Inżynieria Oprogramowania

Jakość, cd.

Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski
www.mimuw.edu.pl/~dabrowski

Walidacja

- Czy system realizuje to co trzeba?
  - "Czy budujemy właściwy system?"
- Trudno to stwierdzić
- Oceny są zazwyczaj subiektywne

Weryfikacja

- Czy oprogramowanie jest zgodne ze specyfikacją?
  - "Czy prawidłowo budujemy system?"
- Może być obiektywne
- Specyfikacje muszą być wystarczająco precyzyjne

Proces weryfikacji i walidacji

- Weryfikacja i walidacja musi wykonywana na każdym etapie tworzenia oprogramowania
- Główne cele:
  - Wykrycie błędów w systemie
  - Ocena czy system jest możliwy do wykorzystania produkcyjnego

Trzy sposoby

- Testowanie
  - System jest uruchamiany z danymi testowymi i sprawdzane jest jego zachowanie
- Inspekcje
  - Analiza statycznej reprezentacji systemu w celu wykrycia problemów
- Metody formalne

Podejście do testowania

- Czym jest skuteczny test?
  - Taki w którym nie znaleziono błędów?
  - Czy taki w którym wykryto co najmniej jeden błąd?
- Kto ma testować?
- Jak testować?
- Co poddawać testom?

- Gdy projekt się opóźnia skraca się czas testowania
  - … Czy to może mieć sens ekonomiczny?

Czym jest testowanie?

- Testowanie oprogramowania jest wykonaniem kodu dla kombinacji danych wejściowych i stanów w celu wykrycia błędów
- [Robert V. Binder, *Testing Object-Oriented Systems. Models, Patterns, and Tools*]

Losowe testy nie wystarczą

```
boolean equal (int x, y) {
/* effects: returns true if
  x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

- Opis strukturalny
- Strategia testowa:
  - wybierz losowe wartości dla x oraz y i przetestuj ich „równość”
- Ale:
  - ...prawdopodobnie nigdy nie przetestujemy pierwszej gałęzi instrukcji „if”

Losowe testy nie wystarczą

