

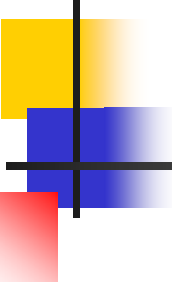
Zagadnienia

- Podejścia do procesu testowania
- Testowanie typu „black box”
- Testowanie typu „white box”
- Weryfikacja statyczna - review



Cel

- Celem testowania jest wykrycie istniejących błędów w oprogramowaniu
- Dobry test → taki który pokazuje że program w danej sytuacji nie funkcjonuje prawidłowo
- Testy dowodzą istnienia błędów, nigdy – faktu że dany system jest wolny od błędów
 - W praktyce niemożliwe ze względu na złożoność dowodów, możliwość istnienia błędów w samym dowodzie itp.



Co testować?

- Aby wykazać że dany program nie posiada błędów, trzeba przeprowadzić wszystkie możliwe testy
 - Niemożliwe w praktyce
- Przy testowaniu ważniejsze jest sprawdzenie funkcjonowania systemu jako całości od weryfikacji działania poszczególnych komponentów
- Dotychczasowo istniejące funkcjonalność – ważniejsza niż nowa!! ← [Użytkownicy](#)
- Testowanie typowych przypadków użycia systemu – ważniejsze niż sprawdzanie przypadków skrajnych



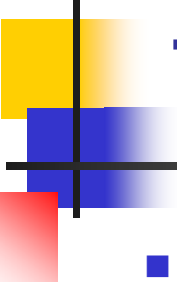
Dane testowe a test case'y

- **Dane testowe** – pewne dane wejściowe systemu
- **Test case** – składa się z danych testowych oraz opisu spodziewanych wyników
 - Przy założeniu że system funkcjonuje zgodnie ze swoją specyfikacją



Testowanie typu white box

- Inne spotykane w literaturze określenia (ang.):
 - glass box testing
 - structural testing
 - clear box testing
 - open box testing
- Technika testowania w której do wyboru danych testowych wykorzystuje się wiedzę na temat budowy testowanego komponentu ← stąd nazwa
- Znajomość kodu komponentu jest również niezbędna do interpretacji wyników danego testu
- Aby test był miarodajny, tester musi znać zaplanowaną funkcjonalność danego komponentu



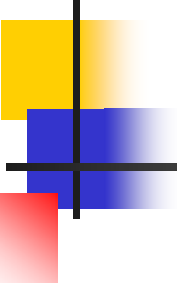
Testowanie typu white box

- Strategie testowania typu white box w swoim założeniu mają na celu
 - Co najmniej jednokrotne wykonanie każdej instrukcji kodu źródłowego systemu
 - Indywidualne przetestowanie każdej procedury/funkcji wchodzącej w skład systemu
- Strategie te nie są w stanie wykryć błędów spowodowanych pominięciem funkcjonalności
 - Testują tylko istniejący kod !
- Najpowszechniej używane w nast. typach testów
 - unit testy, testy integracyjne



Metody testowania typu white box

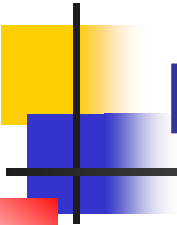
- Pokrycie wyrażeń (*ang. statement coverage*)
- Pokrycie rozgałęzień (*ang. branch coverage*)
 - Dane testowe mają zapewnić przejście wszystkich rozgałęzień w instrukcjach warunkowych
- Pokrycie warunków (*ang. branch condition coverage*)
 - Dane testowe mają zapewnić że wszystkie elementarne warunki w złożonych instrukcjach warunkowych dadzą w wyniku dwie możliwe wartości: T, F
- Testowanie mutacyjne (*ang. mutation testing*)
 - Sposób weryfikacji jakości samych testów
 - Dla danych testowych → wprowadzenie mutacji + test



