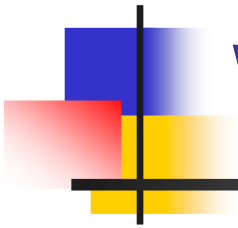
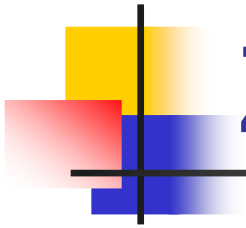


Weryfikacja i walidacja

Metody testowania systemów informatycznych



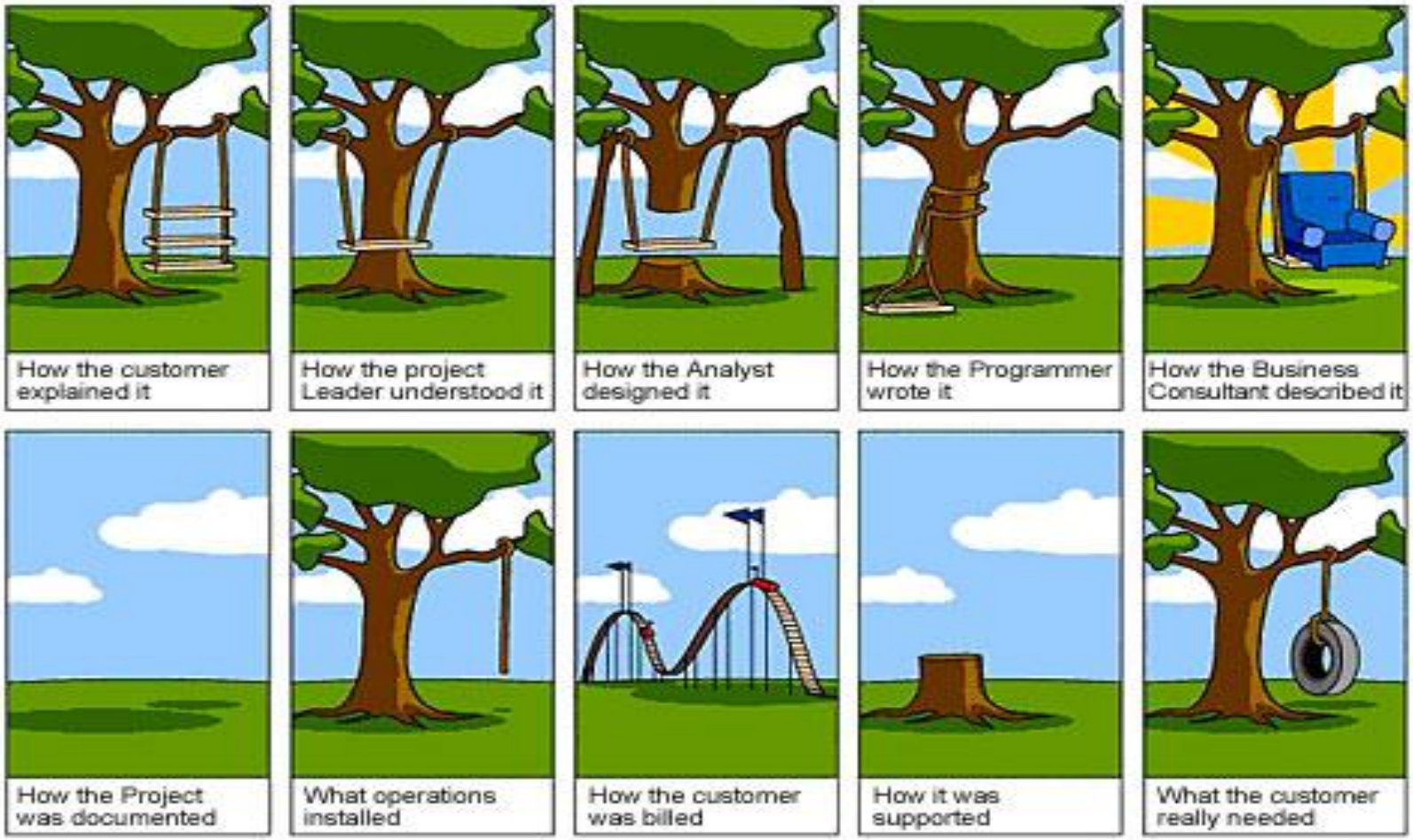
Zagadnienia

- Weryfikacja a walidacja
- Etapy procesu testowania
- Rola planowania w procesie testowania systemów
- Przegląd różnych strategii testowania
- Narzędzia wspomagające proces testowania

