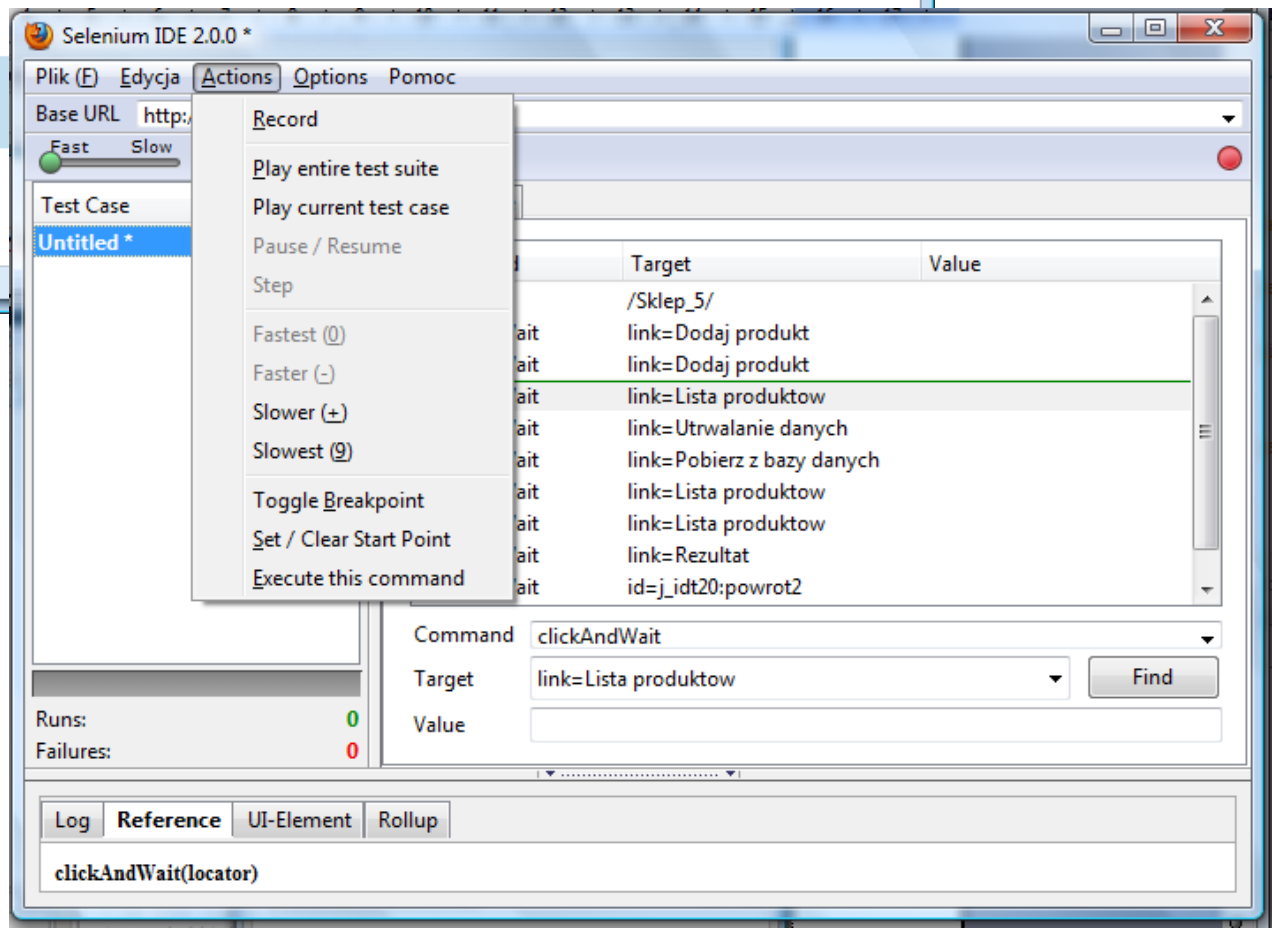
Testy akceptacyjne - Instalacja dodatku Selenium



Testy akceptacyjne - Przygotowanie do nagrywania testu akceptacyjnego za pomocą narzędzia Selenium