```
int maximum (list a)
/* requires: a is a list of
     integers
     effects: returns the maximum
     element in the list
*/
```

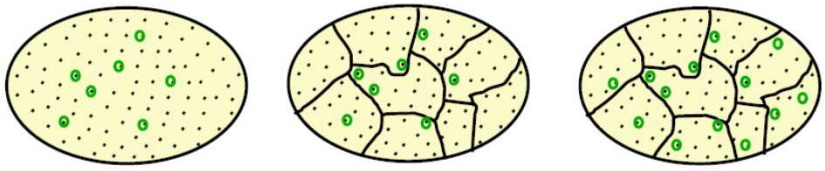
- Opis funkcjonalny
- Strategia testowa:

Input	Output	Correct?
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes
561 13 1024 79 86 222 97	1024	Yes
97 222 86 79 1024 13 561	1024	Yes

- Czy to wystarczy?

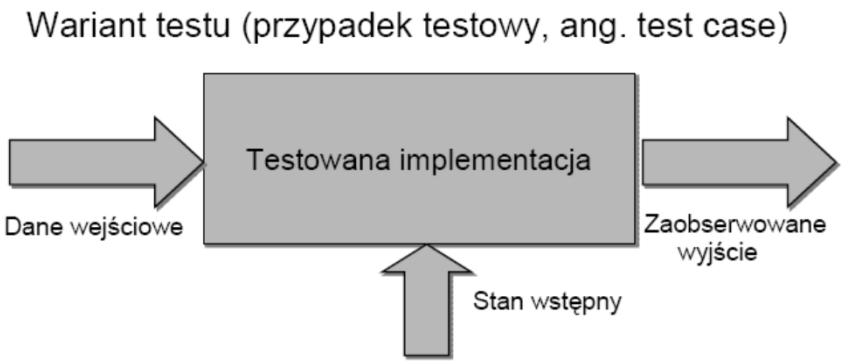
Podejście systematyczne

- Systematyczne testowanie opiera się na partycjonowaniu
  - Dokonaj podziału możliwych zachowań systemu
  - Dla każdej składowej wybierz reprezentatywne próbki
  - Upewnij się, że uwzględnione zostały wszystkie składowe



- Jak zidentyfikować dobry podział?
  - Na tym właśnie polega testowanie!!!
  - Metody wyboru przypadków testowych:
    - czarna skrzynka (black box)
    - przezroczysta skrzynka (white box)
    - ...

Przypadek testowy



11

Czarna skrzynka

- Generowanie przypadków testowych na podstawie specyfikacji
  - Nie patrzymy na kod programu
- Zalety:
  - Unikamy przyjmowania tych samych założeń co programista
  - Dane testowe są niezależne od implementacji
  - Wyniki można interpretować bez wnikania w szczegóły implementacyjne
- Trzy sugestie wyboru przypadków testowych
  - Ścieżki w specyfikacji
    - wybierz przypadki testowe pokrywające każdą z klauzul „wymaga”, „modyfikuje”, „wpływa na” w specyfikacji
  - Warunki brzegowe
    - Wybierz przypadki testowe dla warunków brzegowych zakresu danych wejściowych (lub blisko nich)
    - Szukaj błędów w aliasach (dwa parametry odnoszące się do tego samego obiektu)
  - Przypadki nienominalne
    - Wybieraj testy, które próbują każdego typu niepoprawnych danych wejściowych (program powinien elegancko obsłużyć taki przypadek, bez utraty danych)

12

Przykład

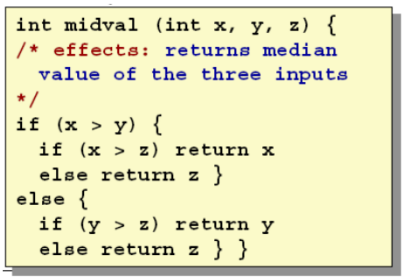
```
char * triangle (unsigned x, y, z) {
/* effects: If x, y and z are the lengths of the sides of a
  triangle, this function returns one of three strings,
  "scalene", "isosceles" or "equilateral" for the given
  three inputs.
*/
```

- Ile przypadków testowych wystarczy?
  - Testy oczekiwane (jeden dla każdego typu): (3,4,5), (4,4,5), (5,5,5)
  - Testy brzegowe (prawie trójkąt): (1,2,3)
  - Testy nienominalne (niepoprawny trójkąt): (4,5,100)
  - Zmiana kolejności wejść dla testów oczekiwanych: (4,5,4), (5,4,4)
  - Zmiana kolejności wejść dla testów granicznych: (1,3,2), (2,1,3), (2,3,1), (3,2,1), (3,1,2)
  - Zmiana kolejności wejść dla testów nienominalnych: (100,4,5), (4,100,5)
  - Wybór dwóch równych parametrów dla testów nienominalnych: (100,4,4)

Przezroczysta skrzynka

- Badanie kodu i testowanie ścieżek w kodzie
  - …ponieważ testowanie czarnej skrzynki nigdy nie gwarantuje, że wykonaliśmy wszystkie fragmenty kodu
- Kompletność instrukcji:
  - Zestaw testów pokrywa komplet instrukcji jeśli każda instrukcja w kodzie jest wykonywana co najmniej raz w zestawie testów
- Kompletność ścieżek:
  - Zestaw testów pokrywa komplet ścieżek jeśli każda ścieżka w kodzie jest wykonana co najmniej raz w zestawie testów

Przykład



```
int midval (int x, y, z) {
/* effects: returns median
  value of the three inputs
*/
if (x > y) {
  if (x > z) return x
  else return z }
else {
  if (y > z) return y
  else return z } }
```

- W kodzie są 4 ścieżki
  - …potrzebujemy więc co najmniej 4 przypadków testowych, np.
    - x=3, y=2, z=1
    - x=3, y=2, z=4
    - x=2, y=3, z=2
    - x=2, y=3, z=4

15

Ile kodu jest pokryte przez testy?

- Pokrycie instrukcji (ang. statement coverage)
  - Każda instrukcja jest sprawdzana
- Pokrycie gałęzi (ang. branch coverage)
  - Każda gałąź była odwiedzona
  - Instrukcja warunkowa musi być przynajmniej raz prawdziwa i przynajmniej raz fałszywa

16

Przykład 100% pokrycia instrukcji i gałęzi

- Pokrycie instrukcji

