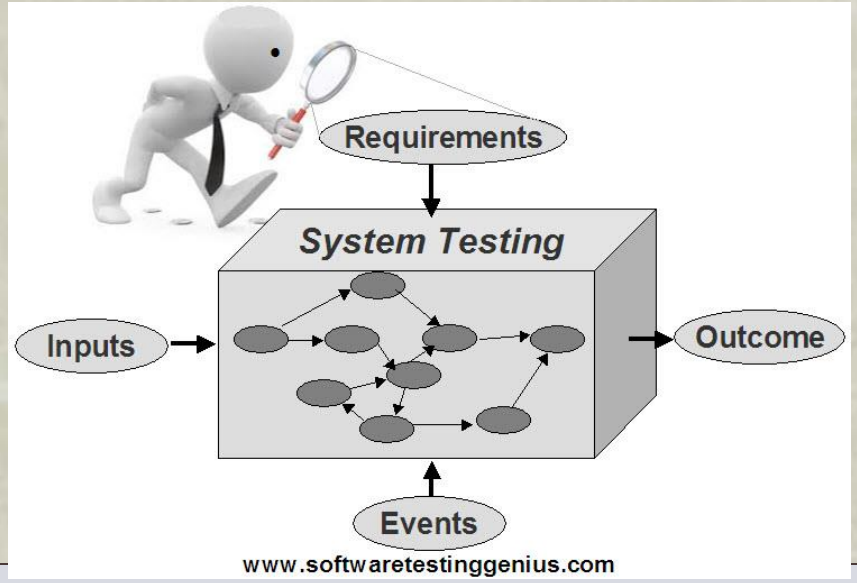
Metody zarządzania projektami informatycznymi

Wykład 6

Testowanie oprogramowania – cz. 1



dr hab. inż. Aneta Poniszewska-Marańda
Instytut Informatyki Politechniki Łódzkiej

Testowanie oprogramowania

- gwałtowny rozwój komputeryzacji życia
- powstaje coraz to większa liczba używanego oprogramowania, którego jakość zależy między innymi od sprawdzenia poprawnego wykonania produktu, czyli od przetestowania
- **testowanie oprogramowania** – proces związany z wytwarzaniem oprogramowania
- jeden z procesów zapewnienia jakości oprogramowania
- testowanie może być wdrożone w dowolnym momencie wytwarzania oprogramowania (w zależności od wybranej metody)
  - w podejściu klasycznym największy wysiłek zespołu następuje po definicji wymagań oraz po zaimplementowaniu wszystkich zdefiniowanych funkcji
  - zwinne metody wytwarzania oprogramowania skupiają się bardziej na jednostkowych testach, wykonywanych przez członków zespołu programistycznego, zanim oprogramowanie trafi do właściwego zespołu testerów

Testowanie oprogramowania

- testowanie ma na celu *weryfikację* oraz *walidację* oprogramowania
- **weryfikacja** pozwala skontrolować, czy wytwarzane oprogramowanie jest zgodne ze specyfikacją ***„Czy budujemy prawidłowo produkt?***
  - potwierdzenie, poprzez przebadanie i dostarczenie oczywistego dowodu, że konkretne wymagania zostały spełnione
  - w projektowaniu i wytwarzaniu dotyczy procesu badania wyniku danej aktywności w celu ustalenia zgodności z podanym dla tej aktywności wymaganiem
- **walidacja** sprawdza, czy oprogramowanie jest zgodne z oczekiwaniami użytkownika ***„Czy budujemy prawidłowy produkt?”***
  - potwierdzenie, poprzez przebadanie i dostarczenie dowodu, że konkretne wymagania w stosunku do specyficznego, zamierzonego użycia spełnione
  - w projektowaniu i wytwarzaniu dotyczy procesu badania produktu w celu ustalenia jego zgodności z potrzebami użytkownika
  - zwykle wykonywana dla produktu końcowego

Testowanie oprogramowania

- główny cel testowania – wyszukiwanie błędów w oprogramowaniu (ang. *software bug*)
- w inżynierii oprogramowania różne pojęcia:
- **defekt** – błąd popełniony przez projektantów lub programistów podczas tworzenia oprogramowania
  - problem odkryty w fazie cyklu życia oprogramowania późniejszej od fazy, w której pojawiła się przyczyna
- **błąd** – problem, nieprawidłowe działanie ujawnione w tej samej fazie, w której pojawiła się przyczyna
- **error** – niezgodność pomiędzy dostarczonym przez funkcję, zaobserwowanym lub zmierzonym rezultatem jej wykonania a oczekiwaną wartością
  - mogą być powodowane celowo w procesie testowania aplikacji

Testowanie oprogramowania

- **…**
- **failure** – niezdolność komponentu lub systemu do wykonania operacji np. w określonym w wymaganiach czasie
- **exception** – nieobsługiwany wyjątek, który powoduje zawieszenie lub przerwanie działania programu
  - może pojawić się w związku z adresowaniem pamięci, danymi, wykonaną operacją, przepełnieniem zmiennej, itp.
- **defect**, **bug**, **fault** – wada modułu lub systemu, która może spowodować, że moduł lub system nie wykona zakładanej czynności
  - defekt, który wystąpi podczas uruchomienia programu, może spowodować awarię modułu lub systemu
- **deviation**, **incident** – każde zdarzenie występujące w procesie testowania, które wymaga zbadania

Testowanie oprogramowania

Błędy ludzkie w oprogramowaniu

- człowiek może zrobić **pomyłkę (error, mistake)**
- ... która powoduje **błąd, defekt** (defect, fault, bug) w kodzie, w oprogramowaniu, w systemie
- ... który może doprowadzić do **awarii, upadku (failure), jeżeli błędny** kod zostanie wykonany
- defekt w oprogramowaniu (dokumentacji) może, ale nie musi doprowadzić do awarii

Testowanie oprogramowania

Rola testowania oprogramowania

- zmniejsza ryzyko wystąpienia awarii w czasie użytkowania oprogramowania
- podnosi jakość oprogramowania
- konieczne do spełnienia wymagań kontraktu, wymagań prawnych, wymagań standardów przemysłowych
- …

Testowanie oprogramowania

Testowanie a jakość

- **testowanie**
  - pozwala zmierzyć jakość oprogramowania
  - pomaga budować zaufanie do oprogramowania
  - zwiększa jakość, jeżeli błędy są znajdowane i usuwane
  - zmniejsza ryzyko upadku systemu
  - pozwala na poprawę procesu produkcji oprogramowania
- testowanie wintegrowane w system zapewnienia **jakości** oprogramowania w organizacji (obok na przykład standardów kodowania, szkoleń, analizy incydentów)

Testowanie oprogramowania – cele

Cele testów

- znalezienie defektów
- uzyskanie informacji o poziomie jakości oprogramowania (zbudowanie zaufania)
- monitorowanie ryzyk produktu
- zapobieganie defektom:
  - projektowanie testów jak najwcześniej
  - przegląd dokumentacji (bazy testowej)
- …

Testowanie oprogramowania – cele

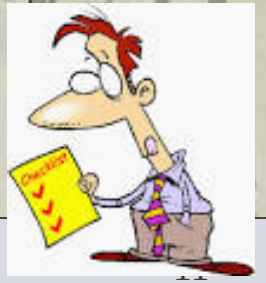
Cele testów

- szczegółowe cele zależne od punktu widzenia i momentu w cyklu życia oprogramowania:
  - *testy developerskie* – znalezienie i eliminacja jak największej liczby błędów
  - *testy akceptacyjne* – potwierdzenie, że oprogramowanie spełnia wymagania
  - *testy utrzymaniowe* – sprawdzenie, czy nowe defekty nie zostały wprowadzone
  - *testy operacyjne* – sprawdzenie charakterystyk niefunkcjonalnych oprogramowania
  - *testy konfirmacyjne* – potwierdzenie, że poprawka błędu jest prawidłowa
  - *testy regresji* – zapewnienie, że nowe błędy w częściach niezmienionych przez poprawki nie zostały wprowadzone/odkryte