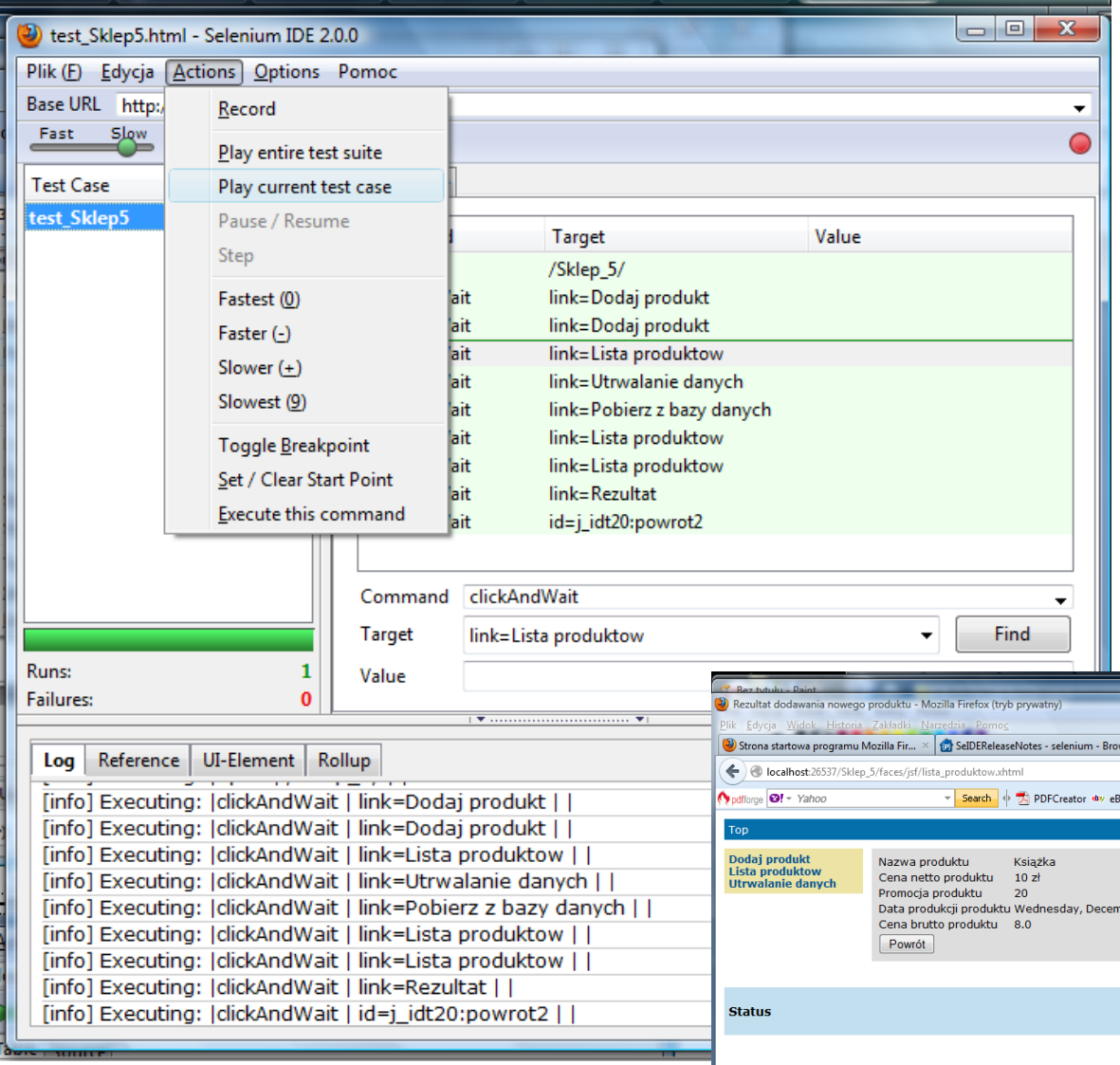


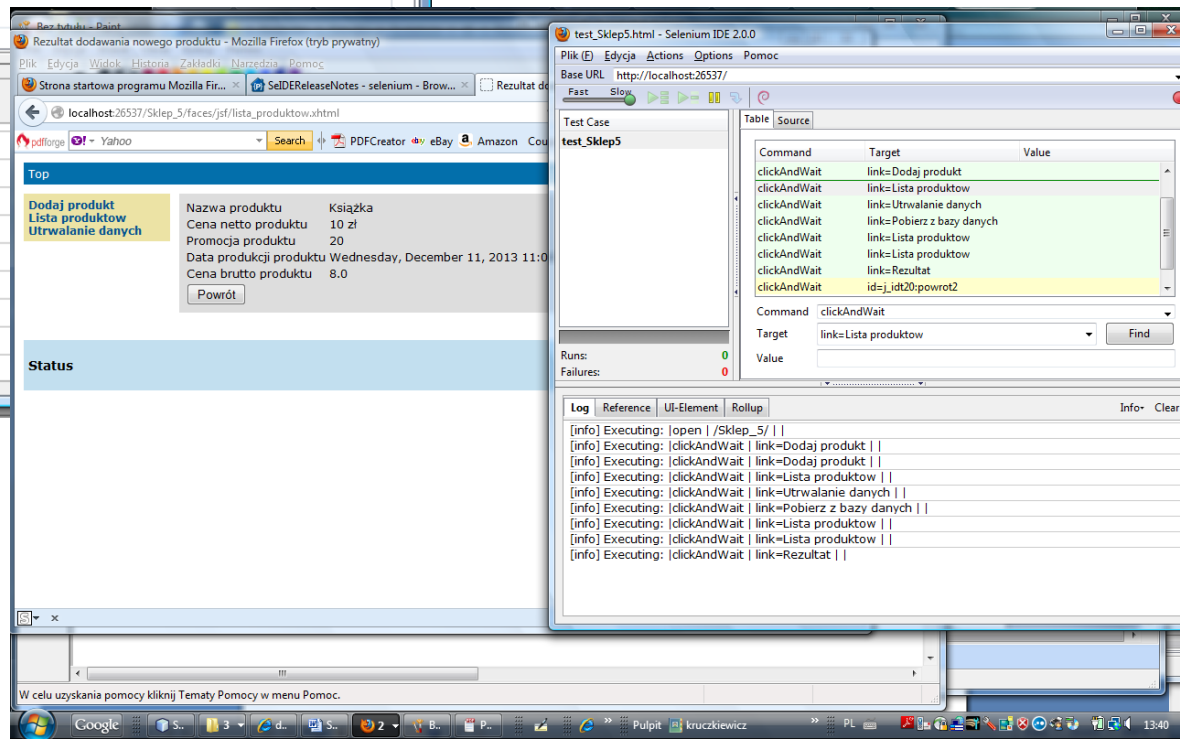




Testy akceptacyjne - Nagrywanie testu akceptacyjnego

Testy akceptacyjne - Odtwarzanie nagranego testu akceptacyjnego



