Środowisko wspomagające testowanie oprogramowania obiektowego

Sebastian Nowak
e-mail: snowak1@ii.pw.edu.pl
Ilona Bluemke
e-mail: I.Bluemke@ii.pw.edu.pl

Instytut Informatyki, Politechnika Warszawska
Nowowiejska 15/19, 00-665 Warszawa

Streszczenie

W pracy przedstawiono środowisko wspomagające testowanie programów obiektowych zaprojektowanych w jednym z dwóch narzędzi CASE – Paradigm Plus i Select Enterprice v 6. Środowisko może być użyte do testowania klas napisanych w językach C++ oraz Java. Zaimplementowane metody testowania oparte są o specyfikację zachowania klasy w postaci diagramu zmian stanów.

1. Wprowadzenie

Testowanie oprogramowania jest jednym ze środków pozwalających zapewnić wysoką jakość oprogramowania. Celem testowania jest wykrycie jak największej liczby błędów wykonania programu przy minimalnym koszcie. Właściwy test powinien wykrywać błędne zachowanie programu – niepoprawne wyniki, czy zachowanie niezgodnie z oczekiwaniami użytkownika. Zaproponowano następujące podejścia do testowania ukierunkowanego na wyszukiwanie defektów:

1. **Funkcjonalne (ang. black box)** – testy są wyprowadzane ze specyfikacji oprogramowania. Jednostka testowana jest traktowana jako „czarna

skrzynka", której zachowanie może być określone na podstawie wejść i odpowiadających im wyjść. Celem testowania jest określenie, czy program spełnia funkcjonalne i niefunkcjonalne (np. wydajnościowe) wymagania.

2. **Strukturalne (ang. white box)** – testy są wyprowadzane na podstawie znajomości struktury testowanej jednostki. Cechą charakterystyczną tego podejścia jest sprawdzanie pokrycia programu np. sprawdzenie, czy wykonano wszystkie instrukcje, skoki, ścieżki przepływu danych.

Powyższe podejścia zostały zaproponowane dla oprogramowania strukturalnego i są także wykorzystywane podczas testowania oprogramowania obiektowego. Jednak od końca lat osiemdziesiątych prowadzone są prace nad metodami testowania dostosowanymi do testowania programów napisanych w językach obiektowych, wiele informacji na ten temat można znaleźć w pracach [1, 2, 3].

Koszty testowania to 40% do 80 % kosztów produkcji oprogramowania stąd redukcja tych kosztów i poprawa jakości testowania są ważnym problemem od lat przyciągającym uwagę informatyków. Powstało szereg narzędzi wspomagających testowanie oprogramowania obiektowego. Przegląd narzędzi można znaleźć w [4]. W większości narzędzi stosowane jest podejście strukturalne do testowania, badane jest np. pokrycie gałęzi w grafie przepływu sterowania. Narzędzia do testowania są samodzielnymi systemami dostosowanymi do testowania programów napisanych w konkretnym języku programowania np. w C++ czy Javie. Spośród komercyjnych narzędzi do testowania jedynie Test Studio firmy Rational współpracuje z narzędziem do projektowania obiektowego o nazwie Rose (tej samej firmy). Ceny narzędzi wspomagających testowanie są bardzo wysokie i, o ile wiadomo autorom, tego typu narzędzia nie są stosowane ani w dydaktyce ani w pracach własnych na polskich uczelniach.

W Instytucie Informatyki PW powstało narzędzie (o nazwie TOSTER - **T**he **O**bject-oriented **S**oftware **T**esting **E**nvi**R**onment)) służące do testowanie oprogramowania obiektowego napisanego w C++ lub w Javie. Narzędzie to pobiera informacje z repozytoriów dwóch komercyjnych narzędzi do projektowania obiektowego tj. Paradigm Plus oraz Select Enterprice. Jednostką poddawaną testowaniu jest klasa. Testowanie może odbywać się dwoma metodami. W rozdziale 2 przedstawiono zarys działania narzędzia TOSTER, pełen opis można znaleźć w [4]. W rozdziale 3 omówiono zrealizowane metody testowania – metodę wektora stanu oraz metodę specyfikacji sekwencji (także opisane w [3]) działające w oparciu o diagram zmian stanów klasy. Przykład testowania z zastosowaniem środowiska TOSTER zamieszczono w rozdziale 4.

