Testowanie oprogramowania

Dr inż. Andrzej Grosser

Literatura

- Andy Hunt, Dave Thomas „JUnit. Pragmatyczne testy jednostkowe w Javie” Helion 2006
- Srinivasan Desikan; Gopalaswamy Ramesh „Software Testing: Principles and Practices” Pearson Education India 2006
- Bogdan Wiszniewski, Bogdan Bereza-Jarociński „Teoria i praktyka testowania programów” PWN 2009

Aksjomaty testowania

- Programu nie da się przetestować całkowicie
- Testowanie jest ryzykowne
- Test nie udowodni braku błędów
- Im więcej błędów znaleziono, tym więcej błędów pozostało do znalezienia
- Nie wszystkie znalezione błędy zostaną naprawione
- Trudno powiedzieć, kiedy błąd jest błędem
- Specyfikacje produktów nigdy nie są gotowe

Definicja błędu

- Oprogramowanie nie robi czegoś co zostało wymienione w jego specyfikacji
- Oprogramowanie wykonuje coś czego według specyfikacji nie powinno robić
- Oprogramowanie robi coś o czym specyfikacji nie wspomina

Definicja błędu

- Oprogramowanie nie wykonuje czegoś o czym specyfikacja nie wspomina mimo że powinno to być wymienione jako istotną częścią systemu.
- Oprogramowanie jest trudne do zrozumienia, powolne lub skomplikowane.

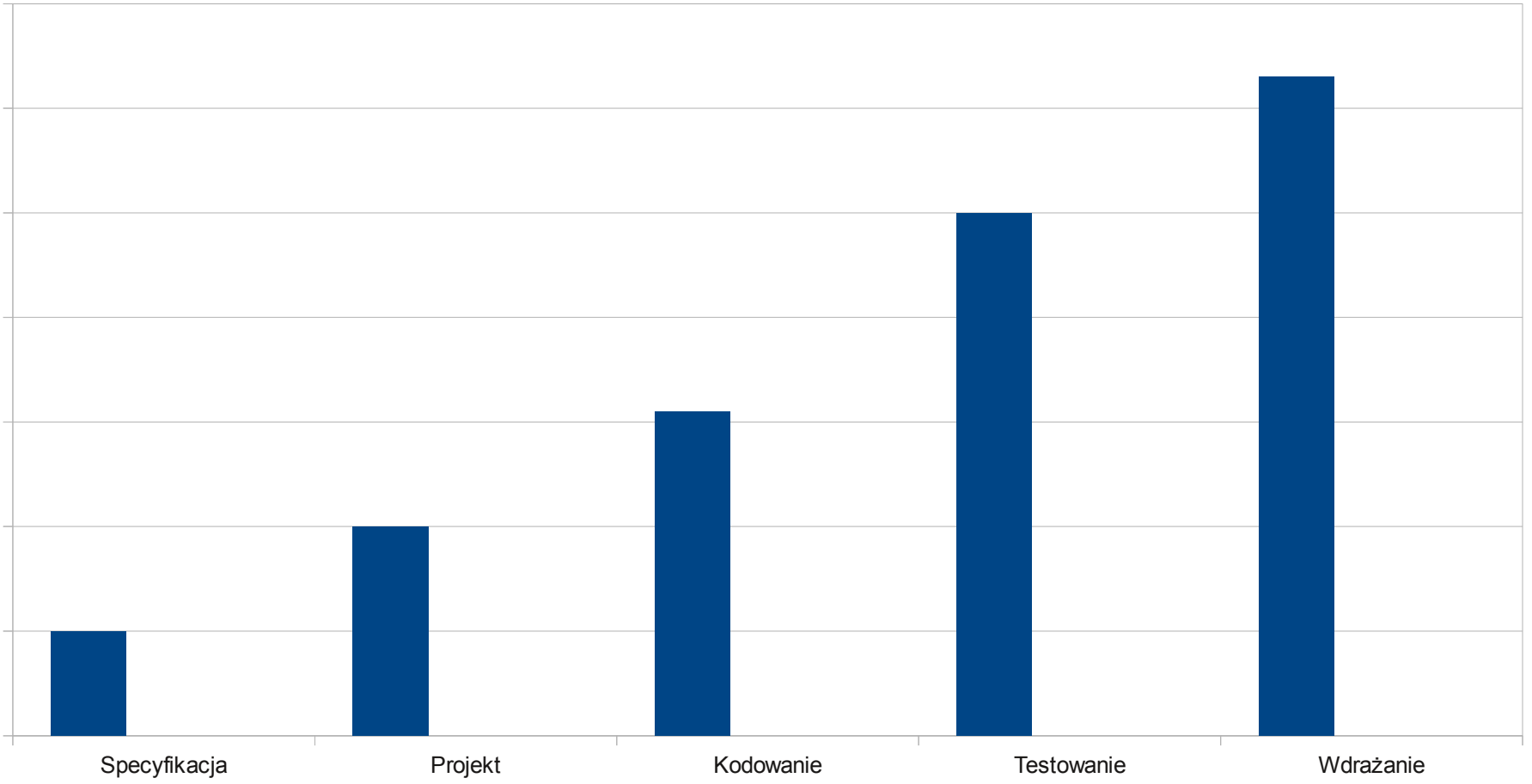
Poziomy testowania oprogramowania

- Testy jednostkowe (modułowe)
- Testy integracyjne
- Testy funkcjonalne
- Testy systemowe
- Testy akceptacyjne
- Testy w fazie utrzymywania systemu

Testowanie w cyklu życia oprogramowania

- W modelu kaskadowym testowanie jest wydzielone jako osobna faza (ostatnia)
- Metodyki zwinne uwzględniają testowanie jako jedna z najważniejszych faz cyklu życia oprogramowania

Koszty błędów



Modele programu

- Modelowanie przez opis przepływów sterowania i danych
- Model składniowy
- Model decyzyjny
- Model aksjomatyczny

Testowanie metodą białej skrzynki

- Wgląd do kodu źródłowego
- Systematyczny sprawdzanie elementów tego kodu.
- Można przeprowadzić w sposób statyczny (statyczna analiza kodu znajdująca źródła potencjalnych problemów w programie, zwana również analizą strukturalną) lub dynamiczny (z wykonaniem programu).

Formalna analiza kodu

- Proces sterujący testowaniem metodą białej skrzynki
- Elementy wyróżniające - identyfikacja problemów (znalezienie błędów i brakujących elementów), postępowanie według narzuconych z góry zasad (np. liczba wierszy kodu podlegającego przeglądowi, ilość czasu na przegląd), przygotowanie do przeglądu (np. podział uczestników na role, jakie będą pełnić w czasie przeglądu) oraz tworzenie raportów (podsumowanie wyników przeglądu)

Standardy i reguły kodowania

- Standardy narzucają ustalone, sztywne zestawy wymagań (zazwyczaj nie dopuszczają wyjątków od takich zasad), określają jakich instrukcji należy używać a jakich nie powinno się używać.
- Reguły kodowania to sugerowane, praktyczne wskazówki, rekomendacje i zalecenia.

Standardy i reguły kodowania

- Powody narzucania standardów i reguł kodowania:
  - podnosi się niezawodność oprogramowania, - standardy nie dopuszczają kiepsko udokumentowanych trików, narzucają również sprawdzone techniki programistyczne
  - łatwiej czytać, zrozumieć i modyfikować oprogramowanie.
  - ułatwiona przenośność pomiędzy różnymi platformami systemowymi.

Analiza pokrycia kodu

- Analiza dynamiczna w testowaniu metodą białej skrzynki
- Pozwala ona na przetestowanie stanu programu i przepływów sterowania pomiędzy tymi stanami
- Sprawdza się wejście i wyjście każdej jednostki programu, dąży się do wykonania każdego wiersza programu i każdej możliwej ścieżki programu.

Analiza pokrycia kodu

- Umożliwia znalezienie wielu błędów - podczas projektowania i implementowania funkcji najwięcej błędów powstaje w fragmentach, które są najrzadziej wykonywane.
- Przypadki szczególne często nie są wychwytywane przez projektantów.
- Fragmenty programu, które miały być w zamierzeniach wykonywane niezwykle rzadko mogą być w rzeczywistości odwiedzane bardzo często.

Analiza pokrycia kodu

- Programy śledzące - umożliwiają wgląd w jaki sposób są wykonywane kolejne wiersze kodu podczas przetwarzania danych testowych.
- Analizatory pokrycia kodu, pozwalają ponadto na sporządzenie szczegółowych statystyk dotyczących wykonania testowanego programu. Umożliwia to uzyskanie informacji, które części kodu nie zostały pokryte przez zastosowane testy.

Analiza pokrycia kodu

- Pokrycie kodu wykonywane jest przez:
  - analizę pokrycia instrukcji (zwaną również pokryciem wiersza kodu) - wykonanie przynajmniej jednokrotne każdej instrukcji w programie
  - pokrycie rozgałęzień programu (testowanie ścieżek) - wykonanie jak największej liczby możliwych ścieżek programu, testowanie rozgałęzień
  - analizę pokrycia warunków logicznych - uwzględnienie złożonych warunków logicznych instrukcji warunkowej.

Testowanie mutacyjne

- Testowanie polega na generowaniu składniowo poprawnych mutantów testowanego programu.
- Mutant w tym przypadku to program wolny od błędów składniowych, różniący się od oryginału drobnymi szczegółami różnych instrukcji.
- Zmienione programy (mutanty) są generowane automatycznie. Ten sposób testowania jest nazywany (od strukturalnego testowania układów elektronicznych) zasiewem błędów.

