Inżynieria Programowania — Testowanie oprogramowania

Arkadiusz Chrobot

Katedra Informatyki, Politechnika Świętokrzyska w Kielcach

Kielce, 19 stycznia 2020

Plan wykładu

Plan wykładu

Plan wykładu

Plan wykładu

Plan wykładu

Plan wykładu

Motto

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

Donald E. Knuth

Wstęp

Proces testowania może obejmować cztery poziomy:

1. testowanie pojedynczych jednostek programów,
2. testowanie interakcji jednostek połączonych w podsystemy i systemy,
3. testowanie systemowe,
4. testowanie na zgodność ze specyfikacją, przy pomocy testów akceptacyjnych.

Wstęp

Proces testowania może obejmować cztery poziomy:

1. testowanie pojedynczych jednostek programów,
2. testowanie interakcji jednostek połączonych w podsystemy i systemy,
3. testowanie systemowe,
4. testowanie na zgodność ze specyfikacją, przy pomocy testów akceptacyjnych.

Wstęp

Proces testowania może obejmować cztery poziomy:

1. testowanie pojedynczych jednostek programów,
2. testowanie interakcji jednostek połączonych w podsystemy i systemy,
3. testowanie systemowe,
4. testowanie na zgodność ze specyfikacją, przy pomocy testów akceptacyjnych.

Wstęp

Proces testowania może obejmować cztery poziomy:

1. testowanie pojedynczych jednostek programów,
2. testowanie interakcji jednostek połączonych w podsystemy i systemy,
3. testowanie systemowe,
4. testowanie na zgodność ze specyfikacją, przy pomocy testów akceptacyjnych.

Fazy testowania

Testowanie obiektowe

Z punktu widzenia testowania systemy obiektowe różnią się od systemów strukturalnych w dwóch zasadniczych kwestiach:

1. W systemach strukturalnych istnieje jasne rozróżnienie między podstawowymi jednostkami programów, a kolekcjami tych jednostek. W systemach obiektowych nie ma takiego rozgraniczenia.
2. Nie ma jasnej hierarchii obiektów (w sensie przepływu sterowania), jaka występuje w systemach strukturalnych.

Testowanie obiektowe

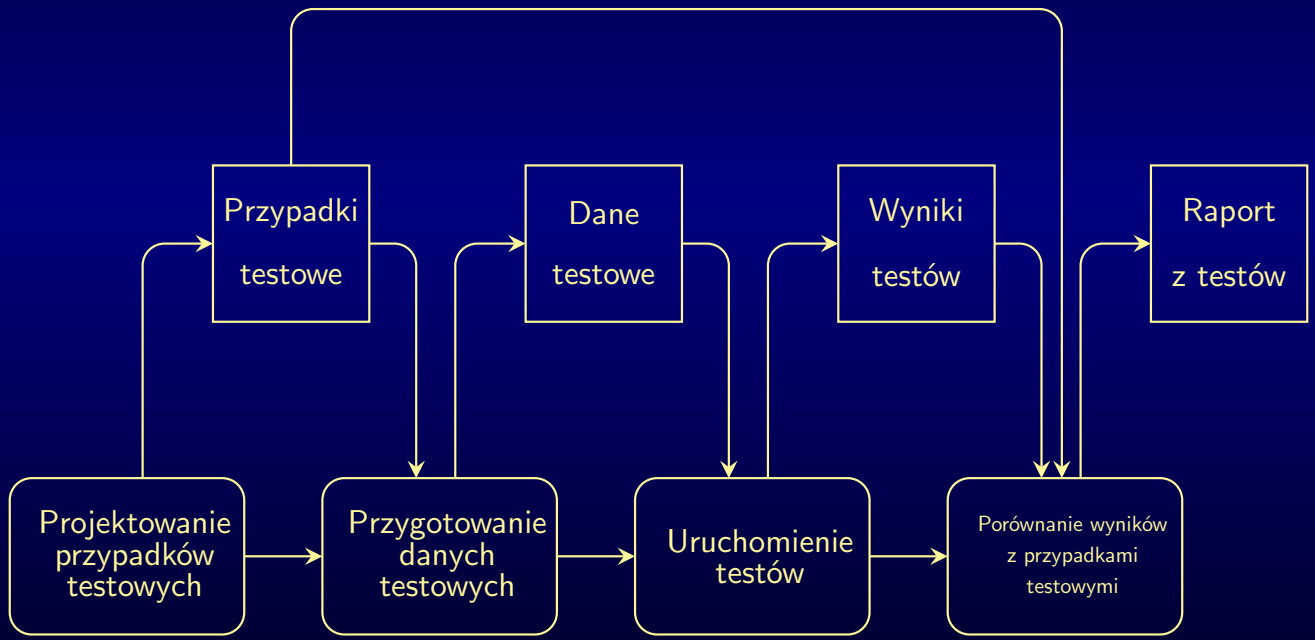
Z punktu widzenia testowania systemy obiektowe różnią się od systemów strukturalnych w dwóch zasadniczych kwestiach:

1. W systemach strukturalnych istnieje jasne rozróżnienie między podstawowymi jednostkami programów, a kolekcjami tych jednostek. W systemach obiektowych nie ma takiego rozgraniczenia.
2. Nie ma jasnej hierarchii obiektów (w sensie przepływu sterowania), jaka występuje w systemach strukturalnych.

Testowanie defektów

Testowanie defektów ma na celu ujawnienie utajnionych defektów w systemie oprogramowania przed jego dostarczeniem. Pozytywny test defektu to taki, który powoduje, że system działa *niepoprawnie*, ujawniając defekt. Z tego wynika podstawowa własność testowania - testowanie wykazuje *obecność*, a nie brak defektu.

Proces testowania defektów



Strategie testowania

Testowanie musi być oparte na pewnym podzbiorze dopuszczalnych przypadków testowych. Strategie wyboru takich przypadków powinny być opracowane przez firmę, a nie przez zespół wytwórczy. Ich podstawą mogą być ogólne strategie testowania lub doświadczenia nabyte podczas testowania systemu. Oto przykłady takich strategii:

1. Należy przetestować wszystkie funkcje systemu dostępne z menu.
2. Należy przetestować kombinacje funkcji (np. formatowanie tekstu) dostępnych z tego samego menu.
3. Należy przetestować wszystkie funkcje, w których użytkownik wprowadza dane, zarówno na poprawnych, jak i niepoprawnych danych wejściowych.

Strategie testowania

Testowanie musi być oparte na pewnym podzbiorze dopuszczalnych przypadków testowych. Strategie wyboru takich przypadków powinny być opracowane przez firmę, a nie przez zespół wytwórczy. Ich podstawą mogą być ogólne strategie testowania lub doświadczenia nabyte podczas testowania systemu. Oto przykłady takich strategii:

1. Należy przetestować wszystkie funkcje systemu dostępne z menu.
2. Należy przetestować kombinacje funkcji (np. formatowanie tekstu) dostępnych z tego samego menu.
3. Należy przetestować wszystkie funkcje, w których użytkownik wprowadza dane, zarówno na poprawnych, jak i niepoprawnych danych wejściowych.

Strategie testowania

Testowanie musi być oparte na pewnym podzbiorze dopuszczalnych przypadków testowych. Strategie wyboru takich przypadków powinny być opracowane przez firmę, a nie przez zespół wytwórczy. Ich podstawą mogą być ogólne strategie testowania lub doświadczenia nabyte podczas testowania systemu. Oto przykłady takich strategii:

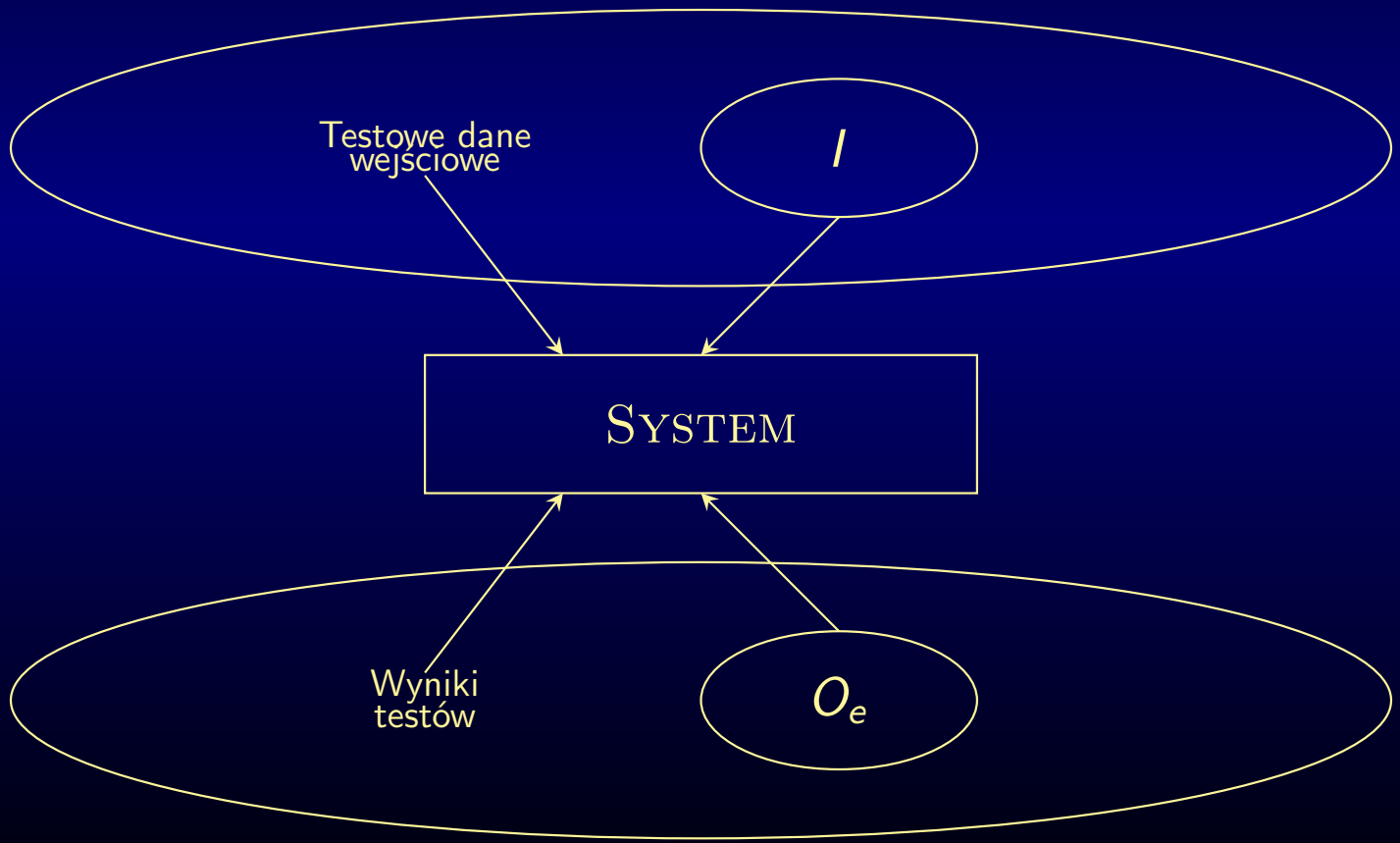
1. Należy przetestować wszystkie funkcje systemu dostępne z menu.
2. Należy przetestować kombinacje funkcji (np. formatowanie tekstu) dostępnych z tego samego menu.
3. Należy przetestować wszystkie funkcje, w których użytkownik wprowadza dane, zarówno na poprawnych, jak i niepoprawnych danych wejściowych.

