Inżynieria oprogramowania 1

Testowanie dynamiczne, część pierwsza

Arkadiusz Chrobot

Katedra Systemów Informatycznych, Politechnika Świętokrzyska

Kielce, 15 stycznia 2025

Plan

Motto

”Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald E. Knuth

Wstęp

W ramach tego wykładu zostanie omówione bardziej szczegółowo zagadnienie testowania dynamicznego. Zostaną przedstawione poziomy oraz metody wykonywania testów.

Poziomy testowania

Można wyróżnić *cztery* poziomy dynamicznych testów:

testy jednostkowe (ang. *unit tests*) dotyczą małych jednostek kodu, takich jak funkcje;

testy integracyjne (ang. *integration tests*) sprawdzają interakcję komponentów;

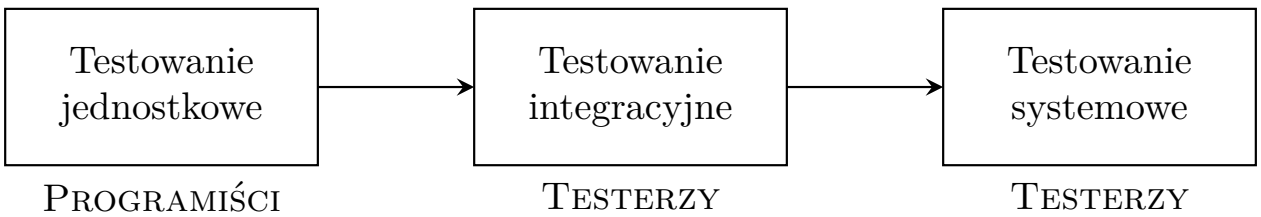
testy systemowe (ang. *system tests*) weryfikują ukończoną *wersję* oprogramowania;

testy akceptacyjne (ang. *acceptance tests*) stanowią potwierdzenie dla interesariuszy, że oprogramowanie spełnia wymagania.

Testy akceptacyjne

W przypadku oprogramowania tworzonego na zamówienie *testy akceptacyjne* mogą być wykonane przez jego twórców, ale częściej są przeprowadzane przez klienta. Ten rodzaj testowania akceptacyjnego nazywa się w języku angielskim *User Acceptance Tests* lub w skrócie *UATs*. Jeśli klient nie dysponuje odpowiednimi kompetencjami, to może zlecić te testy firmie trzeciej.
Testowanie akceptacyjne oprogramowania ogólnego przeznaczenia dzieli się na dwa etapy nazywane *testami alfa* i *testami beta*. *Testowanie alfa* odbywa się w siedzibie producenta oprogramowania, z udziałem wyselekcjonowanej grupy klientów. Testy beta są przeprowadzane z użyciem komputerów klientów, na których zainstalowana jest okrojona oprogramowania lub taka, której działanie jest ograniczone czasowo.

Poziomy testowania

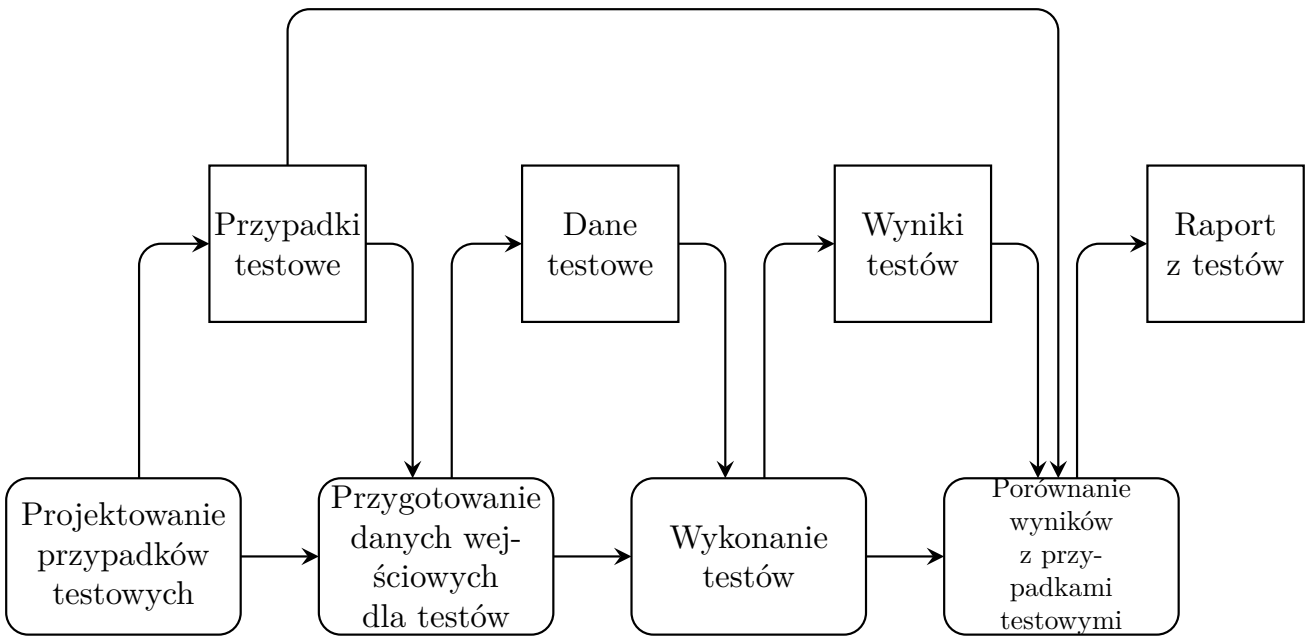


Rysunek: Poziomy testów i odpowiedzialne za nie zespoły

Testowanie defektów

Testowanie dynamiczne, w szczególności testowanie defektów, polega na przeprowadzaniu eksperymentów, w trakcie których sprawdzane jest, czy zachowanie i wyniki oprogramowania odpowiadają jego *specyfikacji*. Opis (formalny lub nieformalny) takiego eksperymentu nazywa się *przypadkiem testowym* (ang. *test case*). Określa on dane wejściowe dla testu oraz spodziewane zachowanie i wyniki oprogramowania podlegającego testowaniu (Rysunek 2).
Główną trudnością w testowaniu jest zaprojektowanie jak najmniejszej liczby przypadków testowych, które dostarczałby jak najlepszych rezultatów (Rysunek 3). Jakość testów może być szacowana przy pomocy *metryk*, które zostaną omówione na następnym wykładzie. Jednakże dobrej jakości testy można uzyskać dzięki użyciu odpowiedniej metody projektowania przypadków testowych. Dwie najczęściej stosowane takie metody to *testowanie funkcjonalne* (ang. *functional testing*) i *testowanie strukturalne* (ang. *structural testing*).