Testowanie oprogramowania – zasady

7 zasad testowania

1. Testowanie udowadnia istnienie błędów
   - testowanie może pokazać, że błędy istnieją, ale nie może udowodnić, że błędów nie ma
2. Testy wyczerpujące są niemożliwe
   - przetestowanie wszystkiego jest niemożliwe – w zamian za to analiza ryzyka i priorytetyzacja testów
3. Testować jak najwcześniej
   - aktywności testowe powinny rozpoczynać się w cyklu życia oprogramowania tak szybko, jak to tylko możliwe
4. Kumulowanie się błędów
   - mała liczba modułów (funkcji) zawiera większość błędów wychodzących w trakcie testów lub użytkowania oprogramowania



Testowanie oprogramowania – zasady

7 zasad testowania

5. Paradoks pestycydów – oprogramowanie uodparnia się na testy
   - te same przypadki testowe powtarzane wielokrotnie nie znajdują nowych defektów
   - przypadki testowe powinny być przeglądane, aktualizowane
   - nowe przypadki testowe powinny być dodawane
6. Testowanie zależy od kontekstu
   - testy powinny być dostosowane do kontekstu oprogramowania
7. Mylne przekonanie o braku błędów
   - znalezienie i eliminacja błędów nie pomoże, jeżeli system jest nieużyteczny i nie spełnia potrzeb i oczekiwań klienta, użytkowników

Testowanie – aksjomaty

- aksjomaty testowania
  1. Programu nie da się przetestować całkowicie
  2. Testowanie jest ryzykowne
  3. Test nie udowodni braku błędów
  4. Im więcej błędów znaleziono, tym więcej błędów pozostało do znalezienia
  5. Nie wszystkie znalezione błędy zostaną naprawione
  6. Trudno powiedzieć, kiedy błąd jest błędem
  7. Specyfikacje produktów nigdy nie są gotowe

Testowanie oprogramowania – błąd

- 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 specyfikacja nie wspomina
  - 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

Testowanie oprogramowania

Testowanie z wykorzystaniem przypadków użycia

- każdy przypadek użycia ma określony cel, a jego wykonanie powinno doprowadzić do określonego rezultatu
- następujące elementy powinny zostać uwzględnione podczas projektowania przypadków testowych:
  - warunki uruchomienia
  - wszystkie możliwe przepływy sterowania
  - warunki zakończenia (stan systemu po zakończeniu przypadku użycia zarówno sukcesem, jak i porażką)

Testowanie oprogramowania

- systemy informatyczne są na tyle złożone, że sprawne zarządzanie projektem przez cały cykl życia aplikacji bardzo pracochłonne, o ile nie wspomagamy się odpowiednimi narzędziami
- *testowania oprogramowania powinno być integralną częścią tego cyklu* – narzędzia do testowania muszą mieć odpowiednią funkcjonalność
- oprogramowanie testujemy na bardzo różne sposoby, zależnie od tego, jaki aspekt jego działania jest w danym momencie ważny
  - jeżeli oprogramowanie podlega szybkim zmianom i by ich wprowadzanie nie naruszało istniejącej funkcji stosujemy *testy regresyjne*
  - jeśli chcemy pokazać klientowi, że aplikacja działa zgodnie z jego wymaganiami przygotowujemy *testy akceptacyjne*
  - *testy obciążeniowe* będziemy stosować w przypadku aplikacji, co do których musimy mieć pewność, że działają wydajnie

Testowanie – podstawowy proces testowy

1. Planowanie i kontrola
2. Analiza i projektowanie
3. Implementacja i wykonanie
4. Obliczanie kryteriów wyjścia i raportowanie
5. Zakończenie testów

Testowanie – proces testowy

1. Planowanie i kontrola testów

- Planowanie testów
  1. Weryfikacja misji testowania
  2. Zdefiniowanie strategii testowania
  3. Zdefiniowanie celów testowania (*test objectives*)
  4. Określenie aktywności testowych mających spełnić cele i misję testowania
- Kontrola testów
  1. Porównywanie aktualnego postępu w stosunku do założonego planu
  2. Raportowanie statusu (szczególnie odchyłek od założonego planu)
  3. Podejmowanie kroków niezbędnych do spełnienia założonej misji i celów testów
  4. Aktywność ciągła w projekcie
  5. Kontrola możliwa tylko dzięki ciągłemu monitorowaniu testów

Testowanie – proces testowy

2. Analiza i projektowanie

- zamiana ogólnych celów testowania (zdefiniowanych na etapie *Planowania*) na rzeczywiste i namacalne **warunki testowe** (*Analiza*) i **przypadki testowe** (*Projektowanie*)
  1. Przegląd bazy testów: wymagania, architektura, projekt systemu, projekt interfejsów
  2. Oszacowanie testowalności bazy testów i celów testów
  3. Identyfikacja i priorytetyzacja warunków testowych na podstawie analizy bazy testowej
  4. Projektowanie i priorytetyzacja przypadków testowych
  5. Identyfikacja koniecznych danych testowych
  6. Projekt środowiska testowego, identyfikacja wymaganej infrastruktury i narzędzi

Testowanie – proces testowy

3. Implementacja i wykonanie

- tworzenie **procedur testowych i skryptów testowych** na podstawie zdefiniowanych przypadków testowych – *Implementacja*
- wykonanie zaimplementowanych procedur testowych – *Wykonanie*

1. Implementacja przypadków testowych poprzez projektowanie i priorytetyzację procedur testowych
2. Wybór danych testowych
3. Przygotowanie automatycznych skryptów testowych
4. Stworzenie zestawów testów z procedur testowych dla wygodniejszego wykonywania testów
5. Weryfikacja przygotowania środowiska testowego
6. Wykonanie testów i logowanie wyników testów
7. Porównanie aktualnych wyników z wynikami spodziewanymi
8. Raportowanie niezgodności jako incydentów
9. Analiza przyczyn incydentów
10. Testy konfirmacyjne poprawionych błędów
11. Testy regresyjne niezmienionego kodu