Testowanie mutacyjne

- W pierwszym kroku generowane są kolejne mutanty testowanego programu.
- Następnie mutanty wraz z oryginalnym programem są równocześnie wykonywane dla tych samych danych wejściowych zadawanych przez testera lub losowo.
- Celem usunięcie z puli jak największej liczby mutantów. Usuwane są takie mutanty, dla których można zaobserwować różnicę w działaniu lub inny wynik niż w przypadku programu testowane

Testowanie metodą czarnej skrzynki

- Koncentruje się na wymaganiach funkcjonalnych stawianych tworzonemu oprogramowaniu.
- Pozwala na sprawdzenie zgodności programu z wymaganiami użytkownika.
- Stosuje się go najczęściej pod koniec testowania systemu.
- Ma na celu wykrycie pominiętych lub niepoprawnie zaimplementowanych funkcjonalności ze specyfikacji użytkownika.

Testowanie danych – warunki graniczne

- Zakłada się, że realizowany przez program algorytm i jego implementacja są poprawne, zaś pojawiające się błędy wynikają z ograniczeń związanych z platformą sprzętową, użytym językiem programowania.
- Celem testowania jest tutaj zaprojektowanie przypadków testowych sprawdzających wartości graniczne i unikatowe związane z architekturą komputera na której będzie uruchamiany testowany program, językiem implementacji i realizowanym algorytmem.

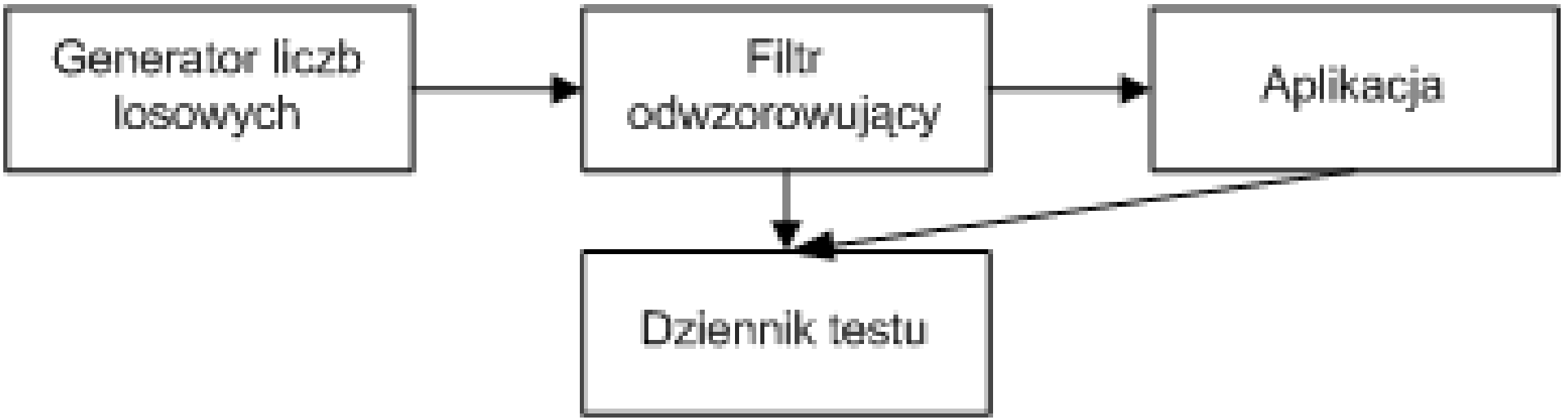
Wartości specjalne i transcendentne

- Problem doboru takich wartości:
  - Dla operacji arytmetycznych można dobrać maksymalne i minimalne wartości operandów, elementy jednostkowe i zerowe.
  - Dla tablic dobór takich wartości mógłby dotyczyć pustej tablicy, wykraczających poza zakres tablicy indeksów i poprawnych wartości indeksów.
  - Dla bardziej skomplikowanych struktur danych tak jak listy i drzewa mogą to być puste wskaźniki (odniesienia), jednoelementowe, zawierające wiele elementów i listy z tak zwanym zerwanym wskaźnikiem.

Metoda klas równoważności

- Metoda ta pozwala na ograniczenie, często ogromnego zbioru zadań testowych do mniejszej, ściśle określonej klasy danych testowych.
- Klasą równoważności jest tutaj określany zbiór zadań testowych, które sprawdzają sprawdzają aplikację (fragment aplikacji) pod kątem tego samego błędu.
- Jedna klasa równoważności zawiera zwykle wiele zestawów testowych, zgodnych (lub niezgodnych) z pewnymi warunkami wejściowymi.

Metoda Monte Carlo i metody genetyczne



Metoda Monte Carlo i metody genetyczne

- Algorytm genetyczny jest wykonywany cyklicznie.
- W pierwszym cyklu populacja jest dobierana losowo.
- W każdym następnym cyklu wykonywane są operacje krzyżowania (wymiana części pomiędzy wektorami bitowymi), mutacji (losowy bit w wektorze bitowym ulega zmianie) i reprodukcji na populacji danych testowych.

Metody Monte Carlo i metody genetyczne

- Elementy biorące udział w tworzeniu nowej populacji są wybierane na podstawie wartości tak zwanej funkcji dopasowania, która opisuje własności populacji.
- Sposób pozwala na zmniejszenie liczby danych testowych, wymaga jednak dodatkowego kosztu związanego z przekształceniem danych wejściowych do wektorów bitowych (tzw. chromosomów).

Testy jednostkowe

- Test jednostkowy – fragment kodu sprawdzający działanie pewnego, niewielkiego, dokładnie określonego obszaru funkcjonalności testowanego kodu
- Zadaniem testów jednostkowych jest udowodnienie, że kod działa zgodnie z założeniami programisty

Testy jednostkowe

- Właściwości poprawnych testów jednostkowych
  - Automatyzacja
  - Kompletność
  - Powtarzalność
  - Niezależność
  - Profesjonalizm

Automatyzacja

- Testy jednostkowe muszą być wykonywane w automatyczny sposób
- Automatyzacja dotyczy tutaj uruchamiania testów i sprawdzania ich wyników
- Testy są wykonywane wielokrotnie, dlatego ich uruchamianie musi być proste

Kompletność

- Testy muszą testować wszystko co może zawieść
- Dwa podejścia:
  - testowanie każdego wiersza kodu i każdego rozgałęzienia sterowania
  - Testowanie fragmentów najbardziej narażonych na błędy
- Narzędzia umożliwiające sprawdzenie jaka część testowanego kodu jest w rzeczywistości wykonywana

Powtarzalność

- Testy powinny być niezależne nie tylko od siebie, ale również od środowiska – wielokrotne wykonywanie testów, nawet w różnej kolejności, powinno dać te same wyniki
- Obiekty imitacji pozwalają odseparować testowane metody od zmian zachodzących w środowisku

Niezależność

- Testy powinny się koncentrować na testowanej w danym momencie metodzie oraz być niezależne od środowiska i innych testów
- Testy muszą testować jeden aspekt działającego kodu – sprawdzać działanie pojedynczej metody lub niewielkiego zestawu takich metod, które współpracując ze sobą dostarczają jakąś funkcjonalność
- Przeprowadzenie testu nie może zależeć od wyniku innego testu

Profesjonalizm

- Kod testujący powinien spełniać te same wymagania co kod produkcyjny
- Muszą być przestrzegane zasady poprawnego projektowania – np. hermetyzacja, reguła DRY.
- Brak testów dla fragmentów, które nie są istotne – np. proste metody dostępowe
- Liczba wierszy kodu testującego porównywalna z liczbą wierszy kodu produkcyjnego

Obszary testowania

- Czy wyniki są poprawne?
- Czy warunki brzegowe są określone poprawnie?
- Czy można sprawdzić relacje zachodzące w odwrotnym kierunku?
- Czy można uzyskać wyniki korzystając z alternatywnego sposobu?
- Czy można wymusić warunki zachodzenia błędu?
- Czy efektywność jest poprawna?

Poprawność wyników

- Wyniki działania kodu znajdują się często w specyfikacji wymagań
- W przypadku braku dokładnej specyfikacji - założenie własnych wymagań odnośnie wyników metod
- Weryfikacja założeń w współpracy z użytkownikami
- Dla zestawu testów wymagających dużej liczby danych wejściowych używanie plików

Warunki brzegowe

- Typowe warunki brzegowe:
  - Wprowadzenie błędnych lub niespójnych danych wejściowych
    - Nieprawidłowy format danych wejściowych
    - Nieodpowiednie wartości
    - Dane przekraczające znacznie oczekiwania
  - Pojawienie się duplikatów na listach
  - Wystąpienie list nieuporządkowanych
  - Zakłócenie typowego porządku zdarzeń

Warunki brzegowe

- Poszukiwanie warunków brzegowych:
  - Zgodność (z oczekiwanym formatem)
  - Uporządkowanie (poprawne uporządkowanie zbioru wartości)
  - Zakres (poprawny zakres danych wejściowych)
  - Odwołanie (do zewnętrznych obiektów znajdujących się poza kontrolą kodu)
  - Istnienie (wartość istnieje)
  - Liczność (dokładnie tyle wartości ile jest oczekiwanych)
  - Czas – zdarzenia zachodzą w oczekiwanej kolejności

Odwrócenie relacji

- Działanie niektórych funkcji można przetestować stosując logikę działania w odwrotnym kierunku (pierwiastkowanie – podnoszenie do kwadratu)
- Zalecana ostrożność – implementacje obu funkcji mogą zawierać podobne błędy

Kontrola wyników na wiele sposobów

- Wyniki działania testowanych metod można sprawdzać na wiele sposobów – zazwyczaj istnieje więcej niż jeden sposób wyznaczania wartości
- Implementacja kodu produkcyjnego z wykorzystaniem najefektywniejszego algorytmu, inne algorytmy można użyć do sprawdzenia czy wersja produkcyjna daje te same wyniki.