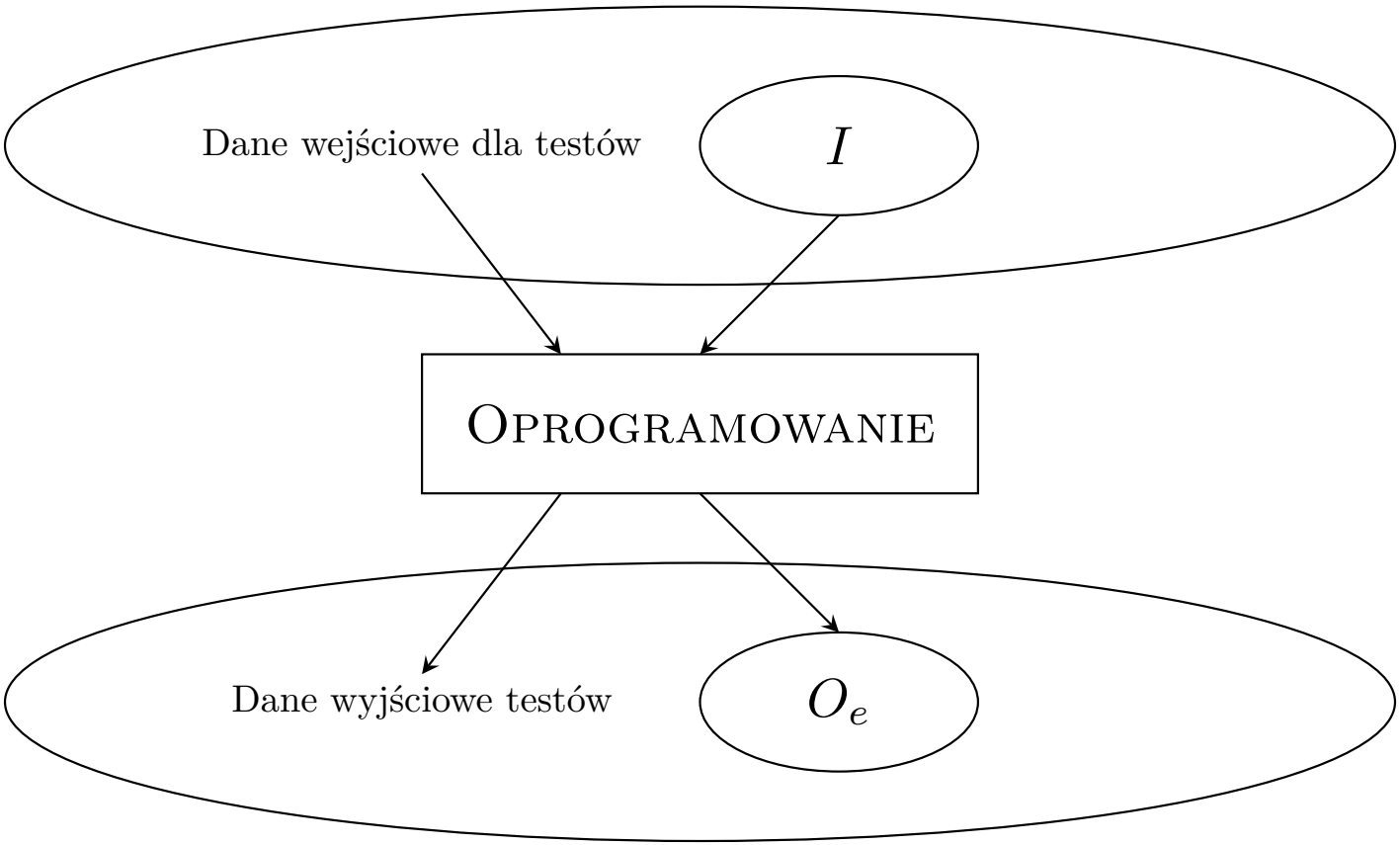
Testowanie defektów



Rysunek: Przebieg testowania

Testowanie defektów

I — dane wejściowe, które powodują błędne działanie oprogramowania
O_e — wyniki ujawniające istnienie defektów