Testowanie funkcjonalne

Testowanie funkcjonalne, nazywane *testowaniem czarnej skrzynki* to podejście, w którym testy wyprowadza się ze specyfikacji programu lub komponentu. System traktuje się jako *czarną skrzynkę*, której działanie można poznać jedynie na podstawie danych wejściowych i związanych z nimi danych wyjściowych.

Testowanie funkcjonalne - model systemu

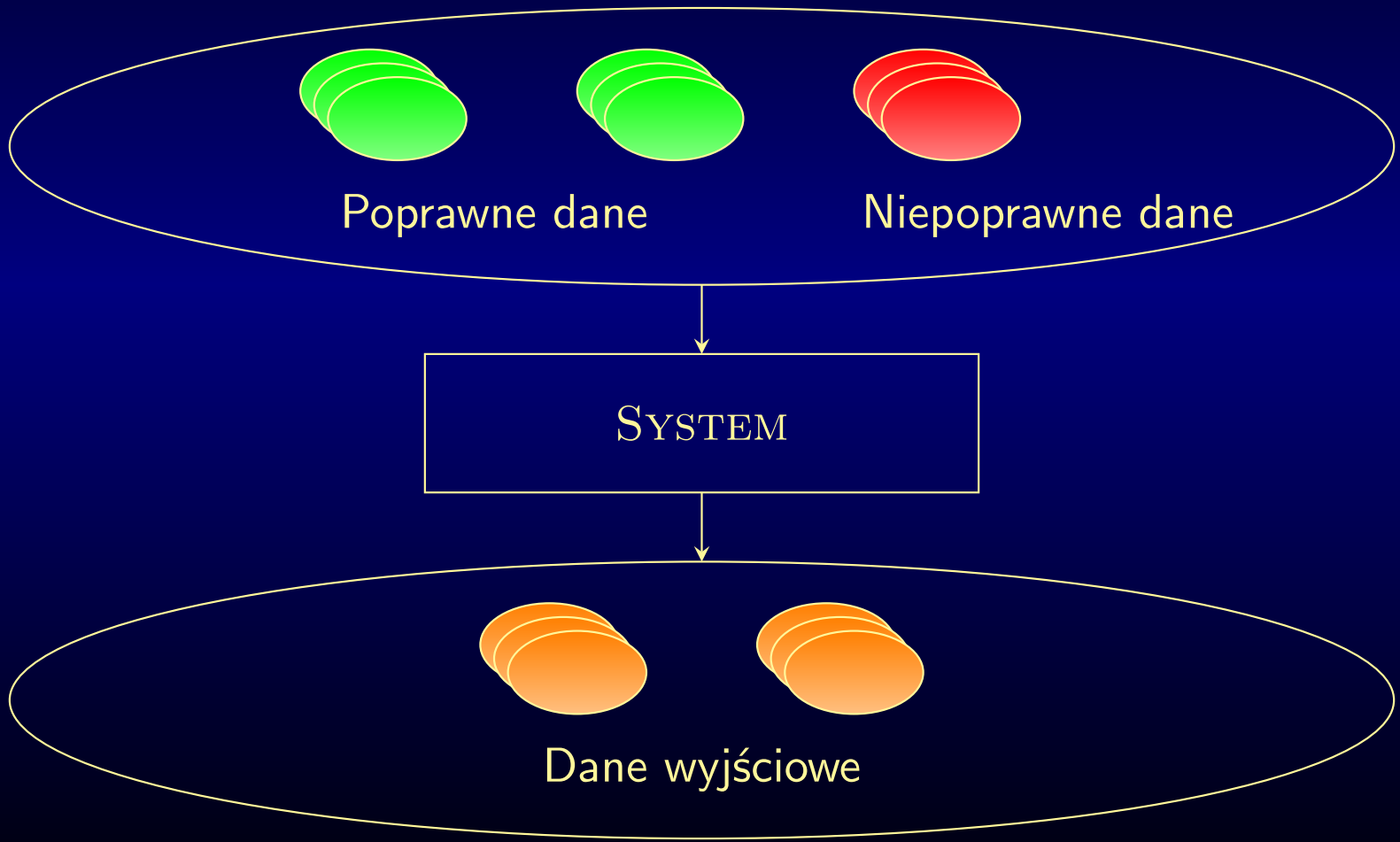
I - dane powodujące błędne zachowanie
O_e - dane wyjściowe świadczące o istnieniu defektu



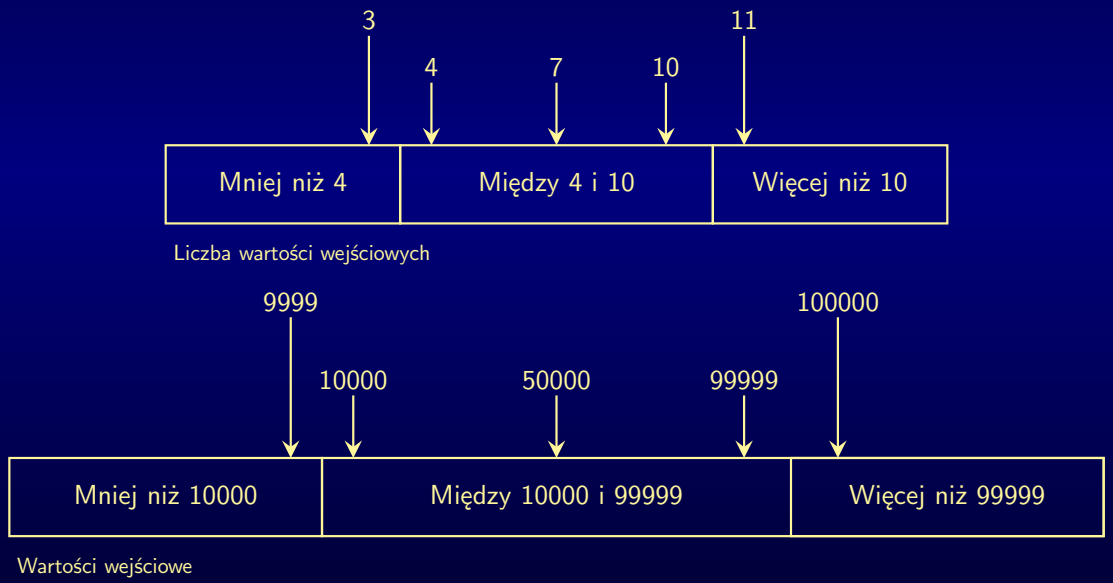
Klasy równoważności

Dane wejściowe programu można podzielić na kilka różnych klas. Podstawą tego podziału są wspólne właściwości np. liczby dodatnie, liczby ujemne, ciągi znaków bez spacji. Programy działają zwykle w porównywalny sposób dla wszystkich elementów tej samej klasy, dlatego nazywamy je *klasami równoważności* lub *dziedzinami*. Jedna z metod systematycznego testowania defektów polega na zidentyfikowaniu wszystkich klas równoważności, które muszą być obsłużone przez program. Przypadki testowe projektuje się tak, aby dane wejściowe i wyjściowe leżały wewnątrz tych klas.

Podział na klasy równoważności



Klasy równoważności



Znajdowanie przypadków testowych

Znajdowanie przypadków testowych zostanie zobrazowane na przykładzie uproszczonej specyfikacji podprogramu wyszukującego.