Wymuszanie warunków powstawania błędów

- Rzeczywisty system narażony jest na różnego rodzaju zewnętrzne błędy – np. brak miejsca na dysku, awarie infrastruktury zewnętrznej – sieci, problemy z rozdzielczością ekranu, przeciążeniem systemu
- Przekazywanie niepoprawnych parametrów, symulacje zewnętrznych błędów z wykorzystaniem obiektów imitacji

Charakterystyka efektywnościowa

- Charakterystyka efektywnościowa – sposób zmiany efektywności kodu w odpowiedzi na rosnącą liczbę danych, rosnącą komplikację problemu.
- Zmiany efektywności programu w zależności od wersji kodu
- Testy sprawdzające czy krzywa efektywności jest stabilna w różnych wersjach programu

Obiekty imitacji

- Działanie kodu może zależeć od wielu czynników niezależnych od kodu oraz innych części systemu
- Testy będą musiały inicjować wiele komponentów systemu – trudne w sytuacji ciągle zmieniącego się systemu
- Te problemy można rozwiązać stosując namiastki rzeczywistych obiektów – obiekty imitacji

Obiekt imitacji

Obiekty imitacji są używane, gdy:

- Obiekt rzeczywisty zachowuje się niedeterministycznie (wynik jego działania jest nieprzewidywalny
- Obiekt rzeczywisty jest trudny do skonfigurowania
- Trudno wywołać istotne z punktu widzenia testowania zachowanie obiektu rzeczywistego

Obiekt imitacji

- Działanie obiektu rzeczywistego jest powolne
- Obiekt rzeczywisty ma graficzny interfejs użytkownika lub jest częścią graficznego interfejsu użytkownika
- Test musi uzyskać informacje o sposobie użycia rzeczywistego obiektu
- Obiekt rzeczywisty jeszcze nie istnieje (częsty problem podczas łączenia kodu z innymi podsystamami)

Obiekty imitacji

- Etapy testowania przy użyciu obiektów imitacji:
  - Stworzenie interfejsu do opisu obiektu
  - Implementacja interfejsu dla kodu produkcyjnego
  - Implementacja interfejsu przez obiekt imitacji dla potrzeb testowania

Pułapki testowania

- Ignorowanie testów jednostkowych, gdy kod działa
- „Testy ognia”
- Program działa na komputerze programisty
- Problemy arytmetyki zmiennoprzecinkowej
- Testy zajmuja zbyt wiele czasu
- Testy ciągle zawodzą
- Testy zawodzą na niektórych maszynach

Kod działa poprawnie – nie przechodzi testów

- Kod działa poprawnie, ale nie przechodzi testów jednostkowych
- Można spróbować zignorować test, ale:
  - Kod może w każdej chwili przestać działać
  - Zmarnowany wysiłek poświęcony na pisanie testów
- Najlepszą strategią – przyjęcie założenia, że kod zawiera błędy

Testy ognia

- Testy ognia – założenie, że metoda nie zawiera błędów, gdy wykona swój kod bez błędów
- Takie testy zawierają zwykle jedną asercję na końcu kodu testującego assertTrue(true) – sprawdza to czy sterowanie doszło do końca kodu
- Za mało – nie sprawdzono żadnych danych ani zachowania kodu
- Testy powinny polegać na sprawdzaniu wyników

Arytmetyka zmiennoprzecinkowa

- Komputer umożliwia jedynie skończoną dokładność reprezentacji liczb zmiennoprzecinkowych i operacji na nich
- Problemy z reprezentacją liczb dziesiętnych w systemie dwójkowym
- Konieczna pewna dokładność z jaką porównywane są wyniki

Testy zajmują zbyt wiele czasu

- Testy jednostkowe powinny być wykonywane szybko – są robione wiele razy
- Wyodrębnić testy wykonywane zbyt wolno, obniżające efektywność testowania
- Testy wykonywane wolno powinny być wykonywane rzadziej, np. raz dziennie

Testy ciągle zawodzą

- Różne testy kończą się stale niepomyślnym wynikiem
- Niewielkie zmiany w kodzie powodują, że część testów przestaje przechodzić pomyślnie testowanie
- Najczęściej oznaka zbyt dużego powiązania z zewnętrznymi danymi lub innymi częściami systemu

Testy zawodzą na niektórych maszynach

- Testy są wykonywane poprawnie jedynie na większości komputerów
- Najczęstszym problemem różne wersje systemu operacyjnego, bibliotek, wersji kompilatora, sterowników, konfiguracji, wydajności i architektury maszyny
- Założenie, że testy powinny być wykonywane poprawnie na wszystkich maszynach

U mnie działa

- Program działa poprawnie na komputerze programisty
- Błędy ze środowiskiem, w którym działa program:
  - Kod nie został wprowadzony do systemu kontroli wersji
  - Niespójne środowisko programowania
  - Prawdziwy błąd, który jest ujawniony w określonych warunkach
- Testy muszą być wykonywane z sukcesem na wszystkich maszynach

Środowiska testowania jednostkowego

- Java
  - JUnit
  - TestNG
- C#
  - framework Microsoft
  - NUnit

Biblioteki do budowy obiektów imitacji

- Java
  - Easy Mock
  - JMock
- C#
  - Moq
  - Rhino Mocks

JUnit

- Framework o otwartych źródłach przeznaczony do testowania jednostkowego kodów napisanych w języku Java
- Integracja z popularnymi środowiskami programistycznymi – Eclipse, NetBeans, IntelliJ
- Wersja 4 korzysta z mechanizmu adnotacji

Testowanie z JUnit

- Umożliwia odseparowanie testów co pozwala na uniknięcie efektów ubocznych
- Adnotacje @Before, @BeforeClass, @After i @AfterClass umożliwiają inicjowananie i odzyskiwanie zasobów
- Zbiór metod assert umożliwia ławe sprawdzanie wyników testowanie

Prosty test

```
import static org.junit.Assert.*;
import org.junit.Test;
public class AddTest {
@Test
public void TestAdd() {
double result = 10 + 20;
assertEquals(30, result, 0);
}
}
```

Kompilacja i uruchamianie z konsoli

- Kompilacja:
  - javac -cp /opt/junit4.8/junit-4.8.jar *.java
- Uruchamianie testu z konsoli
  - Java -cp .:/opt/junit4.8/junit-4.8.jar org.junit.runner.JUnitCore AddTest

Parametryzowanie testów

```
[…]
@RunWith(value=Parameterized.class
public class ParameterizedTest {
  private double expected, valueOne, valueTwo;
  @Parameters
  public static Collection<Integer[]>
getTestParameters() {
   return Array.asList(new Integer[][] {2,1,1},
{3,1,2},{3,2,1}});
}
```

Parametryzowanie testów

```
public ParameterizedTest(double exp, double vo,
double vT) {
 expected = exp; valueOne = vo; valueTwo = vT;
}
@Test
public void sumTest() {
  assertEquals(expected, valueOne + valueTwo,
0);
}
} //class
```

Testowanie rzucania wyjątków

- Framework JUnit umożliwia sprawdzanie czy dany wyjątek został wyrzucony w określonej sytuacji
- ```
  @Test(expected=RuntimeException.class)
  public void testExceptionMethod()
  {//...}
  ```
  ```

Testowanie czasu wykonania

- JUnit umożliwia sprawdzanie czasu wykonania danego fragmentu kodu.
- Parametr timeout pozwala na ustawienie czasu, którego przekroczenie spowoduje, że test nie zostanie zaliczony. Wprowadzona bariera czasowa jest zapisywana w milisekundach
- ```
  @Test(timeout = 120)
  public void testTimeMethod()
  {//...}
  ```
  ```

Ignorowanie testów

- Czasami istnieją powody, żeby zignorować wyniki niektórych testów – np. w sytuacji, gdy do końca nie wiadomo, jaki będzie trzeba ustalić limit czasowy.
- Junit pozwala na ignorowanie wyników niektórych testów za pomocą adnotacji @Ignore
- ```
  @Test
  @Ignore(value=”Zignorowany do ustalenia czasu”)
  public void testIgnore(){}
  ```
  ```

Testy jednostkowe w .NET

- ```
  public class ClassTest
  {
  //Wykonywane raz przed wszystkimi testami klasy
  [ClassInitialize()]
  public void methodClassInitialize(){}
  //Wykonywane raz po wszystkich testach klasy
  [ClassCleanup()]
  public void methodClassCleanup(){}
  ```
  ```

Testy jednostkowe w .NET

```
//Wykonywane przed każdym testem
[TestInitialize()]
public void methodTestInitialize(){}
//Wykonywane po każdym teście
[TestCleanup()]
public void methodTestInitialize(){}
//Metoda testująca
[TestMethod()]
public void testMethod(){}
}
```

Testy jednostkowe w .NET

- Asercje są zdefiniowane w przestrzeni nazw:

  Microsoft.VisualStudio.TestTools.UnitTesting
- Klasami asercji – Assert, StringAssert, CollectionAssert
- Przykładowe metody: Assert.AreEqual, Assert.AreSame, Assert.Fail, Assert.IsTrue, StringAssert.Matches, CollectionAssert.AreEquivalent
- Asercje mogą zwracać Pass, Fail i Inconclusive (Assert.Inconlusive)

Testy jednostkowe w .NET

- ExpectedExceptionAtribute – zaznacza, że dany wyjątek jest spodziewany w czasie wywołania metody testowej

```
[TestMethod()]

[ExpectedException=typeof(MyException)]
```

- TimeoutAttribute – określa okres czasowy dla wykonania testu jednostkowego

```
[TestMethod()]

[Timeout = 120]
```

EasyMock

- EasyMock jest biblioteką dostarczającą klas dla obiektów imitacji.
- Pozwala na korzystanie z atrap bez ich pisania
- Biblioteka generuje obiekty imitacji (atrapy) za pomocą mechanizmu proxy (dostępnego w Javie od wersji 1.3).
- Obiekt sterujący imitacją umożliwia włączenie trybu nagrywania i trybu odtwarzania.