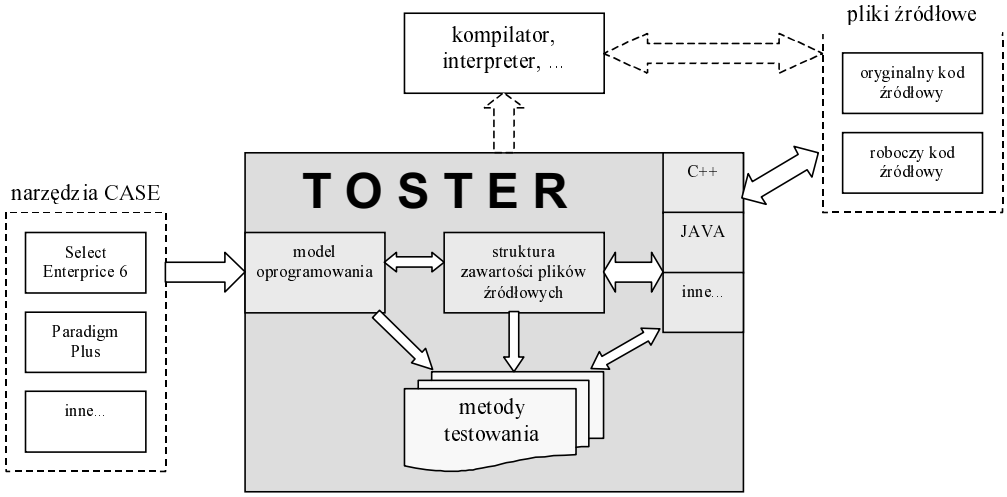
Znane autorom komercyjne narzędzia wspomagające testowanie nie stosują metod opartych o diagramy zmian stanów. Jedynie narzędzie Tootsie (Total Object Oriented Testing Support Environment) opisane w [1] korzysta z diagramów zmian stanów jednak w inny sposób niż TOSTER.

System obiektowy jest zbiorem współdziałających obiektów. Zachowanie całego systemu jest wynikiem interakcji i zachowań indywidualnych obiektów. Wiarygodny system obiektowy wymaga, by każdy z komponentów zachowywał się właściwie, by sumaryczne zachowanie było właściwe oraz, by nie było

możliwości powstania nieprawidłowego zachowania. Testowanie oprogramowania obiektowego powinno bazować na modelu, który ułatwi generowanie przypadków testowych i pozwoli wykryć najbardziej prawdopodobne błędy. Zachowanie klasy jest modelowane (we wszystkich metodach obiektowych) za pomocą diagramu zmian stanów. Wydaje się naturalnym użycie diagramów zmian stanów w procesie testowania zachowania klasy.

2. Architektura środowiska TOSTER

TOSTER (ang. **T**he **O**bject-oriented **S**oftware **T**esting **E**nvi**R**onment) jest otwartym środowiskiem wspomagającym testowanie oprogramowania zorientowanego obiektowo. Środowisko integruje model oprogramowania wykonany w narzędziu CASE z kodem źródłowym oprogramowania oraz udostępnia zestaw narzędzi ułatwiających testowanie oprogramowania. Proces testowania realizowany jest przez moduły, zwane dalej metodami testowania. Metody testowania korzystając z danych oraz narzędzi udostępnianych przez środowisko TOSTER, modyfikują kod źródłowy, uruchamiają testy oraz interpretują oraz prezentują wyniki testowania (rys. 1).



Rysunek 1. Architektura środowiska TOSTER

Otwartość środowiska polega na możliwości:

- implementacji dowolnych metod testowania
- przystosowania do współpracy z niemalże dowolnym obiektowym językiem programowania

- przystosowania do współpracy z różnymi narzędziami CASE

Aktualnie w środowisku zaimplementowano dwie metody testowania: metodę wektora stanu oraz metodę specyfikacji sekwencji opisane w rozdziale 3. TOSTER pozwala na testowanie klas napisanych w językach JAVA lub C++. Wraz ze środowiskiem dostępne jest oprogramowanie pozwalające na korzystanie z modeli w notacji UML utworzonych w narzędziach Select Enterprise 6 oraz Paradigm Plus. Współpraca z narzędziami CASE odbywa się poprzez plik wymiany w formacie XML, co pozwala na stosunkowe łatwe przystosowanie innych narzędzi do współpracy ze środowiskiem TOSTER.