Specyfikacja

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

Pre-condition

--ciąg ma co najmniej jeden element

T'FIRST <=T'LAST

Post-condition

--znaleziono element i wskazuje go L

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

or

--element nie występuje w ciągu

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

Klasy równoważności

Na podstawie specyfikacji można wskazać dwie oczywiste klasy równoważności:

- dane wejściowe, w których element szukany występuje w ciągu (**Found=true**),
- dane wejściowe, w których element szukany nie wstępuje w ciągu (**Found=false**).

Klasy równoważności

Na podstawie specyfikacji można wskazać dwie oczywiste klasy równoważności:

- dane wejściowe, w których element szukany występuje w ciągu (**Found=true**),
- dane wejściowe, w których element szukany nie wstępuje w ciągu (**Found=false**).

Porady dotyczące testowania

Porady dotyczące testowania umożliwiają wyszukiwanie klas równoważności innych, niż te na które wskazuje bezpośrednio specyfikacja. Oto przykłady takich porad dla ciągów wartości:

1. Przetestuj oprogramowanie na ciągach, które składają się tylko z jednego elementu.
2. Użyj odrębnych ciągów w różnych testach.
3. Opracuj testy, w których należy odczytać pierwszy, środkowy i ostatni element w ciągu.

Porady dotyczące testowania

Porady dotyczące testowania umożliwiają wyszukiwanie klas równoważności innych, niż te na które wskazuje bezpośrednio specyfikacja. Oto przykłady takich porad dla ciągów wartości:

1. Przetestuj oprogramowanie na ciągach, które składają się tylko z jednego elementu.
2. Użyj odrębnych ciągów w różnych testach.
3. Opracuj testy, w których należy odczytać pierwszy, środkowy i ostatni element w ciągu.

Porady dotyczące testowania

Porady dotyczące testowania umożliwiają wyszukiwanie klas równoważności innych, niż te na które wskazuje bezpośrednio specyfikacja. Oto przykłady takich porad dla ciągów wartości:

1. Przetestuj oprogramowanie na ciągach, które składają się tylko z jednego elementu.
2. Użyj odrębnych ciągów w różnych testach.
3. Opracuj testy, w których należy odczytać pierwszy, środkowy i ostatni element w ciągu.

Klasy równoważności - uzupełnienie

Korzystając z przytoczonych wcześniej porad, można wskazać dwie dalsze klasy równoważności:

- Ciąg wejściowy zawiera jedną wartość.
- Liczba elementów ciągu wejściowego jest większa niż 1.

Klasy równoważności - uzupełnienie

Korzystając z przytoczonych wcześniej porad, można wskazać dwie dalsze klasy równoważności:

- Ciąg wejściowy zawiera jedną wartość.
- Liczba elementów ciągu wejściowego jest większa niż 1.

Klasy równoważności i przypadki testowe

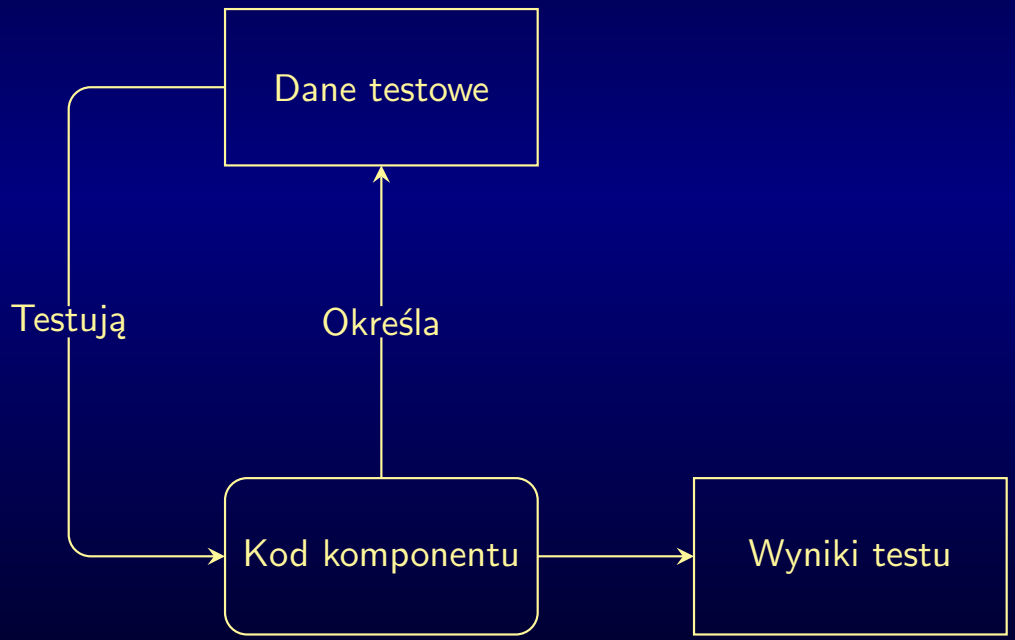
Tablica	**Element**
Jedna wartość	Jest w ciągu
Jedna wartość	Nie ma jej w ciągu
Więcej niż 1 wartość	Jest pierwszym elementem ciągu
Więcej niż 1 wartość	Jest ostatnim elementem ciągu
Więcej niż 1 wartość	Jest środkowym elementem ciągu
Więcej niż 1 wartość	Nie ma jej w ciągu

Ciąg wejściowy (T)	**Klucz(Key)**	**Wynik (Found,L)**
17	17	true,1
17	0	false,??
17,29,21,23	17	true,1
41,18,9,31,30,16,45	45	true,7
17,18,21,23,29,41,38	23	true,4
21,23,29,33,38	25	false,??

Testowanie strukturalne

Testowanie strukturalne nazywane również *testowaniem przezroczystej skrzynki* to metoda, w której opracowuje się testy na podstawie wiedzy o strukturze i implementacji oprogramowania. Stosuje się je zwykle do niewielkich jednostek kodu źródłowego, takich jak podprogramy i metody obiektów.

Testowanie strukturalne - schemat



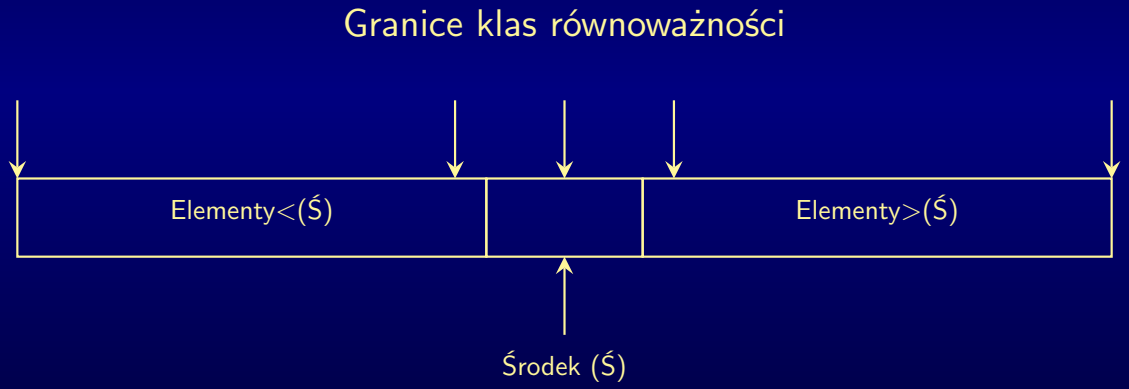
Testowanie strukturalne - przykład

Testowanie strukturalne zostanie przedstawione na przykładzie metody wyszukiwania binarnego zaimplementowanej w języku Java. Należy pamiętać, że taka metoda ma silniejszy warunek wejściowy niż jej specyfikacja - tablica musi być posortowana niemalejąco.

Kod źródłowy

```
class BinSearch {
//Klasa z metodą wyszukiwania binarnego, która pobiera uporządkowaną
//tablicę oraz i zwraca obiekt z dwoma atrybutami:
//index - indeks elementu w tablicy
//found - wartość logiczna wskazująca, czy klucz znajduje się w tablicy
//Atrybut „index" ma wartość -1, jeśli klucza nie ma w tablicy.
 public static void search(int key, int[] elemArray, Result r)
 {
        int bottom=0;
        int top=elemArray.length-1;
        int mid;
        r.found=false; r.index=-1;
        while(bottom<=top)
        {
                int mid=(top+bottom)/2;
                if(elemArray[mid] == key)
                {
                        r.index=mid;
                        r.found=true;
                        return;
                }
                else
                {
                        if(elemArray[mid]<key)
                                bottom=mid+1;
                        else
                                top=mid-1;
                }
        }
 }
}
```