EasyMock

- Utworzyć obiekt imitacji dla interfejsu, który ma być symulowany
- Zapisać spodziewane zachowanie dla metod interfejsu (mogą być tylko wybrane metody interfejsu)
- Przełączyć obiekt imitacji w tryb odtwarzania

EasyMock

```
import static org.easymock.EasyMock.*;
import org.junit.*;
public interface Interface1 {
public int method1(); public void method2();
}
class TestClass {
Interface1 mock;
    @Before
    public void setUp() {
//Krok 1
        mock = createMock(Interface1.class);}
```

EasyMock

```
@Test
public void test() {
    // Krok 2
   mock.method2();
   expect(mock.method1()).andReturn(2);
   // Krok 3
   replay(mock);
  //...
   verify(mock);
}
```

Moq

- Biblioteka wspierająca tworzenie obiektów imitacji
- Prosta w użyciu – nie ma trybów zapisu i odgrywania (record/replay)
- Korzysta z wyrażeń lambda (.NET 3.5)

Moq

- Tworzenie obiektu imitacji

```
var mock = new Mock<Interface>();
```

- Konfiguracja zaślepki

```
mock.Setup(foo =>
foo.method1("foo")).Returns(true);
```

- Odwołanie do obiektu imitacji

```
mock.Object.method1("foo");
```

- Weryfikacja obiektu imitacji

```
mock.VerifyAll();
```

Testowanie wydajności

- Niska wydajność lub brak dostępu do usługi może powodować znaczne straty finansowe (np. klienci przeniosą się do konkurencji)
- Wysokowydajne systemy (serwery, łącza o dużej przepustowości) są kosztowne
- Konieczna jest równowaga pomiędzy tymi dwoma czynnikami

Testowanie wydajności

- Testowanie wydajności:
  - jest skomplikowane i kosztowne związku z ilością zasobów jakich wymaga oraz czasu jakiego należy na nie poświęcić.
  - nie jest jednoznaczne – różne osoby mogą mieć inne oczekiwania co do wydajności
  - duża liczba defektów odsłoniętych podczas testowania może wymagać zmiany projektu systemu

Testowanie wydajności

- Wykonywane w celu zapewnienia, że aplikacja:
  - Przetwarza określoną liczbę transakcji w założonym przedziale czasu
  - Jest dostępna i ma możliwość uruchomienia w różnych warunkach obciążeniowych
  - Umożliwia wystarczająco szybką odpowiedź w różnych warunkach obciążeniowych
  - Umożliwia dostęp do zasobów w zależności od potrzeb aplikacji
  - Jest porównywalna pod względem parametrów wydajnościowych od konkurencyjnych aplikacji

Parametry wydajności

- Przepustowość
  - liczba transakcji/żądań obsługiwanych przez system w określonym przedziale czasu
  - mierzy ile transakcji może być przetwarzanych w określonym przedziale czasu przy zadanym obciążeniu (np. liczbie wykonywanych transakcji, liczbie korzystających z systemu klientów)
  - przepustowość optymalna to maksymalna liczba wykonywanych transakcji

Parametry wydajności

- Czas odpowiedzi
  - opóźnienie pomiędzy momentem wysłania żądania a pierwszą odpowiedzią serwera
- Opóźnienie
  - nie każdy dłuższy czas oczekiwania na odpowiedź jest spowodowany działaniem aplikacji
  - zwłoka spowodowana przez aplikację, system operacyjny, środowisko uruchomieniowe

Metodyka testowania wydajnościowego

- Zbieranie wymagań
- Zapis przypadków testowych
- Automatyzacja przypadków testowych
- Wykonanie przypadków testowych
- Analiza otrzymanych wyników
- Wykonanie benchmarków
- Dostrajanie wydajności
- Rekomendacja właściwej konfiguracji dla klienta (planowanie obciążenia)

Zbieranie wymagań

- Wymagane wyniki mogą zależeć od ustawień środowiska, mogą również nie być znane z góry
- Wymagania powinny być możliwe do przetestowania
- Wymagania muszą jasno określić jakie parametry mają być mierzone i ulepszane
- Wymagania testowe muszą być skojarzone z pożądanym stopniem ulepszenia wydajności aplikacji

Zapis przypadków testowych

- Definiowanie przypadku testowego:
  - Lista operacji lub transakcji przewidzianych do testowania
  - Opis sposobu wykonania tych operacji
  - Lista produktów oraz parametrów wpływających na wydajność
  - Szablon obciążenia
  - Spodziewany wynik
  - Porównanie wyników z poprzednimi wersjami/konkurencyjnymi aplikacjami

Automatyzacja przypadków testowych

- Testy wydajności są wielokrotnie powtarzane
- Testy wydajności muszą być zautomatyzowane,w najgorszym przypadku niemożliwe jest testowanie bez tego ułatwienia
- Testy wydajności muszą dawać dokładne informacje, ręczne przeliczanie może wprowadzać dodatkowy błąd pomiaru

Automatyzacja przypadków testowych

- Testowanie wydajności jest uzależnione od wielu czynników. Mogą one występować w różnych kombinacjach, które trudno wyprowadzić ręcznie.
- Analiza wyników wymaga dostarczenia wielu informacji pomocniczych – logów, sposobów wykorzystania zasobów w określonych przedziałach czasu, co jest trudne lub niemożliwe do uzyskania ręcznie

Wykonanie przypadków testowych

- Zapis czasu początku i końca testu
- Zapis plików zawierających informacje o produkcie i systemie operacyjnym, ważnych dla powtarzalności testów i przyszłego debugowania
- Zużycie zasobów (CPU, pamięć, obciążenie sieci)
- Zapis konfiguracji testowej zawierający wszystkie czynniki środowiskowe
- Zbieranie informacji o wymaganych parametrach

Analiza otrzymanych wyników

- Obliczenia statystyczne dla otrzymanych wyników – średnia i odchylenie standardowe
- Usunięcie szumu informacyjnego i ponowne obliczenie statystyk
- Zróżnicowanie wyników uzyskanych z cache od wyników bezpośrednio uzyskanych od aplikacji
- Odróżnienie wyników testowania uzyskanych w sytuacji, gdy wszystkie zasoby były dostępne od wyników, gdy były uruchomione dodatkowe usługi w tle

Analiza otrzymanych wyników

- Czy zachowana jest wydajność testowanego systemu, gdy testy są wykonywane wielokrotnie?
- Jaka wydajność może być spodziewana w zależności od różnych konfiguracji (np. dostępnych zasobów)?
- Jakie parametry wpływają na wydajność?
- Jaki jest efekt scenariuszy z udziałem kilku różnych operacji dla czynników wydajnościowych?

Analiza otrzymanych wyników

- Jaki jest wpływ technologii (np. zastosowanie cache) na testy wydajnościowe?
- Jaki jest optymalny czas odpowiedzi/przepustowość dla różnego zbioru czynników wpływających na system – obciążenie, liczba zasobów i innych parametrów?
- Jakie obciążenie jest jeszcze możliwe do zaakceptowania i w jakich okolicznościach dochodzi do załamania wydajności?

Analiza otrzymanych wyników

- Czy wymagania wydajnościowe są spełnione i jak wygląda wydajność w porównaniu z poprzednimi wersjami lub spodziewanymi wynikami?
- Gdy nie ma jeszcze dostępu do wysokowydajnych konfiguracji, można na podstawie dostępnych wyników przewidzieć rezultaty na takich maszynach

Dostrajanie wydajności

- Dostrajanie wydajności:
  - forkowanie procesów w celu umożliwienia równoległych transakcji,
  - wykonywanie operacji w tle
  - powiększenie cache i pamięci operacyjnej
  - dostarczenie wyższego priorytetu dla często używanych operacji,
  - zmiana sekwencji operacji

Wykonywanie benchmarków

- Identyfikacja transakcji/scenariuszy i konfiguracji testowych
- Porównanie wydajności różnych produktów
- Dostrajanie parametrów porównywanych produktów w celu uzyskania najlepszej wydajności
- Publikacja wyników testów

Planowanie obciążenia

- Wyszukiwanie jakie zasoby i konfiguracja jest konieczna w celu osiągnięcia najlepszych rezultatów
- Ustalenie jaką wydajność osiągnie klient przy aktualnie dostępnych zasobach zarówno sprzętowych jak i softwerowych.

Narzędzia do testowania wydajnościowego

- Narzędzia do testowania funkcjonalnego – zapis i odtwarzanie operacji w celu uzyskania parametrów wydajnościowych
- Narzędzia do testowania obciążeniowego – symulacja warunków obciążeniowych bez potrzeby wprowadzania wielu użytkowników lub maszyn
- Narzędzia do mierzenia zużycia zasobów

Testowanie regresyjne

- Testowanie regresyjne jest wykonywane w celu upewnienia się, że wprowadzone nowe elementy lub naprawa defektów nie wpłynęła niekorzystnie na zbudowane wcześniej elementy systemu

Typy testów regresyjnych

- Regularne testy regresyjne – wykonywane pomiędzy cyklami testowania w celu zapewnienia, że po wprowadzonych poprawkach elementy aplikacji testowane wcześniej nadal działają poprawnie
- Końcowe testy regresyjne – wykonywane w celu walidacji ostatniej wersji aplikacji przed jej wypuszczeniem na rynek. Są uruchamiane na pewien okres czasu, ponieważ niektóre defekty ujawniają się dopiero po pewnej chwili (np. wycieki pamięci)

Wybór momentu testowania regresyjnego

- Znacząca liczba wstępnych testów została już przeprowadzona
- Została wprowadzona duża liczba poprawek
- Poprawki mogły wprowadzić efekty uboczne

Testowanie regresyjne

Metodyka testowania regresyjnego

- Wykonanie początkowych testów smoke lub sanity
- Zrozumienie kryteriów wyboru przypadków testowych
- Klasyfikacja przypadków testowych pod względem różnych priorytetów

Metodyka testowania regresyjnego

- Metodyka selekcji przypadków testowych
- Ponowne ustawianie przypadków testowych do wykonania
- Podsumowanie cyklu regresyjnego

Smoke test

- Testy „dymne” (ang. smoke test) składają się z:
  - Identyfikacji podstawowej funkcjonalności jaką musi spełniać przygotowywany produkt
  - Przygotowanie przypadku testowego w celu zapewnienia, że podstawowa funkcjonalność działa i dodanie go do zestawu testów
  - Zapewnienie, że ten zestaw przypadków testowych będzie wykonywany po każdej budowie systemu
  - W przypadku niepowodzenia tego testowania poprawienie lub wycofanie zmian

Zrozumienie wyboru przypadków testowych

- Wymaga informacji na temat:
  - Wprowadzonych poprawek i zmian do bieżącej wersji aplikacji
  - Sposobów testowania wprowadzonych zmian
  - Wpływu jaki mogą mieć wprowadzone poprawki na resztę systemu
  - Sposobu testowania elementów, na które mogły mieć wpływ wprowadzone poprawki

Klasyfikacja przypadków testowych

- Podział ze względu na priorytet dla użytkownika
  - Priorytet 0 – Testy sprawdzają podstawową funkcjonalność. Najważniejsze z punktu widzenia użytkownika i twórców systemu
  - Priorytet 1 – Testy używają podstawowych i normalnych ustawień dla aplikacji. Ważne z punktu widzenia użytkownika i twórców systemu
  - Priorytet 2 – Testy dostarczają poglądowych wartości odnośnie projektu.