Rysunek: Ogólny model testowania

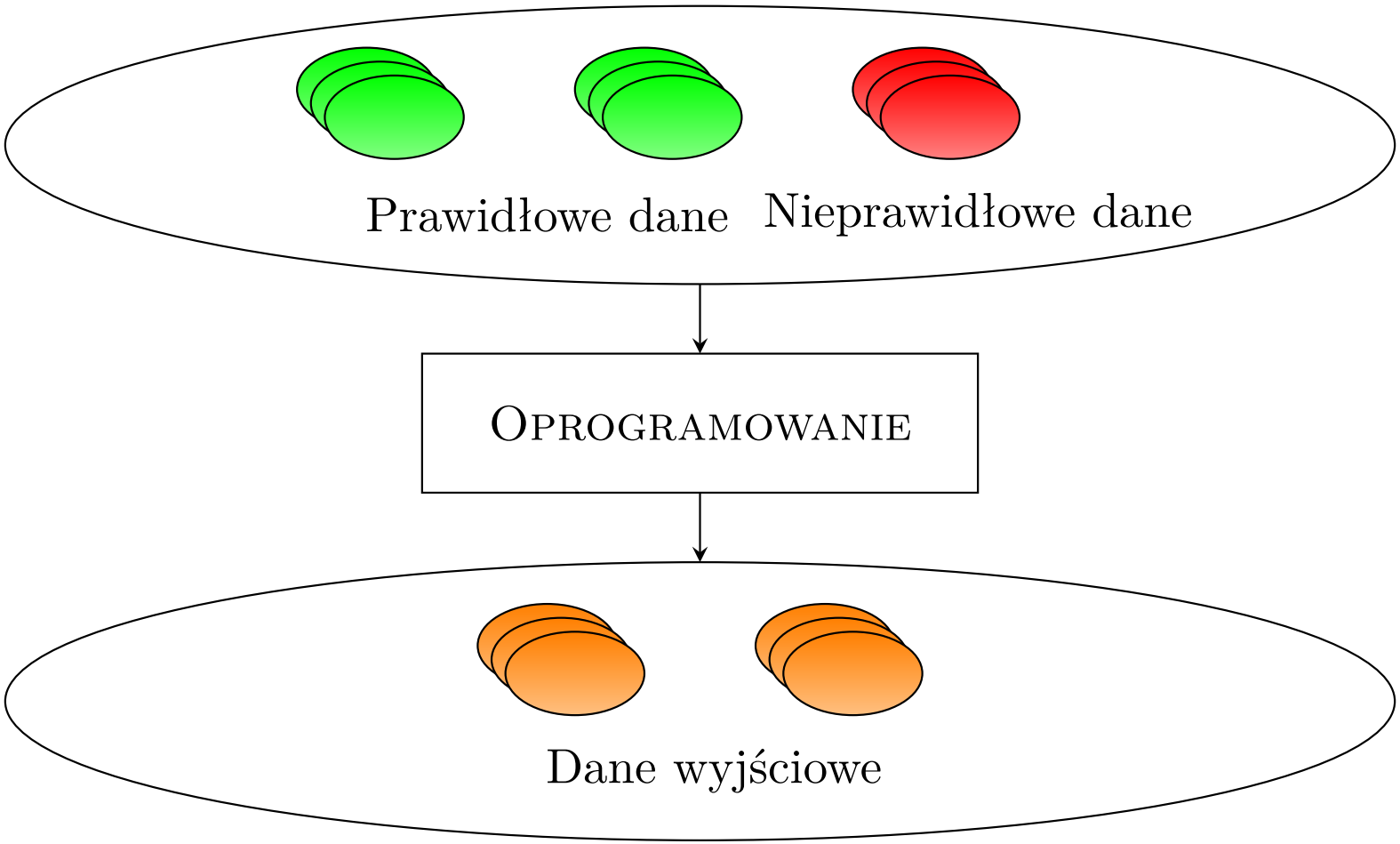
Testowanie funkcjonalne

Podział na klasy równoważności

Testowanie funkcjonalne (ang. *functional testing*) jest również znane jako *testowanie czarnej skrzynki* (ang. *black-box testing*). W tej metodzie kod źródłowy oprogramowania jest albo niedostępny, albo zbyt duży i skomplikowany, aby go przeanalizować. Dostępna jest jedynie specyfikacja opisująca jak to oprogramowanie działa. Ta metoda może być zastosowana na każdym poziomie testowania. Pierwszą czynnością konieczną do zaprojektowania odpowiednich przypadków testowych dla testowania funkcjonalnego jest podzielenie zbioru danych wejściowych na podzbiory nazywane klasami równoważności (ang. *equivalence classes*). Każda z nich zawiera dane o podobnej charakterystyce, a więc przetwarzane w podobny sposób przez testowane oprogramowanie (Rysunek 4).

Testowanie funkcjonalne

Podział na klasy równoważności



Rysunek: Podział na klasy równoważności

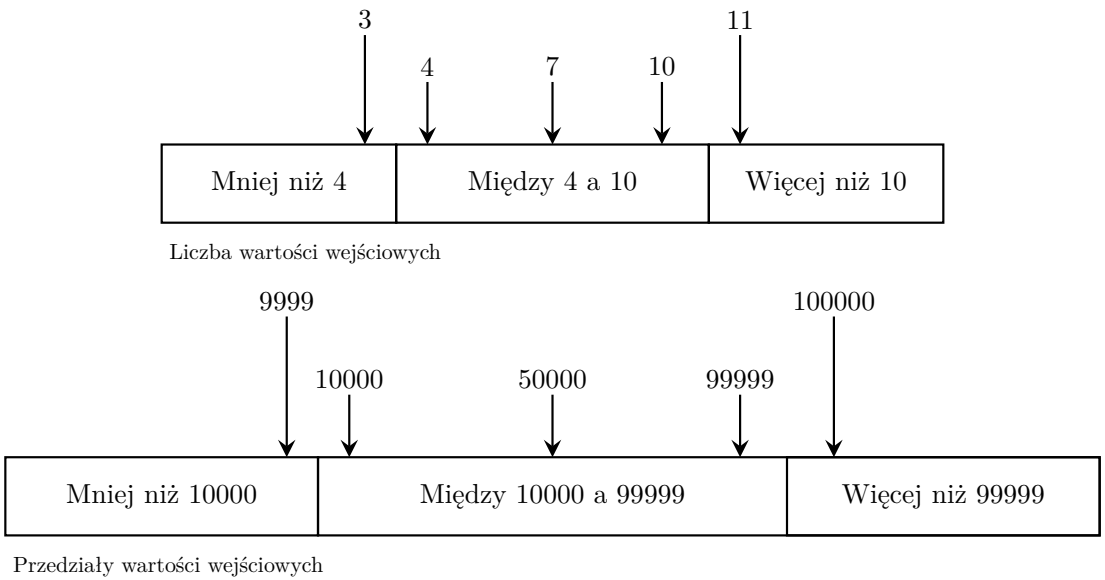
Testowanie funkcjonalne

Testowanie wartości brzegowych

Projektant przypadków testowych powinien nie tylko wybierać dane wejściowe ze „środka” klas równoważności, ale również dane „leżące” na ich granicach (tak zwane *wartości brzegowe* albo *wartości graniczne*) oraz takie, które są blisko tych granic lub leżą tuż poza nimi (Rysunek 5). Testowanie wykorzystujące te dane wejściowe nazywa się *testowaniem wartości brzegowych* (ang. *boundary testing*).