Pokrycie wyrażeń

- Cechy metody

- Każda instrukcja kodu jest wykonana con. 1 raz
- Gwarantuje pełne pokrycie kodu
- Najprostsza z metod typu white box
- Nie analizuje szczegółowo złożonych wyrażeń logicznych w instrukcjach warunkowych
 - Analiza tylko do momentu uzyskania pokrycia



Pokrycie wyrażeń na przykładzie

Problem: wygenerować dane testowe które spowodują przejście sterowania po wszystkich wyrażeniach następującego fragmentu kodu:

```
if((a > 0 and b < 10) or (a < -5 and b > 1))  
{  
    s = 0;  
}  
t = 100;
```

Rozwiązanie: $a = 1$, $b = 5$.

- $a > 0$ and $b < 10 = \text{TRUE}$
- cały warunek = TRUE
- sterowanie wchodzi w blok instrukcji warunkowej

Pokrycie wyrażień na przykładzie

```
LRESULT Comdir_OnTimer(hwnd, message)
HWND      hwnd;
UINT      message;
{
    switch (message) {
        case STTC_TYPE:
            STCTimerProc();
            break;
        case DDSP_TYPE:
            DDSPTimerProc();
            break;
        case TABLES_TYPE:
            TABLESTimerProc();
            break;
    }
    return 0;
}
```

TEST 1,2,3

TEST 1,2,3

TEST 1

TEST 1

TEST 2,3

TEST 2

TEST 2

TEST 3

TEST 3

TEST 3

TEST 1,2,3

No. Test case	message
1.	STTC_TYPE
2.	DDSP_TYPE
3.	TABLES_TYPE



Metody testowania typu BlackBox

- Podział na klasy równoważności (ang. equivalence partitioning)
- Analiza wartości brzegowych (ang. boundary value analysis)
- Finite-state testing
 - Testowanie systemów wyspecyfikowanych bądź nmodelowanych jako maszyna stanów
- Kontrola składni (ang. syntax checking)
 - Przydatne przy testowaniu parserów, itp.



Metody testowania typu BlackBox – podział na klasy równoważności

Opis

- Istotą metody jest pomysł aby wszystkie możliwe zakresy zmiennych wejściowych podzielić na tzw. klasy równoważności. Dla danej klasy równoważności, wszystkie wartości danej zmiennej wejściowej pochodzące z tej klasy powinny być przetwarzane w taki sam względnie podobny sposób. W wyniku tego zakres możliwych wartości parametrów wejściowych zmniejsza się do rozsądnego (możliwego do przetestowania) rozmiaru.

Cechy

- Zmniejszenie rozmiaru zbioru możliwych wartości danych wejściowych
- Nie podaje sposobu wyboru konkretnych danych testowych
- Trudne w użyciu w przypadku złożonego przetwarzania danych
- Założenie że pewien podzbiór danych wejściowych jest obsługiwany w identyczny sposób – niekoniecznie prawdziwe
- Uwaga: metoda ta ma zastosowanie do funkcji której dane wejściowe są od siebie niezależne – więc nie muszą być testowane we wszelkich możliwych kombinacjach

Podział na klasy równoważności (1/3)

- Testowany obiekt: funkcja pobierająca dwa argumenty wejściowe, X i Y. X ma znajdować się w przedziale 10..30, Y ma znajdować się w przedziale 40..80.
- **Krok 1.** Dla każdej danej wejściowej określ zbiór klas równoważności opisując każdą klasę. Testowany program powinien zachowywać się „tak samo” (ściśle: w podobny sposób) dla wszystkich wartości danej wejściowej wchodzących w skład danej klasy równoważności.
- W przypadku naszego systemu, bez żadnych dodatkowych informacji, dla X i Y wyznaczamy nast. klasy równoważności:

X	Y
[10,30] (Poprawna)	[40,80] (Poprawna)
<10 (Niepoprawna)	<40 (Niepoprawna)
>30 (Niepoprawna)	>80 (Niepoprawna)

Podział na klasy równoważności (2/3)