Weryfikacja a walidacja

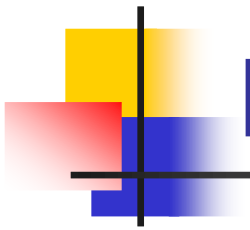
- Weryfikacja (podejście statyczne):
  - „Czy tworzymy system we właściwy sposób"
  - Odpowiada na pytanie czy oprogramowanie odpowiada swojej specyfikacji
- Walidacja (podejście dynamiczne):
  - „Czy tworzymy właściwy system"
  - Odpowiada na pytanie czy tworzony przez nas system działa tak jak tego oczekuje użytkownik

Proces weryfikacji i walidacji



Proces weryfikacji i walidacji

- Proces ten odbywa się na przestrzeni całego cyklu życia projektu. W skład każdego kolejnego etapu cyklu życia projektu musi wchodzić testowanie.
- Dwa podstawowe cele
  - Identyfikacja defektów w obrębie systemu
  - Ocena czy system w warunkach eksploatacji spełnia oczekiwania użytkownika
- Weryfikacja: czy budujemy produkt dobrze
- Walidacja: czy budujemy dobry produkt



Podejście dynamiczne a statyczne

- **Podejście dynamiczne** Skupia się na uruchamianiu i obserwacji zachowania systemu
  - Stosuje się do elementów wykonywalnych systemu, tj. samego programu względnie prototypu (ów)
- **Podejście statyczne** Skupia się na analizie statycznej reprezentacji elementów systemu. Celem jest identyfikacja możliwych problemów
  - Ma zastosowanie do wszystkich produktów (artefaktów) powstających w obrębie systemu
  - Np. dokumenty projektowe, kod, projekty testów (!)

Testowanie oprogramowania

- Ma na celu pokazanie obecności błędów, a nie ich braku !
  - Dowodzenie poprawności programów w praktyce niemożliwe
    - Złożoność
    - Błędy w samym dowodzie
- „Dobry test” – to taki który wykrywa jeden lub więcej błędów
- Jedyny sposób walidacji wymagań niefunkcjonalnych
  - Np. wymagania dot. wydajności
- Testowanie powinno być stosowane wraz z metodami statycznej weryfikacji

Testowanie a usuwanie błędów

- Są to dwa różne procesy
- Testowanie ma na celu stwierdzenie istnienia błędów w oprogramowaniu
- Usuwanie błędów (*ang. debugging*, polski slang - debugowanie) ma na celu identyfikację i usunięcie tych błędów
- Podczas debugowania zachodzi konieczność sformułowania hipotezy dlaczego program zachowuje się niewłaściwie a następnie jej sprawdzenia
  - Przydatna tu jest znajomość typowych błędów programistycznych (np. niezwalnianie pamięci)

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

Etapy procesu testowania

- Testowanie modułów (*ang. unit testing*)
  - Testowanie poszczególnych komponentów systemu w izolacji od pozostałych
- Testowanie integracyjne (*ang. integration testing*)
  - Testowanie interfejsów współpracujących ze sobą modułów lub podsystemów
- Testowanie systemowe (*ang. system testing*)
  - Szczegółowe testowanie funkcjonalności całego, kompletnego systemu
- Testowanie akceptacyjne (*ang. acceptance testing*)
  - Wykonywane przez/w obecności klienta/użytkownika w celu stwierdzenia czy system spełnia wymagania. Zwykle podzbiór testów systemowych
- Testowanie regresyjne (*ang. regression testing*)
  - Ma na celu identyfikację błędów wprowadzonych do istniejącej już i testowanej wcześniej funkcjonalności, np. przy okazji nanoszenia poprawek

Etapy testów systemowych

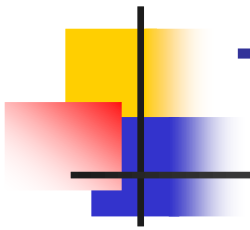
- **Testy systemowe** – wykonywane w środowisku w którym system został utworzony (*ang. development environment*)
- **Alfa testy (*ang. alpha testing*)** – wykonywane w docelowym środowisku, w którym system będzie pracował; testujący - zespół testerów
- **Beta testy (*ang. beta testing*)** – docelowe środowisko pracy systemu (u użytkownika); testujący - użytkownik

Planowanie testów

- Wyznaczenie
  - Głównych faz procesu testowania
  - Odwzorowania testów systemowych na wymagania (!) ← źródło testów
- Oszacowanie planu czasowego oraz potrzebnych zasobów
- Uzgodnienie planu testów z innymi planami projektowymi
  - Eliminacja konfliktów
- Zaplanowanie metody zapisu wyników testów
  - Np. w test case'ach, pole „expected result"

