Studia Podyplomowe: Java EE – produkcja oprogramowania

Produkcja i testowanie (PTE)
Wybrane metody testowania

(do użytku wewnętrznego)

Anna Derezińska
Instytut Informatyki, Politechnika Warszawska

Semestr zimowy 2020/21
Grupa: JA20Z (zesp. JA1-A, JA1-B) - semestr pierwszy

1

Problem wyroczni testu (*oracle*)

Metody weryfikacji wyniku działania programu

1. Bezpośrednia weryfikacja
2. Wielowersyjne przetwarzanie
3. Sprawdzenie spójności
4. Redundancja danych – porównanie wyjść dla różnych dany

Uwaga – asercje w testach jednostkowych są zwykle rodzajem wyroczni dla przypadków testowych lub sprawdzają ogólne warunki poprawności

2

Bezpośrednia weryfikacja

Przykład - sortowanie liczb w tablicy (np. rosnąco)

```
public class SortClass {
        public void fill_in(int tab [], int max) {...}
        public void sort(int tab [], int max) {...}
}
```

Traktując metodę sortowania jako „czarną skrzynkę”, jaki warunek końcowy (*post condition)* ma spełniać tablica na wyjściu?

3

Warunek końcowy 1

Liczby w tablicy są w kolejności rosnącej

```
@Test
public void postCondition1() {
    SortClass s= new SortClass();
    s.fill_in(inputdata, NMAX);
    s.sort(inputdata, NMAX);
    for (int k = 0; k< NMAX; k++)
    { assertTrue(inputdata[k]<= inputdata[k+1],"table not sorted",);
    }
}
```

Ale co z przypadkiem:
wejście: 4, 2, 3, 8
wyjście: 1, 2, 3 ,4 – jest posortowane, test przejdzie, choć inne liczby

4

Warunek końcowy 2

Liczby w tablicy są w kolejności rosnącej i tablica na wyjściu zawiera tylko liczby z tablicy wejściowej

```
@Test
public void postCondition2() {
    boolean flag;
    int [] copydata = new int [NMAX+1];
    System.arraycopy(inputdata, 0, copydata,  0, NMAX+1);
    s.sort(inputdata, NMAX);
    for (int k = 0; k <= NMAX; k++)
        { flag = true;
        for (int j = 0; j <= NMAX; j++)
            { if (inputdata[k] == copydata[j])
                {flag = false; break;}
            }
        assertFalse(flag, "element not from input table", );
} }
```

Ale co z przypadkiem: wejście: 4, 2, 3, 8, wyjście: 2, 2, 2 ,2 – jest posortowane, elementy z tablicy wejściowej

5

Warunek końcowy 3

Liczby w tablicy są w kolejności rosnącej i tablica na wyjściu zawiera permutację liczb z tablicy wejściowej

Test dla tego warunku wykryje więcej błędnych przypadków, ale kod realizujący taki test jest jeszcze bardziej złożony.
Możliwość błędu w kodzie testującym!

package junit. sorterPostCondition

6

Sprawdzanie spójności danych

Przykłady:

P- prawdopodobieństwo, P>1 lub P<0 – błąd

P - data urodzenia żyjącej osoby, P<1850 ??

P – liczba dni w roku, P = 500 ???

P – koszt czegoś, P <0 ??

P – prędkość samochodu, P = 10000 km/h ??

Niezmienniki, asercje, spójność danych, wykrywanie anomalii, ...

7

Testowanie - wyszukiwanie defektów

Podzbiory możliwych przypadków testowych

Strategie wyboru przypadków (tradycyjne):

- **Funkcjonalne , czarnej skrzynki (black box)** – testy wyprowadzone ze specyfikacji
- **Strukturalne, białej skrzynki (white box)** – testy wyprowadzone na podstawie znajomości struktury programu

Jaki poziom abstrakcji jest podstawą doboru testów na podstawie danych kryteriów testowych!