Klasy równoważności dla wyszukiwania binarnego



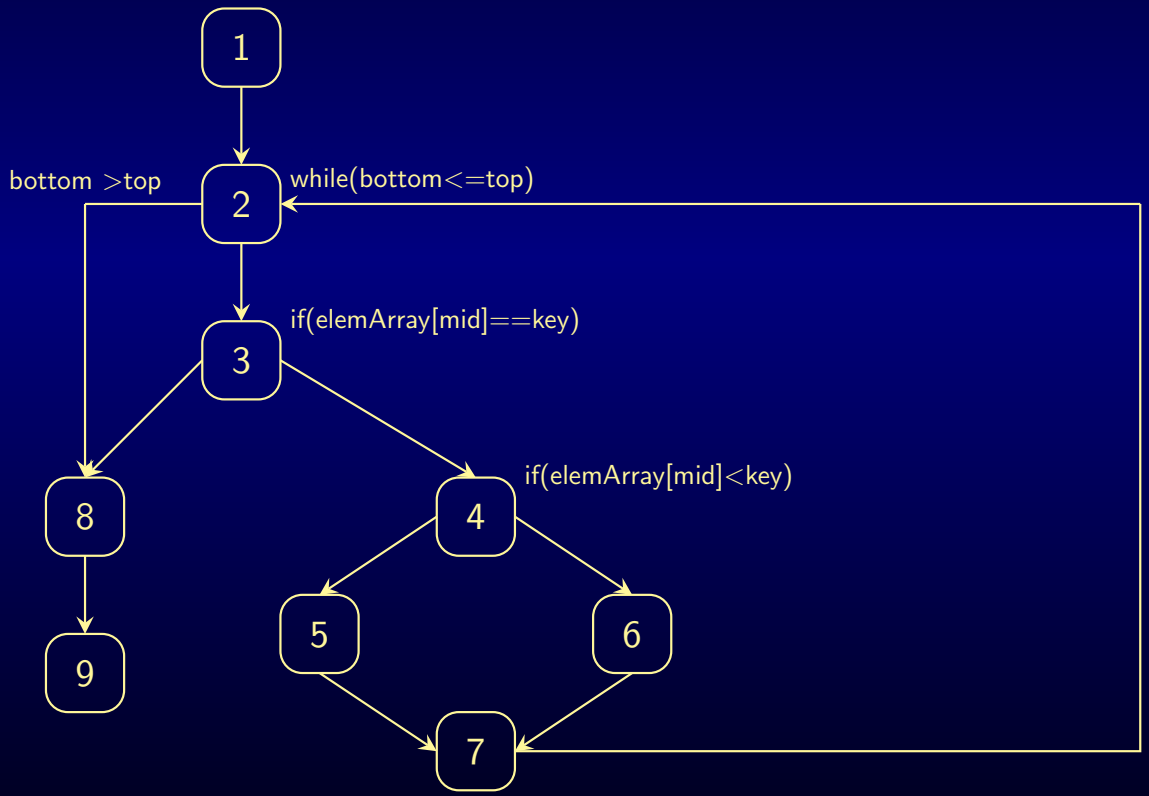
Przypadki testowe dla wyszukiwania binarnego

Tablica 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,??

Testowanie ścieżek

Testowanie ścieżek to metoda której celem jest zbadanie każdej niezależnej ścieżki wykonania komponentu lub programu. Dzięki przetestowaniu każdej możliwej ścieżki uzyskuje się pewność, że wykonano każdą pojedynczą instrukcję komponentu co najmniej raz. Każda instrukcja strukturalna zostaje sprawdzona zarówno w przypadku prawdy jak i fałszu. W tej metodzie nie sprawdza się każdej *kombinacji* wykonania niezależnych ścieżek.

Graf strumieni



Niezależne ścieżki wykonania

Niezależne ścieżki wykonania w omawianym grafie to:
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
Jeśli wykonamy wszystkie te ścieżki, to możemy być pewni, że:

1. Każda instrukcja metody wyszukiwania binarnego została wykonana co najmniej raz.
2. Każde rozgałęzienie zbadano dla przypadku prawdy i fałszu.

Niezależne ścieżki wykonania

Niezależne ścieżki wykonania w omawianym grafie to:
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
Jeśli wykonamy wszystkie te ścieżki, to możemy być pewni, że:

1. Każda instrukcja metody wyszukiwania binarnego została wykonana co najmniej raz.
2. Każde rozgałęzienie zbadano dla przypadku prawdy i fałszu.

Złożoność cykliczna

Liczbę niezależnych ścieżek w programie można wyznaczyć obliczając złożoność cykliczną grafu strumieni programu, według następującego wzoru:

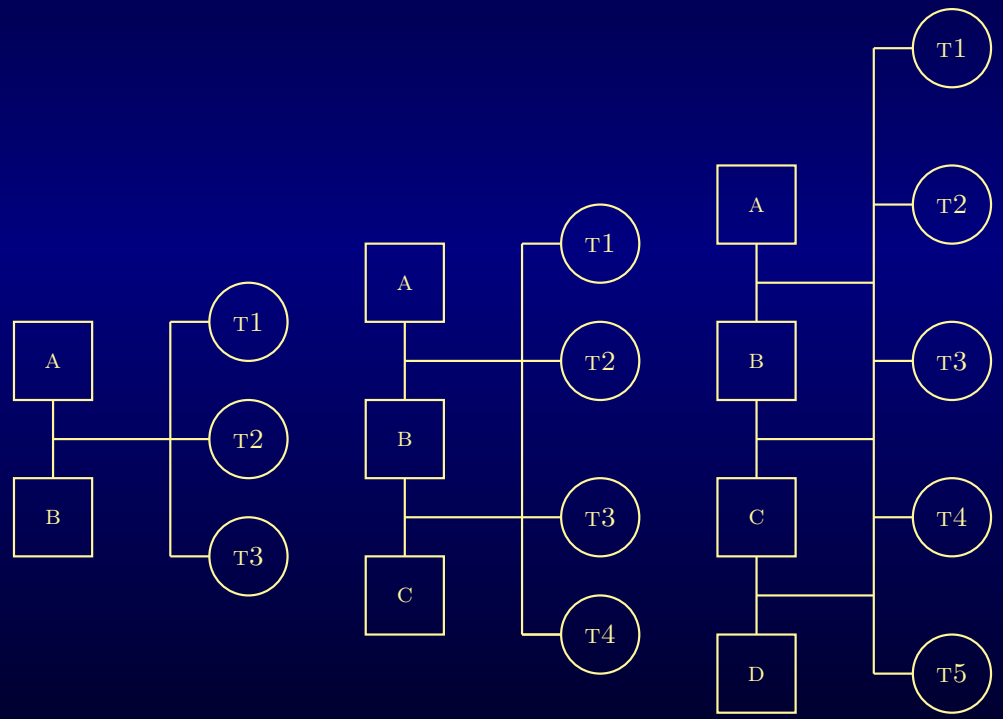
$$ZC(G) = Liczba(krawędzi) - Liczba(węzłów) + 2$$

W przypadku programów bez instrukcji skoku złożoność cykliczna jest o jeden większa niż liczba warunków prostych w programie. Złożoność cykliczna oprócz liczby ścieżek wyznacza również minimalną liczbę przypadków testowych, niezbędnych do ich przetestowania.

Testowanie integracyjne

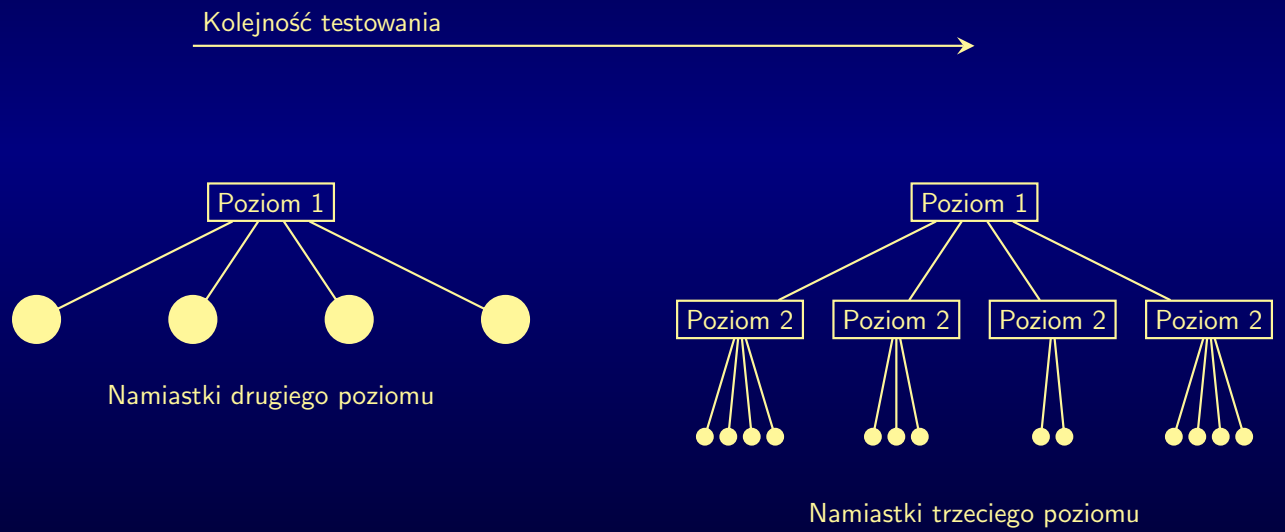
Po odrębnym przetestowaniu komponentów programu należy je zintegrować w gotowy lub częściowo ukończony system. Ten proces integracji polega na zbudowaniu systemu i przetestowaniu go w poszukiwaniu problemów związanych z interakcją komponentów.