- **Krok 2.** Przygotuj test case'y zawierające każda tak dużo jak to tylko możliwe nieprzetestowanych wcześniej poprawnych klas równoważności. Powtarzaj ten krok aż do całkowitego pokrycia wszystkich poprawnych klas równoważności

		Test case #				
		1	2	3	4	5
X	[10,30]					
	<10					
	>30					
Y	[40,80]					
	<40					
	>80					

		Dane wejściowe	
TC#		X	Y
	1	11	75
	2		
	3		
	4		
	5		
	6		
	7		

Podział na klasy równoważności (3/3)

- **Krok 3.** Przygotuj test case'y pokrywające każdą jedną i tylko jedną z nieprzetestowanych wcześniej, niepoprawnych klas równoważności.

		Test case #				
		1	2	3	4	5
X	[10,30]					
	<10					
	>30					
Y	[40,80]					
	<40					
	>80					

		Dane wejściowe	
TC#		X	Y
	1	11	75
	2	15	21
	3	12	110
	4	-7	50
	5	45	70
	6		
	7		



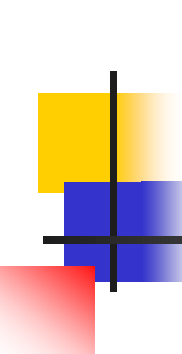
Metody testowania typu BlackBox – Analiza wartości brzegowych

Opis

- Metodę tę wykorzystuje się w połączeniu z podziałem na klasy równoważności. Po przeprowadzeniu podziału danych wejściowych na klasy równoważności dane testowe wyznacza się biorąc pod uwagę wartości graniczne poszczególnych klas.

Cechy

- Metoda ta wykazuje podobne cechy jak metoda podziału na klas równoważności z tym że sugeruje sposób wyboru poszczególnych wartości danych testowych
- Graniczne wartości poszczególnych klas równoważności często powodują ujawnienie się błędów



Analiza wartości brzegowych na przykładzie (1/2)

- Testowany system: czajnik z elektronicznym czujnikiem temperatury (zakres pomiaru: od 0.0 st. C do 120.0 st. C, z dokładnością do 0,2 st. C), emitujący dźwięk w przypadku przekroczenia temperatury 100.0 st. C
- Które wartości brzegowe należy wyznaczyć jako dane testowe?
- **Krok 1.** Dla każdej danej wejściowej wyznacz klasy równoważności, poprawne i niepoprawne
 - $<0.0, 100.0>$ (Poprawna)
 - $<100.1, 120.0>$ (Poprawna)
 - < 0.0 (Niepoprawna)
 - > 120.0 (Niepoprawna)

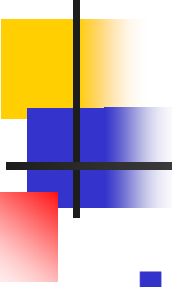


Analiza wartości brzegowych na przykładzie (2/2)

- **Krok 2.** Dla każdej granicy klasy równoważności utwórz dane testowe umieszczone możliwie najbliżej tej granicy, po obu jej stronach.
- Poprawne klasy równoważności powinny być testowane razem (możliwie najwięcej / 1 test case)
- Niepoprawne klasy → pojedynczo (1/1 test case)

Wartości graniczne:

- Spoza zakresu → -0.1, 120.1
- W obrębie zakresu → 0.0, 120.0, oraz dodatkowo
 - Próg emisji dźwięku → 100.1 (emisja dźwięku)
 - Próg emisji dźwięku → 100.0 (brak emisji dźwięku)



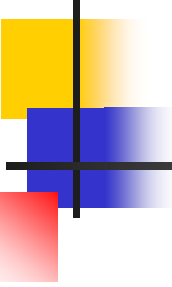
Weryfikacja statyczna

- Metodę tą można stosować do wszelkich dokumentów powstałych w trakcie tworzenia systemu
- Cecha: pomaga identyfikować błędy we wczesnych etapach projektu
- Zwykle jest to tańsza metoda identyfikacji defektów niż późniejsze testowanie (unit testing, itd.)
- Pozwala na łączenie identyfikacji błędów z innymi rodzajami weryfikacji jakości



