Wykłady 8-9

1. Faza implementacji

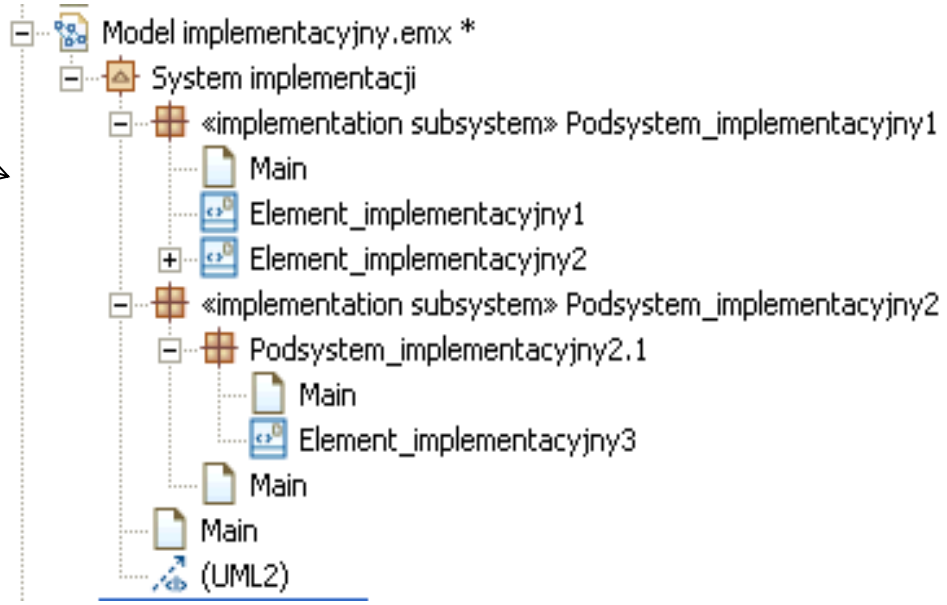
2. Jakość w procesie wytwarzania oprogramowania

- Podstawowe pojęcia i definicje związane z jakością
- Testowanie dynamiczne i statyczne
- Metryki jakości oprogramowania