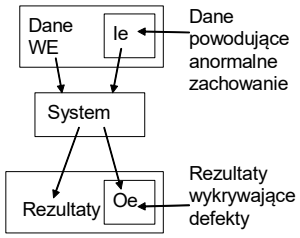
8

Testowanie funkcjonalne czarnej skrzynki (black box)

Wyprowadzenie testów na podstawie specyfikacji

Zachowanie systemu -„czarnej skrzynki" określone na podstawie wejść i odpowiadających im wyjść.

Identyfikacja przypadków, które ujawnią błędy.



9

Podział na klasy równoważności

Głównie testy funkcjonalne

Equivalence partitioning

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.

10

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

11

Przykład specyfikacji metody

public int search(Element tab [], Element key)

Input:

- Tab tablica elementów
- Key wartość szukanego elementu

Output:

- W przypadku znalezienia elementu o podanej wartości zawracany jest jego indeks w tablicy
- Jeśli brak elementu o danej wartości zwraca jest liczba „-1"
- Tablica elementów nie powinna być zmieniona

12

Podziały na bazie specyfikacji

Grupy podziału ze względu na tablicę ***Tab***:

A. - tablica pusta (nie zawsze możliwe)
B. - tablica 1-elementowa,
C. - tablica wielo-elementowa

Grupy podziału ze względu na klucz ***Key***:

I. - nie ma klucza w tablicy
II. - jest klucz w tablicy (w tym podgrupy):
   1. jest pierwszym elementem tablicy
   2. jest ostatnim elementem tablicy
   3. jest wewnętrznym elementem tablicy

13

Testy na bazie specyfikacji funkcji

Przykładowe przypadki testowe wg możliwych kombinacji podziałów na grupy

Input sequence (Tab)	Key (Key)	Output
A:		-1
B I: 5	5	0
B II: 105	5	-1
C I1: 17, 29, 21, 23	17	0
C I2: 41, 18, 9, 31, 30, 16, 45	45	6
C I3: 17, 18, 21, 23, 29, 41, 38	23	3
CII: 1, 2, 3, 4, 5	104	-1

package junit. searchEquivalencePartition

14

Testowanie strukturalne, białej (szklanej) skrzynki (white-box)

Osoba testująca może analizować kod, korzystać ze struktury komponentu do opracowania testu.

Zwykle dla małych jednostek programu

Pokrycie kodu,...

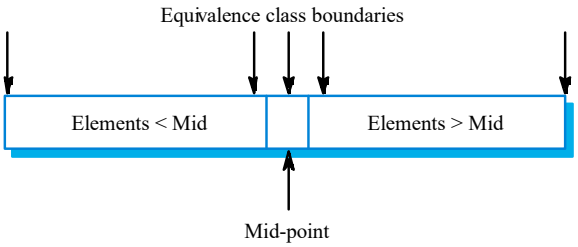
Klasy równoważności

Znajomość algorytmu pozwala na znalezienie dalszych podziałów.

Wewnętrzne wartości graniczne

15

Szukanie połówkowe – klasy równoważności



Warunek wstępny – tablica wejściowa musi być posortowana

16

Szukanie połówkowe – przypadki testowe

Więcej przypadków dla wielo-elementowej tablicy i istniejącego klucza

- Klucz jest w środku oraz jest parzysta/nieparzysta liczba elementów
- Klucz jest na prawo od środka lub na lewo od środka

Input sequence (Tab)	Key (Key)	Output
A:		-1
B I: 5	5	0
B II: 105	5	-1
C I1: 17, 29, 21, 23	17	0
C I2: 41, 18, 9, 31, 30, 16, 45	45	6
C I3a: 17, 18, 21, 23, 29, 41, 38	23	3
C I3b: 17, 18, 21, 23, 29, 41, 38	18	1
C I3c: 17, 18, 21, 23, 29, 41, 38	41	5
C I34: 17, 18, 21, 23, 29, 41	23	3
CII: 1, 2, 3, 4, 5	104	-1

17

Pokrycie kodu (przepływu sterowania)

Pokrycie:

- instrukcji, co najmniej jednokrotne wykonanie każdej instrukcji dla danego testu (zestawu testów)

często zastępowane przez pokrycie linii kodu (line coverage)

- warunków (rozejść decyzyjnych)

każdy elementarny warunek ma zostać co najmniej raz spełniony i co najmniej raz nie spełniony

- złożonych warunków logicznych
- bloków, funkcji – wykonane co najmniej raz
- ścieżek

18

Czy wystarczy 100% pokrycia linii?