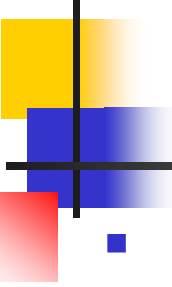
Efektywność procesu weryfikacji

- Przy wykorzystaniu nieformalnych inspekcji kodu wykrywa się ok. 60% błędów tam zawartych
- Przy podejściu bardziej sformalizowanym (metody matematyczne) wykrywa się ok. 90% błędów kodu



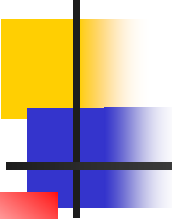
Inspekcje kodu

- Ma na celu identyfikację błędów, a nie ich poprawę
 - Ustala się tylko fakt wystąpienia błędu, nie sugerując sposobu w jaki należy go usunąć
- Identyfikowane błędy
 - Błędy logiczne
 - Cechy kodu genrujące błędy sterowania, np. niezainicjalizowane zmienne
 - Niezgodność z przyjętymi standardami (np. kodowania)



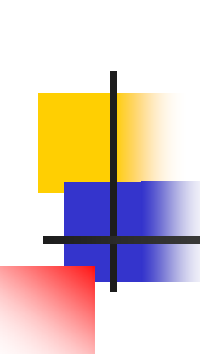
Inspekcja – warunki wejściowe

- Dokładna specyfikacja programu
- Osoby biorące udział w inspekcji powinny znać przyjęte w organizacji standardy
- Kod programu, poprawny składniowo
 - Musi się przynajmniej kompilować
- Lista najczęściej popełnianych błędów (*ang. error checklist*)
 - Uzupełniana poprzez inspekcje !
- Wyższe koszty we wczesnych etapach procesu tworzenia oprogramowania
- Wyniki inspekcji
 - Mają służyć poprawie jakości kodu
 - Nie ocenie developerów !



Etapy procesu inspekcji

- Planowanie, przegląd (*ang. overview*)
 - Wybór członków zespołu, logistyka (rezerwacja sali itp.)
 - Dystrybucja materiałów (wydruki), zgrubne omówienie materiału
- Indywidualne przygotowanie się
 - Konieczne przed spotkaniem; jeden z warunków „go/no go”
- Spotkanie
 - W wyniku – lista zidentyfikowanych błędów
- Poprawa zidentyfikowanych błędów (*ang. rework*)
 - Przeprowadzana przez autora
- Weryfikacja poprawek (*ang. follow-up*)
 - Przeprowadzana przez moderatora
 - Decyduje on o ew. powtórnej inspekcji



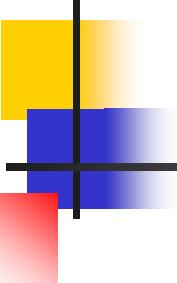
Cechy zespołu prowadzącego inspekcję

- Złożony z con. 4 osób
 - W zasadzie ról → mogą być łączone
- Wśród nich autor kodu
 - Stosunkujący się do zidentyfikowanych błędów
- Prowadzący
 - Czyta kod na głos bądź mówi który fragment kodu aktualnie analizujemy
- Inspektor – 1 lub więcej osób identyfikujących błędy
- Moderator – prowadzący spotkanie, decydujący w kwestiach spornych
- Notujący (*ang. scribe*)



Szybkość procesu inspekcji

- 500 wyrażeń/godz w czasie fazy przeglądu (overview)
- 125 wyrażeń/godz w czasie indywidualnego przygotowywania się
- 90-125 wyrażeń/godz w czasie spotkania
- ➔ Proces inspekcji jest kosztowny
 - Przykładowo inspekcja 500 linii kodu to ok. 40 osobogodzin
 - Koszty ponoszone we wczesnych fazach cyklu życia projektu
 - Zwracają się w postaci mniejszego wysiłku potrzebnego na testowanie i debugowanie w późniejszych fazach