Test plan - zawartość

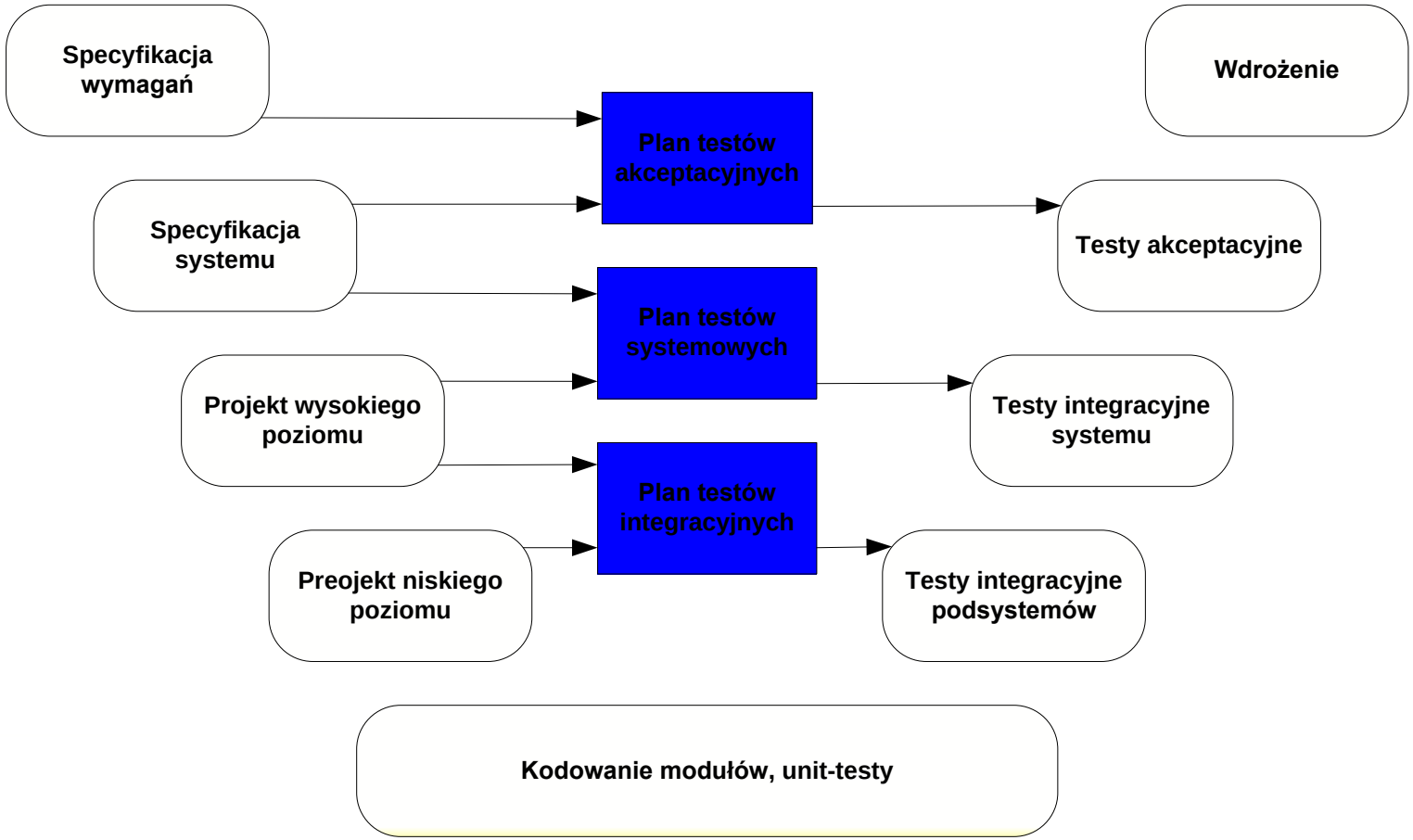
- Opis procesu testowania
- Odwzorowanie testów na wymagania (*ang. requirements traceability*)
  - Weryfikacja pokrycia wymagań
- Wyszczególnienie co będzie podlegać testowaniu
- Plan czasowy
- Procedury przeprowadzania testów
  - Zachowywanie wyników testów
- Wymagania sprzętowe i programowe
- Znane ograniczenia

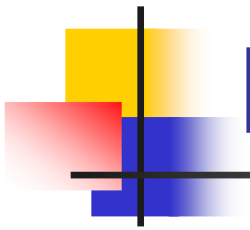


Testerzy

- Kto powinien testować?
- Programista – unit testy
  - Naturalna tendencja „chronienia” swojego „utworu”
  - Ale nie swój kod – daj to komuś innemu, ty przetestuj czyjś
- Specjalista tester – testy systemowe, integracyjne, alfa testy
- Użytkownik systemu – beta testy, testy akceptacyjne systemu
  - Osobiście lub nadzorując

Testy w modelu V cyklu życia projektu





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

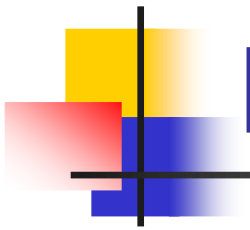
- Inne spotykane w literaturze określenia (ang.):
  - glass box testing
  - structural testing
  - clear box testing
  - open box testing
- Technika testowania w ktrórej do wyboru danych testowych wykorzystuje się wiedzę na temat budowy testowanego komponentu ← stąd nazwa
- Znajomość kodu komponentu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