Testowanie – proces testowy

4. Obliczanie kryteriów wyjścia

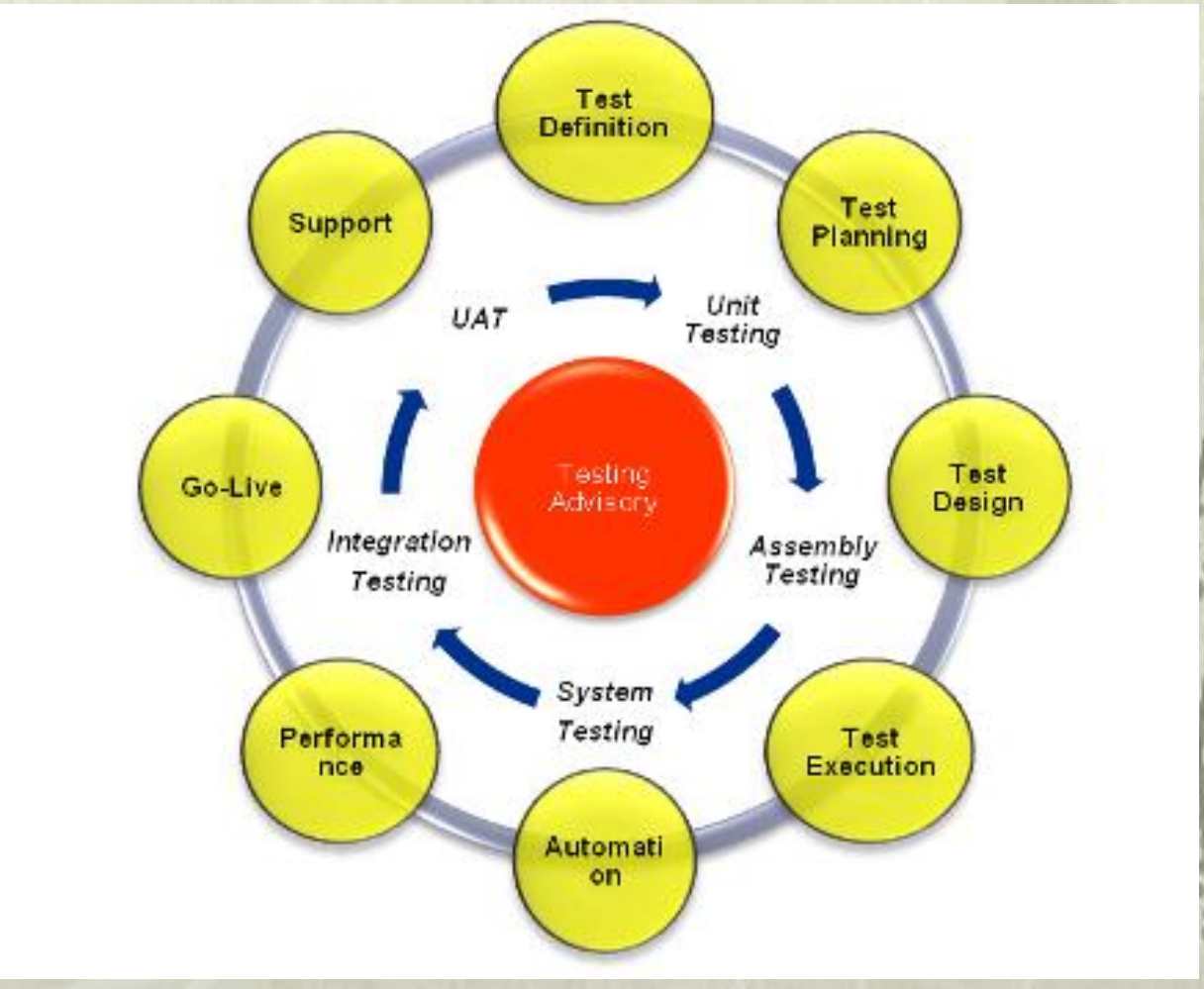
- oszacowanie kryteriów zakończenia testów i porównanie ze zdefiniowanymi celami:
  1. Porównanie wyników testów z kryteriami wyjścia zdefiniowanymi na etapie planowania
  2. Oszacowanie, czy konieczna jest kontynuacja testów
  3. Decyzja o dostarczeniu oprogramowania
  4. Przygotowanie Raportu Podsumowującego (*Test Summary Report*)

Testowanie – proces testowy

5. Zakończenie testów

- zebranie informacji z zakończonych testów w celu zgromadzenia doświadczeń, produktów testów, potrzebnych statystyk:
  1. Dostarczenie dokumentacji z testów
  2. Zamknięcie błędów
  3. Zgłoszenie żądań zmiany
  4. Archiwizacja pełnej dokumentacji testowej
  5. Analiza wniosków w celu ulepszenia procesu testów w przyszłych dostawach

Testowanie – proces testowy



Testowanie – rodzaje testów

- testy można klasyfikować z różnych punktów widzenia
- z punktu widzenia *głównego celu* testy można podzielić na:
  - *testy statystyczne*, których celem jest wykrycie przyczyn najczęstszych błędnych wykonań oraz ocena niezawodności systemu
  - *wykrywanie błędów*, czyli testy, których głównym celem jest wykrycie jak największej liczby błędów w programie
- z punktu widzenia podstawowej techniki wykonywania testów na:
  - *testy dynamiczne*, które polegają na wykonywaniu (fragmentu) programu i porównywaniu uzyskanych wyników z wynikami poprawnymi
  - *testy statyczne*, oparte na analizie kodu

Testowanie – rodzaje testów

- z punktu widzenia *sposobu realizacji* testy można podzielić na:
  - *testy funkcjonalne* – wcielamy się w rolę użytkownika, traktując oprogramowanie jak „czarną skrzynkę”, która wykonuje określone zadania i nie wnikamy w techniczne szczegóły działania programu
    - testy *czarnej skrzynki (black box testing)*
  - *testy strukturalne* – tester ma dostęp do kodu źródłowego oprogramowania, może obserwować jak zachowują się różne części aplikacji, jakie moduły i biblioteki są wykorzystywane w trakcie testu
    - testy *białej skrzynki (white box testing)*
- typowym przykładem testów strukturalnych są ***testy jednostkowe*** (*unit tests*) – tester lub programista tworzy kod, którego jedynym zadaniem jest sprawdzenie działania produkcyjnego kodu aplikacji

Testowanie – rodzaje testów

- z punktu widzenia *automatyzacji* testy można podzielić na:
  - testy ręczne – wykonywane ręcznie przez testera, który przechodzi przez interfejs użytkownika zgodnie z określoną sekwencją kroków
  - testy automatyczne – ich wykonanie nie wymaga udziału testera
  - zautomatyzowane jest przeprowadzanie testów jednostkowych, np. przy użyciu *Jakarta Ant* (narzędzia mają wbudowaną funkcję uruchamiania testów jednostkowych)
  - skomplikowaną sprawą jest automatyzacja testów w schemacie czarnej skrzynki – potrzebne specjalistyczne oprogramowanie, które pozwala uruchamiać napisane lub nagrane przez testera skrypty

Testowanie – rodzaje testów

- z punktu widzenia *zakresu aplikacji*, jaki obejmują testy na:
  - *testy jednostkowe* testują oprogramowanie na najbardziej podstawowym poziomie – na poziomie działania pojedynczych funkcji (metod)
  - *testy integracyjne* pozwalają sprawdzić, jak współpracują ze sobą różne komponenty oprogramowania, czy wszystko razem działa poprawnie, nie ma niezgodności interfejsów itp.
  - *testy systemowe* dotyczą działania aplikacji jako całości, zazwyczaj na tym poziomie testujemy różnego rodzaju wymagania niefunkcjonalne: szybkość działania, bezpieczeństwo, niezawodność, dobrą współpracę z innymi aplikacjami i sprzętem

Testowanie – rodzaje testów

- z punktu widzenia *przeznaczenia* testy można podzielić na:
  - *testy akceptacyjne* – testy wykonywane w celu sprawdzenia na ile oprogramowanie działa zgodnie z wymaganiami klienta
  - *testy funkcjonalne* – testy sprawdzające działanie oprogramowania zgodnie ze specyfikacją,
    - zdarza się, że klient wymaga do akceptacji produktu także wyników testów jednostkowych
  - *testy wydajnościowe i obciążeniowe*
  - ….