Metody testowania działające w środowisku TOSTER w zależności od swoich potrzeb mogą korzystać z następujących struktur danych oraz narzędzi:

- struktury opisujące model testowanego programu pobrany z narzędzia CASE
- struktury danych opisujące zawartość plików źródłowych testowanego programu
- odwzorowanie modelu na zawartość plików źródłowych (wykonane automatycznie oraz interakcyjnie przez użytkownika)
- funkcje służące do modyfikacji zawartości plików źródłowych
- funkcje wspierające technikę Design For Testability pozwalającą na umieszczanie informacji dotyczących testowania w modelu
- funkcje wspomagające uruchamianie kompilacji oraz uruchamianie testowanego programu
- funkcje pozwalające na rejestrowanie przebiegu wykonywania testu.

Testowanie oprogramowania przy użyciu środowiska TOSTER składa się z następujących faz:

- pobranie modelu oprogramowania z narzędzia CASE
- wskazanie plików źródłowych testowanego programu
- uzupełnienie informacji dotyczącej odwzorowania modelu CASE na zawartość plików źródłowych
- wprowadzenie dodatkowych informacji dotyczących między innymi sposobu kompilacji oraz uruchamiania testowanego programu
- wprowadzenie informacji dotyczącej sposobu oraz przebiegu testowania specyficznych dla danej metody testowania
- uruchomienie testu (poprzedzone automatyczną modyfikacją oraz kompilacją kodu źródłowego)
- przeglądanie/zapisanie wyników testowania.

Środowisko TOSTER zostało zaimplementowane w języku JAVA z wykorzystaniem biblioteki SWING, pełny opis znajduje się w [4].

3. Metody testowania dostępne w środowisku TOSTER

W środowisku TOSTER zaimplementowano dwie metody testowania:

1. metodę wektora stanu oraz
2. metodę specyfikacji sekwencji.

Obie metody służą do testowania zgodności implementacji ze specyfikacją wykonaną w postaci diagramów stanów w notacji UML. Są to metody, które trudno jednoznacznie zakwalifikować do metod funkcjonalnych czy strukturalnych. Ze względu na to, iż korzysta się w nich ze specyfikacji zachowania klasy, można by zaliczyć je do metod funkcjonalnych. Specyfikacja w postaci diagramu zmian stanów, określa w pewien sposób „wnętrze" klasy, a ze znajomości struktury wewnętrznej korzystają metody strukturalne.

3.1. Metoda wektora stanów

Metoda wektora stanu powstała na bazie metody stanów i podstanów zaproponowanej przez C.D. Turner'a and D.J. Robson'a w [6]. Metoda pozwala na wyspecyfikowanie każdego ze stanów diagramu zmian stanów, jako wektora składającego się z odpowiednio zdefiniowanych wartości atrybutów klasy.

Metoda wektora stanów kwalifikuje każdy komunikat wysłany do obiektu testowanej klasy w zależności od jego wpływu na stan obiektu. Pozwala to wykryć błędy polegające na:

- wywołaniu metod niedozwolonych dla obiektu w danym stanie
- wywołaniu metod wprowadzających obiekt do stanu niezdefiniowanego
- wywołaniu metod powodujących niepoprawną zmianę stanu obiektu.

Pierwszy typ błędów wynika z niepoprawnego użycia obiektu testowanej klasy, zaś pozostałe dwa mogą być wynikiem błędów w implementacji metod.

Metoda wektora stanu polega na zdefiniowaniu każdego stanu poprzez odpowiednie wartości atrybutów klasy. Każdy stan może być zdefiniowany przy użyciu dowolnego podzbioru wszystkich atrybutów (włącznie z atrybutami odziedziczonymi z nadklasy).

Wartości atrybutów zostały podzielone na trzy rodzaje w zależności od sposobu ich określania na wartości szczegółowe, wartości ogólne oraz wartości dynamiczne. Dla każdego atrybutu specyfikuje się wartości, które może on przyjmować. Wartości te służą do definiowania wektorów stanów.

W przypadku, gdy stany zawierają podstany, wektory stanów powinny być zdefiniowane w ten sposób, by definicja podstanu zawierała się w definicji nadstanu.

Po zdefiniowaniu stanów dla każdej metody testowanej klasy, tworzone są zbiory stanów wejściowych (stan obiektu w momencie wywołania metody) oraz wyjściowych (stan obiektu po wykonaniu metody). Zbiory te tworzone są na podstawie specyfikacji klasy, a w szczególności w sposób automatyczny na podstawie diagramów stanów. Zbiory metod wejściowych oraz wyjściowych zostaną użyte do określenia poprawności przejść stanów.