Testowanie typu white box

- Strategie testowania typu white box w swoim założeniu mają na celu
  - Conajmniej 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 min. 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* = TRUE
- cały warunek = TRUE
- sterowanie wchodzi w blok instrukcji warunkowej

Pokrycie wyrażeń na przykładzie

```
LRESULT Comdir_OnTimer(hwnd, message)
HWND    hwnd;
UINT    message;
{
  switch (message) {          TEST 1,2,3
    case  STTC_TYPE:          TEST 1,2,3
      STTCTimerProc();        TEST 1
      break;                  TEST 1
    case  DDSP_TYPE:          TEST 2,3
      DDSPTimerProc();        TEST 2
      break;                  TEST 2
    case  TABLES_TYPE:        TEST 3
      TABLESTimerProc();      TEST 3
      break;                  TEST 3
  }
  return 0;                   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 modelowanych 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 rónoważności 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 każda 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 0,1 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 w trakcie inspekcji !
- 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 zidentryfikowanych 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

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

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 ✓ Niezadeklarowane 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/wy, 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 scieżki 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) ;
}
```

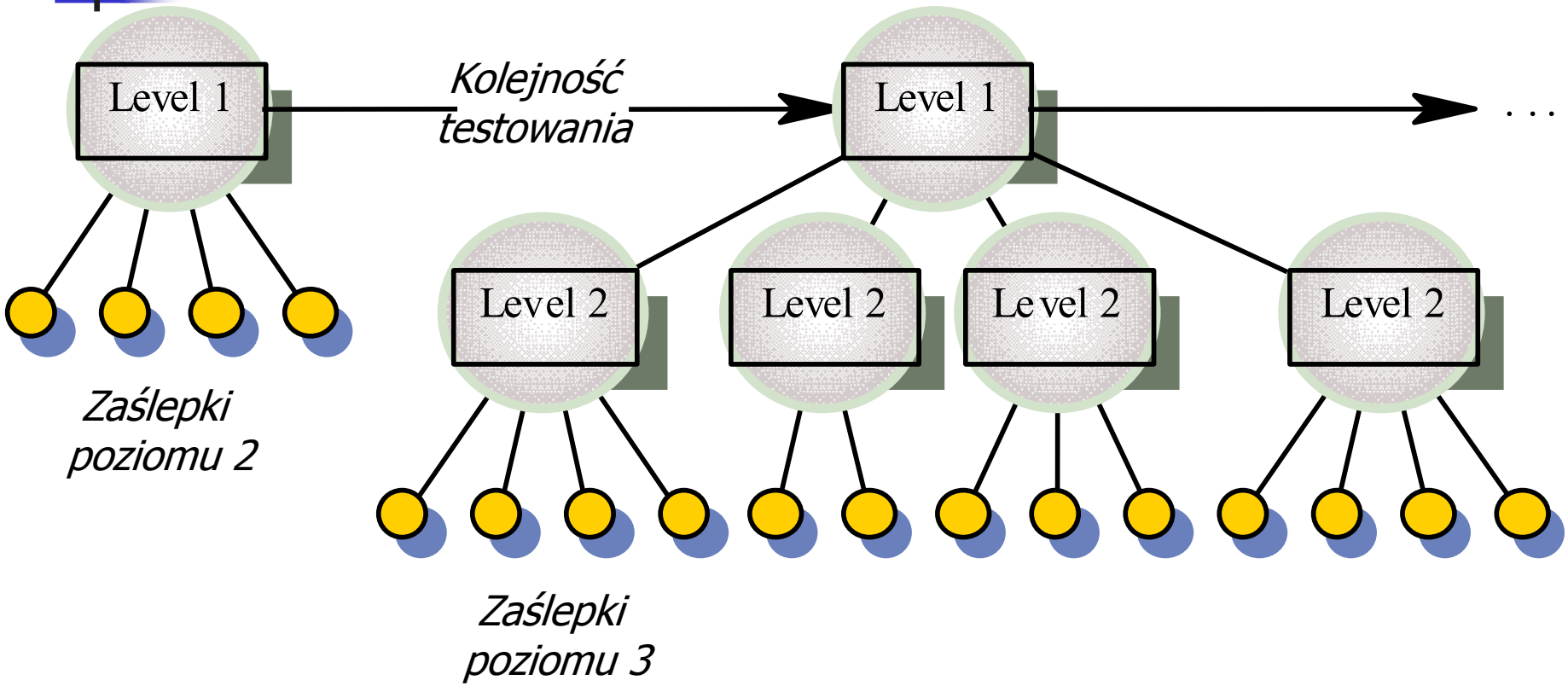
Brak błędów!

```
> 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
```

