Techniki testowania

1. Kategorie

Kandydat potrafi wyjśnić cechy charakterystyczne i elementy wspólne czarnoskrzynkowych technik testowania, białoskrzynkowych technik testowania oraz technik testowania opartych na doświadczeniu, a także różnice między nimi.

1. Czarnoskrzynkowe (techniki behawioralne, oparte na specyfikacji)
2. Białoskrzynkowe
3. Oparte na doświadczeniu
4. Czarnoskrzynkowe techniki

Kandydat potrafi zaprojektować przypadki testowe na podstawie podanych

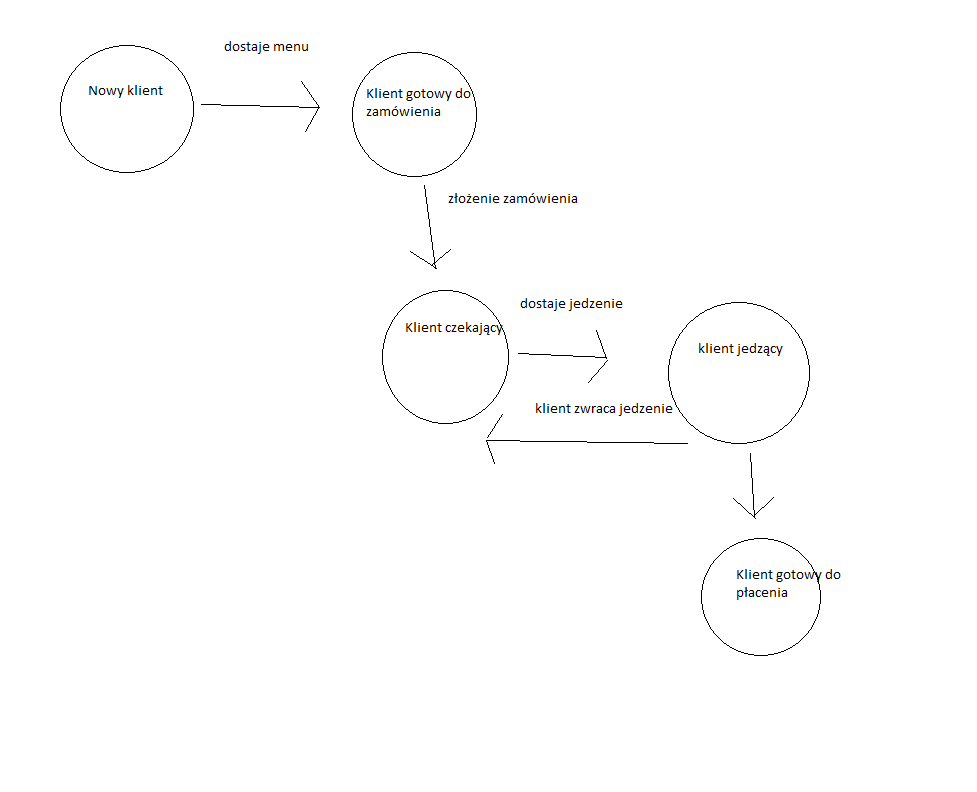
wymagań metodą podziału na klasy równoważności. Kandydat potrafi zaprojektować przypadki testowe na podstawie podanych wymagań metodą analizy wartości brzegowych. Kandydat potrafi zaprojektować przypadki testowe na podstawie podanych wymagań metodą testowania w oparciu o tablicę decyzyjną. Kandydat potrafi zaprojektować przypadki testowe na podstawie podanych

wymagań metodą testowania przejść pomiędzy stanami. Kandydat potrafi wyjaśnić, w jaki sposób można wyprowadzać przypadki testowe z przypadku użycia.

1. Podział na klasy równoważności (w teście – 2-3 pytania o to!)
   1. Technika w której dzielimy nasz problem na takie same części logiczne, które różnią się danymi
   2. Można ją wyznaczyć dla danych poprawnych i niepoprawnych
   3. Np. system dla liczb od 0-2 wydaje krzyk, dla 2-5 pisk a od 5 jęk, mamy więc 3 klasy równoważności (system wydaje „jakiś” dźwięk) 0-2,2-5,5-…
   4. Powyższy przykład można też wyznaczyć dla danych niepoprawnych (np. liczby mniejsze niż 0)
   5. Każda testowana wartość (w naszym powyższym przypadku liczba) może należeć do TYLKO jednej klasy równoważności
   6. Omawialiśmy to mówiąc o testowaniu danymi, podajemy zawsze dane najbardziej newralgiczne dla systemu (np. ujemne, na granicach przedziałów itd.)
   7. Aby uzyskać 100% pokrycie przy tej technice należy napisać test dla każdej(!) zidentyfikowanej klasy równoważności
2. Analiza warunków brzegowych
   1. Jeśli system ma napisane że przyjmuje wartości od 50 do 200 włącznie(!), to są to jego wartośći brzegowe. Od razu możemy znaleźć 3 klasy równoważności (0-49, 50-200, 201-xxx). 2 zewnętrzne klasy równoważności są klasami niepoprawnymi (te dane nie powinny być akceptowane w systemie), środkowa – jest poprawną klasą równoważności)
   2. Dla naszego przykładu (50-200 włącznie) musimy otestować okolice wartości brzegowych
      1. Dla 2 poziomowej analizy (49,50,200,201)
      2. Dla 3 poziomowej analizy : (49,50,51,199,200,201)
3. Testowanie oparte o tablicę decyzyjną (1-2 pytania w teście)
   1. Tworzymy specjalną tablkę według opisu ISTQB
   2. warunki
      1. T – TAK (lub True)
      2. N – Nie (Lub czasami F – False)
      3. „-„ – wartość nie ma znaczenia
   3. Wnioski lub akcje wynikające z tablicy
      1. X – akcja powinna być wykonana
      2. Puste pole – akcja nie powinna być wykonana