Przyrostowe testowanie integracyjne

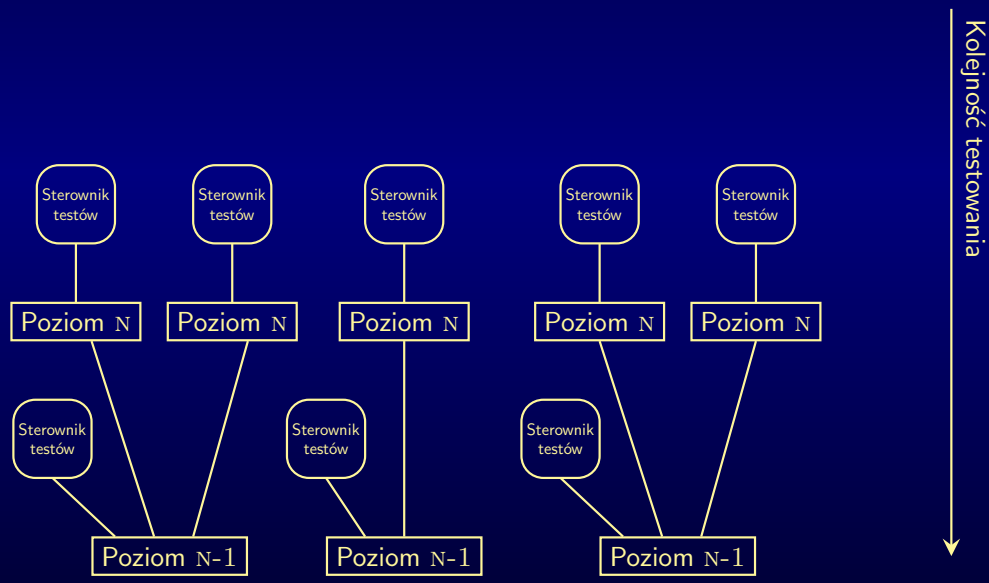


Krok 1 Krok 2 Krok 3

Testowanie zstępujące



Testowanie wstępujące



Porównanie testowania wstępującego i zstępującego

1. *Zatwierdzanie architektury* - testowanie zstępujące daje większą szansę znalezienia błędów w architekturze systemu i projekcie wysokiego poziomu już we wczesnej fazie procesu tworzenia. Przy testowaniu wstępującym do zatwierdzenia projektu wysokiego poziomu dochodzi dopiero w późnej fazie procesu.
2. *Prezentacja systemu* - przy tworzeniu zstępującym ograniczony, ale działający system jest dostępny już we wczesnej fazie budowy. Stanowi to dowód wykonalności systemu oraz pozwala na rozpoczęcie zatwierdzania. Jeśli system jest zbudowany z komponentów wielokrotnego użycia, to pewną formę jego prezentacji można przedstawić także w przypadku podejścia wstępującego.
3. *Implementacja testowania* - testowanie ściśle zstępujące jest trudne w realizacji, ze względu na konieczność opracowania namiastek symulujących niższe poziomy systemu. W przypadku testowania wstępującego trzeba pisać sterowniki testów do badania komponentów niższych poziomów.
4. *Obserwacja testów* - testowanie wstępujące i testowanie zstępujące wiążą się z problemami obserwacji testów. W wypadku wielu systemów ich wyższe poziomy, które trzeba na początku zaimplementować, nie wytwarzają danych wyjściowych. Aby je przetestować, należy je do tego zmusić. W przypadku testowania wstępującego może też zajść potrzeba stworzenia sztucznego środowiska, aby móc obserwować działanie komponentów niższego poziomu.

Porównanie testowania wstępującego i zstępującego

1. *Zatwierdzanie architektury* - testowanie zstępujące daje większą szansę znalezienia błędów w architekturze systemu i projekcie wysokiego poziomu już we wczesnej fazie procesu tworzenia. Przy testowaniu wstępującym do zatwierdzenia projektu wysokiego poziomu dochodzi dopiero w późnej fazie procesu.
2. *Prezentacja systemu* - przy tworzeniu zstępującym ograniczony, ale działający system jest dostępny już we wczesnej fazie budowy. Stanowi to dowód wykonalności systemu oraz pozwala na rozpoczęcie zatwierdzania. Jeśli system jest zbudowany z komponentów wielokrotnego użycia, to pewną formę jego prezentacji można przedstawić także w przypadku podejścia wstępującego.
3. *Implementacja testowania* - testowanie ściśle zstępujące jest trudne w realizacji, ze względu na konieczność opracowania namiastek symulujących niższe poziomy systemu. W przypadku testowania wstępującego trzeba pisać sterowniki testów do badania komponentów niższych poziomów.
4. *Obserwacja testów* - testowanie wstępujące i testowanie zstępujące wiążą się z problemami obserwacji testów. W wypadku wielu systemów ich wyższe poziomy, które trzeba na początku zaimplementować, nie wytwarzają danych wyjściowych. Aby je przetestować, należy je do tego zmusić. W przypadku testowania wstępującego może też zajść potrzeba stworzenia sztucznego środowiska, aby móc obserwować działanie komponentów niższego poziomu.

Porównanie testowania wstępującego i zstępującego

1. *Zatwierdzanie architektury* - testowanie zstępujące daje większą szansę znalezienia błędów w architekturze systemu i projekcie wysokiego poziomu już we wczesnej fazie procesu tworzenia. Przy testowaniu wstępującym do zatwierdzenia projektu wysokiego poziomu dochodzi dopiero w późnej fazie procesu.
2. *Prezentacja systemu* - przy tworzeniu zstępującym ograniczony, ale działający system jest dostępny już we wczesnej fazie budowy. Stanowi to dowód wykonalności systemu oraz pozwala na rozpoczęcie zatwierdzania. Jeśli system jest zbudowany z komponentów wielokrotnego użycia, to pewną formę jego prezentacji można przedstawić także w przypadku podejścia wstępującego.
3. *Implementacja testowania* - testowanie ściśle zstępujące jest trudne w realizacji, ze względu na konieczność opracowania namiastek symulujących niższe poziomy systemu. W przypadku testowania wstępującego trzeba pisać sterowniki testów do badania komponentów niższych poziomów.
4. *Obserwacja testów* - testowanie wstępujące i testowanie zstępujące wiążą się z problemami obserwacji testów. W wypadku wielu systemów ich wyższe poziomy, które trzeba na początku zaimplementować, nie wytwarzają danych wyjściowych. Aby je przetestować, należy je do tego zmusić. W przypadku testowania wstępującego może też zajść potrzeba stworzenia sztucznego środowiska, aby móc obserwować działanie komponentów niższego poziomu.

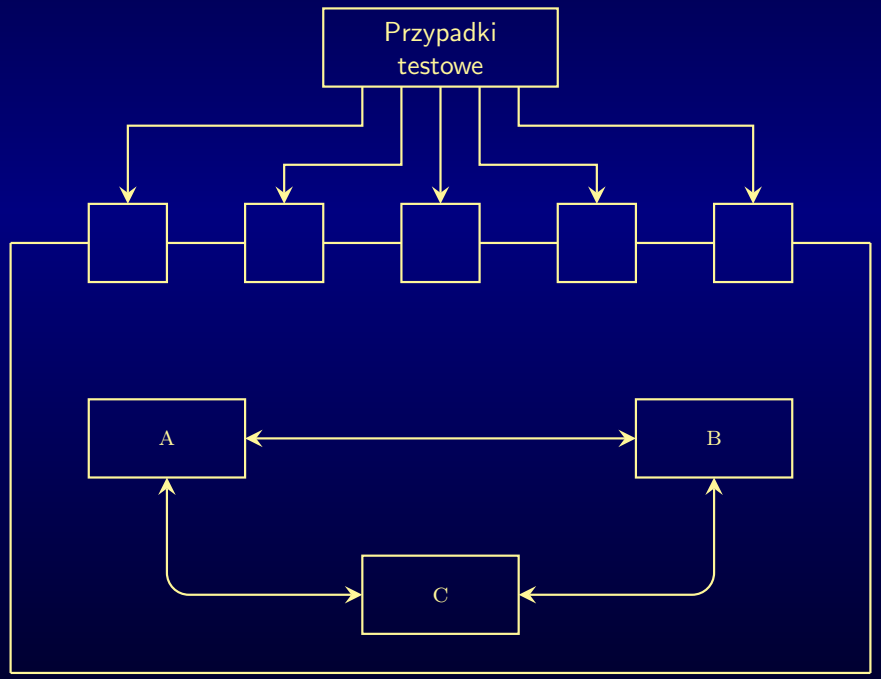
Porównanie testowania wstępującego i zstępującego

1. *Zatwierdzanie architektury* - testowanie zstępujące daje większą szansę znalezienia błędów w architekturze systemu i projekcie wysokiego poziomu już we wczesnej fazie procesu tworzenia. Przy testowaniu wstępującym do zatwierdzenia projektu wysokiego poziomu dochodzi dopiero w późnej fazie procesu.
2. *Prezentacja systemu* - przy tworzeniu zstępującym ograniczony, ale działający system jest dostępny już we wczesnej fazie budowy. Stanowi to dowód wykonalności systemu oraz pozwala na rozpoczęcie zatwierdzania. Jeśli system jest zbudowany z komponentów wielokrotnego użycia, to pewną formę jego prezentacji można przedstawić także w przypadku podejścia wstępującego.
3. *Implementacja testowania* - testowanie ściśle zstępujące jest trudne w realizacji, ze względu na konieczność opracowania namiastek symulujących niższe poziomy systemu. W przypadku testowania wstępującego trzeba pisać sterowniki testów do badania komponentów niższych poziomów.
4. *Obserwacja testów* - testowanie wstępujące i testowanie zstępujące wiążą się z problemami obserwacji testów. W wypadku wielu systemów ich wyższe poziomy, które trzeba na początku zaimplementować, nie wytwarzają danych wyjściowych. Aby je przetestować, należy je do tego zmusić. W przypadku testowania wstępującego może też zajść potrzeba stworzenia sztucznego środowiska, aby móc obserwować działanie komponentów niższego poziomu.

Testowanie interfejsu

Testowanie interfejsu jest wykonywane po zintegrowaniu modułów lub podsystemów w większe systemy. Każdy moduł posiada zdefiniowany interfejs publiczny, który jest wywoływany przez inne komponenty programu. Celem testowania interfejsu jest wykrycie usterek, które pojawiły się w systemie z powodu błędów w interfejsie lub nieprawdziwych założeń o interfejsach.