Metodyka wyboru przypadków testowych

- Gdy wpływ wprowadzonych poprawek na resztę kodu jest mały można wybrać tylko wybrane testy z zbioru przypadków testowych. Można wybrać testy o priorytecie 0, 1 lub 2.
- Gdy wpływ wprowadzonych poprawek jest średni należy wybrać wszystkie testy o priorytecie 0 i 1. Wybór testów o priorytecie 2 jest możliwy, lecz nie jest konieczny
- Gdy wpływ wprowadzonych poprawek jest duży należy wybrać wszystkie testy o priorytecie 0 i 1 oraz starannie wybrane testy o priorytecie 2.

Ustawianie przypadków testowych

- Jest wykonywane gdy:
  - Wystąpiły duże zmiany w produkcie
  - Nastąpiła zmiana w procedurze budowania wpływająca na system
  - Występują duże cykle wprowadzania kolejnych wersji, podczas których pewne przypadki testowe nie były wykonywane przez dłuższy czas
  - Produkt jest w finalnej wersji z kilkoma wybranymi przypadkami testowymi
  - Wystąpiła sytuacja, gdy spodziewane wyniki testów są zupełnie inne w porównaniu z poprzednim cyklem

Ustawianie przypadków testowych

- Przypadki testowe będące w relacji z wprowadzonymi poprawkami mogą być usunięte, gdy kolejne wersje nie powodują zgłaszania błędów
- Usunięto z aplikacji jakąś funkcjonalność, przypadki testowe powiązane z nią również powinny być usunięte
- Przypadki testowe kończą się za każdym razem wynik pozytywnym
- Przypadki testowe powiązane z kilkoma negatywnymi warunkami testowymi (nie powodującymi wykrywanie defektów) mogą być usunięte

Podsumowanie cyklu regresyjnego

- Przypadek testowy zakończył się sukcesem w poprzedniej wersji i niepowodzeniem przy obecnej – regresja zakończona niepowodzeniem
- Przypadek testowy zakończył się niepowodzeniem w poprzedniej wersji i przeszedł pomyślnie test w obecnej wersji – można założyć, że wprowadzone poprawki rozwiązały problem

Podsumowanie cyklu regresyjnego

- Przypadek testowy kończy się niepowodzeniem w obu wersjach (poprzedniej i obecnej) przy braku wprowadzonych poprawek dla tego przypadku testowego – można oznaczać, że test nie powinien być wliczany do wyników testowania. Może również oznaczać, że należy dany przypadek testowy usunąć.
- Przypadek testowy kończy się niepowodzeniem w poprzedniej wersji a w obecnej działa z dobrze udokumentowanym obejściem problemu, to regresję można uznać za zakończoną sukcesem

Porady dotyczące testowania regresyjnego

- Regresje mogą być używane ze wszystkimi wersjami (wprowadzającymi poprawki, nową funkcjonalność)
- Powiązanie problemu z przypadkiem testowym zwiększa jakość regresji
- Testy regresyjne powinny być wykonywane codziennie
- Lokalizacja i usunięcie problemu powinna ochronić produkt przed wpływem problemu i poprawki

Testowanie doraźne

- Wymagania zmieniają się wraz ze ewolucją oprogramowania
- Planowane test wymagają dostosowania do zmian w produkcie – żmudne i czasochłonne
- Planowane testy mogą pomijać istotne elementy, które pojawiły się w czasie tworzenia oprogramowania

Przyczyny testowania

- Brak dokładnie wyspecyfikowanych wymagań – testy wprowadzone wcześniej nie biorą pod uwagę doświadczeń zdobytych w procesie tworzenia oprogramowania
- Brak umiejętności niezbędnych do wykonania testów – testy napisane wcześniej nie biorą pod uwagę wzrostu umiejętności testerów
- Brak czasu na projektowanie testów – można pominąć istotne z punktu widzenia systemu perspektywy

Testowanie doraźne

- Analiza istniejących przypadków testowych
- Planowanie testów
- Wykonanie testów
- Generowanie raportów z testów
- Projektowanie przypadku testowego

Planowanie testów doraźnych

- Po wykonaniu pewnej liczby zaplanowanych testów – mogą być uzyskane nowe sposoby spojrzenia na produkt a także mogą być odsłonięte nowe defekty
- Przed zaplanowanymi testami – w celu zdobycia lepszego spojrzenia na wymagania i osiągnięcia z góry lepszej jakości produktu

Wady testowania doraźnego

- Trudności z zapewnieniem, że wiedza zdobyta w czasie testowania doraźnego będzie użyta w przyszłości
- Brak pewności, czy zostały pokryte przez testy doraźne najważniejsze aspekty systemu
- Trudności z śledzeniem kroków potrzebnych do powtórzenia testu
- Brak danych do analizy metrycznej

Metody testowania ad hoc

- Testowanie losowe
- Testowanie koleżeńskie
- Testowanie parami
- Testowanie badawcze
- Testowanie iteracyjne
- Testowanie zwinne i ekstremalne
- Zasiew defektów

Testowanie koleżeńskie

- Dwaj członkowie zespołu (zazwyczaj programista i tester) pracują nad identyfikacją i usunięciem problemu w oprogramowaniu
- Sprawdza się np.: standardy kodowania, definicje zmiennych, kontrole błędów, stopień udokumentowania kodu
- Używa się testów białej i czarnej skrzynki

Testowanie parami

- Wykonywane przez dwóch testerów na tej samej maszynie
- Celem jest wymiana pomysłów pomiędzy osobami, lepsze zrozumienie wymagań stawianych projektowi, zwiększenie umiejętności testowania oprogramowania
- Podział na role – tester i skryba – jedna osoba wykonuje testy, druga robi notatki

Testowanie badawcze

- Na podstawie poprzednich doświadczeń z technologią, podobnymi systemami wnioskuje się jakie błędy program może zawierać
- Technika użyteczna szczególnie w przypadku systemów nieprzetestowanych wcześniej, nieznanych albo niestabilnych
- Używana, gdy nie do końca wiadomo jakie testy należy jeszcze dołożyć do istniejącej bazy testów

Techniki testowania badawczego

- Zgadywanie – na podstawie wcześniejszych doświadczeń z podobnymi produktami tester zgaduje, która część programu zawiera najwięcej błędów
- Użycie diagramów architektonicznych i przypadków użycia – tester używa do testowania diagramów – architektoniczne przedstawiają jak wyglądają powiązania pomiędzy modułami, przypadki użycia natomiast dostarczają informacji o systemie z punktu widzenia użytkownika

Techniki testowania badawczego

- Studiowanie poprzednich defektów – pozwala na ustalenie jakie części systemu są najbardziej narażone na defekty
- Obsługa błędów – system po wykonaniu niepoprawnej operacji użytkownika może być niestabilny. Obsługa błędów powinna dostarczać czytelnych komunikatów diagnostycznych i/lub sposobów zaradzenia problemowi. Testuje się sytuacje, w których dochodzi do tego błędów.

Techniki testowania badawczego

- Dyskusja – szczegóły implementacyjne użyteczna dla testowania systemu są omawiane na spotkaniach
- Użycie kwestionariuszy i wykazów – odpowiedzi na pytania (np. jak, gdzie, kto i dlaczego) pozwalają ustalić jakie obszary należy przebadać w systemie

Testowanie iteracyjne

- Używany, gdy dodawane są wymagania systemowe z każdą wersją oprogramowania (np. model spiralny)
- Ma zapewniać, że zaimplementowana wcześniej funkcjonalność nadal jest testowana – wymagane powtarzalne testy
- Zmiany w testach wykonywane w każdej iteracji
- Nie wszystkie rodzaje testów są wykonywane w wcześniejszych iteracjach (np. testy wydajnościowe)

Testowanie zwinne i ekstremalne

- Ma na celu zapewnienie, że wymagania użytkownika są spełnione w odpowiednim czasie
- Użytkownik jest częścią zespołu przygotowującego oprogramowanie (rozprasza wszelkie wątpliwości, odpowiada na pytania)
- Podobnie jak w poprzednim modelu oprogramowanie rozwija się w iteracjach

Testowanie zwinne i ekstremalne

- Testerzy nie są traktowani jako osobna grupa wykonująca swoje zadania
- Testerzy nie muszą wysyłać raportów i czekać na odpowiedź innych członków zespołu
- Testerzy traktowani jako pomost pomiędzy programistami (znającymi technologie i sposoby implementacji pomysłów) a użytkownikami (znającymi swoje wymagania) – wyjaśniają różne punkty widzenia

Zasiew błędów

- Zazwyczaj polega na tym, że część członków zespołu wprowadza do programu błędu, które pozostała część zespołu próbuje zidentyfikować
- Wprowadzane błędy są podobne do rzeczywistych defektów
- W czasie wyszukiwania zasianych błędów można zidentyfikować również błędy, które nie były wprowadzone celowo