Wysokie pokrycie kodu – warunek konieczny ale NIE wystarczający dobrych testów

```
public class ArithmeticOperation {
        public int multiply(int arg1, int arg2){
                return arg1 + arg2;       }
}
@Test
public void multiply_zero() {
 ArithmeticOperation a = new ArithmeticOperation();
 Assert.assertEquals(0*0, a.multiply(0, 0));
}
```

Test daje 100% pokrycia metody *multiply()* ale nie wykrywa błędu w mnożeniu

package junit. simpleCoverage

19

Wyznaczanie pokrycia

Narzędzia - **analizatory** pokrycia kodu:

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

Testowanie **wyczerpujące** (exaustive) przejście przez każdą możliwą ścieżkę wykonania programu – praktycznie niemożliwe.

20

Ćwiczenie – Coverage (EclEmma)

1. Sprawdzić pokrycie kodu dla klasy *ArithmeticOperation w przypadku wykonaniaTest2ArithmeticOperation*
   (okno *Coverage,* kolory w kodzie źródłowym klasy, % w repozytorium)
2. Sprawdzić pokrycie kodu klasy *ArithmeticOperation w przypadku Test3ArithmeticOperation*
3. Połączyć wyniki sesji testów (*merge*) i sprawdzić pokrycie.
4. Porównać rezultat z wynikami pokrycia dla *Test4ArithmeticOperation*

21

Pokrycie *Money* – praca domowa B

- Dla rozbudowanej klasy *Money* (zad. A.1 i A.2) sprawdzić pokrycie kodu oraz pokrycie gałęzi (ang. *branches*).
- W przypadku niedostatecznego pokrycia dopisać testy jednostkowe. Warto skorzystać z testów sparametryzowanych.

22

Test z jednym parametrem prostym

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
@ParameterizedTest
@ValueSource(strings = {"CHF","PLZ","USD"})
public void addSameCurrency1(String currency) {
      Money m12curr = new Money(12, currency);
      Money m14curr = new Money(14, currency);
      Money expected = new Money(26, currency);
      Money result = m12curr.add(m14curr);
      assertTrue(expected.equals(result));
}
```

@ValueSource - string, int, long, double

23

Test z wieloma parametrami 1

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
@ParameterizedTest
@CsvSource({"12, 14, 26", "2, 3, 5"})
public void addPLZ(int a, int b, int expect_sum){
      Money aPLZ = new Money(a, "PLZ");
      Money bPLZ = new Money(b, "PLZ");
      Money expected=new Money(expect_sum, "PLZ");
      Money result = aPLZ.add(bPLZ);
      assertTrue(expected.equals(result)); }
```

@CsvSource - string, możliwe konwersje

24

Test z wieloma parametrami 2

```
@ParameterizedTest
@CsvSource({"12, 14, 26, PLZ", "2, 3, 5, PLZ",
"12, 14, 26, USD", "2, 3, 5, USD"})

public void testSimpleAdd(int a, int b, int
expect_sum, String currency) {
  Money acurr = new Money(a, currency);
  Money bcurr = new Money(b, currency);
  Money expect = new Money(expect_sum, currency);
Money result = acurr.add(bcurr);
assertTrue(expect.equals(result));
}
```

25

Test z jednym parametrem złożonym

```
import java.util.stream.Stream;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;
@ParameterizedTest
@MethodSource("name_ofMethod")
public void addSameEqualMultiplyBy2(Money m) {
      assertTrue(m.add(m).equals(m.multiply(2)));
}
private static Stream<Money> name_ofMethod(){
      return Stream.of(new Money(5,"PLZ"),
      new Money (5,"USD"), new Money (13,"PLZ"));
 }
```