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Warunki Tempomatu | T1 | T2 | T3 | T4 |
| Prędkość powyżej 50kmh | TAK | TAK | NIE | NIE |
| Drugi samochód bliżej niż 15metrów | TAK | NIE | TAK | NIE |
|  |  |  |  |  |
| Akcja – hamujemy? | TAK | NIE | TAK | NIE |
|  |  |  |  |  |

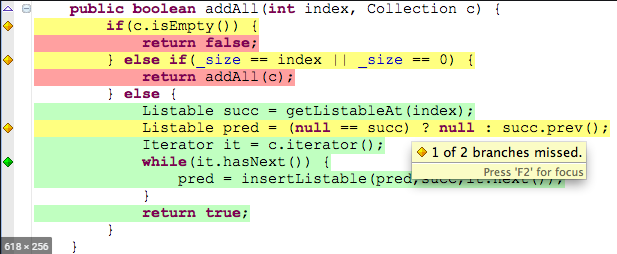
* 1. Powyższa tabelka opisuje nam 4 przypadki jakie mogą zaistnieć w testowanym urządzeniu – tempomacie
  2. Pisząc test – chcemy być pewni, że przetestowaliśmy tempomat dobrze, tworzymy więc tabelkę która obrazuje jakie powinien podjąć decyzje
  3. Tabelkę analizujemy kolumnami T1 – TAK,TAK -> TAK, T2 – TAK,NIE -> NIE, T3 – NIE,TAK -> TAK, T4 – NIE,NIE -> NIE
  4. Powinniśmy napisać przynajmniej 1 test dla każdego przejścia – w powyższym przypadku 4 testy

1. Testowanie przejść pomiędzy stanami
   1. Stany i przejścia to sposób do opisywania jakiegoś systemu
   2. Np. system zamówień w restauracji:
      1. 
      2. Graf zawiera STANY (okręgi z tekstem) i PRZEJŚCIA STANÓW (strzałki)
      3. Możemy dzięki temu ocenić w jakim stanie jest dany klient (ten czeka, tamten płaci, ten ogląda menu) i mamy jasną informację czego oczekuje klient, żeby przejść do następnego stanu
      4. Tak można modelować zachowanie większości systemów
      5. Tak zamodelowany system można przetestować, symulując stany klienta i przejścia między nimi (czy klient nowy dostał menu? Jeśli tak – to czy jest w stanie gotowy do zamówienia?)
      6. Możemy też testować czy nie ma miejsca nielegalne przejście między stanami (klient nowy nie może przejść od razu do klienta jedzącego)
      7. Możemy testować taki graf również tabelką przejść między stanami (Jest takie pytanie w teście!)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Numer TC | 1 | 2 | 3 | 4 | 5 |
| Stan początkowy | Nowy klient | Klient gotowy do zamówienia | Klient oczekujący | Klient jedzący | Klient jedzący |
| Przejście | Dostaje menu | zamawia | Dostaje jedzenie | Zwraca jedzenie | Zjadł |
| Spodziewany stan końcowy | Klient gotowy do zamówienia | Klient oczekujący | Klient jedzący😊 | Klient oczekujący | Gotowy do płacenia |

* + 1. Teraz dla każdego przejścia ze stanu do stanu możemy napisać lub przeprowadzić test
    2. Dzięki temu, że listę testów mamy zamkniętą – możemy przeliczyć pokrycie testami
    3. Często używany gdy testujem aplikację która ma menu rozwijalne – zaznaczenie menuA->podmenu B->opcja C sugeruje przejścia więc ta technika się do tego idealnie nada
    4. Ważne – 1 test powinien symulować 1 pełne przejście między stanami (jeśli przejściem wrócimy do stanu, z którego zaczynaliśmy – powinniśmy skończyć test)

1. Testowanie oparte na przypadkach użycia (patrz projekt-testowy/UML/PrzypadkiUzycia.png)
   1. Wygląda podobnie do przejść pomiędzy stanami, ale tutaj strzałki nie mówią o przejściach tylko odpowiedzialnościach (który użytkownik – Aktor – może korzystać z których funkcjonalności systemu)
   2. Każdy przypadek użycia (Use Case) określa konkretne działanie/zachowanie tej części systemu
   3. Testy można opisać w kategorii interakcji użytkownika z daną częścią systemu
   4. Najczęściej UseCase’y testuje się tworząc do nich odpowiadające im TC (TestCase’y). Jeden przypadek użycia może mieć kilka TestCase’ów
   5. Pokrycie – ilość przechodzących TestCase’ów
   6. Przykładowe TC dla naszego obrazka.
      1. Relacja kierowca – start silnika
         1. TC1 – kierowca włącza silnik
         2. TC2 – kierowca wyłącza silnik
         3. TC3 – kierowca wyciąga kluczyk w trakcie jazdny (lub wciska przycisk stop)
      2. Relacja Mechanik – silnik
         1. TC1 – mechanik wlewa olej
         2. TC2 – mechanik odkręca śrubkę
         3. TC3 – mechanik robi cośtam co ma spowodować cośtam innego
         4. Itd.