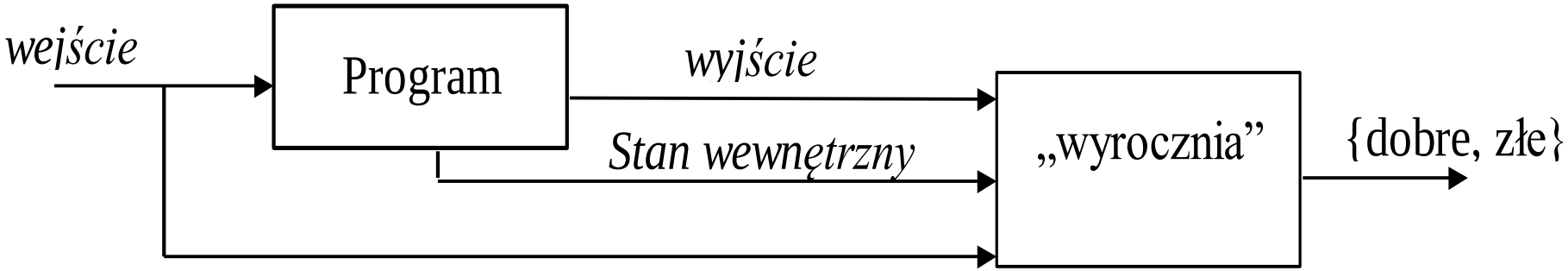


Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne
7. Plan testowania błędów zakresu produktu
8. Ocena niezawodności programu – testowalność [2]

Testowalność (niezawodność)