Strategie testowania

- Istnieją różne sposoby podejścia do procesu testowania
- Zwykle w różnych fazach procesu testowania stosuje się różne strategie
  - Najbardziej odpowiednie dla danej fazy
- Przykłady strategii
  - Podejście typu „top-down”
  - Podejście typu „bottom-up”
  - Testowanie przeciążeniowe (*ang. stress testing*)

Testowanie metodą „top-down”

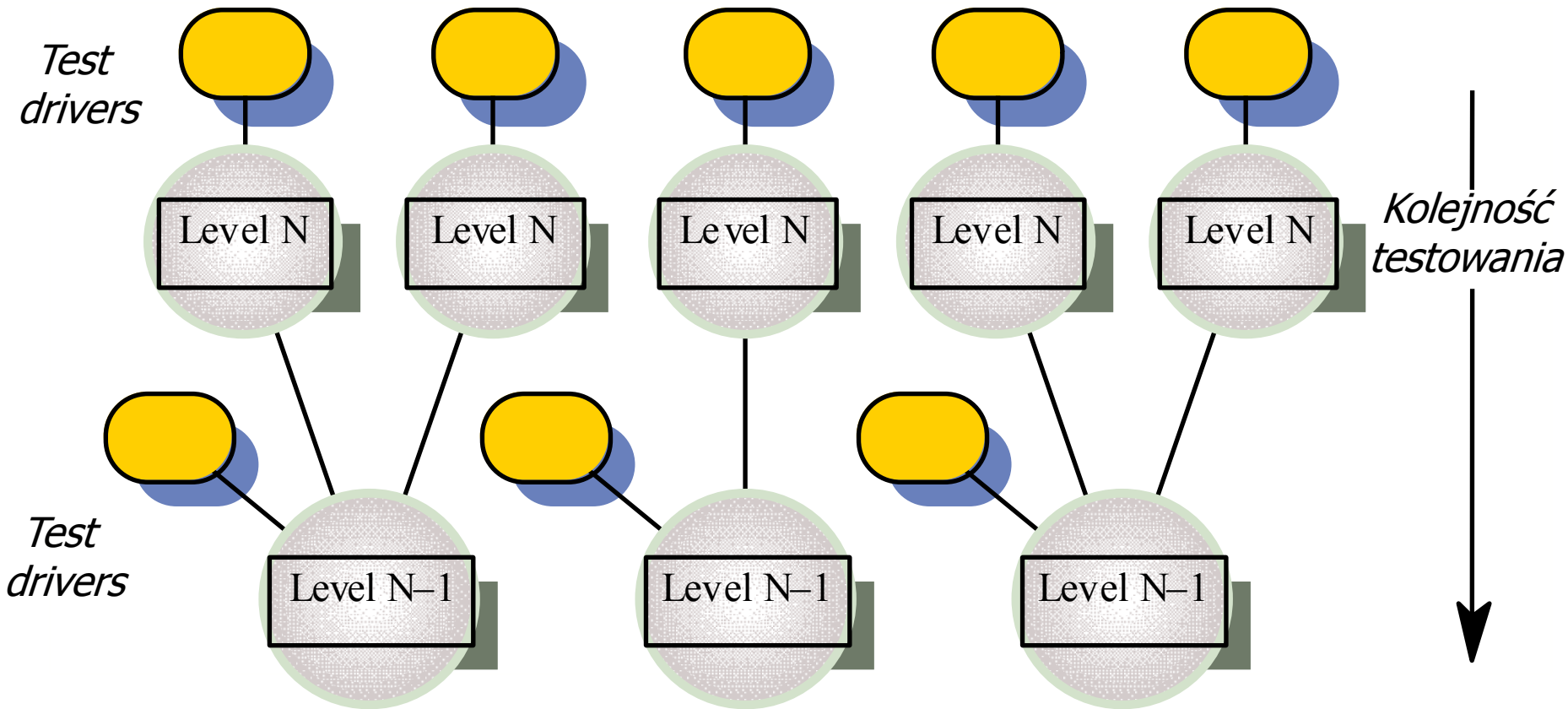


Wg Ian Somerville, (c) 1995

Testowanie metodą „top-down”

- Początek – komponenty najwyższych rzędów
- Kierunek – komponenty coraz niższych rzędów
- Używane w połączeniu z programowaniem „top-down”
  - Kodujemy najpierw komponenty najwyższych rzędów
  - Zamiast komponentów niższych rzędów – zaślepki (ang. stubs)
- Pozwala na wczesną lokalizację błędów architektury
- Trudności
  - Może się pojawić konieczność dostępu do docelowej infrastruktury systemu – nie wszystko da się zastąpić zaślepkami
  - Implementacja zaślepek – uniwersalność vs. wysiłek potrzebny na utworzenie

Testowanie metodą „bottom-up”



Wg Ian Somerville, (c) 1995

Testowanie metodą „bottom-up”