```
void func(int liczba) {
  if ((liczba % 2) == 0)
   System.out.println(„liczba parzysta");
  for (; liczba < 5; liczba++)
   System.out.println(„liczba "+liczba);
}
```

Wystarczy przypadek testowy z liczba = 4

- Pokrycie gałęzi

```
void func(int liczba) {
  if ((liczba % 2) == 0)
   System.out.println(„liczba parzysta");
  for (; liczba < 5; liczba++)
    System.out.println(„liczba "+liczba);
}
```

Dwa przypadki testowe: liczba = 4 i liczba = 13

17

Słabość metody badającej kompletność ścieżki

- Testowanie metodą białej skrzynki nie wystarczy
  - Np.

```
int midval (int x, y, z) {
/* effects: returns median
   value of the three inputs
*/
return z; }
```

  - Pojedynczy przypadek testowy x=4, y=1, z=2 daje komplet ścieżek w programie
    - program działa poprawnie w danym przypadku testowym
    - ale program nie jest poprawny!!
- Pokrycie kompletu ścieżek jest zazwyczaj niemożliwe
  - Np.

```
for (j=0, i=0; i<100; i++)
  if a[i]=true then j=j+1
```

  - Istnieje 2^100 ścieżek przez dany segment programu
  - Problemem są pętle:
    - Testujemy 0, 1, 2, n-1, aż do n iteracji (n jest maksymalną możliwą ilością iteracji)
    - Lub próbujemy przeprowadzić analizę formalną (znaleźć niezmiennik pętli!!)

18

Jakie powinno być testowanie?

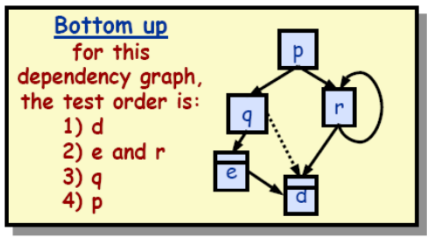
- Powtarzalne
  - W przypadku znalezienia błędu należy móc powtórzyć test by pokazać błąd innym zainteresowanym osobom
  - W przypadku naprawy błędu należy powtórzyć test by upewnić się, że dany błąd już nie występuje
- Systematyczne
  - Losowe testowanie nie wystarcza
  - Należy wybierać zestawy testów pokrywających cały zakres działania programu
  - Należy wybierać testy które są reprezentatywne dla rzeczywistych zastosowań systemu
- Dokumentowane
  - Należy śledzić jakie testy zostały przeprowadzone i jakie osiągnięto wyniki

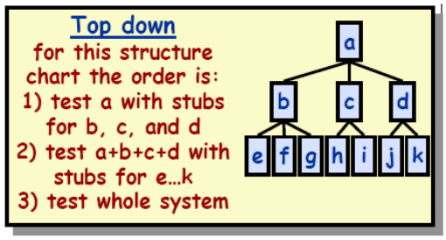
Testy jednostkowe

- Spopularyzowane przez Extreme Programming
- Każda jednostka (np. klasa, pakiet) jest testowana oddzielnie
  - Sprawdzamy czy spełnia specyfikację
  - Nie ma modułu bez testu
- Pisz test, zanim napiszesz moduł
- Przypadek użycia → przypadek testowy

Testy integracyjne

- Testy integracji
  - Testujemy czy moduły współpracują ze sobą
  - Strategie:





- Testowanie integracji jest trudne:
  - Dużo trudniej zidentyfikować klasy równoważności
  - Pojawiają się problemy skali
  - Często wykrywamy błędy specyfikacji a nie błędy integracji

21

Testy systemowe

- Testy funkcjonalności
  - Warunki testowe
  - Skrypty testowe
  - Dane testowe

22

Testy systemowe

- Testy pozafunkcjonalne
  - facility testing – Czy system zapewnia wszystkie wymagane funkcje?
  - volume testing – Czy system radzi sobie z dużą ilością danych?
  - stress testing – Czy system radzi sobie z dużym obciążeniem?
  - endurance testing – Czy system zachowuje parametry w długim okresie?
  - usability testing – Czy system jest łatwy w użyciu?
  - security testing – Czy system wytrzymuje ataki?
  - performance testing – Jak dobry jest czas reakcji systemu?
  - storage testing – Czy pojawiają się problemy ze składowaniem danych?
  - configuration testing – Czy system działa na wszystkich platformach?
  - installability testing – Czy da się skutecznie zainstalować system?
  - reliability testing – Jak zmienia się niezawodność systemu w czasie?
  - recovery testing – Jak skutecznie system podnosi się po awarii?
  - serviceability testing – Czy daje się pielęgnować system?
  - documentation testing – Czy dokumentacja jest dokładna? Adekwatna?
  - operations testing – Czy instrukcje operatorów są poprawne?

Testowanie regresji

- Testowanie regresji
  - Ponowne wykonanie opracowanych wcześniej testów
- Test uruchomieniowy (ang. *smoke test*)
  - Czy program nadal się uruchamia?
  - Wersja minimalna testowania regresywnego
- Testy powinny być powtarzane po każdej ważnej modyfikacji programu!

Automatyzacja testów

- W świecie idealnym – testy wpełni zautomatyzowane
  - można łatwo powtarzać testy po każdej modyfikacji kodu (testowanie regresji)
  - mniejszy wysiłek prowadzenia testów
  - bardziej rozległe testowanie
- Potrzebne
  - Motor testów
    - automatyzuje proces uruchamiania zestawu testów
    - definiuje środowisko
    - wykonuje odwołania do testowanego modułu
    - zapisuje wyniki i sprawdza ich poprawność
    - generuje podsumowanie dla deweloperów
  - Przypadek testowy
    - symuluje fragment programu wywoływanego przez testowany moduł
    - sprawdza czy motor poprawnie zainicjował środowisko
    - sprawdza czy motor wprowadził sensowne parametry
    - zwraca wartości wynikowe (zgodne z przypadkiem testowym)
    - może być interaktywny i prosić użytkownika o dostarczenie wartości

Przykład narzędzia