26

Testowanie obiektowe

Poziomy

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

intra-method, inter-method, intra-class, inter-class

27

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)

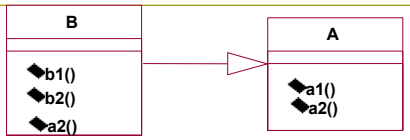
28

Zbiory obiektów zależnych

- **Testowanie hierarchii klas**
  - operacje odziedziczone
  - funkcje wirtualne dla obiektów klasy bazowej i potomnych, polimorfizm,
  - dziedziczenie wielopoziomowe
  - operacje przy realizacji wielu interfejsów

29

Testowanie klasy B



- Testowanie użycia atrybutów z klasy B, i innych dostępnych atrybutów (np. chronionych atrybutów z pakietu zawierającego B)
- Testowanie strukturalne i funkcjonalne metod.
- Testowanie odziedziczonej metody a1() w kontekście obiektów klasy B.
- Testowanie użycia obiektów klasy B w możliwych stanach.

package junit. Inheritance Zadanie C

30

Dziedziczenie – praca domowa C

- Napisać nową klasę zawierającą testy jednostkowe testujące klasę *Manager* z pakietu *inheritance*.
- Uwzględnić dziedziczenie z klasy *Employee*

To znaczy: testujemy wszystkie metody klasy *Manager,* oraz wszystkie metody klasy *Employee* wywołane na rzecz obiektu klasy *Manager.*

Sprawdzić pokrycie klas *Manager* i *Employee*.

31

Testowanie eksploracyjne

- Program traktujemy jako specyfikację i systematycznie badamy udostępniane funkcje - jak dla czarnej skrzynki.

  Np. gdy nie ma specyfikacji wymagań ani systemu
- Brak predefiniowanego zbioru testów.
- Równoległy proces tworzenia przypadków testowych, ich wykonywania i oceny wyników.
- Nie da się odnaleźć brakujących funkcji

35

Testowanie systemu

Testowanie wymagań niefunkcjonalnych
- Obciążenie , - Ochrona , - Wydajność
- Przenośność - różne konfiguracje sprzętowe, biblioteki,...
- Kompatybilność (np. nowej wersji)
- Instalowalność
- Niezawodność
- Odporności na niepożądane zdarzenia (roboustness)

zanik zasilania, awaria sprzętu, "głupota" użytkowania
- Samonaprawialność (recovery)

testowanie mechanizmów tolerowania błędów
- Sprawność serwisu
- Przydatność dokumentacji

36

Testowanie granicznego obciążenia (stress testing)

Testy sprawdzające jak system „radzi sobie" ze zwiększanym obciążeniem:
- bardzo duże dane wejściowe,
- dużo danych w krótkim czasie.

Systemy rozproszone, transakcyjne (np. nominalnie 100 tr/sec)

Testowanie jest kontynuowane po przekroczeniu planowanego obciążenia aż do upadku systemu.

Cele:
- Badanie jak nastąpi upadek systemu, czy dane nie zostały zniszczone, „łagodny upadek"
- Wykrycie defektów, które przy normalnej pracy nie ujawniły się, rzadko występujących

37

Testowanie regresyjne

Testowanie regresyjne
- ponowne wykonanie opracowanych wcześniej testów,
- zwykle dla zmodyfikowanego programu (po usunięciu błędu, zmianie/rozszerzeniu funkcjonalności)
- ale dla większych zmian – konieczna modyfikacja testów

Test na dym (ang. *smoke test*)
- uproszczone testowanie regresyjne – Czy program nadal się uruchamia?

Dla testów regresyjnych – ważna **jakość:** efektywność, spełnienie różnych kryteriów, minimalizacja, kolejność wykonania (priorytety) zbioru testów

38

Testowanie interfejsów

- Testy na podstawie specyfikacji i znajomości wewnętrznych interfejsów
- Błędy w interfejsie lub w założeniach interfejsu.
- Integracja modułów, systemy obiektowe, komponentowych (wielokrotnego użycia).