Efektywność testowania metodą wektora stanu zależy od specyfiki samej klasy. Dla takich klas jak: klasy kontrolujące pliki, analizatory leksykalne i składniowe oraz abstrakcyjne typy danych (np. listy) testowanie takie może wykryć wiele błędów niewykrywalnych przez inne metody. Stopień efektywności zależy od ilości atrybutów kontroli przepływu w danej klasie.

3.2. Metoda specyfikacji sekwencji

Metoda specyfikacji sekwencji zaproponowana została przez S. Kirami i W.T. Tsai w [5] i polega na specyfikowaniu poprawnych sekwencji wywołań metod danej klasy. Metoda specyfikacji sekwencji pozwala na specyfikowanie sekwencji przy pomocy wyrażeń regularnych lub przy pomocy diagramów stanów w notacji UML.

W przypadku opisu sekwencji przy użyciu wyrażeń regularnych, alfabetem nad którym są one konstruowane jest zbiór nazw metod testowanej klasy. Przykładowo, dla klasy `Konto` alfabetem będzie zbiór: {Utwórz, Otwórz, Zamknij, Usuń, Pobierz, Powiększ}. Wyrażenie regularne dla klasy `Konto` może wyglądać (w zależności od specyfiki działania takiej klasy) następująco:

Utwórz (Otwórz Powiększ (Powiększ | Pobierz)* Zamknij)* Usuń

Symbol * oznacza zero lub więcej powtórzeń. Dopuszczalna sekwencja to np.: Utwórz Otwórz Powiększ Pobierz Zamknij Usuń.

Specyfikacja sekwencji pokazuje prawidłowe użycie wszystkich obiektów klasy. Obiekt może otrzymywać jedynie te komunikaty, które są podzbiorem dopuszczalnej sekwencji komunikatów. Otrzymanie komunikatu, który nie należy do wyspecyfikowanej sekwencji sygnalizowane jest jako wystąpienie błędu. Specyfikacja sekwencji może więc być użyta do weryfikacji implementacji.

W przypadku, gdy specyfikacja sekwencji tworzona jest na podstawie diagramu stanów, wówczas jest on traktowany jako automat skończony dla którego alfabetem będą metody wykonywane w odpowiedzi na konkretne zdarzenia.

Metoda specyfikacji sekwencji pozwala sprawdzić, czy na rzecz obiektu danej klasy została wywołana sekwencja metod, która należy do specyfikacji tej klasy. Testowana jest poprawność sposobu wykorzystania obiektów testowanej klasy przez program lub obiekty innych klas, co może być podstawą do przeprowadzenia przy użyciu tej metody testów integracyjnych. Pozwala ona na wykrycie błędów w implementacji testowanej klasy tylko w przypadku, gdy sekwencja wywoływanych metod zależy od wyników ich działania.

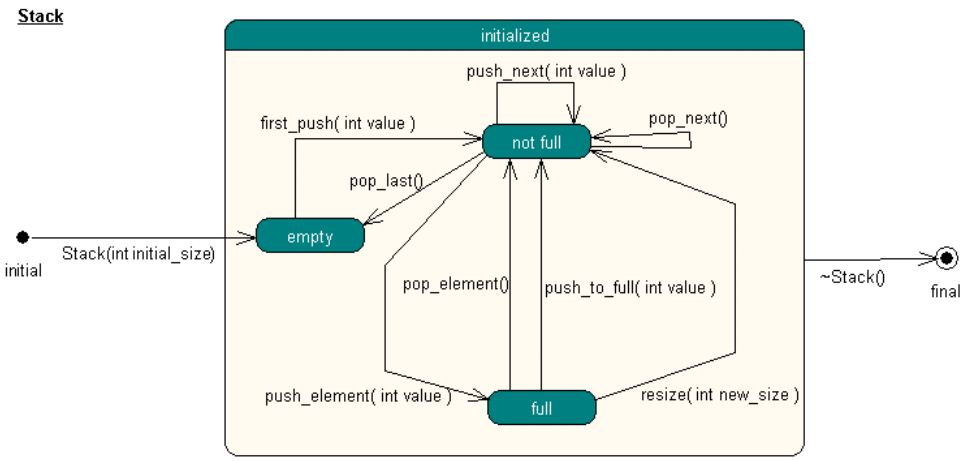
4. Przykład zastosowań

Klasa `Stack` [4] powstała na potrzeby przedstawienia procesu testowania metodą wektora stanu. Została ona zaimplementowana w języku C++ i realizuje funkcjonalność stosu w tablicy.