```
public static TestSuite() {
  TestSuite suite= new TestSuite();

  suite.addTest(new MoneyTest("testMoneyEquals"));
  suite.addTest(new MoneyTest("testBagEquals"));
  suite.addTest(new MoneyTest("testSimpleAdd"));

  return suite;
}
```

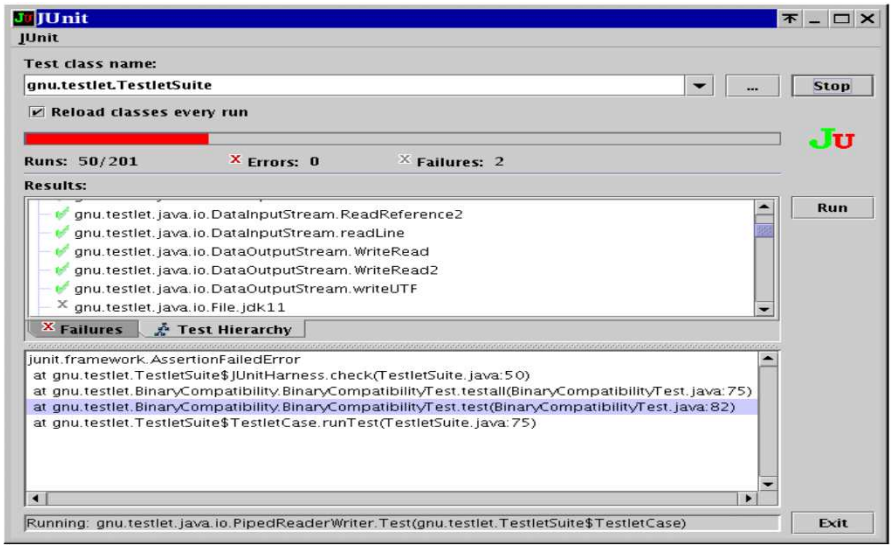
Przykład narzędzia

```
public class MoneyTest extends TestCase {
    //...
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");
        Money m14CHF= new Money(14, "CHF");