Testowanie interfejsu



Rodzaje interfejsów

1. *Interfejsy parametryczne* - są to interfejsy, w których dane lub niekiedy odwołania do funkcji są przekazywane od jednego komponentu do drugiego.
2. *Interfejsy pamięci dzielonej* - są to interfejsy, w których blok pamięci służący do wymiany danych jest dzielony między podsystemami.
3. *Interfejsy proceduralne* - są to interfejsy, w których jeden podsystem obudowuje zbiór procedur wywoływanych przez inne podsystemy.
4. *Interfejs z przekazywaniem komunikatów* -, są to interfejsy, w których jeden podsystem żąda usługi innego przez przesłanie mu komunikatu.

Rodzaje interfejsów

1. *Interfejsy parametryczne* - są to interfejsy, w których dane lub niekiedy odwołania do funkcji są przekazywane od jednego komponentu do drugiego.
2. *Interfejsy pamięci dzielonej* - są to interfejsy, w których blok pamięci służący do wymiany danych jest dzielony między podsystemami.
3. *Interfejsy proceduralne* - są to interfejsy, w których jeden podsystem obudowuje zbiór procedur wywoływanych przez inne podsystemy.
4. *Interfejs z przekazywaniem komunikatów* -, są to interfejsy, w których jeden podsystem żąda usługi innego przez przesłanie mu komunikatu.

Rodzaje interfejsów

1. *Interfejsy parametryczne* - są to interfejsy, w których dane lub niekiedy odwołania do funkcji są przekazywane od jednego komponentu do drugiego.
2. *Interfejsy pamięci dzielonej* - są to interfejsy, w których blok pamięci służący do wymiany danych jest dzielony między podsystemami.
3. *Interfejsy proceduralne* - są to interfejsy, w których jeden podsystem obudowuje zbiór procedur wywoływanych przez inne podsystemy.
4. *Interfejs z przekazywaniem komunikatów* -, są to interfejsy, w których jeden podsystem żąda usługi innego przez przesłanie mu komunikatu.

Rodzaje interfejsów

1. *Interfejsy parametryczne* - są to interfejsy, w których dane lub niekiedy odwołania do funkcji są przekazywane od jednego komponentu do drugiego.
2. *Interfejsy pamięci dzielonej* - są to interfejsy, w których blok pamięci służący do wymiany danych jest dzielony między podsystemami.
3. *Interfejsy proceduralne* - są to interfejsy, w których jeden podsystem obudowuje zbiór procedur wywoływanych przez inne podsystemy.
4. *Interfejs z przekazywaniem komunikatów* -, są to interfejsy, w których jeden podsystem żąda usługi innego przez przesłanie mu komunikatu.

Błędy dotyczące interfejsów

1. *Niewłaściwe użycie interfejsu* - komponent wywołujący inny komponent popełnia błąd użycia tego interfejsu.
2. *Niezrozumienie interfejsu* - twórcy komponentu wywołującego źle zrozumieli specyfikację komponentu wywoływanego i przyjęli nieprawdziwe założenia o jego zachowaniu.
3. *Błędy synchronizacji* - takie błędy zdarzają się w systemach czasu rzeczywistego, które zawierają pamięć dzieloną lub interfejs z przekazywaniem komunikatów.

Błędy dotyczące interfejsów

1. *Niewłaściwe użycie interfejsu* - komponent wywołujący inny komponent popełnia błąd użycia tego interfejsu.
2. *Niezrozumienie interfejsu* - twórcy komponentu wywołującego źle zrozumieli specyfikację komponentu wywoływanego i przyjęli nieprawdziwe założenia o jego zachowaniu.
3. *Błędy synchronizacji* - takie błędy zdarzają się w systemach czasu rzeczywistego, które zawierają pamięć dzieloną lub interfejs z przekazywaniem komunikatów.

Błędy dotyczące interfejsów

1. *Niewłaściwe użycie interfejsu* - komponent wywołujący inny komponent popełnia błąd użycia tego interfejsu.
2. *Niezrozumienie interfejsu* - twórcy komponentu wywołującego źle zrozumieli specyfikację komponentu wywoływanego i przyjęli nieprawdziwe założenia o jego zachowaniu.
3. *Błędy synchronizacji* - takie błędy zdarzają się w systemach czasu rzeczywistego, które zawierają pamięć dzieloną lub interfejs z przekazywaniem komunikatów.

Porady dotyczące testowania interfejsów

1. Przeanalizuj testowany kod i jawnie wypisz wszystkie wywołania zewnętrznych komponentów. Opracuj zbiór testów, w których wartości parametrów zewnętrznych komponentów leżą na granicach zakresów. Takie graniczne wartości dają największą szansę wykrycia niespójności interfejsów.
2. Gdy przez interfejs przekazuje się wskaźniki, zawsze przetestuj ten interfejs z zerowymi wartościami wskaźników.
3. Tam, gdzie komponent jest wywoływany przez interfejs proceduralny, opracuj testy, które powinny spowodować awarię komponentu. Różniące się założenia o awariach są jednym z najczęściej występujących nieporozumień co do specyfikacji.
4. W wypadku systemów z przekazywaniem komunikatów wykonaj testy obciążeniowe. Opracuj testy, które powodują wygenerowanie znacznie większej liczby komunikatów, niż jest to możliwe w praktyce. Mogą one doprowadzić do wykrycia problemów z synchronizacją.
5. Tam, gdzie występuje interakcja kilku komponentów przez pamięć dzieloną, opracuj testy różniące się porządkiem uruchamiania tych komponentów. Testy mogą doprowadzić do wykrycia niejawnych, przyjętych przez programistę założeń o porządku produkcji i konsumpcji dzielonych danych.

Porady dotyczące testowania interfejsów

1. Przeanalizuj testowany kod i jawnie wypisz wszystkie wywołania zewnętrznych komponentów. Opracuj zbiór testów, w których wartości parametrów zewnętrznych komponentów leżą na granicach zakresów. Takie graniczne wartości dają największą szansę wykrycia niespójności interfejsów.
2. Gdy przez interfejs przekazuje się wskaźniki, zawsze przetestuj ten interfejs z zerowymi wartościami wskaźników.
3. Tam, gdzie komponent jest wywoływany przez interfejs proceduralny, opracuj testy, które powinny spowodować awarię komponentu. Różniące się założenia o awariach są jednym z najczęściej występujących nieporozumień co do specyfikacji.
4. W wypadku systemów z przekazywaniem komunikatów wykonaj testy obciążeniowe. Opracuj testy, które powodują wygenerowanie znacznie większej liczby komunikatów, niż jest to możliwe w praktyce. Mogą one doprowadzić do wykrycia problemów z synchronizacją.
5. Tam, gdzie występuje interakcja kilku komponentów przez pamięć dzieloną, opracuj testy różniące się porządkiem uruchamiania tych komponentów. Testy mogą doprowadzić do wykrycia niejawnych, przyjętych przez programistę założeń o porządku produkcji i konsumpcji dzielonych danych.

Porady dotyczące testowania interfejsów

1. Przeanalizuj testowany kod i jawnie wypisz wszystkie wywołania zewnętrznych komponentów. Opracuj zbiór testów, w których wartości parametrów zewnętrznych komponentów leżą na granicach zakresów. Takie graniczne wartości dają największą szansę wykrycia niespójności interfejsów.
2. Gdy przez interfejs przekazuje się wskaźniki, zawsze przetestuj ten interfejs z zerowymi wartościami wskaźników.
3. Tam, gdzie komponent jest wywoływany przez interfejs proceduralny, opracuj testy, które powinny spowodować awarię komponentu. Różniące się założenia o awariach są jednym z najczęściej występujących nieporozumień co do specyfikacji.
4. W wypadku systemów z przekazywaniem komunikatów wykonaj testy obciążeniowe. Opracuj testy, które powodują wygenerowanie znacznie większej liczby komunikatów, niż jest to możliwe w praktyce. Mogą one doprowadzić do wykrycia problemów z synchronizacją.
5. Tam, gdzie występuje interakcja kilku komponentów przez pamięć dzieloną, opracuj testy różniące się porządkiem uruchamiania tych komponentów. Testy mogą doprowadzić do wykrycia niejawnych, przyjętych przez programistę założeń o porządku produkcji i konsumpcji dzielonych danych.

Porady dotyczące testowania interfejsów

1. Przeanalizuj testowany kod i jawnie wypisz wszystkie wywołania zewnętrznych komponentów. Opracuj zbiór testów, w których wartości parametrów zewnętrznych komponentów leżą na granicach zakresów. Takie graniczne wartości dają największą szansę wykrycia niespójności interfejsów.
2. Gdy przez interfejs przekazuje się wskaźniki, zawsze przetestuj ten interfejs z zerowymi wartościami wskaźników.
3. Tam, gdzie komponent jest wywoływany przez interfejs proceduralny, opracuj testy, które powinny spowodować awarię komponentu. Różniące się założenia o awariach są jednym z najczęściej występujących nieporozumień co do specyfikacji.
4. W wypadku systemów z przekazywaniem komunikatów wykonaj testy obciążeniowe. Opracuj testy, które powodują wygenerowanie znacznie większej liczby komunikatów, niż jest to możliwe w praktyce. Mogą one doprowadzić do wykrycia problemów z synchronizacją.
5. Tam, gdzie występuje interakcja kilku komponentów przez pamięć dzieloną, opracuj testy różniące się porządkiem uruchamiania tych komponentów. Testy mogą doprowadzić do wykrycia niejawnych, przyjętych przez programistę założeń o porządku produkcji i konsumpcji dzielonych danych.