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

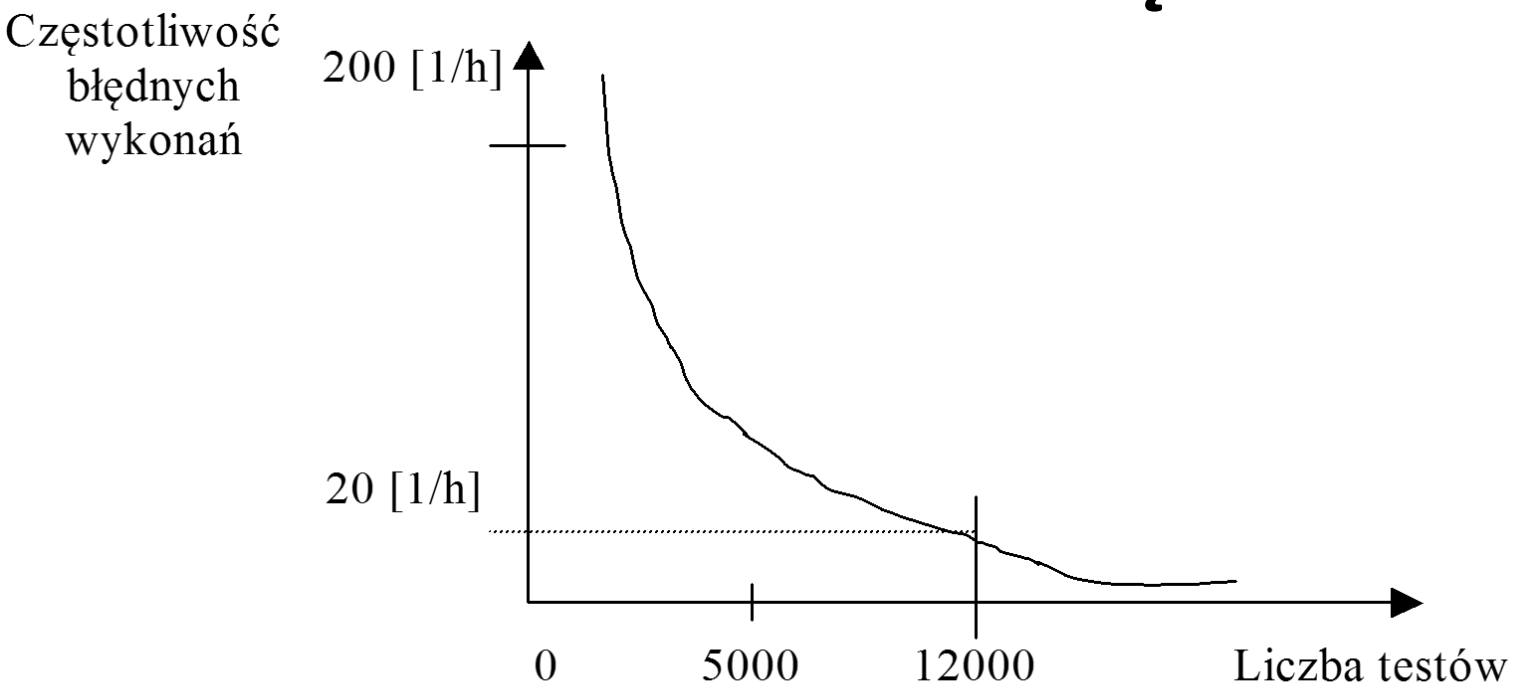


***Wyrocznia* jest następującą funkcją:**

Wyrocznia: D × R ×(zbiór_wartości_stanów_programu) →{dobry, zły}

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

Związek między liczbą przeprowadzonych testów i niezawodnością



***Niezawodność* programu** jest częstotliwością jego błędnych wykonań.

Miara niezawodności: MTBF = MTTF + MTTR

MTBF – mean time between failure, MTTF (mean time to failure), MTTR (mean time to repair)

Rośnie ona logarytmicznie wraz ze wzrostem liczby przeprowadzonych testów i usuwaniu błędów.

Struktura wykładu

1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne
7. Plan testowania błędów zakresu produktu
8. Ocena niezawodności programu – testowalność
9. Ocena wykrywalności błędów

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 przed 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.