Testowanie funkcjonalne

Testowanie wartości brzegowych



Rysunek: Testowanie wartości brzegowych

Testowanie funkcjonalne

Przykład

W przykładzie należy wykonać testy komponentu, który próbuje znaleźć daną wartość w sekwencji liczb. Specyfikacja formalna tego komponentu znajduje się na następnym slajdzie.

Testowanie funkcjonalne

Przykład

Specyfikacja

procedure Search (Key:ELEM; T:ELEM ARRAY; Found: **in out** BOOLEAN; L: **in out** ELEM_INDEX);

Pre-condition

-- Sekwencja musi mieć co najmniej jeden element.

T'FIRST <=T'LAST

Post-condition

-- Wartość jest odnaleziona, a jej położenie zapisane jest w L.

(Found **and** T(L)=Key)

or

-- Wartość nie występuje w sekwencji.

(**not** Found **and not**(**exists** i, T'FIRST <=i<=T'LAST, T(i) = Key))

Testowanie funkcjonalne

Przykład

Po przeanalizowaniu specyfikacji i zastosowaniu metod projektowania przypadków testowy, które zostały opisane wcześniej, można określić następujące klasy równoważności:

Sekwencja	**Wartości**
Tylko jeden element.	Jest w sekwencji.
Tylko jeden element.	Nie występuje w sekwencji.
Więcej niż jeden element.	Jest w pierwszym elemencie.
Więcej niż jeden element.	Jest w ostatnim elemencie.
Więcej niż jeden element.	Jest w środkowym elemencie.
Więcej niż jeden element.	Nie występuje w sekwencji.

Testowanie funkcjonalne

Przykład

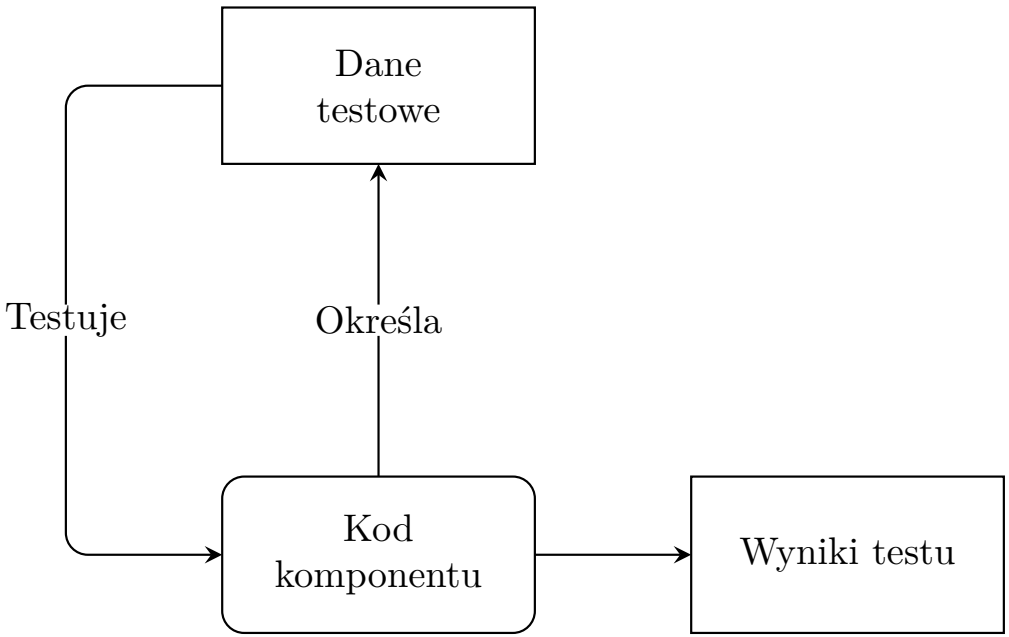
Posługując się klasami równoważności można zaprojektować następujące przypadki testowe:

Sekwencja wejściowa (T)	**Klucz (Key)**	**Wynik (Found, L)**
17	17	true, 0
17	0	false, ??
17,29,21,23	17	true, 0
41,18,9,31,30,16,45	45	true, 6
17,18,21,23,29,41,38	23	true, 3
21,23,29,33,38	25	false, ??

Testowanie strukturalne

W *testowaniu strukturalnym* (ang. *structural testing*), znanym także jako *testowaniem białej skrzynki* (ang. *white-box testing*) lub *testowaniem przezroczystej skrzynki* (ang. *clear box testing, transparent box testing*) albo *testowaniem szklanej skrzynki* (ang. *glass box testing*) kod źródłowy testowanego komponentu jest dostępny. Na jego podstawie projektant testów może opracować dodatkowe przypadki testowe (Rysunek 6). Ta metoda może być zastosowana w testach jednostkowych i do pewnego stopnia w testach integracyjnych.

Testowanie strukturalne



Rysunek: Testowanie strukturalne

Testowanie strukturalne

Przykład — wyszukiwanie binarne

Następny slajd zawiera napisany w Javie kod źródłowy metody, która próbuje zlokalizować wartość (`key`) w tablicy (`elemArray`) stosując algorytm wyszukiwania binarnego. Zastosowanie takiego algorytmu narzuca dodatkowy warunek wstępny: wartości w tablicy powinny być posortowane w porządku niemalejącym.

Testowanie strukturalne

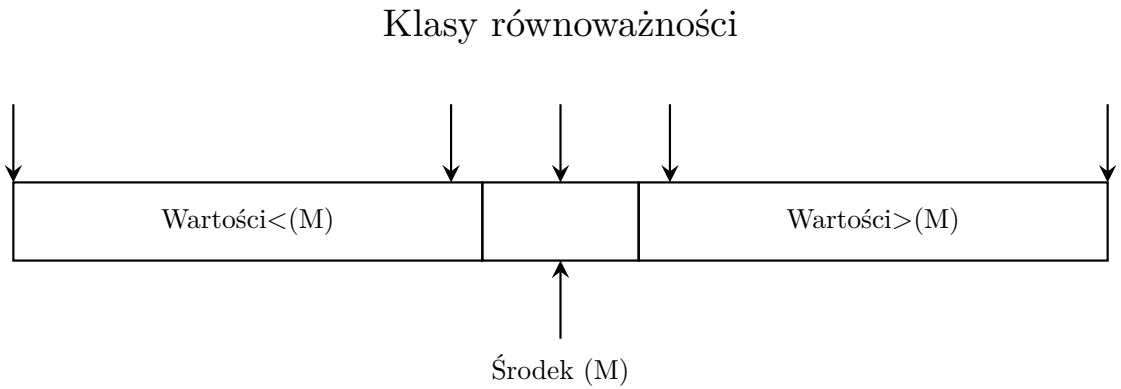
Przykład — wyszukiwanie binarne

```
public static Result search(int key, int elemArray[]) {
    int bottom = 0;
    int top = elemArray.length-1;
    while(bottom<=top) {
        int middle = bottom + (top-bottom)/2;
        if(elemArray[middle]==key)
            return new Result(middle,true);
        if(elemArray[middle]<key)
            bottom=middle+1;
        else
            top = middle-1;
    }
    return new Result(-1,false);
}
```