Porady dotyczące testowania interfejsów

1. Przeanalizuj testowany kod i jawnie wypisz wszystkie wywołania zewnętrznych komponentów. Opracuj zbiór testów, w których wartości parametrów zewnętrznych komponentów leżą na granicach zakresów. Takie graniczne wartości dają największą szansę wykrycia niespójności interfejsów.
2. Gdy przez interfejs przekazuje się wskaźniki, zawsze przetestuj ten interfejs z zerowymi wartościami wskaźników.
3. Tam, gdzie komponent jest wywoływany przez interfejs proceduralny, opracuj testy, które powinny spowodować awarię komponentu. Różniące się założenia o awariach są jednym z najczęściej występujących nieporozumień co do specyfikacji.
4. W wypadku systemów z przekazywaniem komunikatów wykonaj testy obciążeniowe. Opracuj testy, które powodują wygenerowanie znacznie większej liczby komunikatów, niż jest to możliwe w praktyce. Mogą one doprowadzić do wykrycia problemów z synchronizacją.
5. Tam, gdzie występuje interakcja kilku komponentów przez pamięć dzieloną, opracuj testy różniące się porządkiem uruchamiania tych komponentów. Testy mogą doprowadzić do wykrycia niejawnych, przyjętych przez programistę założeń o porządku produkcji i konsumpcji dzielonych danych.

Testowanie systemowe

Testowanie systemowe polega na sprawdzeniu cech funkcjonalnych i niefunkcjonalnych systemu w środowisku zbliżonym do tego, w którym docelowo system będzie zainstalowany. Może ono obejmować następujące rodzaje testów:

- *testy funkcjonalne* - sprawdzenie poprawności działania wszystkich funkcji systemowych,
- *testy wydajnościowe* - sprawdzenie działania systemu przy normalnym obciążeniu,
- *testy przeciążeniowe* - sprawdzenie działania systemu przy przekroczonym maksymalnym założonym obciążeniu,
- *testy bezpieczeństwa* - sprawdzenie zabezpieczeń zasobów systemu,
- *testy odporności* - sprawdzenie zachowania systemu w warunkach awarii,
- *testy zgodności* - sprawdzenie czy oprogramowanie może pracować na różnych platformach systemowych,
- *testy dokumentacji* - sprawdzenie poprawności dokumentacji.

Testowanie obciążeniowe

Testy efektywności projektuje się po to, aby upewnić się, że system może działać przy założonym obciążeniu. W przypadku niektórych systemów przeprowadza się testy nawet po osiągnięciu maksymalnego zaprojektowanego obciążenia. Ten rodzaj testowania spełnia dwie funkcje:

1. Umożliwia badanie awaryjnego zachowania systemu - testowanie pozwala stwierdzić, czy system *miękko się załamuje*, czy załamanie następuje gwałtownie.
2. Obciążą system i może doprowadzić do ujawnienia defektów, które normalnie pozostałyby nieodkryte.

Testy obciążeniowe obejmują zatem swoim zakresem testy wydajnościowe, przeciążeniowe i odporności.

Testowanie obciążeniowe

Testy efektywności projektuje się po to, aby upewnić się, że system może działać przy założonym obciążeniu. W przypadku niektórych systemów przeprowadza się testy nawet po osiągnięciu maksymalnego zaprojektowanego obciążenia. Ten rodzaj testowania spełnia dwie funkcje:

1. Umożliwia badanie awaryjnego zachowania systemu - testowanie pozwala stwierdzić, czy system *miękko się załamuje*, czy załamanie następuje gwałtownie.
2. Obciążą system i może doprowadzić do ujawnienia defektów, które normalnie pozostałyby nieodkryte.

Testy obciążeniowe obejmują zatem swoim zakresem testy wydajnościowe, przeciążeniowe i odporności.

Testowanie obiektowe

Testowanie obiektowe obejmuje te same etapy, co testowanie systemów strukturalnych, ale występują pewne różnice:

1. Obiekty jako oddzielne komponenty są zwykle większe niż poszczególne funkcje.
2. Obiekty integrowane w podsystem są zwykle luźno powiązane i nie ma jasnego „wierzchołka” systemu.
3. Jeśli obiektów użyto wielokrotnie, to osoby testujące mogą nie mieć dostępu do kodu źródłowego komponentu, a zatem nie mogą przeprowadzić jego analizy.

Testowanie obiektowe

Testowanie obiektowe obejmuje te same etapy, co testowanie systemów strukturalnych, ale występują pewne różnice:

1. Obiekty jako oddzielne komponenty są zwykle większe niż poszczególne funkcje.
2. Obiekty integrowane w podsystem są zwykle luźno powiązane i nie ma jasnego „wierzchołka” systemu.
3. Jeśli obiektów użyto wielokrotnie, to osoby testujące mogą nie mieć dostępu do kodu źródłowego komponentu, a zatem nie mogą przeprowadzić jego analizy.

Testowanie obiektowe

Testowanie obiektowe obejmuje te same etapy, co testowanie systemów strukturalnych, ale występują pewne różnice:

1. Obiekty jako oddzielne komponenty są zwykle większe niż poszczególne funkcje.
2. Obiekty integrowane w podsystem są zwykle luźno powiązane i nie ma jasnego „wierzchołka” systemu.
3. Jeśli obiektów użyto wielokrotnie, to osoby testujące mogą nie mieć dostępu do kodu źródłowego komponentu, a zatem nie mogą przeprowadzić jego analizy.

Poziomy testowania

1. Testowanie poszczególnych metod związanych z obiektami.
2. Testowanie poszczególnych klas obiektów.
3. Testowanie gron obiektów.
4. Testowanie systemu obiektowego.

Poziomy testowania

1. Testowanie poszczególnych metod związanych z obiektami.
2. Testowanie poszczególnych klas obiektów.
3. Testowanie gron obiektów.
4. Testowanie systemu obiektowego.

Poziomy testowania

1. Testowanie poszczególnych metod związanych z obiektami.
2. Testowanie poszczególnych klas obiektów.
3. Testowanie gron obiektów.
4. Testowanie systemu obiektowego.

Poziomy testowania

1. Testowanie poszczególnych metod związanych z obiektami.
2. Testowanie poszczególnych klas obiektów.
3. Testowanie gron obiektów.
4. Testowanie systemu obiektowego.

Testowanie klas obiektów

Przy testowaniu obiektów pełne pokrycie testami powinno obejmować:

1. Oddzielne testowanie wszystkich metod związanych z obiektem.
2. Ustalanie i odczytywanie wszystkich atrybutów związanych z obiektem.
3. Badanie obiektu we wszystkich możliwych stanach, co oznacza, że należy zasymulować wszystkie zdarzenia powodujące zmianę stanu obiektu.

Testowanie klas obiektów

Przy testowaniu obiektów pełne pokrycie testami powinno obejmować:

1. Oddzielne testowanie wszystkich metod związanych z obiektem.
2. Ustalanie i odczytywanie wszystkich atrybutów związanych z obiektem.
3. Badanie obiektu we wszystkich możliwych stanach, co oznacza, że należy zasymulować wszystkie zdarzenia powodujące zmianę stanu obiektu.

Testowanie klas obiektów

Przy testowaniu obiektów pełne pokrycie testami powinno obejmować:

1. Oddzielne testowanie wszystkich metod związanych z obiektem.
2. Ustalanie i odczytywanie wszystkich atrybutów związanych z obiektem.
3. Badanie obiektu we wszystkich możliwych stanach, co oznacza, że należy zasymulować wszystkie zdarzenia powodujące zmianę stanu obiektu.

Integracja obiektów

Istnieją trzy możliwe metody testowania integracyjnego z których można skorzystać:

1. *Testowanie przypadków użycia lub scenariuszy.* Przypadki użycia lub scenariusze są opisami jednego trybu użycia systemu. Podstawą testowania mogą być opisy scenariuszy i grona obiektów utworzone w celu realizacji przypadków użycia związanych z tym trybem użycia.
2. *Testowanie wątków* - polega na testowaniu reakcji systemu na konkretne zdarzenie wejściowe lub zbiór zdarzeń wejściowych. Systemy obiektowe często są sterowane zdarzeniami, a zatem jest to szczególnie dobry sposób testowania. Aby skorzystać z tego podejścia, należy rozpoznać, w jaki sposób przetwarzanie zdarzeń przewija się przez ten system.
3. *Testowanie interakcji obiektów.* - pośredni poziom testowania integracyjnego może być oparty na identyfikacji ścieżek *metoda-komunikat*. Są to ślady poprzez ciągi interakcji obiektów, które kończą się, gdy operacja obiektu nie wywołuje dalszych usług innych obiektów. *Atomowe Funkcje Systemu (AFS)* składają się z pewnego zdarzenia wejściowego i następującego po nim ciągu ścieżek metoda-komunikat zakończonych zdarzeniem wyjściowym. AFS jest podobna do pojęcia wątku w systemach czasu rzeczywistego.