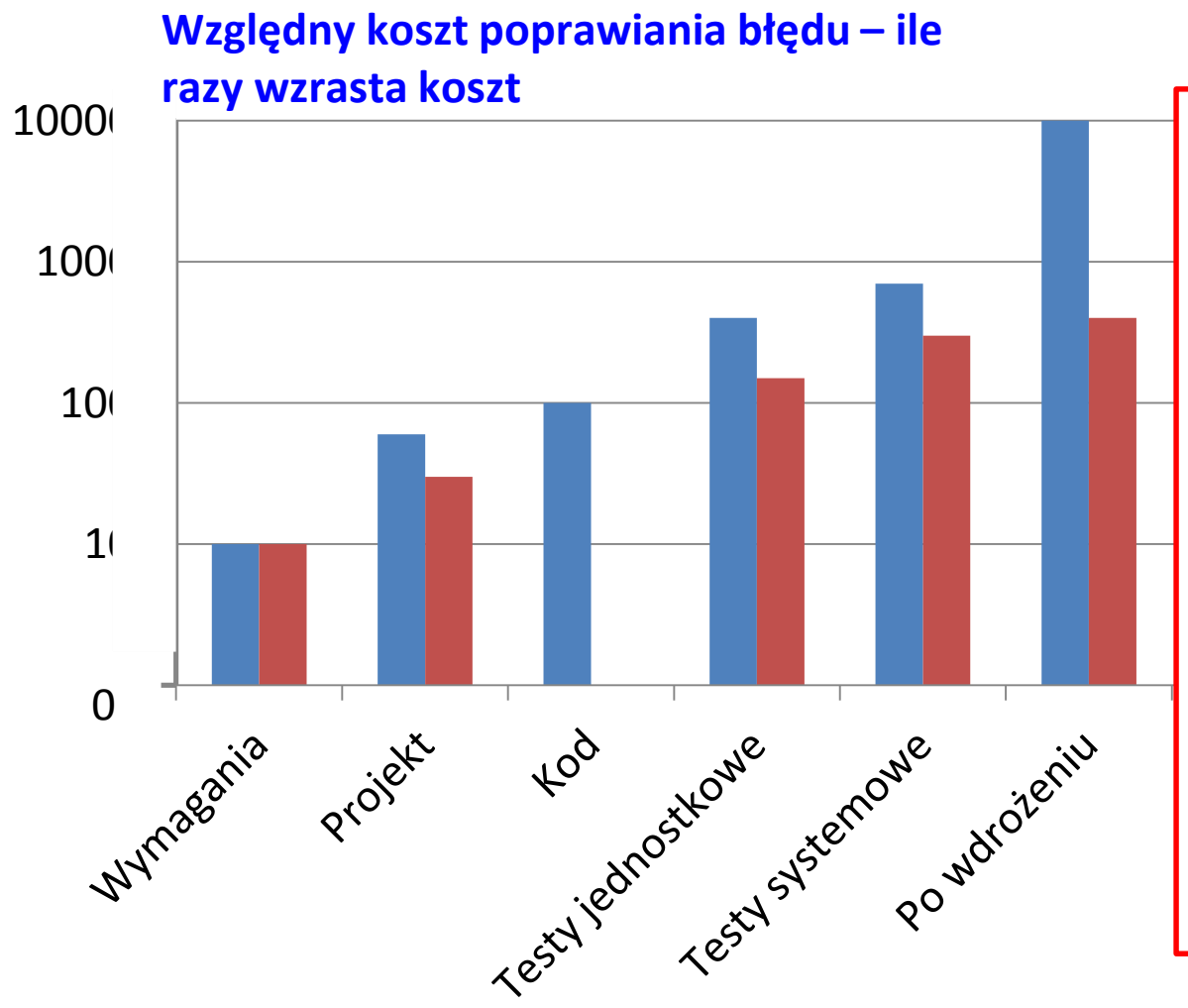
Metody posiewania błędów

- losowe zakłócenia w przypisywaniu danych
- losowe mutacje kodu - zmiany kodu źródłowego modyfikującego sterowanie lub dane w programie
- losowe zakłócenia między interfejsami modułów