        Money expected= new Money(26, "CHF");
        Money result=m12CHF.add(m14CHF);

        Assert.assertTrue(expected.equals(result));
    }
}
```

27

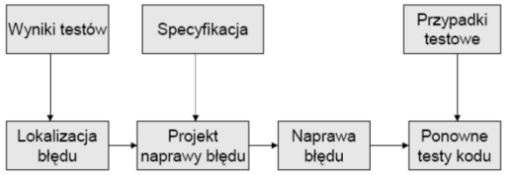
Przykład narzędzia

- JUnit



28

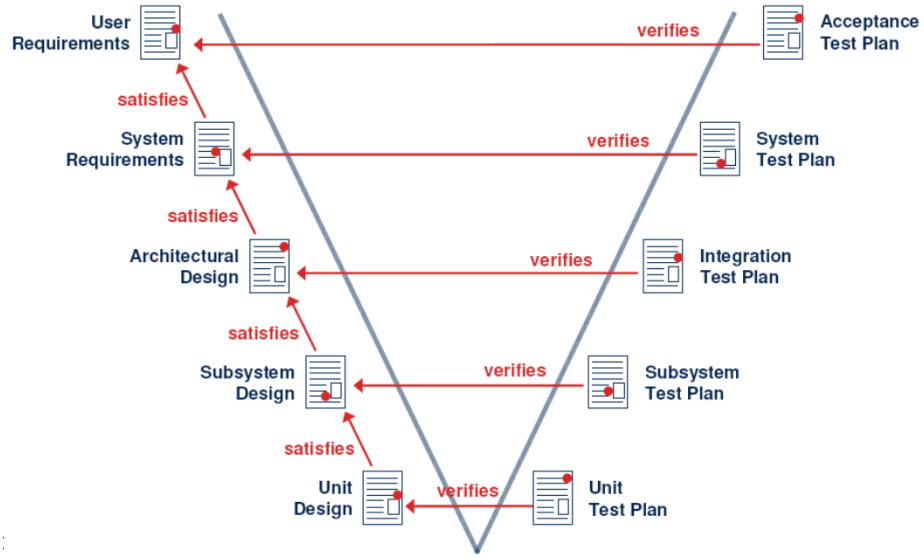
Testowanie != Debugowanie



- Testowanie
  - koncentruje się na znajdywaniu błędów
- Debugowanie
  - zajmuje się ich lokalizacją i usuwaniem

29

Kiedy testować



Aksjomaty testowania

- Nie można założyć, że z poszczególnych poprawnych części zawsze powstaje poprawna całość
  [Elaine Weyuker, AT&T]
- Zestaw testów pokrywający jedną implementację danej specyfikacji nie musi pokrywać jej innej implementacji
  [Antyekstensjonalność]
- Pokrycie uzyskane dla testowanego modułu nie zawsze jest uzyskane dla modułów przez niego wywoływanych
  [Antydekompozycja]
- Zestawy testów, z których każdy osobno jest adekwatny dla elementów modułu, niekoniecznie są odpowiednie dla modułu jako całości
  [Antykompozycja]

31

Samo testowanie nie wystarczy

„Testowanie może ujawnić obecność błędów, ale nigdy ich brak”
[Edsger Dijkstra]

32

- Inspekcje

Inspekcje

- Bardzo efektywna technika znajdowania błędów
- Angażują ludzi do przeglądania źródłowej reprezentacji systemu
- Nie wymagają uruchomienia systemu, więc mogą być użyte przed jego stworzeniem
- Mogą być zastosowane dla dowolnej reprezentacji systemu (wymagania, projekt itd.)
- Poziom formalizmu
  - Nieformalne: od spotkań przy kawie po regularne spotkania zespołów
  - Formalne: harmonogram spotkań, przygotowani uczestnicy, zdefiniowany przebieg spotkania, spisana dokumentacja spotkania

Zalety inspekcji

- W przypadku projektów programistycznych:
  - Większość programów po przeglądach działa poprawnie przy pierwszym użyciu
  - Dla porównania: ponad 10 prób przy podejściu testy - debugowanie
- Informacje pochodzące z dużych projektów
  - Bell-Northern Research:
    - Koszt inspekcji: 1 godzina na usterkę
    - Koszt testowania: 2-4 godziny na usterkę
    - Koszt po wydaniu: 33 godziny na usterkę
  - Redukcja usterek 5-krotna (niekiedy nawet 10-krotna)
  - Zwiększenie produktywności: 14% do 25%
  - Ilość błędów wykrytych w trakcie inspekcji: 58% do 82%
- Wpływ na kompetencje zespołu:
  - Zwiększone morale, mniejsza rotacja
  - Lepsze oszacowania i harmonogramowanie (więcej wiedzy o profilach usterek)
  - Lepsze rozpoznanie możliwości zespołu

35

Ograniczenia inspekcji

- Ilość osób
  - Min = 3, max = 7