- Użyteczne przy testowaniu kluczowych komponentów architektury
- Początek – komponenty najniższych rzędów
- Kierunek – komponenty wyższych rzędów
- ➔ Implementacja tzw. „test drivers”
- Możliwość identyfikacji błędów projektu (design) systemu dopiero w późnych fazach procesu
- Szeroko stosowane dla systemów obiektowych

Testowanie przeciążeniowe

- Ma na celu sprawdzenie zachowania systemu w warunkach obciążenia większego niż projektowane dopuszczalne
  - Zwykle w takiej sytuacji – ujawnianie się różnych błędów
- Obciążanie systemu – pokazuje zachowanie systemu w takiej sytuacji
  - System nie może powodować nieakceptowalnej utraty funkcjonalności bądź danych – konieczność implementacji odp. mechanizmów
- Szczególnie przydatne przy testowaniu systemów rozproszonych
  - Częsta sytuacja – chwilowe/dłuższe przeciążenie sieci
  - Mechanizmy obsługi - ??

Testowanie regresyjne

- Odpowiada na pytanie
  - Czy wraz ze zmianami wprowadzonymi do systemu nie wprowadzono również nowych błędów
- Stosowane do istniejących, uprzednio już testowanych komponentów systemu
  - Nowo dodawana funkcjonalność ➔ przechodzi unit testy przed dodaniem; co z dotychczasową ?
- Powszechnie używane w *Extreme Programming*
- Wymaga dostępu do narzędzi umożliwiających automatyzację procesu testowania
  - Często powtarzane te same testy

Narzędzia do automatyzacji testów

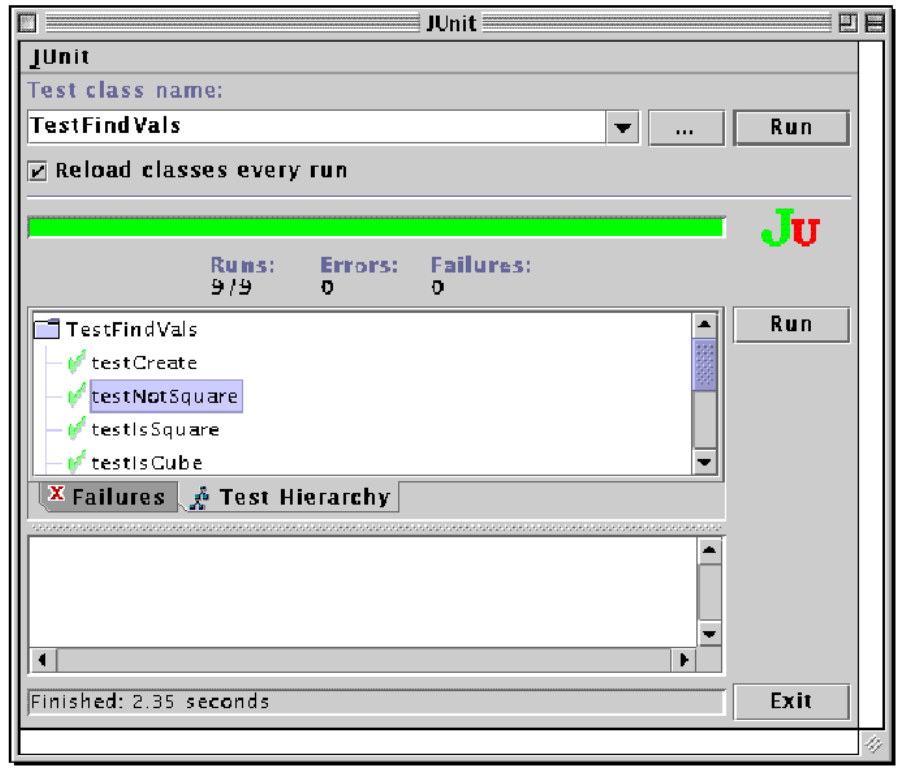
- **Jako przykład**
  - Prosta procedura, napisana „z palca”
  - ➔Tak prosta że musi być poprawna...

```
int sumPrimes() {
   int sum = 0;
   for (int i=1; i < 100; i++ ) { // loop over possible primes
    bool prime = true;
    for (int j=1; j < 10; j++) { // loop over possible factors
       if (i % j == 0) prime = false;
    }
    if (prime) sum += i;
   }
   return sum;
}
```

  - ➔ Ale nie jest poprawna!
  - Jak to sprawdzić ?
    - patrząc na kod
    - uruchomić program i sprawdzić czy daje dobry wynik – wiadomo od razu!

Jak testować kolejne poprawki?

- Najprościej: Uruchamiamy i patrzymy na wynik
  - Szybko staje się nudne!
  - Jak często można to wykonywać?
- Bardziej realistycznie: zakodować procedury testowe dostarczające danych wejściowych i sprawdzające wyniki
  - Bardzo szybko się rozrastają
- Najwygodniej: wykorzystać tzw. *test framework*
  - Raporty z wyników testów
  - Lepsza kontrola procesu testowania



Testing Frameworks:CppUnit, Junit, ...