Struktura wykładu

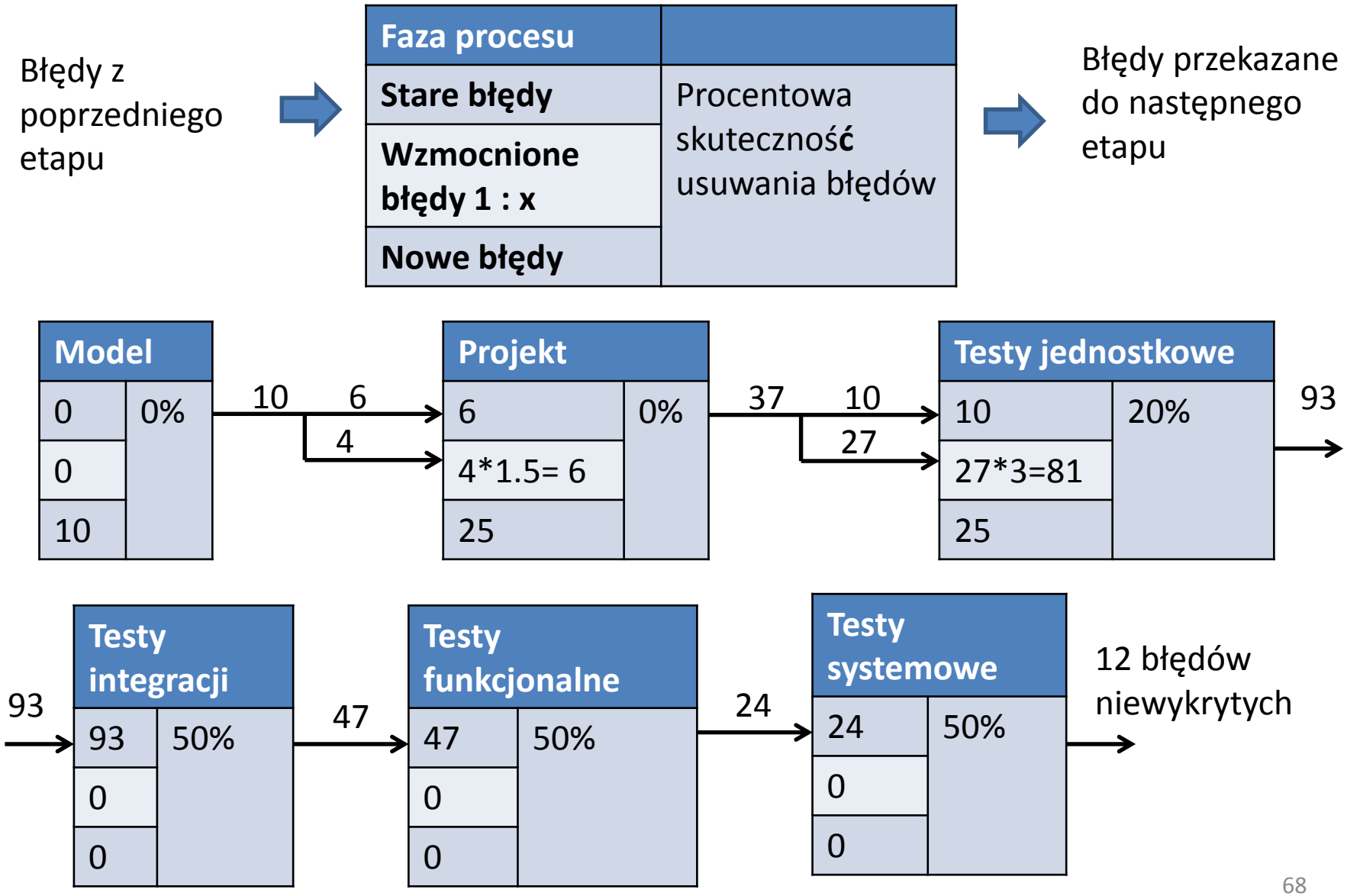
1. Wstęp
2. Klasyfikacje błędów i testów
3. Problemy testowania i lokalizacji błędów
4. Testowanie błędów
5. Zalecane techniki weryfikacji przeprowadzane podczas cyklu życia produktu
6. Testy statyczne – testowanie symboliczne
7. Plan testowania błędów zakresu produktu
8. Ocena niezawodności programu – testowalność
9. Ocena wykrywalności błędów
10. Koszt wykrywalności błędów

Względny koszt poprawiania błędów



Przykład:
LOC =200 000,
czas_usuwania_błędów =7053h,
koszt_h =40 USD,
liczba_błędów = 3112,
cały_koszt=282120 USD,
koszt_usuwania_błędu ≈ 91 USD,
a po wdrożeniu:
3640 ÷ 91000 USD

Usuwanie i wzmacnianie się błędów bez przeglądów technicznych



Usuwanie i wzmacnianie się błędów po wprowadzeniu przeglądów technicznych