Testowanie – rodzaje testów

- z punktu widzenia *przeznaczenia* testy można podzielić na:
  - ...
  - *testy regresyjne* – bardzo ważny rodzaj testów, pełniących zasadniczą rolę w kwestii jakości oprogramowania
  - celem testów regresyjnych jest sprawdzenie, czy dodając nową funkcję lub poprawiając błędy nie naruszyliśmy niespodziewanie innej funkcji oprogramowania
    - powinny być wykonywanie zarówno na poziomie kodu aplikacji (jeśli to możliwe) – zazwyczaj są to testy jednostkowe – jak i na wyższym poziomie działania całej aplikacji
    - aplikacja testowana w ten sposób, że przechodzimy przez wybrane ścieżki działania oprogramowania tak, jakby to robił jego użytkownik
  - ...

Testowanie – rodzaje testów

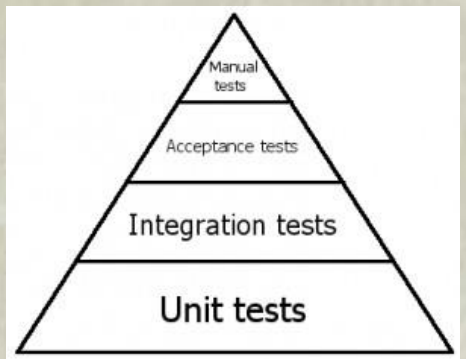
- z punktu widzenia *przeznaczenia* testy można podzielić na:
  - ...
  - *testy instalacyjne/testy konfiguracji* – służą, by sprawdzić, jak oprogramowanie zachowuje się na różnych platformach sprzętowych, systemach operacyjnych, różnych wersjach tych systemów, przy różnym zestawie oprogramowania, jakie może mieć odbiorca
  - *testy wersji alfa i beta* – służą głównie zdobyciu informacji zwrotnej od użytkowników – wybranej grupie przekazujemy wstępne wersje produktu, następnie zbieramy ich opinie i komentarze dotyczące działania produktu
  - ...

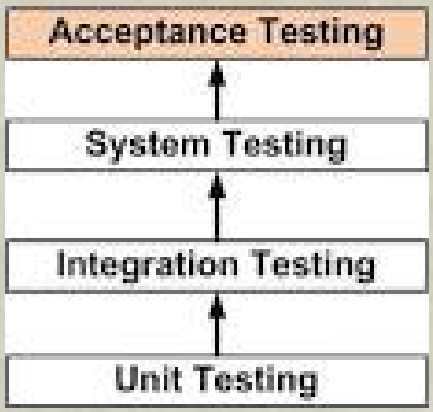
Testowanie – rodzaje testów

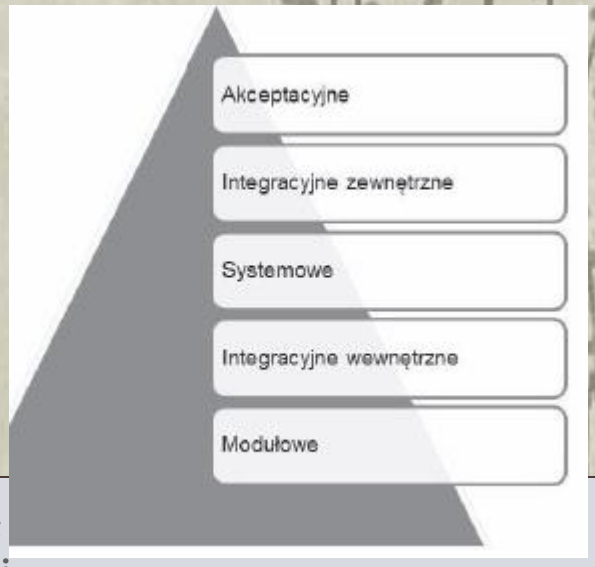
- z punktu widzenia *przeznaczenia* testy można podzielić na:
  - ...
  - *testy używalności* – tzw. *usability tests*, służą, by sprawdzić, jak szybko potencjalni użytkownicy mogą opanować działanie aplikacji, na ile użyteczna i jasna jest dokumentacja, itp.
  - *testy post-awaryjne* – testy służące sprawdzeniu, czy aplikacja zachowuje się poprawnie po wystąpieniu sytuacji awaryjnej
    - w pewnych przypadkach jest to bardzo ważny rodzaj testów, na przykład producent bazy danych powinien sprawdzić, na ile awaria wpłynie na integralność przechowywanych danych

Testowanie – poziomy testów

- Poziomy testów
  - Testy modułowe (*unit/component testing*)
  - Testy integracyjne (*integration testing*)
  - Testy systemowe (*system testing*)
  - Testy integracyjne zewnętrzne
  - Testy akceptacyjne (*acceptance testing*)



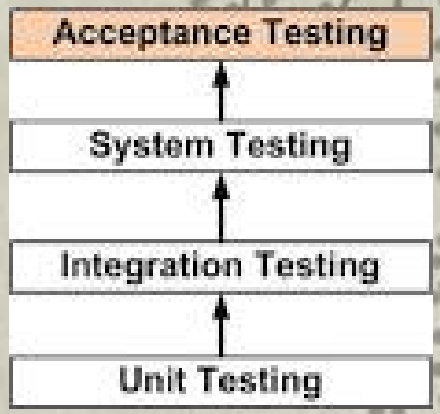




dr hab. inż. Aneta Poniszewska-Marańda
Instytut Informatyki Politechniki Łódzkiej

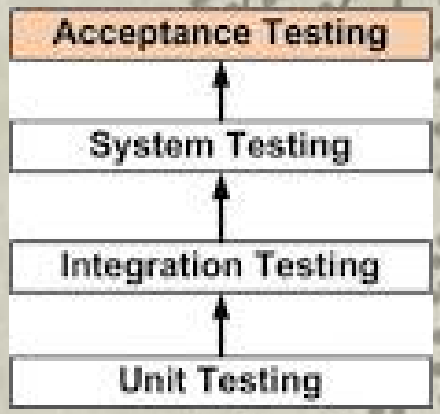
Testowanie – poziomy testów

1. Testy modułowe
   - analiza ścieżek (*path analysis*)
   - użycie klas równoważności (*equivalence partition*)
   - testowanie wartości brzegowych
   - testowanie składniowe
2. Testy integracyjne
   - pomiędzy modułami
     - funkcjonalne
     - wydajnościowe
   - pomiędzy systemami
     - funkcjonalne
     - wydajnościowe
     - regresywne



Testowanie – poziomy testów

3. Testy systemowe
   - instalacyjne
   - funkcjonalne
   - interfejsu (użyteczności)
   - wydajnościowe
   - regresywne
   - bezpieczeństwa
4. Testy akceptacyjne
   - funkcjonalne
   - wydajnościowe
   - bezpieczeństwa



Testowanie – testy jednostkowe

- testy jednostkowe (ang. *unit test), test modułowy* – metoda testowania oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania **pojedynczych elementów** (jednostek) programu
  - np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym
- testowanie na najniższym poziomie
- poszczególne metody (funkcje) testowane pojedynczo, w oderwaniu od reszty aplikacji
- celem sprawdzenia pod kątem zgodności ze zdefiniowanym typem/zakresem danych wejściowych
  - udowodnienie, że kod działa zgodnie z założeniami programisty

Testowanie – testy jednostkowe

- wykonywane na etapie wytwarzania kodu
- uczestnikami najczęściej programiści, którzy w fazie implementacji poddają weryfikacji własny kod
- wykonywanie wybranego fragmentu instrukcji w celu zweryfikowania, czy realizuje ona swoją funkcję zgodnie z założeniami
- odnosić się do małych podmiotów, a ich wynik powinien być zależny od innych elementów, które mają znaleźć się w gotowej aplikacji
- nie powinny wnikać w szczegóły procesu biznesowego
- analizie i ocenie podlega jedynie mały i wyizolowany fragment kodu
  - kiedy nabierzemy zaufania do testowanych fragmentów kodu, rozpocząć składanie ich do postaci gotowego produktu lub większego komponentu
- wykonuje się zwykle w środowisku deweloperskim z dostępem do kodu źródłowego
- wymaga to od testera umiejętności czytania i wykonywania kodu oraz pisania własnych skryptów (fragmentów kodu)