Testowanie strukturalne

Klasy równoważności dla wyszukiwania binarnego

Wiedząc, że testowana metoda stosuje algorytm wyszukiwania binarnego można zdefiniować dwie dodatkowe klasy równoważności — sekwencje liczb, w których szukana wartość znajduje się w lewym i prawym sąsiedzie elementu środkowego (Rysunek 7).



Rysunek: Klasy równoważności dla wyszukiwania binarnego.

Testowanie strukturalne

Przypadki testowe dla wyszukiwania binarnego

Dla metody `search()` mogą zostać zdefiniowane następujące przypadki testowe:

Sekwencja wejściowa (T)	**Klucz (Key)**	**Wynik (Found, L)**
17	17	true,0
17	0	false,??
17,21,23,29	17	true,0
9,16,18,30,31,41,45	45	true,6
17,18,21,23,29,38,41	23	true,3
17,18,21,23,29,33,38	21	true,2
12,18,21,23,32	23	true,3
21,23,29,33,38	25	false,??

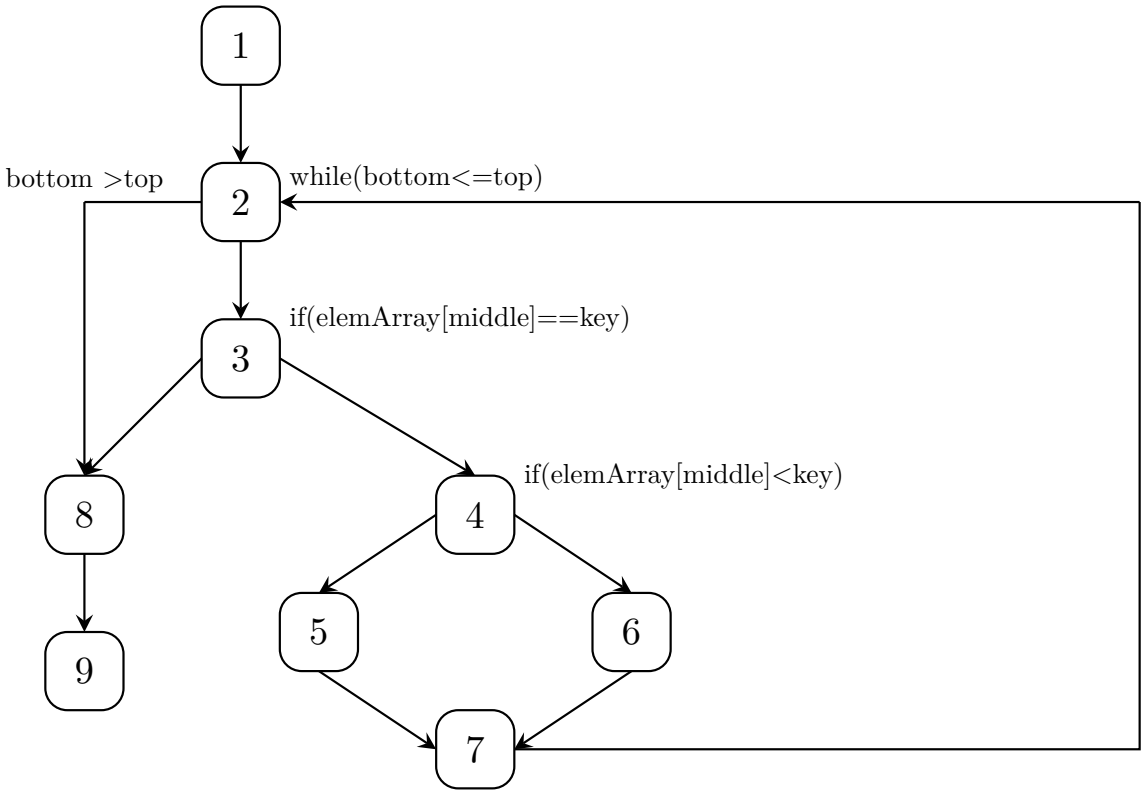
Dodatkowo powinna zostać użyta bardzo długa sekwencja (do 2147483647 elementów), z szukaną wartością zlokalizowaną na jej końcu.

Testowanie strukturalne

Testowanie strukturalne pozwala testerom zweryfikować każdą *ścieżkę pierwotną* (ang. *prime path*) w testowanym komponencie. *Ścieżka* (ang. *path*) to sekwencja wykonywanych wspólnie instrukcji. Rozpoczyna się ona *punktem wejściowym* (ang. *entry point*) i kończy *punktem wyjściowym* (ang. *exit point*). *Ścieżką pierwotną* jest każda ścieżka, która różni się od pozostałych co najmniej jedną instrukcją.
Jeśli w komponencie zostaną przetestowane wszystkie ścieżki pierwotne, to będzie to znaczyło, że wszystkie jego instrukcje zostały przynajmniej raz wykonane i że każdy *prosty warunek* został sprawdzony wtedy gdy jest prawdziwy i gdy jest fałszywy. Warunek prosty to część wyrażenia warunkowego, która nie zawiera operatorów logicznych (and, or). Nie testuje się kombinacji wykonania ścieżek pierwotnych, bo byłoby to zbyt kosztowne. Celem znalezienia wszystkich ścieżek pierwotnych w kodzie można zastosować *graf przepływu sterowania* (ang. *control-flow graph*), w skrócie (*CFG*). Taki graf dla metody `search()` przestawia Rysunek 8.