- Czas
  - Nie dłużej niż 2 godziny (spada koncentracja)
- Wyniki
  - Wszyscy muszą zgodzić się co do wyniku
  - Szczegółowa lista zagadnień + podsumowanie (raport)
- Zakres
  - Skupiamy się na niewielkim fragmencie projektu, nie na całości
  - Rzędu 150 LOC / h
- Harmonogram
  - Przegląd produktu po zakończeniu prac przez autora
  - Nie nazbyt wcześnie
    - Produkt nie gotowy – znajdziemy problemy, których autor jest świadom
  - Nie nazbyt późno
    - Produkt w użyciu – kosztowna naprawa błędów

36

Wytyczne dla inspekcji

- Przed przeglądem
  - Przeglądy formalne należy uwzględnić w planie projektu
  - Należy przeszkolić wszystkich uczestników
  - Należy umożliwić uczestnikom przygotowanie się z wyprzedzeniem
- Podczas przeglądu
  - Dokonujemy przeglądu produktu, nie autora!!!
  - Komentarze konstruktywne, zorientowane na zadanie
  - Należy trzymać się planu spotkania!!!
  - Lider musi utrzymywać kierunek dyskusji
  - Należy ograniczać debaty / dygresje
  - Należy notować zagadnienia na potrzeby przyszłej dyskusji
  - Należy identyfikować problemy ale ich nie rozwiązywać!!!
  - Należy robić notatki!!!
- Po przeglądzie
  - Należy dokonać przeglądu procesu inspekcji

Przebieg inspekcji

- 1 Wstęp
  - Wymiana informacji o module
  - Rozdanie materiałów
  - Współczynnik: 500 LOC na godzinę
- 2 Przygotowanie
  - Wszyscy uczestnicy pracują indywidualnie
  - Przeglądają materiały w poszukiwaniu usterek
  - Współczynnik: 100 LOC na godzinę
- 3 Inspekcja
  - Autor przedstawia projekt
  - Dyskusja: identyfikacja i zapisanie problemów (bez rozwiązań)
  - Współczynnik: 150 LOC na godzinę
- 4 Przeróbki
  - Wszystkie błędy/problemy rozwiązywane przez autora
  - Współczynnik: 16-20 godzin na 1000 LOC
- 5 Podsumowanie
  - Moderator upewnia się, że poprawiono wszystkie błędy
  - Jeśli więcej niż 5% przeróbek, wówczas produkt podlega ponownej inspekcji; dokonuje zespół z pierwszej inspekcji

Sposób organizacji dyskusji

- Checklisty
  - Przegląd odbywa się według listy pytań / punktów kontrolnych
- Produkty
  - Przegląd odbywa się według modułów / produktów
  - Jedna osoba prezentuje każdy moduł (krok po kroku)
- Szybki przegląd
  - Każdy uczestnik przeglądu ma 3 minuty na przejrzenie rozważanego fragmentu, następnie przekazuje inicjatywę kolejnej osobie
- Dodatkowo:
  - Adwokat diabła
    - celowa próba przyjęcia przeciwnego stanowiska
  - Zapluskwianie
    - celowe umieszczanie błedów
    - nagroda za ich znalezienie!!