Typy interfejsów
- **parametryczne** (wskazania na dane, funkcje)
- **z dzieloną pamięcią** (shared memory)
- **proceduralne**
- **z przekazywaniem komunikatów** (message passing)

39

Klasy błędów interfejsów

- **Użycia** – częsty w interfejsach parametrycznych (błędny typ, kolejność, liczba parametrów)
- **Błędnego zrozumienia** – komponent wywołujący zakłada błędnie, jakie ma być zachowanie komponentu wywoływanego
  np. że wektor ma być uporządkowany a nie jest
- **Błędy synchronizacji** – systemy czasu rzeczywistego, komunikacja poprzez pamięć dzieloną lub przekazywanie komunikatów
  np. Producent i konsument danych pracują z różnymi prędkościami. Konsument może dostać „stare" dane.

40

Wskazania do testowania interfejsów

- Wyszukaj wywołania zewnętrznych komponentów.
- Testuj ekstremalne wartości parametrów w swoich zakresach
- Sprawdź interfejs dla wskazań równych null
- Interfejs proceduralny – zaprojektuj test, który spowoduje błąd komponentu
- Systemy z przekazywaniem komunikatów – testuj stresująco. Generowanie większej liczby komunikatów niż zakładano może ujawnić problemy czasowe.
- Komunikacja poprzez pamięć dzieloną – testuj różną kolejność dostępu do pamięci

41

Testowanie aplikacji internetowych

Cechy aplikacji internetowych:

- multi-tier nature,
- bogate sterowane zdarzeniami
- struktura hyperlinków
- bogaty interfejs graficzny
- dołączenie – server side scripting

Błędy:

1) błędy nawigacji
2) błędy interfejsów
3) błędy integracji
4) błędy w skryptach funkcyjnych i innym kodzie

42

GUI i aplikacje internetowe

Pokrycie:

- hyperlinków
  – przejście przynajmniej raz za pomocą każdego hyperlinku
- elementów GUI (przycisków, rozwijane menu, okienka wprowadzania danych)
- zdarzeń
  - uaktywnienie przynajmniej raz każdego zdarzenia aplikacji

43

Nagrywanie - Odtwarzanie

Recording – Plying back
Capture and Replay (CR)
Zarejestruj-odtwórz, odtwarzająco-przechwytujące

- odwołania do pozycji
- odwołania do obiektów

- Testowanie gotowej aplikacji
- Do testowania regresyjnego (tak, ale)
- Do testowania przed kodowaniem (test-first, TDD) NIE, ale:
  – skrypty testujące (np. Java Script)
  – generacja z modeli lub specyfikacji

44

Zasada pracy z "capture and replay"

1. a)"Przeklikanie" sekwencji testującej z użyciem GUI
   b) Rejestracja zdarzeń w skrypcie
   c) Wstawienie punktów kontrolnych (weryfikacyjnych) podczas nagrywania skryptu
2. Ręczna edycja skryptu (opcjonalnie)
3. Odtwarzanie skryptu testującego (możliwe wielokrotnie) – wyniki testów otrzymane z punktów kontrolnych

45

Typowe narzędzie

- Testowanie funkcjonalne aplikacji Java, Windows, HTML
  - Wspiera tworzenie i wykonywanie zautomatyzowanych testów interfejsów graficznych i funkcjonalności
- System tworzenia i edycji skryptów testowych
  - Np. JavaScript w środowisku Eclipse
- Mechanizm testowania oddzielony od technologii
  - Niezależnie od technologii sposób tworzenia i wykonywania testów jest taki sam