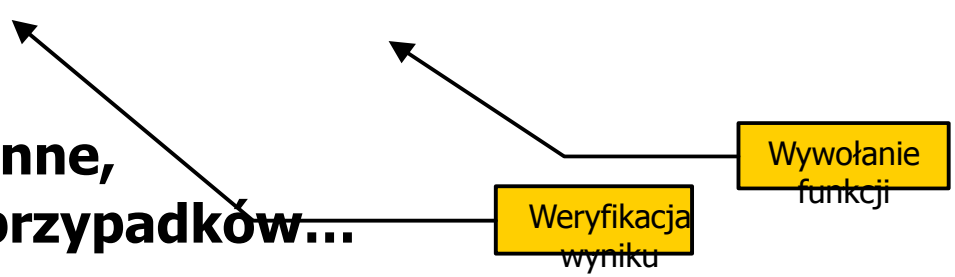
- **W celu przetestowania funkcji:**

```
public class FindVals {
// Test whether an number is a square
  boolean isSquare(int val) {
     double root = Math.floor(Math.pow(val, 0.5));
     if (Math.abs(root*root - val) < 1.E-6 ) return true;
     else return false;
  }
}
```

- **Piszemy test:**

```
public void testIsSquare() {
  FindVals s = new FindVals();
  Assert.assertTrue( s.isSquare(4) );
}
```

- **Jak również inne, dla innych przypadków...**



Testing frameworks ... (c.d.)

- Osadzamy testy w narzędziu
  - Zbieramy wszystkie testy razem

```
// define test suite
public static Test suite() {
  // all tests from here down in hierarchy
  TestSuite suite = new TestSuite(TestFindVals.class);
  return suite;
}
```

  - I zaczynamy testowanie
    - Aby uruchomić testy:

```
junit.textui.TestRunner.main(TestFindVals.class.getName());
```

    - To samo z użyciem GUI:

```
junit.swingui.TestRunner.main(TestFindVals.class.getName());
```

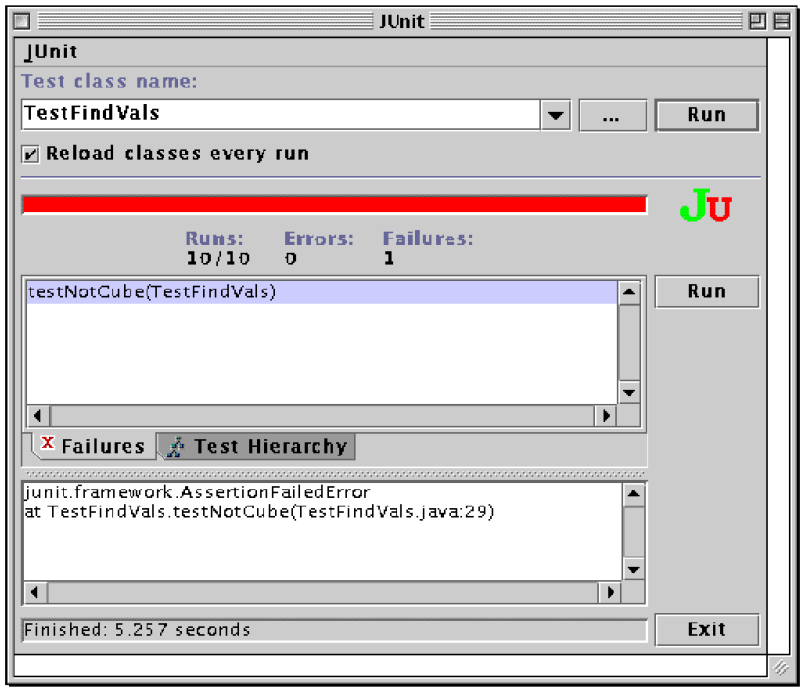
  - To wszystko!