Deklaracja klasy w języku C++ wygląda następująco:

```
class Stack
{
public:
    Stack(int initial_size);
    ~Stack();
    void Push(const int value);
    int Pop();
protected:
    int size;     //rozmiar tablicy pStack
    int top;      //indeks pierwszego wolnego elementu w pStack
    int * pStack;
    void Resize(const int new_size);
};
```

Diagram stanów opisujący zachowanie klasy przedstawia rysunek 2:



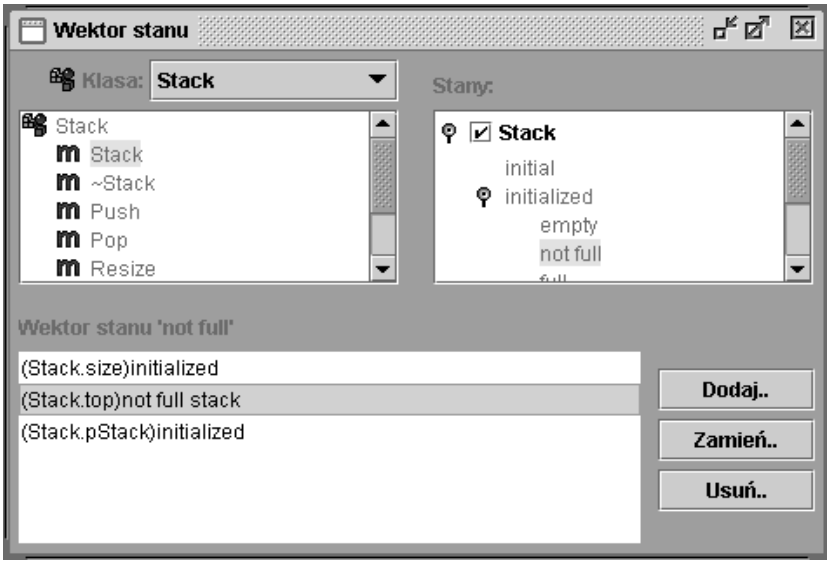
Rysunek 2 Diagram stanów dla klasy Stack

Zgodnie z ogólnym schematem postępowania dla metody wektora stanu, przed przystąpieniem do wykonywania testów konieczne jest określenie wartości dla atrybutów oraz wektorów stanu dla wszystkich stanów. Dla atrybutów klasy Stack zdefiniowano następujące wartości:

Atrybut	Nazwa wartości	Typ wartości	Wartość
size	initialized	ogólna	size>0
top	correct value	ogólna	top>=0
	empty stack	szczegółowa	0
	full stack	dynamiczna	top==size
	not full stack	dynamiczna	top>0 && top<size
pStack	initialized	ogólna	pStack!=NULL

Korzystając z powyższych wartości, dla każdego stanu zdefiniowano następujące wektory stanów (rysunek 3):

- initialized = [size.initialized, top.correct value, pStack.initialized]
- empty = [size.initialized, top.empty stack, pStack.initialized]
- full = [size.initialized, top.full_stack, pStack.initialized]
- not_full = [size.initialized] top.not_full_stack, pStack.initialized]



Rysunek 3 Okno definicji wektorów stanów

Ze względu na specjalny charakter stanów **initial** oraz **final** nie są dla nich definiowane wektory stanów. Metoda wektora stanu pozwala dla danej metody na pominięcie procedury ustalania stanu obiektu i określenie stanu „odgórnie”.