Zasiew błędów

- Może być wskazówką efektywności procesu testowania, może mierzyć liczbę usuniętych błędów i pozostających w kodzie

  liczukb = (liczbz / liczbwzb) * liczwab

  - liczukb – liczba ukrytych błędów
  - liczbz – liczba błędów zasianych
  - liczbwzb – liczba wykrytych zasianych błędów
  - liczwab – liczba wykrytych autentycznych błędów

Specyfika systemów obiektowych

- Znaczne powiązanie danych z metodami działającymi na tych danych
- Operacje wykonywane jedynie z wykorzystaniem publicznego interfejsu, ukrywanie informacji
- Dziedziczenie umożliwia definiowanie nowych klas z już istniejących. Istotne także ze względu na wiązanie dynamiczne i polimorfizm

Testowanie systemów obiektowych

- Testowanie jednostkowe klas
- Testowanie współpracy klas ze sobą (testy integracyjne klas)
- Testowanie systemowe
- Testy regresyjne

Testowanie klas

- Klasy są zaprojektowane do częstego używania. Błędy w kodzie mogą być odczuwalne w każdej instancji klasy.
- Wiele błędów pojawiło się w kodzie klasy od razu jak została zdefiniowana. Zwłoka w wykryciu tych błędów będzie skutkowała tym, że oprogramowanie korzystające z klasy będzie narażone na ich skutki.
- Klasy mają dostarczają różnych funkcjonalności, oprogramowanie może korzystać z różnych funkcjonalności

Testowanie klas

- Klasa jest kombinacją danych i metod. Jeśli dane i metody nie są ze sobą zsynchronizowane na potrzeby testów jednostkowych, może to powodować pojawienie się ciężkich do wykrycia błędów
- Systemy obiektowe korzystają z dodatkowych własności np. dziedziczenia, które powodują dodawanie nowego kontekstu do budowanych bloków kodu. Może to powodować powstawanie błędów w przyszłości

Metody użyteczne w testowanie klas

- Prawie każda klasa posiada zmienne, które mogą być efektywnie testowane z wykorzystaniem analizy wartości granicznych, metody klas równoważności
- Nie wszystkie metody są jednocześnie wykonywane przez oprogramowanie klienckie, testowanie pokrycia kodu jest użyteczne do zapewnienia, że każda metoda jest przetestowana

Metody użyteczne w testowaniu klas

- Każda klasa może posiadać metody, które posiadają logikę proceduralną, które testowanie z wykorzystaniem pokrycia rozgałęzień kodu, pokrycia warunków i złożoności kodu
- Ponieważ obiekty klas mogą być tworzone wielokrotnie przez różne oprogramowanie klienckie, więc mogą być użyteczne różnego rodzaju testy obciążeniowe

Założenia dla testowania klas

- Testowanie obiektu w całym jego okresie życia od utworzenie aż do zniszczenia
- Testowanie bardziej skomplikowanych metod po testach metod prostych
- Testowanie metod od prywatnych do publicznych
- Wysyłanie komunikatów do każdej metody przynajmniej raz

Założenia do testowania klas

- Testowanie na początku konstruktorów
  - gdy istnieje wiele sposobów inicjowania obiektów powinny być również przetestowane
- Testowanie metod dostępowych
- Testowanie metod modyfikujących pola składowe klasy
- Testowanie niszczenia obiektu
  - wszystkie zasoby przydzielone przez obiekt powinny być zwolnione

Testowanie integracyjne

- Systemy obiektowe są budowane z wielu małych komponentów – testy integracyjne są dlatego bardzo istotne
- Komponenty systemów obiektowych są opracowywane równolegle – wyższa potrzeba testów integracyjnych
- Biorąc pod uwagę równoległą pracą na komponentami systemu należy również przetestować dostępność klas podczas testów integracyjnych

Testy systemowe i współdziałania

- Klasy mogą składać się z wielu części, które nie są uruchamiane w jednym czasie. Użycie danej części może powodować błędy dopiero później
- Klasy mogą być w różny sposób zestawiane przez oprogramowanie klienckie co może prowadzić do nowych błędów
- Obiekt może nie zwalniać wszystkich zasobów, problemy z tym związane mogą być uwidocznione dopiero w czasie testów systemowych

- Enkapsulacja i dziedziczenie
  - Wprowadza dodatkowy kontekst, który należy uwzględnić w czasie testowania aplikacji
  - Wykorzystywane testowanie inkrementacyjne
- Klasy abstrakcyjne
  - Wymagają ponownego przetestowania przy każdej zmianie interfejsu

Testowanie własności klas

- Polimorfizm
  - Każda metoda nosząca tą są samą nazwę powinna być testowana oddzielnie
  - Kłopoty z utrzymaniem kodu
- Wiązanie dynamiczne
  - Konwencjonalne metody testowania pokrycia kodu zmodyfikowane na potrzeby testowania wiązania dynamicznego
  - Wyższa możliwość powstawania późniejszych błędów

Testy właściwości klas

- Komunikacja wewnętrzna z wykorzystaniem komunikatów
  - sekwencje komunikatów
  - diagramy sekwencji
- Ponowne użycie obiektów i równoległa implementacja klas
  - częstsze testy integracyjne i regresyjne
  - testy integracyjne i jednostkowe połączone ze sobą
  - potrzeba testowania interfejsów

Testowanie użyteczności i dostępności

- Testy użyteczności są subiektywne
- Postrzeganie dobrej użyteczności zależy od użytkownika
- Interfejs użytkownika może być stworzony w czasie projektowania systemu – tworzenie interfejsu użytkownika nie wpływa jednak w bezpośredni sposób na wymagania funkcjonalne stawiane aplikacji

Testowanie użyteczności

- Użyteczność testuje się z punktu widzenia użytkowników
- Sprawdza czy produkt jest łatwy do użycia przez różne grupy użytkowników
- Jest procesem sprawdzającym rozbieżności pomiędzy zaprojektowanym interfejsem a oczekiwaniami użytkownika

Testowanie użyteczności

- Łatwość użycia
  - może być różna dla różnych grup użytkowników
- Szybkość działania
  - zależy od wielu czynników – np. systemowych czy sprzętowych
- Estetyka aplikacji
  - zależna od postrzegania aplikacji przez użytkownika

Testowanie użyteczności

- Testowanie użyteczności obejmuje
  - aplikację
    - pozytywne i negatywne testowanie
  - dokumentację programu
  - komunikaty aplikacji
  - media z jakich korzysta aplikacja

Podejścia do użyteczności

- Czynniki obiektywne w testowaniu użyteczności np.:
  - liczba kliknięć myszy
  - liczba klawiszy do naciśnięcia
  - liczba menu do przejścia
  - liczba poleceń potrzebnych do wykonania zadania

Dopasowanie użytkowników

- Dwie grupy najbardziej przydatne do testowania użyteczności
  - Typowi przedstawiciele aktualnej grupy użytkowników – umożliwiają wyodrębnienie wzorców korzystania z aplikacji
  - Nowi użytkownicy nieposiadający przyzwyczajeń – mogą zidentyfikować problemy z użytecznością niedostrzegalne przez użytkowników korzystających z aplikacji dłuższy czas

Moment wykonania testów użyteczności

- Planowanie ↔ Uwzględnienie testów użyteczności w planie testowania
- Projektowanie ↔ Projektowanie walidacji użyteczności
- Kodowanie ↔ Testowanie użyteczności

Projektowanie użyteczności

- Arkusze stylów
  - grupują elementy interfejsu użytkownika
  - zapewniają spójność interfejsu użytkownika
- Prototypy ekranów aplikacji
  - są projektowane tak jak będą wyglądały dla użytkownika, bez połączenia z funkcjonalnością produktu – umożliwia odrębne testowanie
  - symulują wykonywanie różnych ścieżek programu – wyświetlanie komunikatów i okien

Projektowanie użyteczności

- Projektowanie na papierze
  - projekt okien aplikacji, układu elementów, menu narysowany na papierze jest wysyłany do użytkownika w celu akceptacji
- Projektowanie layoutu
  - pomocne przy układaniu różnorodnych elementów aplikacji na ekranie
  - zmiana układu elementów aplikacji może prowadzić do błędów użytkownika

Problemy z testowaniem użyteczności

- Grupy użytkowników posiadające niezgodne ze sobą wymagania
- Użytkownicy mogą nie być w stanie wyrazić użycia produktu prowadzących do problemów
- Różne grupy użytkowników
  - eksperci – mogą znajdować obejścia istniejących problemów
  - początkujący – nie mają jeszcze odpowiedniej wiedzy o użyciu produktu

Testowanie użyteczności

- Obserwacja wzorców zachowania użytkowników
  - W jaki sposób użytkownicy wykonują operacje w celu osiągnięcia określonego zadania?
  - Ile im zajmuje to czasu?
  - Czy czas odpowiedzi systemu jest satysfakcjonujący?
  - W jakich okolicznościach napotykają na problemy?
  - Jak je omijają?
  - Jakie zachowania aplikacji powodują zmieszanie użytkownika?