Testowanie – testy jednostkowe

- ***Dlaczego należy wykonywać testy modułowe?***
  - błędy wykryte we wczesnej fazie produkcji oprogramowania kosztują znacznie mniej niż poprawa oraz usuwanie ich skutków w kolejnych fazach wytwarzania lub użytkowania produkcyjnego aplikacji
  - wykryte problemy usuwane natychmiast – minimalizuje ryzyko propagacji negatywnego wpływu wadliwego kodu na pozostałe moduły
  - wykryte błędy nie znajdują odzwierciedlenia w postaci formalnego zgłoszenia, co przyspiesza udoskonalanie kodu i nie niesie ze sobą ryzyka krytycznych uwag osób trzecich
  - bezpieczna i efektywna forma testów własnego kodu
  - ujawnione błędy mogą mieć przyczynę głęboko ukrytą w kodzie.
  - pozwalają na wyeliminowanie typowych problemów w podstawowej ścieżce obsługi systemu
  - im mniej problemów przeniesionych z fazy implementacji do fazy formalnych testów, tym szybciej gotowy produkt zostanie przekazany do odbiorcy, a jego jakość wzrośnie

Testowanie – testy jednostkowe

- testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, rzucone wyjątki) z oczekiwanymi wynikami – **pozytywnymi i negatywnymi**
  - niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu
- frameworki dla języka programowania Java : *JUnit*, *TestNG*

Testowanie – testy jednostkowe

JUnit

- JUnit pozwala:
  - stworzyć test sprawdzający, czy wyniki działania metod spełniają wymagane zależności (*AssertTrue*, *AssertEquals* itd.)
  - organizować testy w spójne zestawy, testujące określoną część funkcji projektu lub wybrane klasy
  - uruchamiać napisane testy przy użyciu GUI i z linii poleceń

TestNG

- TestNG zapewia:
  - wykorzystanie adnotacji Java
  - elastyczną konfigurację testów
  - obsługę parametryzacji testów
  - testowanie w środowisku rozproszonym
  - rozbudowany mechanizm tworzenia zestawów testów
  - rozszerzalność za pomocą języka skryptowego *BeanShell*

Testowanie – testy jednostkowe

- cechy poprawnych testów jednostkowych
  - *automatyzacja* – uruchamianie testów musi być łatwe
  - *kompletność* – należy testować wszystko, co może zawieść
  - *powtarzalność* – wielokrotne wykonanie testu daje te same wyniki
  - *niezależność* – od środowiska i od innych testów
  - *profesjonalizm* – kod testujący jest tak samo ważny, jak kod dostarczany klientowi

Testowanie – testy jednostkowe

Automatyzacja

- testy jednostkowe muszą być wykonywane w automatyczny sposób
- automatyzacja dotyczy uruchamiania testów i sprawdzania ich wyników
- testy 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

dr hab. inż. Aneta Poniszewska-Marańda
Instytut Informatyki Politechniki Łódzkiej

Testowanie – testy jednostkowe

Powtarzalność

- testy powinny być niezależne nie tylko od siebie, ale również od środowiska – wielokrotne wykonywanie testów, nawet w rożnej 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 funkcję
- przeprowadzenie testu nie może zależeć od wyniku innego testu

Testowanie – testy jednostkowe

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

Testowanie – testy jednostkowe

Co testować?

1. Czy wyniki są poprawne (klasy ekwiwalencji)?
2. Czy warunki brzegowe zostały prawidłowo określone?
3. Czy można sprawdzić relacje zachodzące w odwrotnym kierunku?
4. Czy można sprawdzić wyniki w alternatywny sposób?
5. Czy można wymusić błędy?
6. Czy efektywność jest zadowalająca?

Testowanie – testy jednostkowe

Poprawność wyników

- wyniki działania kodu znajdują się często w specyfikacji wymagań
- w przypadku braku dokładnej specyfikacji – założenie własnych wymagań odnośnie wyników metod
- weryfikacja założeń we współpracy z użytkownikami

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

Testowanie – testy jednostkowe

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
- 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

Testowanie – testy jednostkowe

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

Testowanie – testy jednostkowe

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

Testowanie – testy jednostkowe, pułapki

Pułapki testowania

1. Ignorowanie testów jednostkowych, gdy kod działa
2. „Testy ognia”
3. Program działa na komputerze programisty
4. Problemy arytmetyki zmiennoprzecinkowej
5. Testy zajmują zbyt wiele czasu
6. Testy ciągle zawodzą
7. Testy zawodzą na niektórych maszynach

Testowanie – testy jednostkowe, pułapki

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

dr hab. inż. Aneta Poniszewska-Marańda
Instytut Informatyki Politechniki Łódzkiej

Testowanie – testy jednostkowe, pułapki

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

Testowanie – testy jednostkowe, pułapki

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

Testowanie – testy jednostkowe, pułapki

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

Testowanie – testy integracyjne

- testy integracyjne to testy sprawdzające, czy przetestowane w ramach testów jednostkowych komponenty (klasy, metody) dobrze ze sobą współpracują
- wykonywane w celu wykrycia błędów w interfejsach i interakcjach pomiędzy modułami/komponentami:
  - niekompatybilne interfejsy komponentów
  - komponent wysyła dane o niepoprawnej syntaktyce
  - komponenty na różny sposób interpretują dane, które między sobą przesyłają
  - niespełnione ograniczenia czasowe nałożone na komunikację pomiędzy komponentami

Testowanie – testy integracyjne

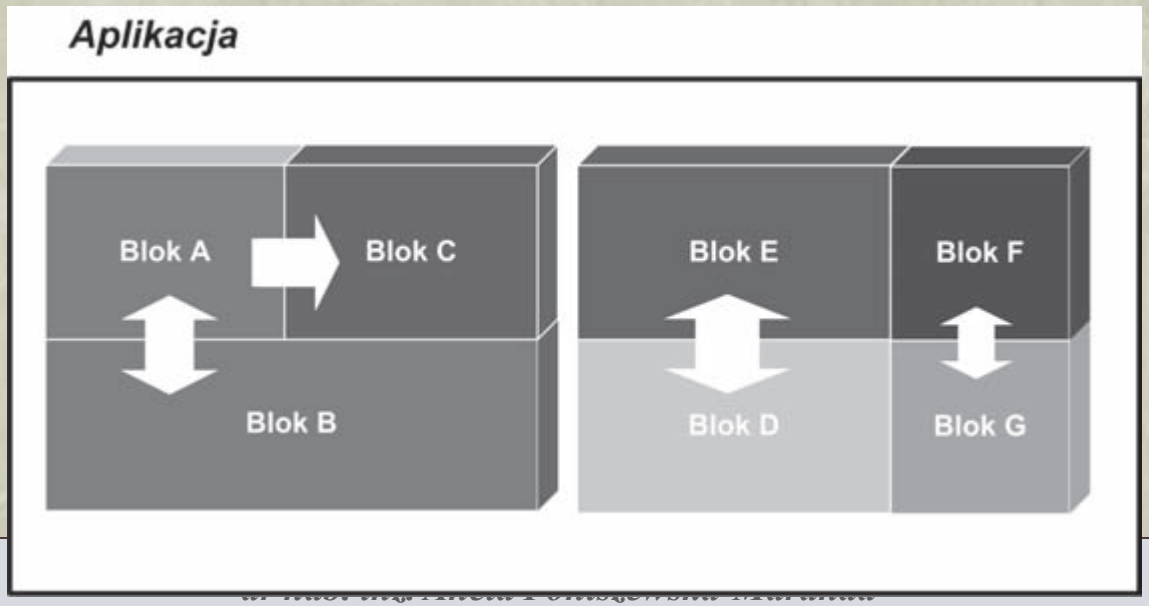
- opieranie jednego procesu biznesowego na wielu różnych systemach, podsystemach i aplikacjach
- przeprowadzenie pełnego testu biznesu w oparciu o scalone środowisko
- zweryfikować interakcję pomiędzy poszczególnymi modułami
- spójność systemu otwiera drogę do pełnych testów funkcjonalnych w całościowym ujęciu obsługi biznesu
- zbadanie współpracy i wzajemnego oddziaływania dwóch lub więcej modułów systemu