Integracja obiektów

Istnieją trzy możliwe metody testowania integracyjnego z których można skorzystać:

1. *Testowanie przypadków użycia lub scenariuszy.* Przypadki użycia lub scenariusze są opisami jednego trybu użycia systemu. Podstawą testowania mogą być opisy scenariuszy i grona obiektów utworzone w celu realizacji przypadków użycia związanych z tym trybem użycia.
2. *Testowanie wątków* - polega na testowaniu reakcji systemu na konkretne zdarzenie wejściowe lub zbiór zdarzeń wejściowych. Systemy obiektowe często są sterowane zdarzeniami, a zatem jest to szczególnie dobry sposób testowania. Aby skorzystać z tego podejścia, należy rozpoznać, w jaki sposób przetwarzanie zdarzeń przewija się przez ten system.
3. *Testowanie interakcji obiektów.* - pośredni poziom testowania integracyjnego może być oparty na identyfikacji ścieżek *metoda-komunikat*. Są to ślady poprzez ciągi interakcji obiektów, które kończą się, gdy operacja obiektu nie wywołuje dalszych usług innych obiektów. *Atomowe Funkcje Systemu (AFS)* składają się z pewnego zdarzenia wejściowego i następującego po nim ciągu ścieżek metoda-komunikat zakończonych zdarzeniem wyjściowym. AFS jest podobna do pojęcia wątku w systemach czasu rzeczywistego.

Integracja obiektów

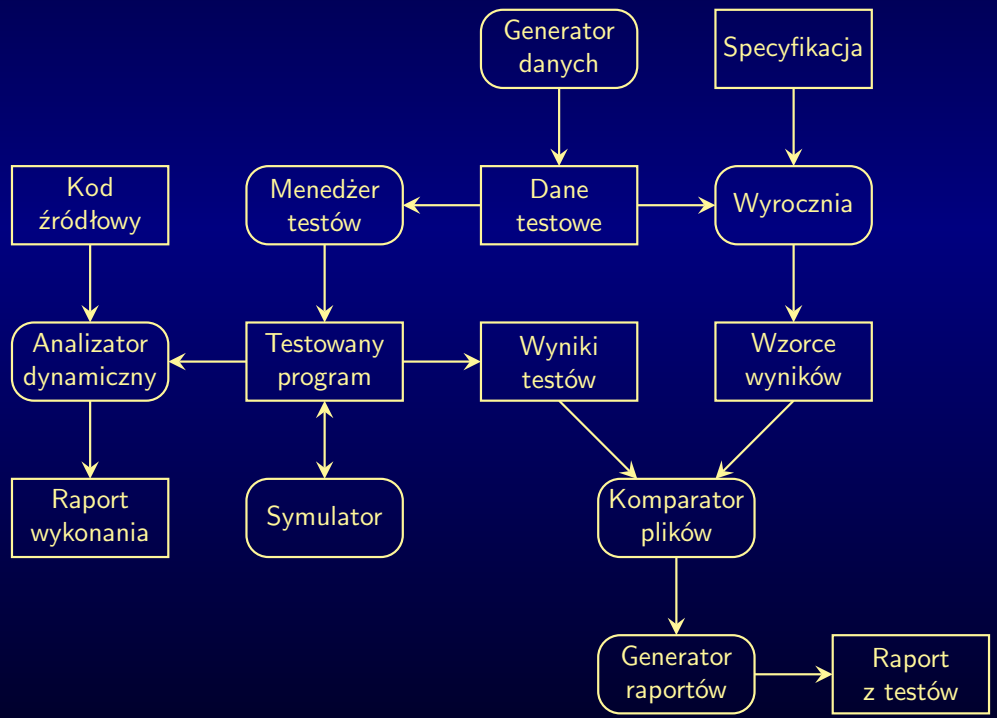
Istnieją trzy możliwe metody testowania integracyjnego z których można skorzystać:

1. *Testowanie przypadków użycia lub scenariuszy.* Przypadki użycia lub scenariusze są opisami jednego trybu użycia systemu. Podstawą testowania mogą być opisy scenariuszy i grona obiektów utworzone w celu realizacji przypadków użycia związanych z tym trybem użycia.
2. *Testowanie wątków* - polega na testowaniu reakcji systemu na konkretne zdarzenie wejściowe lub zbiór zdarzeń wejściowych. Systemy obiektowe często są sterowane zdarzeniami, a zatem jest to szczególnie dobry sposób testowania. Aby skorzystać z tego podejścia, należy rozpoznać, w jaki sposób przetwarzanie zdarzeń przewija się przez ten system.
3. *Testowanie interakcji obiektów.* - pośredni poziom testowania integracyjnego może być oparty na identyfikacji ścieżek *metoda-komunikat*. Są to ślady poprzez ciągi interakcji obiektów, które kończą się, gdy operacja obiektu nie wywołuje dalszych usług innych obiektów. *Atomowe Funkcje Systemu (AFS)* składają się z pewnego zdarzenia wejściowego i następującego po nim ciągu ścieżek metoda-komunikat zakończonych zdarzeniem wyjściowym. AFS jest podobna do pojęcia wątku w systemach czasu rzeczywistego.

Warsztaty do testowania

Narzędzia do testowania tworzą tzw. *warsztat do testowania*. Celem jego zastosowania jest minimalizacja kosztów testowania, które w przypadku dużych projektów mogą obejmować nawet 50% całkowitych kosztów budowy systemu.

Warsztaty do testowania - schemat



Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Narzędzia

W skład narzędzi tworzących warsztat do testów mogą wchodzić:

1. *Menedżer testów.* Zarządza wykonywaniem testów programów. Menedżer testów przechowuje informacje o danych testowych, spodziewanych wynikach i testowanych udogodnieniach programu.
2. *Generator danych testowych.* Generuje dane testowe dla testowanego programu. Może to robić przez wybór danych z bazy danych lub za pomocą wzorców do generowania losowych danych w poprawnej postaci.
3. *Wyrocznia.* Generuje prognozy spodziewanych wyników testów. Wyrocznie mogą być wcześniejszymi wersjami programu albo systemami prototypowymi. Testowanie ramię w ramię polega na jednoczesnym uruchomieniu wyroczni i testowanego programu. Wychwytuje się różnice w wytworzonych przez nie wynikach.
4. *Narzędzie do porównywania plików.* Porównuje wyniki testów programu z poprzednimi wynikami testów i informuje o różnicach między nimi. Narzędzia do porównywania są szczególnie przydatne do testowania regresywnego. Porównuje się wtedy wyniki testów starej i nowej wersji. Różnice między tymi wynikami wskazują potencjalne kłopoty z nową wersją systemu.
5. *Generator raportów.* Oferuje udogodnienia do definiowania i generowania raportów z wyników testowania.
6. *Analizator dynamiczny.* Dodaje do programu kod, którego zadaniem jest zliczanie liczby wykonań każdej instrukcji. Po wykonaniu testów generuje się charakterystykę wykonania, z której wynika, jak często uruchamiano poszczególne instrukcje programu.
7. *Symulator.* Można wykorzystać różne rodzaje symulatorów. Symulatory platformy docelowej imitują maszynę, na której program ma działać. Symulatory interfejsu użytkownika są programami skryptowymi, które imitują wiele jednoczesnych interakcji z użytkownikiem. Użycie symulatorów wejścia-wyjścia daje możliwość powtarzalnego testowania synchronizacji ciągów transakcji.

Dostosowanie

Warsztaty testowe należy dostosować do planu testów każdego systemu:

1. Mogą być potrzebne narzędzia do testowania specyficznych właściwości programu użytkowego.
2. Może zajść potrzeba napisania skryptów do symulatorów interfejsu użytkownika i zdefiniowania wzorców dla generatorów danych wejściowych.
3. Jeśli nie ma żadnej wcześniejszej wersji programu, która mogłaby odegrać rolę wyroczni, to niezbędne może być manualne opracowanie zbiorów oczekiwanych wyników testów.
4. Może zajść potrzeba napisania specjalistycznych narzędzi do porównywania plików.

Dostosowanie

Warsztaty testowe należy dostosować do planu testów każdego systemu:

1. Mogą być potrzebne narzędzia do testowania specyficznych właściwości programu użytkowego.
2. Może zajść potrzeba napisania skryptów do symulatorów interfejsu użytkownika i zdefiniowania wzorców dla generatorów danych wejściowych.
3. Jeśli nie ma żadnej wcześniejszej wersji programu, która mogłaby odegrać rolę wyroczni, to niezbędne może być manualne opracowanie zbiorów oczekiwanych wyników testów.
4. Może zajść potrzeba napisania specjalistycznych narzędzi do porównywania plików.

Dostosowanie

Warsztaty testowe należy dostosować do planu testów każdego systemu:

1. Mogą być potrzebne narzędzia do testowania specyficznych właściwości programu użytkowego.
2. Może zajść potrzeba napisania skryptów do symulatorów interfejsu użytkownika i zdefiniowania wzorców dla generatorów danych wejściowych.
3. Jeśli nie ma żadnej wcześniejszej wersji programu, która mogłaby odegrać rolę wyroczni, to niezbędne może być manualne opracowanie zbiorów oczekiwanych wyników testów.
4. Może zajść potrzeba napisania specjalistycznych narzędzi do porównywania plików.

Dostosowanie

Warsztaty testowe należy dostosować do planu testów każdego systemu:

1. Mogą być potrzebne narzędzia do testowania specyficznych właściwości programu użytkowego.
2. Może zajść potrzeba napisania skryptów do symulatorów interfejsu użytkownika i zdefiniowania wzorców dla generatorów danych wejściowych.
3. Jeśli nie ma żadnej wcześniejszej wersji programu, która mogłaby odegrać rolę wyroczni, to niezbędne może być manualne opracowanie zbiorów oczekiwanych wyników testów.
4. Może zajść potrzeba napisania specjalistycznych narzędzi do porównywania plików.

Pytania

?

Koniec

Dziękuję Państwu za uwagę.