Testowanie strukturalne

Graf przepływu sterowania



Rysunek: Graf przepływu sterowania

Testowanie strukturalne

W CGF dla metody `search()` są 4 ścieżki pierwotne, t.j.:

$1, 2, 3, 8, 9$
$1, 2, 3, 4, 6, 7, 2$
$1, 2, 3, 4, 5, 7, 2$
$1, 2, 3, 4, 6, 7, 2, 8, 9$

Liczba tych ścieżek stanowi miarę *złożoności cyklomatycznej* (and. *cyclomatic complexity*) zdefiniowanej przez Thomasa J. McCabe'a Sr. Można ją wyznaczyć korzystając ze wzoru:

$$CC(G) = \#edges - \#nodes + 2$$

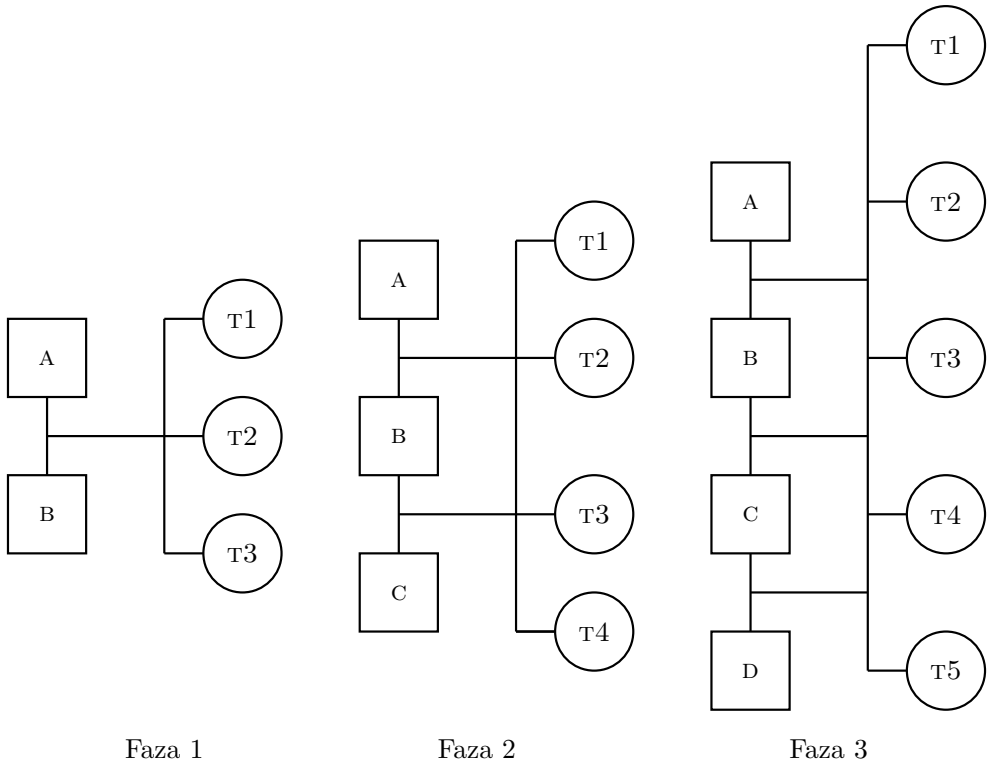
(gdzie $\#edges$ to liczba krawędzi grafu, a $\#nodes$ to liczba jego węzłów) lub zliczając wszystkie proste warunki w kodzie i dodając do wyniku jeden. Obie metody nie nadają się do zastosowania w kodzie wielowątkowym, rekurencyjnym lub zawierającym osławioną instrukcję `goto`. Złożoność cyklomatyczna określa minimalną liczbę przypadków testowych koniecznych do weryfikacji testowanego kodu.

Testowanie integracyjne

Głównym celem *testów integracyjnych* (ang. *integration testing*) jest weryfikacja interakcji współpracujących komponentów. Oznacza to, że te testy mogą być przeprowadzone tylko wtedy, gdy co najmniej dwa takie komponenty są dostępne. Testowanie integracyjne jest procesem przyrostowym (Rysunek 9). Jeśli tylko dwa komponenty są gotowe do testów, to mogą zostać przygotowane trzy zbiory przypadków testowych: dwa testujące w izolacji komponenty (np. T1 i T2) i jeden sprawdzający jak one współpracują (np. T3). Jeżeli zostanie dodanych więcej komponentów, to istniejące przypadki testowe powinny zostać zaktualizowane oraz powinny być dodane nowe ich zbiory, by zweryfikować interakcję nowych elementów z resztą systemu.

Testowanie integracyjne

Przyrostowe testowanie integracyjne



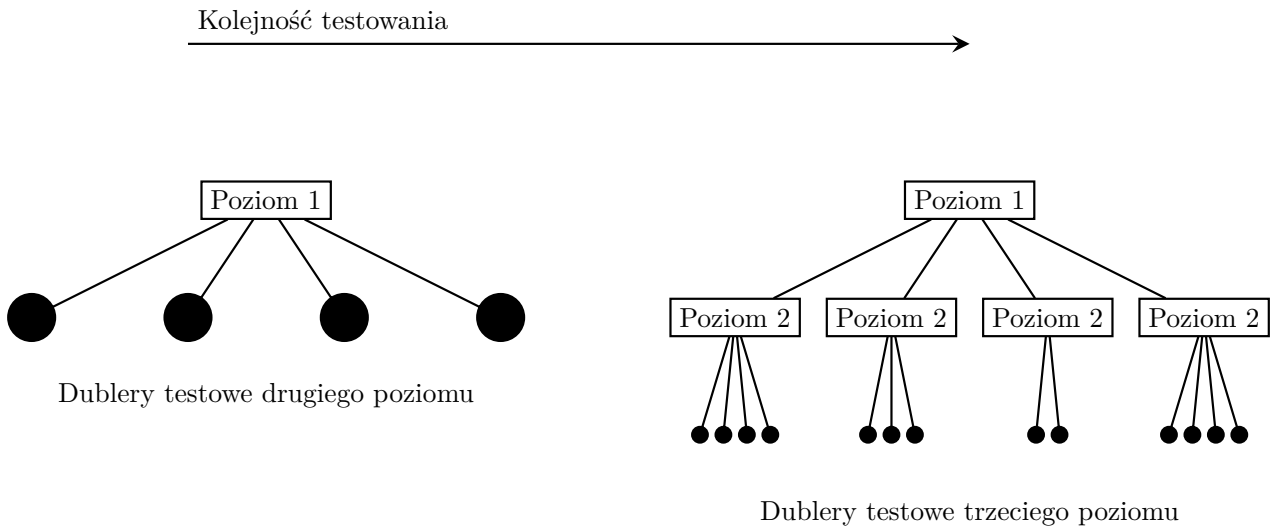
Rysunek: Przyrostowe testowanie integracyjne