Czynniki jakości użyteczności

- Czytelność
  - produkt i jego dokumentacja posiada prostą i logiczną strukturę
  - operacje są grupowane na podstawie scenariuszy uzyskanych od użytkownika
  - często wykonywane operacje są bardziej widoczne w interfejsie użytkownika
  - komponenty programu używają terminologii użytkownika

Czynniki jakości użyteczności

- Spójność
  - z uznanymi standardami
  - wyglądem aplikacji dla określonej platformy
  - poprzednimi wersjami aplikacji
  - innymi produktami danego producenta
- Nawigowalność
  - łatwość wyboru różnych operacji wykonywanych przed produkt
  - np. minimalizacja liczby kliknięć myszy potrzebnych do wykonania operacji

Czynniki jakości użyteczności

- Czas odpowiedzi aplikacji
  - szybkość reakcji produktu na żądania użytkownika – różne od wydajności aplikacji
  - np. wizualizacja ile procent danych zostało przetworzonych przez aplikację

Testowanie estetyki

- Często subiektywne, zależne od odczuć użytkownika
- Dotyczy wszelkich aspektów aplikacji związanych z jej wyglądem – kolorów, okien, komunikatów i obrazów
- Nie każdy produkt musi być dziełem sztuki, wystarczy, że będzie miły dla oka

Testowanie dostępności

- Oznacza testowanie aplikacji przez niepełnosprawnych użytkowników
- Dostarczanie dla tej grupy użytkowników alternatywnych sposobów wykonywania operacji – narzędzia wspierające te operacje
- Podział na:
  - Podstawową dostępność – dostarczana przez system operacyjny i sprzęt
  - Dostępność produktu

Planowanie testowania

- Wybór zakresu testowania wraz z identyfikacją co powinno być przetestowane a co można nie testować
- Wybór sposobu wykonania testów – podział zadania na mniejsze zadania i przyjęcie strategii przeprowadzenia zadania
- Wybór zasobów jakie należy przetestować – zarówno komputerowe jak i te związane z ludźmi

Planowanie testowania

- Przyjęcie ram czasowych, w których będą wykonywane testy
- Ocena ryzyka jakie musi być podjęte z wykonaniem wcześniej wymienionych zadań, wraz z odpowiednim planem awaryjnym i migracyjnym

Zasięg testowania

- Wyodrębnienie podstawowych składowych końcowego produktu
- Podział produktu na zbiór podstawowych składowych
- Ustawienie priorytetów testów dla różnych aspektów oprogramowania
- Zadecydowanie co należy testować a co nie
- Estymacja liczby zasobów potrzebnych do testowania na podstawie zebranych wcześniej danych

Wybór priorytetów

- Cechy, które są nowe i krytyczne dla nowego wydania produktu
  - Spełnienie oczekiwań użytkownika
  - Spodziewane w nowym kodzie wiele nowych błędów
- Cechy, których nieprawidłowe działanie może spowodować katastrofalne skutki – np. mechanizm odzyskiwania bazy danych

Wybór priorytetów

- Cechy oprogramowania, dla których spodziewane jest skomplikowane testowanie
- Cechy, które są rozszerzeniem mechanizmów z obecnej wersji oprogramowania, które były źródłem wielu błędów
- Wymienione cechy rzadko można zidentyfikować jako występujące samodzielne, częściej spotykane są w różnych kombinacjach

Wybór strategii testowania

- Wybór testów potrzebnych do przetestowania funkcjonalności produktu
- Wybór konfiguracji i scenariuszy potrzebnych do testowania funkcjonalności produktu
- Wybór testów integracyjnych, koniecznych do sprawdzenie czy zestawienie elementów prowadzi do poprawnej ich pracy
- Wybór potrzebnej walidacji lokalizacji produktu
- Wybór koniecznych testów niefunkcjonalnych

Ustanowienie kryteriów testowania

- Wybór jasnych kryteriów dla danych testowych i dla spodziewanych wyników
- Wybór warunków, w których testy powinny zostać wstrzymane
  - Wykryto szereg błędów
  - Napotkano na problem, który uniemożliwia dalsze testowanie
  - Wydano nową wersję, która ma na celu naprawienie błędów w obecnej wersji

Identyfikacja zakresu obowiązków, personelu i wymagań szkoleniowych

- Testowanie wymaga wymaga inżynierów testowania, prowadzących testy i managerów testowania
- Wyznaczenie:
  - Obowiązków w celu wykonania określonego zadania
  - Jasnej listy obowiązków dla członków personelu
  - Wzajemnego uzupełniania zadań personelu
- Szkolenia w razie braku odpowiednich umiejętności potrzebnych do wykonywania testów

Identyfikacja wymagań odnośnie zasobów

- Wymagania sprzętowe dla przeprowadzenie testów
- Liczby konfiguracji systemowych dla testowanego produktu
- Koszty ogólne narzędzi potrzebnych do automatyzacji testów
- Narzędzia wspierające – kompilatory, generatory danych testowych itd
- Wymagana liczba licencji narzędzi potrzebnych do wykonania zadania

Identyfikacja dostarczanych wyników testowania

- Planowanie samych testów
- Identyfikacja przypadków testowych
- Przypadki testowe wraz z ich automatyzacją
- Logi wyprodukowane przez wykonane testy
- Raporty podsumowujące testy

Ocena rozmiaru i wysiłku koniecznego do wykonania testów

- Ocena rozmiaru – oszacowanie liczby koniecznych do wykonania testów
  - rozmiar produktu przeznaczonego do testowania
    - liczba linii kodu
    - liczba punktów funkcyjnych
    - liczba ekranów, raportów i transakcji
  - zakres wymaganej automatyzacji
  - liczba platform i środowisk koniecznych do rozważenia

Struktura podziału pracy

- Liczba przypadków testowych
- Liczba scenariuszy testowych
- Liczba konfiguracji do przetestowania
- Czynniki wpływające:
  - Dane dotyczące produktywności
  - Możliwości ponownego użycia testów
  - Odporność procesu
    - Dobrze udokumentowane standardy, sprawdzone sposoby wykonania operacji, obiektywne metody mierzenia efektywności

Podział i planowanie działań

- Identyfikacja zewnętrznych i wewnętrznych zależności
- Ustalenie kolejności działań bazując na spodziewanym czasie trwania działania
- Identyfikacja czasu dla każdego działania
- Monitorowanie postępu zarówno pod względem czasu jak i wykonanego wysiłku
- Jeśli konieczne zmiana planów i zasobów

Zarządzanie ryzykiem

- Identyfikacja potencjalnego ryzyka
- Pomiar potencjalnego ryzyka
  - Prawdopodobieństwo wystąpienia
  - Potencjalny wpływ
- Planowanie sposobów złagodzenia ryzyka
- Odpowiedź na realnie występujące ryzyko

Problemy

- Niejasne wymagania
- Zależności pomiędzy zadaniami w planie
- Niewystarczający czas dla testowania
- Występowanie defektów blokujących pracę
- Obecność testerów z odpowiednimi umiejętnościami i motywacją
- Brak narzędzi automatyzujących testowanie

Zarządzanie testami

- Wybór standardów
  - Konwencje nazewnictwa
  - Standardy dokumentowania testów
  - Standardy kodowania i raportowania testów
- Zarządzanie infrastrukturą testów – baza danych przypadków testowych, repozytorium błędów, narzędzia konfiguracyjne
- Zarządzanie ludźmi
- Synchronizacja testów z planami wydania

Proces testowania

- Specyfikacja przypadku testowego
  - Cel testu
  - Elementy testowane
  - Konieczne środowisko uruchomieniowe
  - Dane wejściowe dla testu
  - Kroki potrzebne do wykonania testu
  - Spodziewane wyniki
  - Kroki potrzebne do porównania wyników z spodziewanymi wynikami
  - Relacje pomiędzy testem a innymi testami

Proces testowania

- Aktualizacja macierzy śledzenia
- Identyfikacja potencjalnych kandydatów dla automatyzacji
- Zaprogramowanie i przyłączenie przypadków testowych
- Wykonanie przypadków testowych
- Zbieranie i analiza metryk
- Przygotowanie raportu podsumowywującego

Raporty z testów

- Raporty wykonania testu
- Raporty z cyklów testowania
- Raporty podsumowujące testowanie
  - Zależne od fazy testowania, tworzone po wykonaniu każdej fazy testowania
  - Raporty końcowe

Automatyzacja testów

- Automatyzacja pozwala zaoszczędzić czas – testy mogą być uruchamiane i wykonywane szybciej. Umożliwia to:
  - zaprojektowanie dodatkowych przypadków testowych umożliwiających np. lepsze pokrycie kodu
  - wykonanie bardziej skomplikowanych testów np. testów doraźnych
  - przeprowadzenie dodatkowe testy manualne

Automatyzacja testów

- Automatyzacja uwalnia testerów od przyziemnych zadań i pozwala wykorzystać zaoszczędzony czas na bardziej produktywne zajęcia
- Testowanie automatyczne może być bardziej wiarygodne
  - Większa możliwość powstawania pomyłek przy ręcznym uruchomieniu testów

Automatyzacja testów

- Automatyzacja pomaga w bezpośrednim testowaniu
  - Nie ma dużej przerwy pomiędzy programowaniem a testami
- Automatyzacja pozwala ochronić organizację przed wypaleniem zawodowym inżynierów testowania
  - Lepszy transfer umiejętności pomiędzy testerami
  - Testy stają się mniej zależne od konkretnych testerów

Automatyzacja testów

- Automatyzacja otwiera możliwość lepszego wykorzystania globalnych zasobów
  - Testy mogą być uruchamiane przez większość czasu w różnych strefach czasowych
- Niektóre testy nie mogą być uruchomione bez automatyzacji
  - np. testy obciążeniowe i wydajnościowe systemu wymagającego zalogowania tysięcy użytkowników

Automatyzacja testowania

- Nagrywanie i odtwarzanie
  - Nagrywanie czynności użytkownika – kliknięcia myszą, akcje wykonywane klawiaturą – i ich późniejsze odtwarzanie w tej samej kolejności
    - Łatwe nagrywanie i odtwarzanie skryptów
    - Skrypty mogą być odtwarzane wielokrotnie
    - Wartości wprowadzane ręcznie – może być trudne uzyskanie ogólnych skryptów

Automatyzacja skryptów

- Testy sterowane danymi
  - Metoda pozwala na generowanie testów na podstawie zestawów danych wejściowych i wyjściowych
- Testy sterowane akcjami
  - Wszystkie akcje jakie mogą być wykonane przez aplikacje są generowane automatycznie na podstawie zbioru kontrolek jakie zostały zdefiniowane do testowania

Identyfikacja testów podatnych na automatyzację

- Testowanie wydajności, niezawodności, obciążeniowe
  - Testy wymagające wielu użytkowników i dużego czasu przeznaczonego na ich wykonanie
- Testowanie regresyjne
  - Powtarzalne, uruchamiane wielokrotnie
- Testowanie funkcjonalne
  - Może wymagać skomplikowanych przygotowań – automatyzacja umożliwi uruchamianie przygotowanych przez ekspertów testów mniej doświadczonym testerom