Testowanie – testy integracyjne

- testy integracyjne mają wykazać:
  - czy moduły poprawnie współpracują, czy nie wystąpiły przeszkody natury technologicznej oraz czy wzajemnie świadczone usługi spełniają oczekiwania (logika)
  - jak zachowują się poszczególne elementy w sytuacji awarii, błędów lub niestabilnej pracy w przypadku dysfunkcji jednego z nich (wzajemne oddziaływanie)
  - czy kojarzone wzajemnie elementy realizują założony proces biznesowy (logika biznesu)
  - czy infrastruktura techniczna zapewnia optymalne warunki pracy dla skomplikowanego systemu (wielomodułowego)
  - czy nie ma luk w logice biznesu, tj. czy nie ujawniły się problemy i/lub potrzeby, które nie zostały przewidziane na etapie projektowania rozwiązania

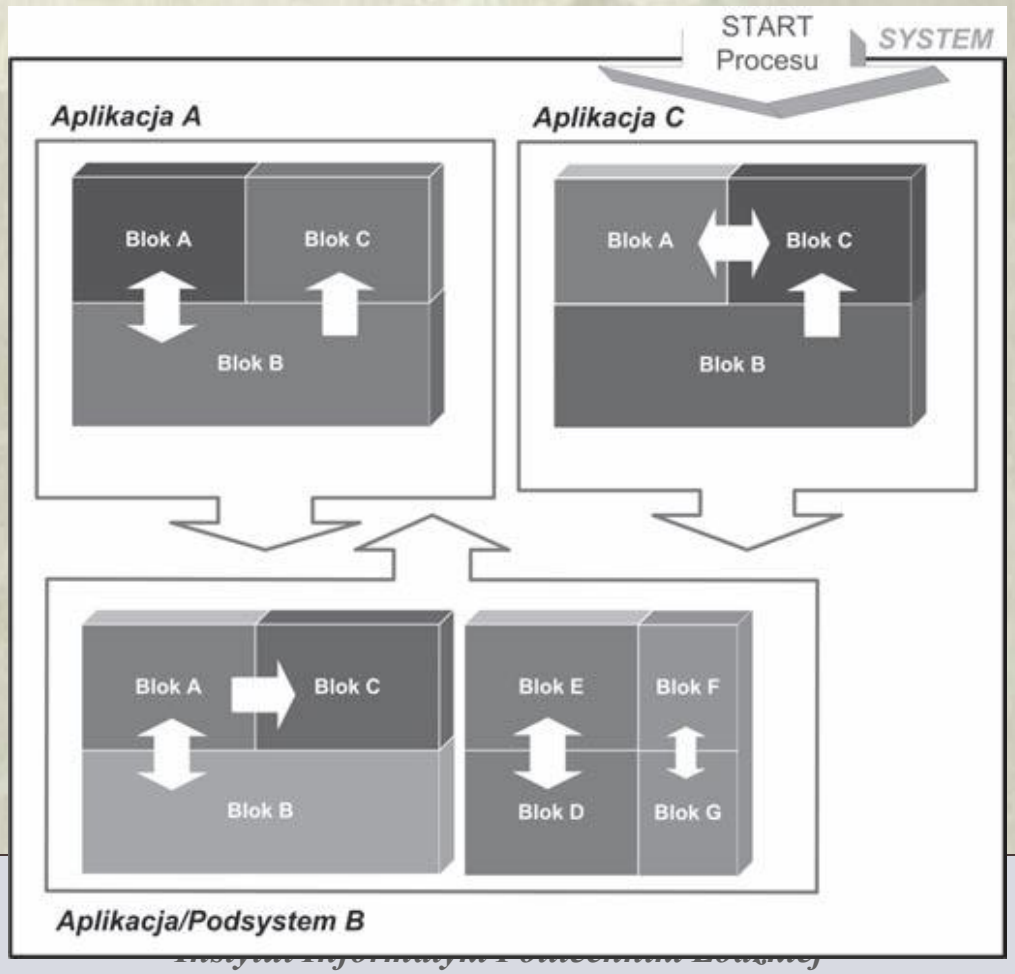
Testowanie – testy integracyjne

- prace integracyjne rozpoczynają się już w momencie łączenia kodu dwóch lub więcej modułów tej samej aplikacji (integracja wewnętrzna)
- komponenty mogą być przygotowane przez zupełnie niezależnych programistów, a finalnie podlegają integracji
- prace podejmować jak najbliżej kodu, na etapie prac programistycznych i testów wewnętrznych
- *schemat wzajemnego oddziaływania bloków jednej aplikacji*



Testowanie – testy integracyjne

- kolejnym momentem w procesie produkcji oprogramowania, w którym wykonywane testy integracyjne, jest zestawianie odrębnych modułów na etapie weryfikacji funkcjonalnej
- *schemat interakcji trzech niezależnych aplikacji*



Testowanie – testy integracyjne

- podejście do testów integracyjnych:
  - *Top-down*
  - podstawowe, ogólne zintegrowane moduły są testowane, a gałęzie modułu są testowane krok po kroku do końca danego modułu
  - *Bottom-up*
  - najniższy poziom składników testowany jako pierwszy, a następnie używany do ułatwienia testowania na wyższym poziomie składników
  - proces powtarzany, dopóki składnik na szczycie hierarchii jest testowany
  - *Big bang*
  - wszystkie lub większość rozwijanych modułów jest łączonych razem do postaci pełnego systemu informatycznego lub jego znacznej części, a następnie używane do testowania integracyjnego
  - jeżeli przypadki testowe nie są poprawie połączone, integracja procesu będzie bardziej skomplikowana i może uniemożliwić osiągnięcie celu

Testowanie – testy systemowe

- testy systemowe wykonywane po zakończeniu integracji
  - podczas ich przeprowadzania testowany system powinien być uruchomiony w środowisku możliwie bliskiemu docelowemu
  - celem skonfrontowanie działania systemu ze stawianymi przed nim wymaganiami funkcjonalnymi
- podczas testów systemowych cały system weryfikowany pod kątem zgodności z:
  - wymaganiami funkcjonalnymi
  - wymaganiami niefunkcjonalnymi (wydajność, użyteczność, niezawodność)
- system jest testowany **całościowo** z użyciem *technik czarnej skrzynki*
- wiedza o kodzie lub wewnętrznej strukturze aplikacji nie jest wymagana