Testowanie integracyjne

Testowanie integracyjne zazwyczaj nie przebiega tak prosto, jak to pokazuje Rysunek 9. Nowo dodane przypadki testowe mogą ujawnić defekty w interakcji poprzednio zintegrowanych komponentów lub w ich wnętrzu, które nie zostały odkryte we wcześniejszych testach.
Może się również zdarzyć, że integracja tylko dwóch komponentów będzie niemożliwa i trzeba będzie od razu zintegrować ich więcej. Takie podejście jest nazywane *testowaniem integracyjnym metodą wielkiego wybuchu* (ang. *big bang integration testing*). Jednakże częściej są stosowane dwa inne podejścia: *zstępujące testowanie integracyjne* (ang. *top-down integration testing*) i *wstępujące testowanie integracyjne* (ang. *bottom-up integration testing*). Są one ściśle powiązane z metodami tworzenia oprogramowania.
W podejściu zstępującym najpierw są tworzone i testowanie komponenty wysokopoziomowe. Oznacza to, że te niskopoziomowe, od których zależą wysokopoziomowe, są najczęściej niedostępne i muszą być na czas testowania zastąpione *dublerami testowymi* (ang. *test doubles*). Później, kiedy brakujące komponenty zostaną opracowane, to będą one podczas testów korzystały z innych dublerów testowych (Rysunek 10).

Testowanie integracyjne

Testowanie zstępujące



Rysunek: Testowanie zstępujące

Testowanie integracyjne

Dublery testowe

Dubler testowy (ang. *test double*) jest komponentem oprogramowania używanym w testach. Zazwyczaj udaje on inny komponent, którego ostateczna wersja nie jest dostępna w czasie testów, ale działanie weryfikowanego oprogramowania od niego zależy. Dublery mogą wykonywać w trakcie testów dodatkowe zadania, jak rejestrowanie ile razy zostały wywołane przez testowany komponent. Następująca ▸ klasyfikacja dublerów testowych została zaproponowana przez Gerarda Meszarosa, pracownika firmy Microsoft:

atrapa (ang. *dummy*) Dubler testowy przekazywany jako argument wywołania podprogramu. Nic nie robi.

fałszywka (ang. *fake*) Posiada ona roboczą implementację, która jednakże nie nadaje się do środowiska produkcyjnego (np. baza danych utrzymywana w RAM).

namiastka (ang. *stub*) Dostarcza ona odpowiedzi na określone zapytania. Nie reaguje na żadne inne.

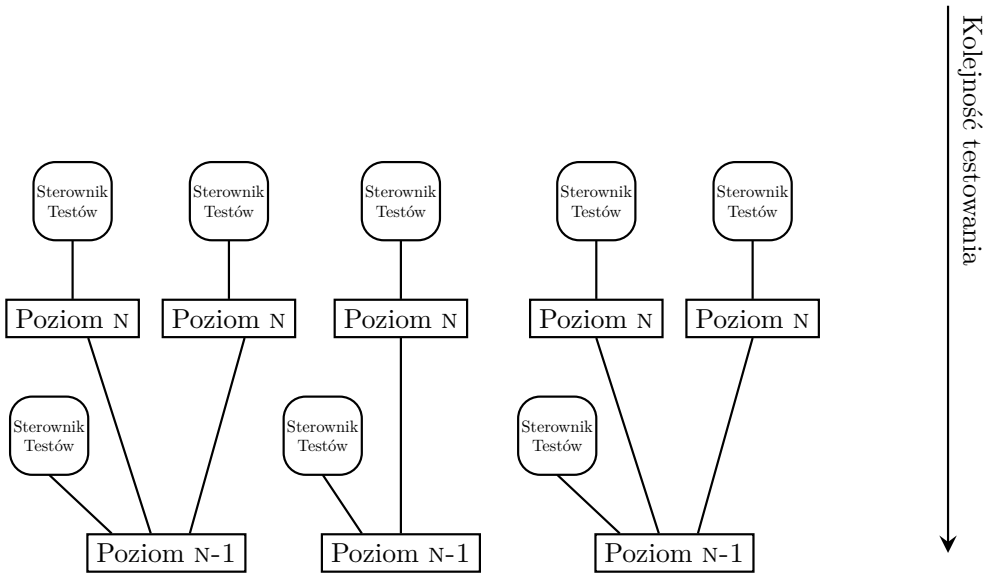
szpieg (ang. *spy*) To namiastka, która rejestruje w ograniczony sposób informacje o tym jak była wywoływana podczas testów.

Testowanie integracyjne

makieta (ang. *mock*) Makieta *sprawdza* jak jest wywoływana. Może ona wyrzucić wyjątek, jeśli jest używana w sposób nieprawidłowy przez testowany komponent. Podobnie jak szpieg rejestruje i sprawdza ile razy została wywołana w czasie testów.

Testowanie integracyjne

Testowanie wstępujące



Rysunek: Testowanie wstępujące

Testowanie integracyjne

W podejściu wstępującym najpierw są tworzone i testowane komponenty niskopoziomowe. Niestety, nie są one niezależne i w związku z tym muszą zostać opracowane specjalne *sterowniki testów* (ang. *test drivers*), aby móc je przetestować. Kiedy ostatecznie uda się te komponenty zintegrować, w komponenty wyższego poziomu, to potrzebny będzie nowy zbiór sterowników testów, aby móc je sprawdzić.