Automatyzacja testowania

- Automatyzacja powinna być rozważana przede wszystkim dla elementów, które nie zmieniają się lub zmieniają się rzadko
- Automatyzacja testów, które odnoszą się do zgodności z standardami np. czy dane oprogramowanie spełnia wymagania stawiane jakiemuś ustandaryzowanemu protokołowi

Architektura dla automatyzacji testowania

- Moduły zewnętrzne
  - Baza przypadków testowych
  - Baza błędów
- Moduły plików konfiguracyjnych i scenariuszy
  - Scenariusze – informacja jak uruchomić określony test
  - Pliki konfiguracyjne – zawierają zbiór zmiennych używanych w automatyzacji

Architektura dla automatyzacji oprogramowania

- Moduły przypadków testowych i frameworku testowego
  - Przypadki testowe wzięte z bazy przypadków testowych
  - Przypadki testowe są wykonywane dla innych modułów, nie przeprowadzają żadnych interakcji
  - Framework testowy łączy przypadek testowy z sposobem jego wykonania
  - Framework testowy jest rdzeniem architektury automatyzacji testowania

Architektura dla automatyzacji oprogramowania

- Moduły narzędzi i wyników
  - Narzędzia wspierają proces tworzenia przypadków testowych
  - Wyniki są zachowywane w celu dalszej analizy, umożliwiają również przygotowanie przypadków testowych dla dalszych faz tworzenia oprogramowania
- Moduły generacji raportów i metryk
  - Generowanie raportów podsumowujących daną fazę testowania oprogramowania

Wymagania stawiane testom

- Brak ustawionych na sztywno wartości dla zestawów testowych
- Zestawy testowy powinny mieć możliwość dalszej rozbudowy
  - Dodany test nie powinien wpływać na inne przygotowane wcześniej testy
  - Dodanie nowego testu nie powinno wymagać ponownego przeprowadzenia już przeprowadzonych testów
  - Dodanie zestawu testowego nie powinno wpływać na inne zestawy testowe

Wymagania stawiane testom

- Zestawy testowe powinny mieć możliwość ponownego użycia
  - Test powinien robić jedynie to co jest spodziewane, że robi
  - Testy powinny być modularne
- Automatyczne ustawianie i czyszczenie po teście
- Niezależność przypadków testowych

Wymagania stawiane testom

- Izolacja testów w czasie wykonania
  - Elementy środowiska mogą wpływać na wyniki testów – blokowanie niektórych zdarzeń
- Standardy kodowania i struktury katalogowej
  - Ułatwia zrozumienie nowy pracownikom przypadków testowych
  - Ułatwia przenoszenie kodu pomiędzy platformami
  - Wymuszenie struktury katalogowej może ułatwić zrównoleglenie testów

Wymagania stawiane testom

- Selektywne wykonanie przypadków testowych
  - Ułatwienie w wyborze z: Wiele zestawów testowych → wielu programów testujących → wielu przypadków testowych
- Losowe wykonanie przypadków testowych
  - Wybór niektórych testów z zestawu przypadków testowych
- Równoległe wykonanie przypadków testowych

Wymagania stawiane testom

- Zapętlenie przypadków testowych
  - Testy wykonywane iteracyjnie dla znanej z góry liczby iteracji
  - Testy wykonywane w określonym przedziale czasu
- Grupowanie przypadków testowych
  - Umożliwia wykonanie testów w przedefiniowanej kombinacji scenariuszy
- Wykonanie przypadków testowych w oparciu na wcześniejsze wyniki

Wymagania stawiane testom

- Zdalne wykonanie przypadków testowych
  - Testy mogą wymagać więcej niż jednej maszyny do ich uruchomienia
  - Umożliwienie uruchomienia testów, zbieranie logów, informacji o postępie testów
- Automatyczne zachowywanie danych testowych
- Schematy raportowania
  - Jakie informacje mają być uwzględnione w raportcie

Wymagania stawiane testom

- Niezależność od języka
  - Framework testowy niezależny od języka programowania
  - Framework powinien dostarczać interfejsów dla najpopularniejszych języków
- Przenośność na różne platformy sprzętowe, systemowe

Metryki przydatne przy testowaniu

- Metryki wyprowadzają informacje z surowych danych w celu ułatwienia podejmowanie decyzji
  - Relacje pomiędzy danymi
  - Przyczyny i efekty korelacji pomiędzy zaobserwowanymi danymi
  - Wskaźniki jak dane mogą być użyte w dalszym planowaniu i ulepszaniu produktu

Metryki

- Odpowiednie parametry muszą być mierzone, parametry dotyczące produktu lub procesu jego tworzenia.
- Właściwa analiza musi być wykonana na mierzonych danych w celu wyprowadzenia poprawnych konkluzji dotyczących poprawności produktu lub procesu
- Wyniki analizy muszą być zaprezentowane w odpowiedniej formie dla zainteresowanych stron w celu podjęcia właściwych decyzji

Metryki

- Wysiłek – czas spędzony na jakiejś częściowej aktywności lub fazie.
- Kroki w zdobywaniu metryki programu
  - Identyfikacji tego co należy zmierzyć
  - Transformacja tego co zostało zmierzone do metryki
  - Podjęcie decyzji co do wymagań operacyjnych
  - Wykonanie analiz metryk
  - Wybór i wykonanie akcji
  - Sprecyzowanie miar i metryk

Wybór miar

- To co jest mierzone powinno być dobrane odpowiednio do zamierzonych celów (wysiłek poświęcony na testowanie, liczba przypadków testowych, liczba raportowanych błędów).
- Mierzone wielkości powinny być naturalne i nie powinny powodować nadmiernych kosztów.
- To co jest mierzone powinno być na właściwym poziomie ziarnistości w celu osiągnięcia celów dla których pomiar jest dokonywany.

Przydatność metryk w testowaniu

- Mierzenie postępu w testowaniu
  - Produktywność wykonywanych testów
  - Data zakończenia testów (oszacowana)
    - Na podstawie ilorazu liczby testów do wykonania przez liczbę testów wykonywanych na dzień
  - Liczba dni potrzebnych na usunięcie błędów
    - (liczba błędów do usunięcia + liczba przewidywanych błędów) / zdolność usuwania błędów

Przydatność metryk w testowaniu

- Oszacowanie daty wydania finalnej wersji
  - Max (liczba dni potrzebnych na testy, liczba dni potrzebnych do usunięcia błędów)
- Oszacowanie jakości wydanego oprogramowania
- Wybór elementów jakie powinny być wzięte pod uwagę w planowaniu wydania oprogramowania

Typy metryk

- Metryki projektu – jak projekt jest planowany i wykonywany
- Metryki postępu – śledzące jak wygląda postęp w różnych rodzajach działalności związanych z projektem
  - Aktywność związana z programowaniem
  - Aktywność związana z testowaniem
- Metryki produktywności – zbierające różnego rodzaju miary, które mają wpływ na planowanie testowania

Typy metryk

- Metryki projektu
  - Wariancja włożonego wysiłku
  - Wariancja harmonogramu
  - Rozkład wysiłku
- Metryki postępu
  - Wskaźnik odnalezionych błędów
  - Wskaźnik naprawionych błędów
  - Wskaźnik pozostających błędów
  - Wskaźnik priorytetu pozostających błędów

Typy metryk

- Trend błędów
- Ważony trend błędów
- Metryki produktywności
  - Liczba błędów na 100 godzin testowania
  - Liczba przypadków testowych na 100 godzin testowania
  - Liczba zaimplementowanych przypadków testowania na 100 godzin
  - Liczba błędów na 100 przypadków testowych

Metryki projektu

- Różnego rodzaju aktywności, podstawowy wkład do projektu i harmonogram prac są wejściem dla początku projektu
- Aktualny wkład i czas poświęcony na działalność są dodawane w miarę jak jest rozwijany projekt.
- Skorygowany wkład i harmonogram – ponownie obliczane, gdy jest to potrzebne

Wariancja wysiłku

- Podstawowa estymacja wysiłku, uaktualniona estymacja wysiłku i aktualny wysiłek
- Wariancja % = (Aktualny wysiłek – Uaktualniony oszacowany wysiłek)/Uaktualniony oszacowany wysiłek * 100
- Wariancja większa od 5 % oznacza, że należy poprawić estymację
- Może być ujemna w przypadku przesacowania

Wariancja harmonogramu

- Zgodność wykonywanych zadań z harmonogramem
- Podobnie jak poprzednia metryka jest odchyleniem aktualnego harmonogramu od harmonogramu oszacowanego
- Akceptowalny poziom to 0 – 5 %

Metryki postępu

- Pozwalają ocenić jak wygląda sprawa postępu w spełnianiu wymagań jakości oprogramowania
- Głównym wskaźnikiem jakości oprogramowania – liczba błędów oprogramowania – znalezionych i oszacowanych, że pozostają w oprogramowaniu
- Pozwalają ocenić jak błędy wpływają na oprogramowanie

Metryki postępu

Priorytet	Co miara oznacza
1	Najwyższy poziom (przed następną wersją)
2	Wysoki poziom(przed następnym cyklem testowania)
3	Średni poziom (jeśli czas pozwoli usunięcie przed wydaniem produktu)
4	Niski poziom (można odłożyć usunięcie)

Dotkliwość błędu	Co miara oznacza
1	Dotyka podstawowej funkcjonalności
2	Niespodziewany błąd lub niepracująca funkcjonalność
3	Dotyka mniej znaczącej funkcjonalności
4	Problem kosmetyczny

Metryki produktywności

- Łączą kilka miar i parametrów z wysiłkiem włożonym w powstawanie produktu
- Pomagają w:
  - Estymacji czasu pojawienia się produktu
  - Estymacji liczby błędów
  - Estymacji jakości
  - Estymacji kosztów

Metryki produktywności

- Pozwalają sprawdzić:
  - efektywność testowania
  - czy wzrasta jakość produktu
  - czy czas poświęcony na testowanie nie jest marnowany