Listy częstych błędów

- Powinny być używane przy prowadzeniu spotkania inspekcyjnego
- Postać listy jest zależna od języka w jakim został napisany kod
- Im słabszy jest mechanizm kontroli typów danego języka, tym obszerniejsza lista 😊
- Np. inicjalizacja zmiennych, nazewnictwo stałych, warunki zakończenia pętli, kontrola granic tablic, itp.



Listy błędów (przykład)

Klasa błędu	Pozycje na liście błędów
Błędy danych	✓ Czy wszystkie zmienne programu zostały zainicjalizowane przed pierwszym użyciem? ✓ Czy wszystkie stałe mają przypisane nazwy? ✓ Czy dolny indeks tablicy powinien wynosić 0,1, ...? ✓ Czy górny indeks tablicy powinien być równy rozmiarowi czy rozmiar-1 ?
Błędy sterowania	✓ Czy w wyrażeniach warunkowych warunki są poprawne? ✓ Czy każda pętla ma jasno określone kryterium jej zakończenia? ✓ Czy złożone wyrażenia są poprawnie otoczone nawiasami? ✓ Czy w wyrażeniach case, zostały uwzględnione wszystkie przypadki?



Listy błędów c.d.

- Lista częstych błędów powinna być uzupełniana na podstawie wyników dotychczasowych inspekcji
- Także powiązane dokumenty
- Przykład:
 - Na kolejnych inspekcjach stwierdzono częste popełnianie błędu w wyrażeniach warunkowych: zamiast
`if (var == CONST)` pojawiało się
`if (var = CONST)`
 - Wniosek: uzupełnienie standardu kodowania o definicję postaci wyrażenia warunkowego:
`if (CONST == var)`
 - Uniemożliwia to popełnienie ww. błędu



Listy błędów c.d. (1/2)

```
#define pl_tolower(c) ( c >64 && c <96 ? c +32 : \  
    ((unsigned char) c == 165 ? 185 : \  
    ((unsigned char) c == 198 ? 230 : \  
    ((unsigned char) c == 202 ? 234 : \  
    ((unsigned char) c == 163 ? 179 : \  
    ((unsigned char) c == 209 ? 241 : \  
    ((unsigned char) c == 211 ? 243 : \  
    ((unsigned char) c == 140 ? 156 : \  
    ((unsigned char) c == 143 ? 159 : \  
    ((unsigned char) c == 175 ? 191 : c )))))))
```



Listy błędów c.d. (2/2)

```
#define pl_tolower(c) ((c)>64 && (c)<96 ? (c)+32 : \  
    ((unsigned char)(c) == 165 ? 185 : \  
    ((unsigned char)(c) == 198 ? 230 : \  
    ((unsigned char)(c) == 202 ? 234 : \  
    ((unsigned char)(c) == 163 ? 179 : \  
    ((unsigned char)(c) == 209 ? 241 : \  
    ((unsigned char)(c) == 211 ? 243 : \  
    ((unsigned char)(c) == 140 ? 156 : \  
    ((unsigned char)(c) == 143 ? 159 : \  
    ((unsigned char)(c) == 175 ? 191 : (c) )))))))
```



Narzędzia do analizy statycznej

- Operują na kodzie źródłowym programu
- Ich zadanie polega na identyfikacji potencjalnych błędów w kodzie
- Bardzo przydatne jako narzędzia wspomagające procesu inspekcji. Nie mogą jednak go zastąpić !



Narzędzia – co potrafią wykryć?