Testowanie integracyjne

Porównanie metody zstępującej i wstępującej

- W testowaniu zstępującym architektura oprogramowania jest zatwierdzana wcześniej, niż w testowaniu wstępującym.
- Dowód wykonywalności może być dostarczony wcześnie w obu metodach.
- Implementacja testów jest równie trudna.
- W obu metodach obserwacja wyników testów wymaga dodatkowych zabiegów.

W rzeczywistych projektach dotyczących opracowania oprogramowania, obie metody tworzenia i testowania są stosowane równocześnie, więc podana lista zalet i wad nie ma szczególnego znaczenia.

Testowanie integracyjne

Testowanie interfejsów

Większość przypadków testowych w testowaniu integracyjnym jest zaprojektowana aby weryfikować *interfejsy* sprawdzanych komponentów, niż ich wewnętrzną implementację (Rysunek 12). Do najczęściej stosowanych typów interfejsów należą:

parametry używane do przekazywana danych lub referencji do funkcji z jednego komponentu do drugiego,

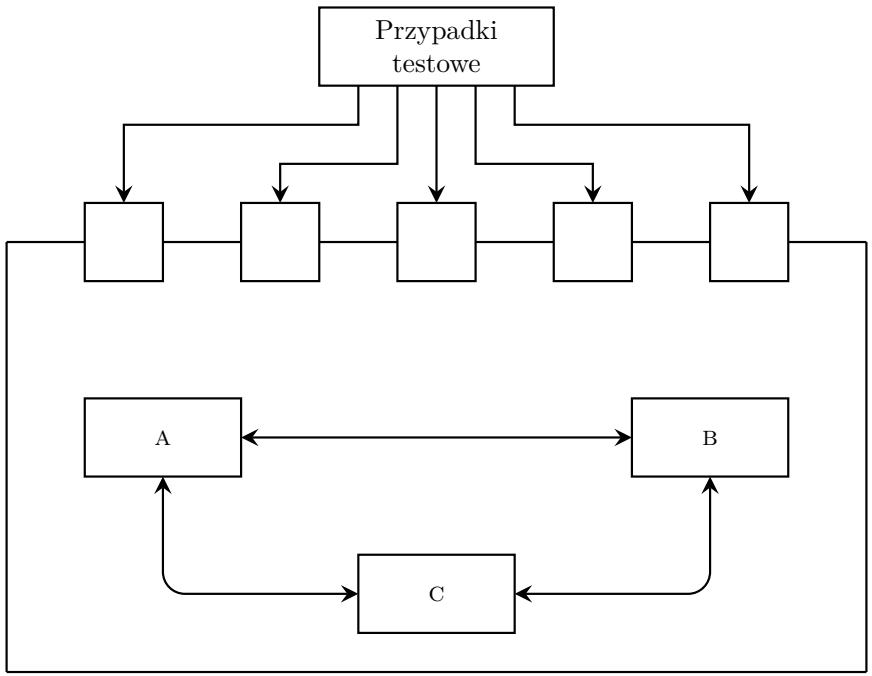
pamięć dzielona to obszar pamięci wykorzystywany przez dwa lub więcej współbieżnych wątków lub procesów do wymiany danych,

intefejsy proceduralne jeden z komponentów dostarcza usług wywoływanych przez drugi,

przekazywanie komunikatów jeden z komponentów żąda usług drugiego poprzez przesłanie komunikatu.

Testowanie integracyjne

Testowanie interfejsów



Rysunek: Testowanie interfejsów

Testowanie integracyjne

Testowanie interfejsów — zalecenia

Projektując przypadki testowe do testowania każdego z opisanych typów interfejsów, należy zastosować inne podejście:

1. W przypadku parametrów można zastosować testowanie wartości brzegowych. Szczególną uwagą należy zwrócić na referencje i wskaźniki. W ich przypadku testy powinny być przeprowadzone również dla przypadków, kiedy ich wartości są równe `null`.
2. Pamięć dzielona powinna być tak testowana, aby procesy lub wątki z niej korzystające były uruchamiane w różnej kolejności.
3. Niektóre przypadki testowe dotyczące interfejsu proceduralnego powinny sprawdzać co się stanie, jeśli komponent w niego wyposażony ulegnie awarii. To pozwoli sprawdzić testerom, czy programiści nie przyjęli złych założeń dotyczących sygnalizowania przez ten komponent wyjątków.
4. Do weryfikacji interfejsów bazujących na przekazywaniu komunikatów należy zastosować *testowanie przeciążeniowe* (ang. *stress testing*).

Testowanie systemowe

Testowanie systemowe (ang. *system testing*) może zostać wykonane kiedy zakończą się prace rozwojowe nad co najmniej wczesną wersją oprogramowania. Niektórzy eksperci uważają je za przedłużenie testowania integracyjnego, ale testy wykonywane na tym poziomie mają inny charakter. One nie tylko sprawdzają poprawność działania usług oferowanych przez to oprogramowanie, ale również weryfikują, czy spełnia ono wymagania niefunkcjonalne:

testy funkcjonalne (ang. *functional tests*) sprawdzają, czy oprogramowanie dostarcza poprawnych usług;

testy wydaności (ang. *efficiency tests*) badają jak oprogramowanie się zachowuje pracując pod normalnym obciążeniem;

testy przeciążeniowe (ang. *stress tests*) sprawdzają zachowanie oprogramowania pracującego ze znacząco wyższym obciążeniem niż to, dla którego zostało zaprojektowane;

testy bezpieczeństwa (ang. *security tests*) badają zabezpieczenia zasobów oprogramowania;

testy zgodności (ang. *compliance tests*) sprawdzają czy oprogramowanie spełnia wymagane standardy;

Testowanie systemowe

testy przenośności (ang. *portability tests*) weryfikują czy oprogramowanie działa poprawnie na różnych platformach systemowych lub sprzętowych;

testy niezawodności (ang. *reliability tests*) sprawdzają czy oprogramowanie działa niezawodnie.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!