1. Białoskrzynkowe techniki (Tu pytania najczęściej odnośnie pokrycia kodu i decyzji)
   1. Pokrycie instukcji kodu
      1. 
      2. Mamy jakiś kod, ma on każdą linijkę pokolorowaną (raz na czerwono, raz na zielono)
      3. To znaczy – pokrycie instrukcji (linijek) kodu = 100% (każda linijka jest uruchamiana przez przynajmniej 1 test)
      4. Gdyby była tu jakaś linijka bez żadnego koloru tła – żaden test jej nie wywołał, nie jest przetestowana – pokrycie linijek spada w dół ( już nie jest 100%)
   2. Pokrycie decyzji kodu
      1. Omawialiśmy algorytmy na 2 spotkaniu (patrz prezentacja) i ich zapis blokowy
      2. Każde „rozgałęzienie” się algorytmu to miejsca, w których kod podejmuje decyzję.
      3. W obrazku z powyższego przykładu – widzimy napis „1 of 2 branches missed”
      4. Linijka żółta
      5. Oznacza to, że nasze testy nie podjęły wszystkich możliwych decyzji jakie są dostępne w kodzie
      6. Pokrycie decyzji – 50%
      7. Testowanie służy do sprawdzenia decyzji zawartych w kodzie oraz kodu wykonywanego na podstawie wyników decyzji
      8. Pokrycie mierzy się jako iloraz liczby wyników decyzji wykonanych przez testy przez łączną liczbę możliwych wyników decyzji (w procentach)
         1. Np. poniżej mamy 3 instrukcje warunkowe, ale test wykonuje tylko 3 z 4 warunków (opis dokładny w nagraniu)
         2. Czyli pokrycie to : 3 pokryte decyzje / 4 decyzje łącznie = 75%
   3. Wnioski
      1. 100% pokrycia decyzji gwarantuje 100% pokrycia kodu (w końcu – każdy kod się musiał wykonać, nie ma innej opcji)
      2. 100% pokrycia kodu **nie gwarantuje** 100% pokrycia decyzji – jak na powyższym obrazku
      3. Jeśli w kodzie występują decyzję – każda z nich zwiększa wymaganą ilość testów wykładniczo
         1. 1 warunek – potrzeba 2 testów
         2. 2 warunki – potrzeba 4 testów
         3. 3 warunki – potrzeba 8 testów
   4. Korzyści z testowania instrukcji i decyzji
      1. Uzyskanie 100% pokrycia kodu gwarantuje, że każda linijka została wywołana i jest pod naszą kontrolą (jej zachowanie jest przewidywalne)
      2. Idąc tą logiką – 50% pokrycia oznacza, że nie wiemy jak zachowa się (lub już teraz zachowuje) połowa naszego systemu (może działa – może nie działa – nie wiadomo nikt tego nie sprawdza)
2. Techniki oparte na doświadczeniu
   1. Wymaga doświadczonych testerów z już wykształconą intuicją
   2. Zgadywanie błędów – pewne błędy występują w podobnych systemach, są charakterystyczne dla danego systemu – i ich istnienia można się domyślić i je trafnie odgadnąć
   3. Można tworzyć listę potencjalnych błędów
   4. Testowanie eksploracyjne (jest pytanie do tego)
      1. Wymaga doświadczonych testerów!!!
      2. Najbardziej przydatne gdy system nie jest w pełni opisany, brakuje specyfikacji konkretnych fragmentów, trzeba poeksplorować i odkryć jak dana część produktu działa
      3. Można tworzyć takie testy zarówno biało- jak i czarnoskrzynkowe
   5. Testowanie oparte o listę kontrolną (jest o to pytanie)
      1. Testerzy piszą testy aby spełniały założenia z listy kontrolnej
      2. Lista kontrolna może zawierać podstawę do testów funkcjonalnych jak i niefunkcjonalnych (system robi ABC – funkcjonalne, system się nie przegrzewa – niefunkcjonalne)
      3. Zagadnienia na listach kontrolnych są dość ogólne i opisane na wysokim poziomie (mało szczegółów, bardziej opisane są zachowania)
      4. Listy kontrolne można tworzyć na podstawie
         1. własnego doświadczenia
         2. danych z przeglądów i informacji o potencjalnych defektach
         3. znajomości oczekiwań użytkowników
         4. ogólnej wiedzy o częstych błędach w oprogramowaniu