Klasa błędu	Sprawdzane objawy
Błędy danych	✓ Zmienne używane bez inicjalizacji ✓ Zadeklarowane zmienne nie wykorzystywane nigdzie później ✓ Zmienne którym jest dwukrotnie przypisywana wartość oraz nie są używane pomiędzy tymi przypisaniami ✓ Możliwe przekroczenie zakresu tablicy ✓ Niezadklarowane zmienne
Błędy sterowania	✓ Martwy (nieosiągalny) kod ✓ Bezwarunkowe skoki do wnętrza pętli
Błędy interfejsów	✓ Niezgodne typy parametrów ✓ Błędna liczba parametrów ✓ Niewykorzystywanie wyników zwracanych przez funkcję ✓ Niewywoływane funkcje
Błędy w zarządzaniu pamięcią	✓ Niezainicjalizowane wskaźniki



Etapy analizy statycznej (1/2)

- **Analiza przepływu sterowania.** Ma na celu identyfikację pętli z wieloma punktami we/ey, nieosiągalnego kodu, itp.
- **Analiza dostępu do danych.** Wykrywa niezainicjalizowane zmienne, zadeklarowane zmienne niewykorzystywane nigdzie później, dwukrotne przypisanie wartości do zmiennej bez występującego w międzyczasie wykorzystania wartości, itp.
- **Analiza interfejsów.** Sprawdza spójność deklaracji i definicji procedur/funkcji ze składnią ich wywołań.



Etapy analizy statycznej (2/2)

- **Analiza przepływu informacji.** Identyfikuje zależności zmiennych wyjściowych. Generuje informacje przydatne do inspekcji kodu
- **Analiza ścieżek sterowania.** Identyfikuje ścieżki w programie wraz z poleceniami wykonywanymi w obrębie tych ścieżek. Wykorzystanie informacji – w procesie inspekcji.
- Oba powyższe etapy generują olbrzymie ilości informacji. Wyników tych etapów należy używać z zachowaniem zdrowego rozsądku 😊

Przykład narzędzia - LINT

lint_ex.c

```
#include <stdio.h>

printarray (Anarray)
int Anarray;
{
    printf("%d",Anarray);
}

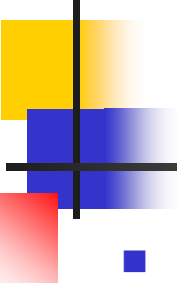
main ()
{
    int Anarray[5];
    int i; char c;
    printarray (Anarray, i, c);
    printarray (Anarray) ;
}
```

Brak błędów!

> **cc lint_ex.c**

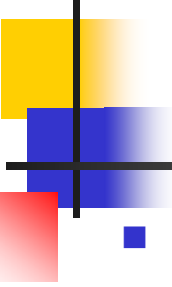
> **lint lint_ex.c**

```
lint_ex.c(10): warning: c may be used
before set
lint_ex.c(10): warning: i may be used
before set
printarray: variable # of args.
    lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used
    inconsistently lint_ex.c(4) ::
    lint_ex.c(10)
printarray, arg. 1 used
    inconsistently lint_ex.c(4) ::
    lint_ex.c(11)
printf returns value which is always
ignored
```



Podsumowanie (1/2)

- Przy testowaniu należy położyć nacisk na komponenty najczęściej używane
- Testowanie typu „black-box” jest oparte o specyfikację systemu
- Testowanie typu „white-box” jest oparte o znajomość struktury kodu
 - Ma na celu skonstruowanie takich testów które wymuszają przejście sterowania po wszystkich możliwych ścieżkach w programie



Podsumowanie (2/2)

- Pokrycie testowe (*ang. test coverage*) określa czy wszystkie wyrażenia w kodzie zostały wykonane con. 1 raz
 - W praktyce niemożliwe jest wykonanie wszystkich możliwych ścieżek
- Weryfikacja (statyczna) jest oparta o analizę kodu źródłowego. Stanowi uzupełnienie a NIE zamiennik procesu testowania
- Narzędzia służące do analizy statycznej kodu
 - Wykrywają nietypowe sekwencje sterujące, konstrukcje, itp. w programie które mogą (niekoniecznie są) przyczyną błędów