Testowanie – testy systemowe

- przedmiotem testów cała aplikacja lub jej samodzielny fragment, który znajduje odwzorowanie w projekcie, tj. wchodzi w zakres projektu
- formą testów funkcjonalnych jest zastosowanie techniki czarnoskrzynkowej
- technika białej skrzynki jako uzupełnienie głównego wątku testów
- testy w środowisku jak najbardziej zbliżonym do produkcyjnych warunków funkcjonowania aplikacji
  - weryfikacja aplikacji w warunkach możliwie wiernie odwzorowujących docelowe środowisko pracy zmniejsza ryzyko przeoczenia błędów i problemów, które mogą wynikać z różnic w specyfice obu środowisk
- do wykonywania testów najlepiej niezależny zespół testerów – zachowuj dużą autonomiczność wobec zespołu programistycznego w aspekcie środowiska testów oraz zasobów ludzkich

Testowanie – testy systemowe

- testy systemowe ujmować wymagania funkcjonalne i niefunkcjonalne
- faza, w której ocenia się globalnie produkt bez nadmiernego nacisku na zgłębianie wewnętrznej architektury i instrukcji kodu aplikacji
- odnoszą się do aplikacji w ujęciu całościowym
- weryfikacja wszystkich funkcji wraz z analizą korelacji pomiędzy nimi
- dobry kandydat do automatyzacji testów
- testy systemowe to
  - testy funkcjonalne
  - testy wydajnościowe
  - testy regresywne
  - testy ergonomii
  - testy instalacji
  - testy bezpieczeństwa

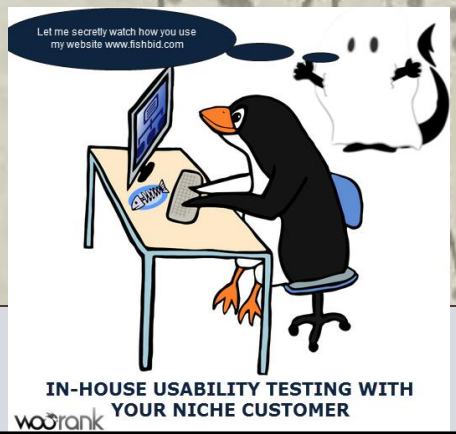
Testowanie – testy akceptacyjne

- testy akceptacyjne skoncentrowane na punkcie widzenia klienta
  - celem podjęcie decyzji, czy system (lub jego poszczególne funkcje) nadają się do wdrożenia
  - w podejściu formalnym w ramach tych testów weryfikuje się, czy wypełniony został kontrakt
  - konieczne jest zaangażowanie klienta zamawiającego system
  - w metodykach zwinnych testy akceptacyjne mogą zastępować specyfikację wymagań
- walidacja systemu pod kątem *zgodności z wymaganiami klienta*, który *w swoim środowisku* wykonuje przypadki testowe przy udziale przedstawicieli projektu
- produkcyjne testy akceptacyjne
- testowanie akceptacyjne w środowisku użytkownika

dr hab. inż. Aneta Poniszewska-Marańda
Instytut Informatyki Politechniki Łódzkiej

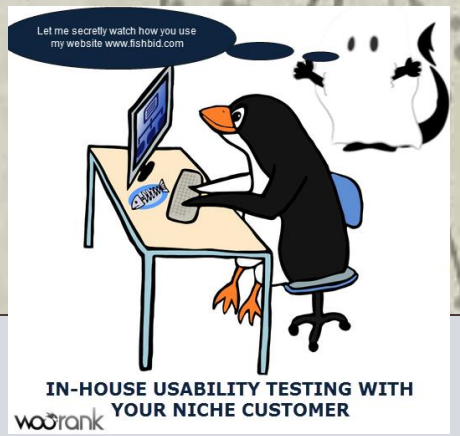
Testowanie – testy akceptacyjne

- testy odbiorcze wykonywane bezpośrednio przez zamawiającego w oparciu o własne zasoby lub poprzez zlecenie prac niezależnemu zespołowi testerskiemu
- testy mają potwierdzić zgodność weryfikowanego produktu z zapisami w umowie i obowiązującymi przepisami prawa
- celem testów akceptacyjnych jest weryfikacja i potwierdzenie, czy wszystkie zapisy w kontrakcie zostały zrealizowane w sposób zaspokajający oczekiwania
- pozytywne zamknięcie fazy testów odbiorczych stanowi podstawę do finansowego rozliczenia kontraktu
  - odnosi się do formalnie spisanych kryteriów odbioru oprogramowania, uzgodnionych na etapie negocjacji kontraktu



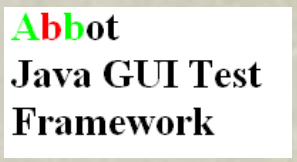
Testowanie – testy akceptacyjne

- dla aplikacji „pudełkowych” dwa typy testów akceptacyjnych: *alfa i beta*
- *testy alfa* wykonywane wewnątrz organizacji, która wyprodukowała oprogramowanie, ale weryfikacji dokonuje niezależny zespół, czyli zespół, który nie brał udziału w procesie wytwarzania
- *testy beta* realizowane poza organizacją wykonującą kod przez grupę użytkowników docelowych
- uzyskanie informacji zwrotnej (potwierdzenie) o wysokiej jakości własnego produktu przed oficjalnym wprowadzeniem go na rynek
- również testy akceptacyjne w aspekcie obowiązujących przepisów prawa



Testowanie – testy akceptacyjne

- *testy akceptacyjne* to testy funkcjonalne, których celem jest wykazanie, że wyspecyfikowane wymagania zostały poprawnie zaimplementowane
- w metodykach lekkich (np. XP) często stanowią integralną część specyfikacji i są automatyzowane przy pomocy jednego z wielu dostępnych narzędzi (*Fitnesse*, *Fit*, *Selenium*, *BadBoy*, *Proven*, *Abbot*, *jfcUnit*, *AutoIt*)
- kiedy wszystkie testy akceptacyjne przypisane do historii użytkownika (przypadku użycia) zostaną poprawnie przeprowadzone, historia jest uważana za poprawnie zaimplementowaną



Testowanie – testy

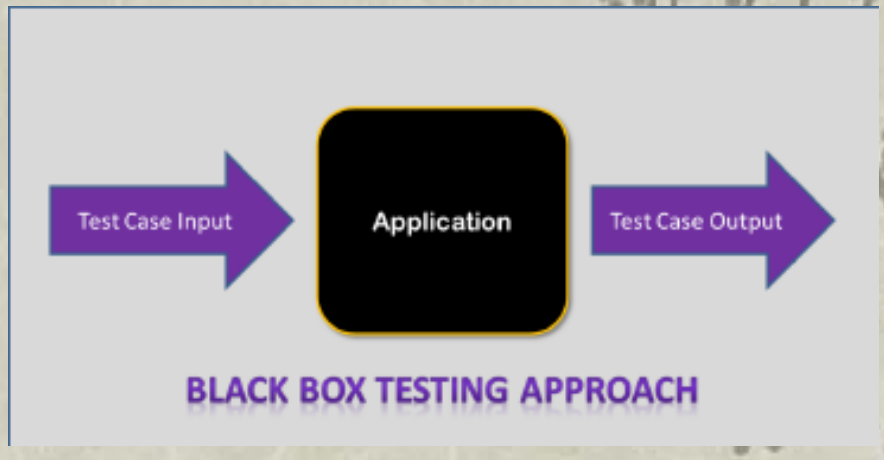
Testy akceptacyjne a jednostkowe

Filippo Ricca: Automatic Acceptance Testing with FIT/FitNesse

Testy akceptacyjne	Testy jednostkowe
Przygotowywane przez klienta i analityka systemowego	Przygotowywane przez programistów
Kiedy żaden z testów nie zawodzi przestań programować – system jest gotowy (XP)	Kiedy żaden z testów nie zawodzi napisz nowy test, który zawiedzie (XP, TDD)
Celem jest wykazanie poprawności działania wyspecyfikowanej funkcji	Celem jest znajdowanie błędów
Używane do weryfikowania kompletności implementacji; jako testy integracyjne i regresyjne; do wskazywania postępu w tworzeniu aplikacji; jako część kontraktu; jako dokumentacja wysokiego poziomu. Używane do znajdowania błędów w modułach (klasach, funkcjach, metodach, komponentach) kodu źródłowego; jako dokumentacja niskiego poziomu	Używane do znajdowania błędów w modułach (klasach, funkcjach, metodach, komponentach) kodu źródłowego; jako dokumentacja niskiego poziomu
Pisane przed implementacją, a wykonywane po niej	Pisane i wykonywane w trakcie implementacji
Wyzwalane przez wymaganie użytkownika (przypadek użycia, historia użytkownika...)	Wyzwalane przez potrzebę dodania nowych metod, klas