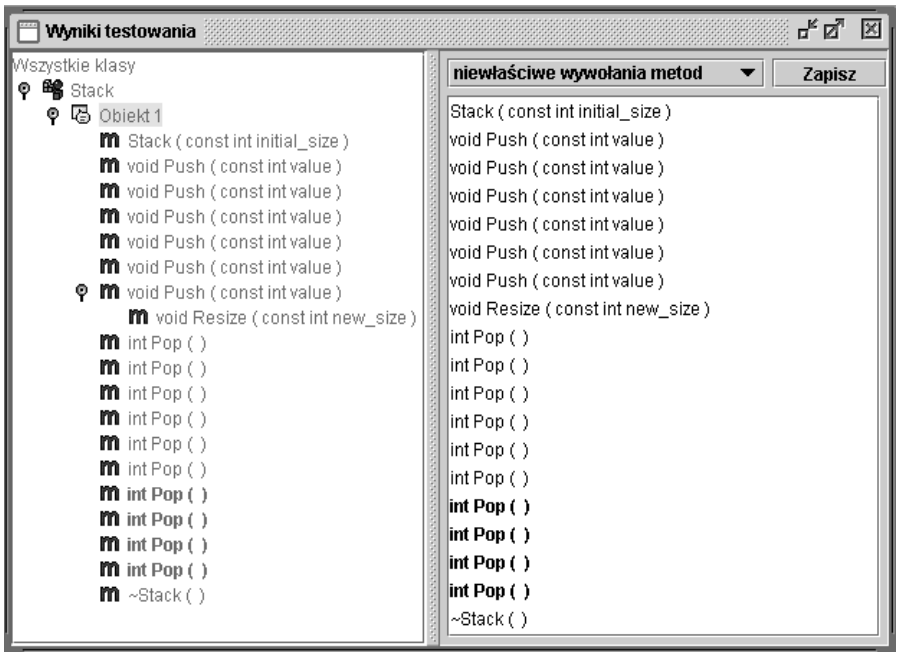
Sytuacja ta ma miejsce w przypadku konstruktora oraz destruktora oraz stanów **initial** oraz **final** z nimi związanymi.

Test wykonany dla przypadku testowego określonego jako {rozmiar początkowy 5; sekwencja komunikatów: push, push, push, push, push, push, pop, pop, pop, pop, pop, pop, pop, pop, pop, pop } spowodował zasygnalizowanie wystąpienia wywołań metod, dla których nie znaleziono odpowiedniego przejścia w diagramie zmian stanów. W oknie rezultatów środowiska TOSTER (Rysunek 4) wywołania błędne zaznaczone są pogrubioną czcionką.

Dla dowolnej z metod, które zostały zakwalifikowane jako niepoprawne (prezentowane pogrubioną czcionką na rysunku 4) system generuje również raport:

```
Metoda została wywołana dla niedozwolonego stanu obiektu.
Przed wykonaniem metody obiekt znajdował się w następujących
stanach:
Stack/initialized
Stack/initialized/empty

Dopuszczalne metody:
void Push ( const int value )
~Stack (  )
```



Rysunek 4 Okno wyników po wykonaniu testu dla klasy Stack

5. Uwagi końcowe

Środowisko TOSTER zrealizowane w Instytucie Informatyki PW wspomaga testowanie klas napisanych w językach C++ i Java a będących częścią systemów zaprojektowanych w narzędziach Paradigm Plus lub Select Enterprice. Testowanie klas w izolacji nie jest wystarczające [2], konieczne jest testowanie integracyjne klas. Obecnie TOSTER nie posiada możliwości testowania grupy współpracujących klas, jednak ze względu na otwartość środowiska dodanie tej możliwości może wkrótce nastąpić.

Testowanie w oparciu o diagram zmian stanów opisujący zachowanie w czasie obiektu jest podejściem właściwym (patrz [1]) dla systemów obiektowych. Zgodnie z informacjami autorów komercyjne narzędzia wspomagające testowanie nie stosują w chwili obecnej tego typu metod testowania.

Bibliografia

[1]. R. V. Binder: „Testing Object-Oriented Systems" Addison Wesley 2000

[2]. I. Bluemke: „Podstawy testowania oprogramowania obiektowego", Pro Dialog nr 12, 2001

[3]. I. Bluemke, A. Lasota, S. Nowak: „Testowanie oprogramowania obiektowego" Raport Badawczy nr 01/2001, Instytut Informatyki, Politechnika Warszawska; styczeń 2001.

[4]. A. Lasota, S. Nowak „Testowanie oprogramowania obiektowego" praca magisterska; Instytut Informatyki Politechniki Warszawskiej, marzec 2001 r.

[5]. S. Kirami W.T. Tsai: „Method sequence specification and verification of classes", J. Object-Oriented Programming, Oct. 1994, pp.28-38

[6]. C.D. Turner and D.J. Robson „The state-based testing of object-oriented programs" Proc. IEEE Conf. Software Maintenance. 1993. pp. 302-310