46

Podstawowe elementy testu

- Skrypt wykonania
- Punkty weryfikacyjne
  - Danych
  - Właściwości
- Zbiory danych
  - Zróżnicowanie testów w trakcie wykonywania (np. różni użytkownicy podczas logowania)
  - Powtarzanie testu dla różnych danych wejściowych
- Mapa obiektów
  - Umożliwia łatwe uaktualnianie właściwości badanych obiektów, na podstawie których opiera się nasz test (wspomaga testowanie regresyjne)

47

Punkty weryfikacyjne - przykłady

- sprawdzamy postać graficzną obiektu – po zmianie będzie błąd
- układ menu – dla danego obiektu automatycznie "przeklikane" elementy hierarchii, wybieramy elementy do testowania

48

Wskazówki do wyboru testów

- Testowanie możliwości systemu ważniejsze od testowania komponentów.
- Identyfikować błędy wstrzymujące, uniemożliwiające pracę użytkownika np. utraty danych.
- Testowanie starych możliwości ważniejsze niż nowych.
- Testowanie typowych sytuacji ważniejsze niż sytuacji brzegowych
- Testowanie „czarnych skrzynek" efektywniejsze od „szklanych skrzynek".
- Najlepiej metody mieszane - różne typy błędów.

56

Miary testowania

1) Oszacowanie pracochłonności i liczby testów

2) Kontrola dokładności testowania

- **Zasięg testowania** - ile przetestowano wymagań ze specyfikacji
- **Głębokość testów** - stosunek liczby przetestowanych niezależnych ścieżek do liczby wszystkich niezależnych ścieżek
- Miary strukturalne np. **% pokrycia**
- Statystyka wykrytych błędów odniesiona do profilu błędu - klasyfikacji błędów w zależności od ich dotkliwości

57

Podatność na testowanie (J. Bach)

- **Operatywność**
  - Błędy nie przerywają procesu testowania
  - Można uruchamiać niegotowy produkt
- **Obserwowalność**
  - Dla każdych danych wejściowych program generuje widoczne dane wyjściowe
  - Można obserwować stany systemu
  - Łatwo rozpoznać niepoprawne dane wyjściowe
  - Wewnętrzne błędy są wykrywane automatycznie za pomocą mechanizmów autotestowania

58

Podatność na testowanie (2)

- **Sterowalność**
  - Za pomocą kombinacji danych wejściowych można uzyskać wszystkie możliwe dane wyjściowe,
  - Można pokryć każdy fragment kodu
  - Spójne formaty danych we/wy
  - Testy można automatyzować i powtarzać
- **Podzielność**
  - System zbudowany z niezależnych modułów
  - Moduły można testować oddzielnie
  - Zmniejszenie propagacji błędów, szybsza lokalizacja przyczyn błędów

59

Podatność na testowanie (3)

- **Prostota**
  - Tylko funkcje niezbędne do spełnienia wymagań
  - Zmodularyzowana architektura
  - Kod zwięzły, zgodny ze standardami łatwy do analizy i pielęgnacji
- **Stabilność**
  - Zmiany nie są zbyt częste i są kontrolowane
- **Zrozumiałość**
  - Zrozumiały projekt, jasne zależności pomiędzy składnikami
  - Rozgłaszane zmiany w projekcie
  - Czytelna struktura dokumentacji

60

Lokalizacja i usuwanie błędów

Metoda brutalna (mało efektywna)
- zrzuty pamięci, śledzenie wykonania pod debuggerem

Metoda powrotów
- przeglądanie kodu wstecz od miejsca pojawienia się błędu (trudna dla dużej liczby ścieżek)

Metoda eliminacji przyczyn
- podzielić problem na części i prześledzić hipotezy robocze (testy)

Po znalezieniu błędu:
- *Czy ten sam błąd występuje w innych częściach programu?*
- *Jakie nowe błędy można wprowadzić poprawiając ten błąd?*
- *Jak można było go uniknąć?*

61

Koniec testowania

- koniec czasu i/lub pieniędzy
- wszystkie testy poprawne (tzn. bez sukcesu)
- spełnione zadane kryterium i bez sukcesu
- liczba (%) znalezionych błędów w stosunku do oczekiwanej liczby

62