Testy funkcjonalne

- Testy czarnoskrzynkowe
  - Functional Testing
  - Stress Testing
  - Load Testing
  - Ad-hoc Testing
  - Exploratory Testing
  - Usability Testing
  - Performance Testing
  - Smoke Testing
  - Recovery Testing
  - Volume Testing
  - Domain Testing
  - Scenario testing
  - Regression Testing
  - User Acceptance
  - Alpha and Beta Testing



Testy funkcjonalne

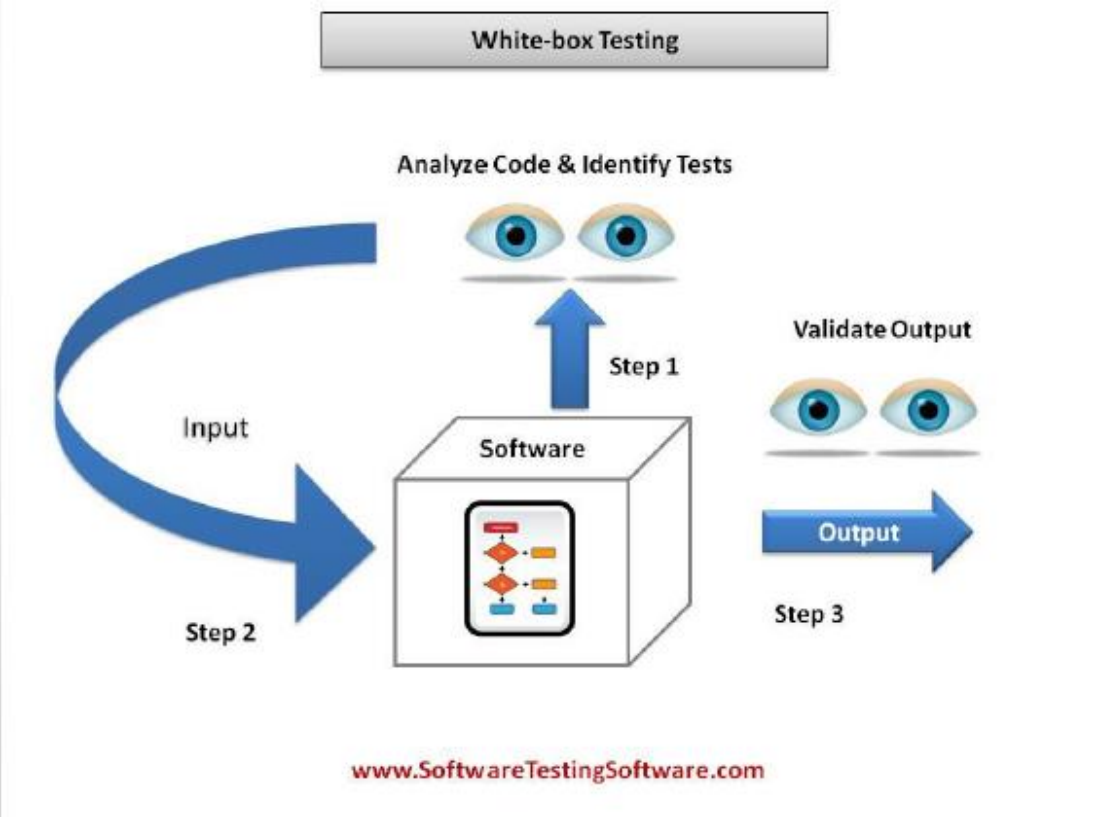
- koncentrują się na wymaganiach funkcjonalnych stawianych tworzonemu oprogramowaniu
  - pozwalają na sprawdzenie zgodności programu z wymaganiami użytkownika
- stosuje się je najczęściej pod koniec testowania systemu
- wykrycie pominiętych lub niepoprawnie zaimplementowanych funkcji ze specyfikacji 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 zaprojektowanie przypadków testowych, sprawdzających wartości graniczne i unikatowe związane z architekturą komputera, na której będzie uruchamiany testowany program, z językiem implementacji i realizowanym algorytmem

Testy funkcjonalne

- zalety testowania metodą czarnej skrzynki
  - testy są powtarzalne
  - testowane jest środowisko, w którym przeprowadzane są testy
  - zainwestowany wysiłek może być użyty wielokrotnie
- wady testowania metodą czarnej skrzynki
  - wyniki testów mogą być szacowane nazbyt optymistycznie
  - nie wszystkie właściwości systemu mogą zostać przetestowane
  - przyczyna błędu nie jest znana

Testy strukturalne

- Testy białoskrzynkowe
  - Unit Testing
  - Static & dynamic Analysis
  - Statement Coverage
  - Branch Coverage
  - Security Testing
  - Mutation Testing



Testy strukturalne

- wgląd do kodu źródłowego
- systematyczne sprawdzanie elementów tego kodu
  - w sposób statyczny – statyczna analiza kodu, znajdująca źródła potencjalnych problemów w programie, zwana analizą strukturalną
  - lub w sposób dynamiczny – z wykonaniem programu
- formalna analiza kodu
- 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
  - tworzenie raportów – podsumowanie wyników przeglądu

Testy strukturalne

Analiza pokrycia kodu

- analiza dynamiczna w testowaniu metodą białej skrzynki
- pozwala 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
- umożliwia znalezienie wielu bledów – 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

Testy strukturalne

Analiza pokrycia kodu

- programy śledzące – umożliwiają wgląd, w jaki sposób wykonywane kolejne wiersze kodu podczas przetwarzania danych testowych
- analizatory pokrycia kodu pozwalają na sporządzenie szczegółowych statystyk wykonania testowanego programu
  - uzyskanie informacji, które części kodu nie zostały pokryte przez zastosowane testy
- pokrycie kodu wykonywane jest przez:
  - analizę pokrycia instrukcji, zwaną 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

Testy strukturalne

- zalety testowania metodą białej skrzynki
  - wymagana jest znajomość struktury kodu – łatwo określić, jaki typ danych wejściowych/wyjściowych jest potrzebny, aby efektywnie przetestować aplikację
  - pomaga też zoptymalizować kod aplikacji
  - pozwala dokładnie określić przyczynę i miejsce, w którym znajduje się błąd
- wady testowania metodą białej skrzynki
  - wymagana jest znajomość struktury kodu – do przeprowadzenia testów potrzebny jest tester ze znajomością programowania, co podnosi koszty
  - prawie niemożliwym przeglądniecie każdej linii kodu w poszukiwaniu ukrytych błędów, co może powodować błędy po fazie testów