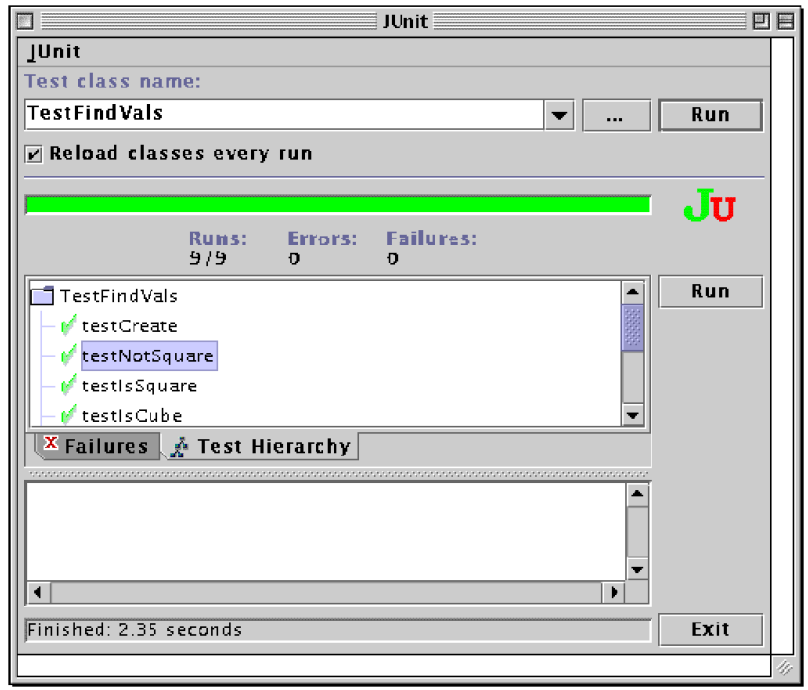
JUnit wykorzystuje nazwę klasy do lokalizacji testów

Wywołanie testów dla danej klasy

Testing frameworks... (c.d.)

- Wykonywanie testów

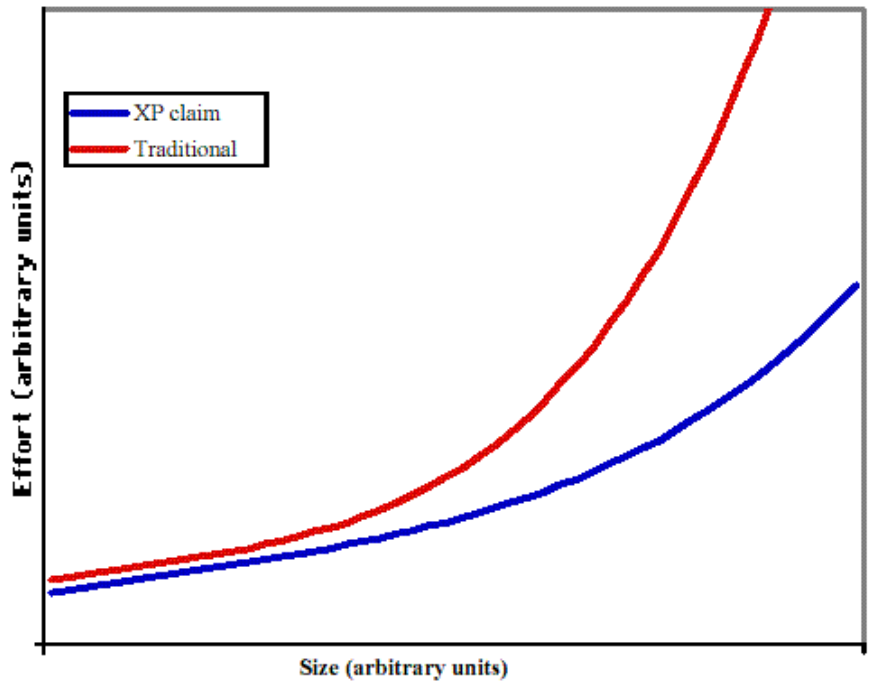




Testing frameworks … (c.d.)

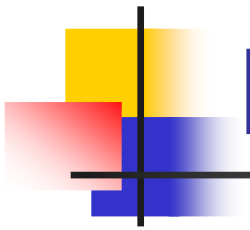
Po co to podejście zamiast po prostu przetestować funkcję ręcznie?

- W przypadku jednego testu
  - W najlepszym wypadku zaoszczędzi sie kilka razy parę sekund
- Ale konsekwentne tworzenie testów wraz z kodem ma swoje zalety
  - W przypadku wprowadzenia błędu przy poprawnianiu bug'a czy dodawania funkcjonalności
    - Testy wykryją to automatycznie
  - Zestaw testów jest formą dokumentacji kodu – wiadomo jak kod ma się zachowywać
  - Naprzemienne kodowanie oraz tworzenie testów narzucają inkrementalny tryb pracy
    - Eliminacja przykrych niespodzianek
- Extreme Programming zaleca pisanie unit testów przed odpowiednim fragmentem kodu



Podsumowanie (1/2)

- Weryfikacja != walidacja
- Testowanie – cel: stwierdzenie istnienia błędu(ów) w systemie
- W zakres testowania wchodzą
  - unit testy, testy integracyjne
  - testy akceptacyjne, systemowe, regresyjne



Podsumowanie (2/2)

- Testowanie musi być uwzględnione od początku w planach projektu
  - Również w alokacji zasobów do projektu
  - Test plan – niezbędny dokument projektowy
- Proces testowania można do pewnego stopnia zautomatyzować
  - Narzędzia do automatycznego wykonywania testów (testing frameworks)