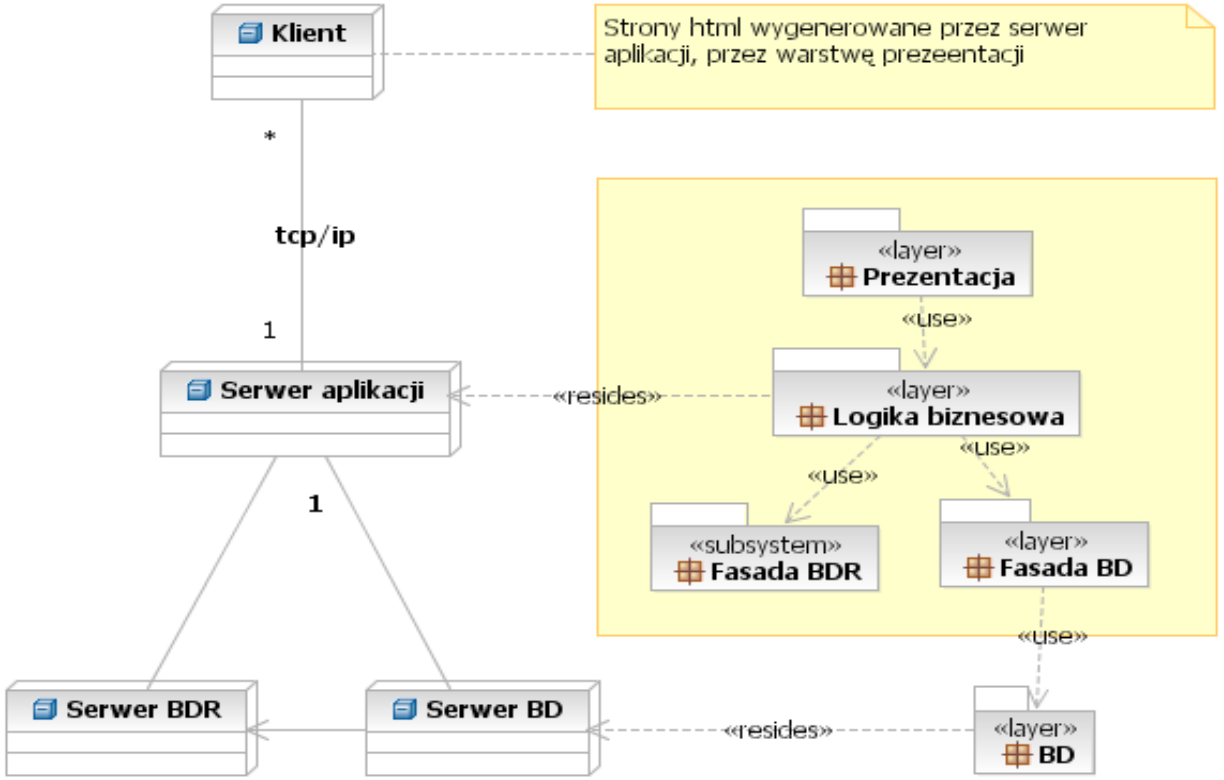
Implementacja -

Faza implementacji → proces przekształcania specyfikacji systemu w działający system (na postawie projektu); jest to *kodowanie* projektu do postaci wymaganej przez wybrane środowisko i język programowania.

Struktura modelu implementacyjnego



MODEL PROJEKTOWY vs. DIAGRAMY IMPLEMENTACYJNE *(przykład)*



PLAN INTEGRACJI *Przyrostu*

Przyrost (ang. build) - wykonywalna wersja systemu lub jego części; efekt iteracji

PLAN INTEGRACJI Przyrostu opisuje sekwencję przyrostów kodu, które mają być zbudowane w kolejnych iteracjach

PLAN INTEGRACJI Przyrostu zawiera dla każdego przyrostu:

- funkcjonalność, która ma zostać zaimplementowana w przyroście (lista przypadków użycia lub ich części, lub/i scenariuszy; lista może dotyczyć wymagań dodatkowych)

- wskazanie części modeli implementacyjnych (podsystemów, artefaktów, komponentów) składających się na przyrost

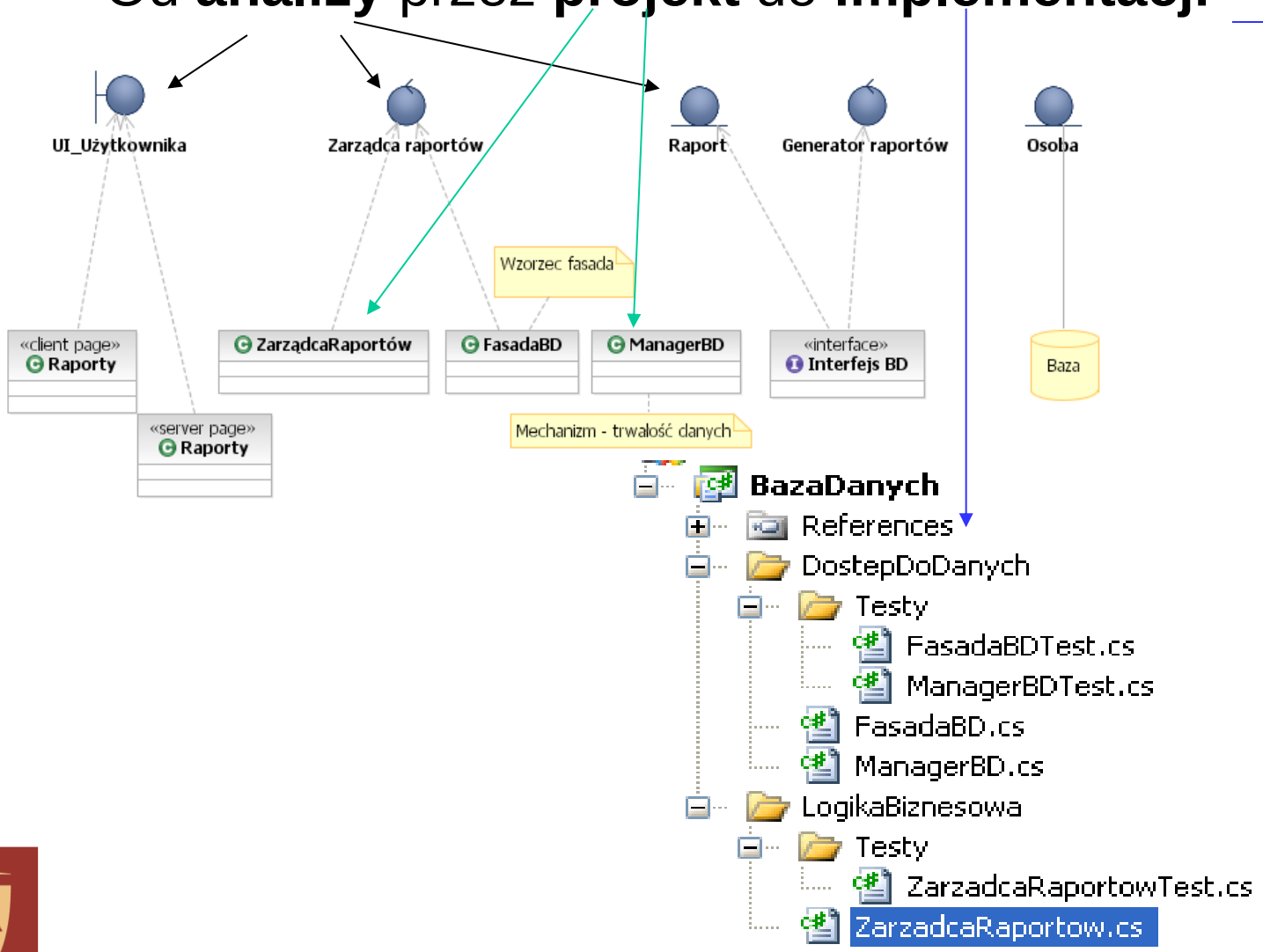
STRATEGIA DEKOMPOZYCJI NA Przyrosty:

- kolejny przyrost powinien dodawać funkcjonalność w stosunku do poprzednich buildów

- nie należy wprowadzać zbyt wielu nowych elementów do przyrostu (trudności w integracji i testowaniu)

- początkowe przyrosty implementują niższe warstwy aplikacji, późniejsze - warstw bliższych warstwie aplikacji

Od **analizy** przez **projekt** do **implementacji**



Jakość oprogramowania

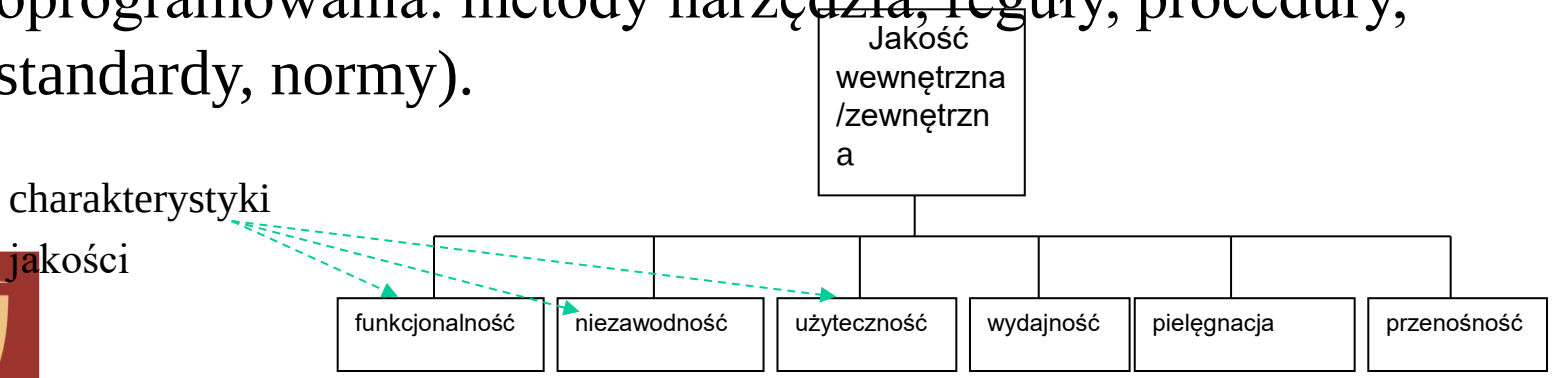
ogół właściwości produktu wiążących się z jego zdolnością do zaspokojenia potrzeb stwierdzonych i oczekiwanych

[wg ISO 9126]

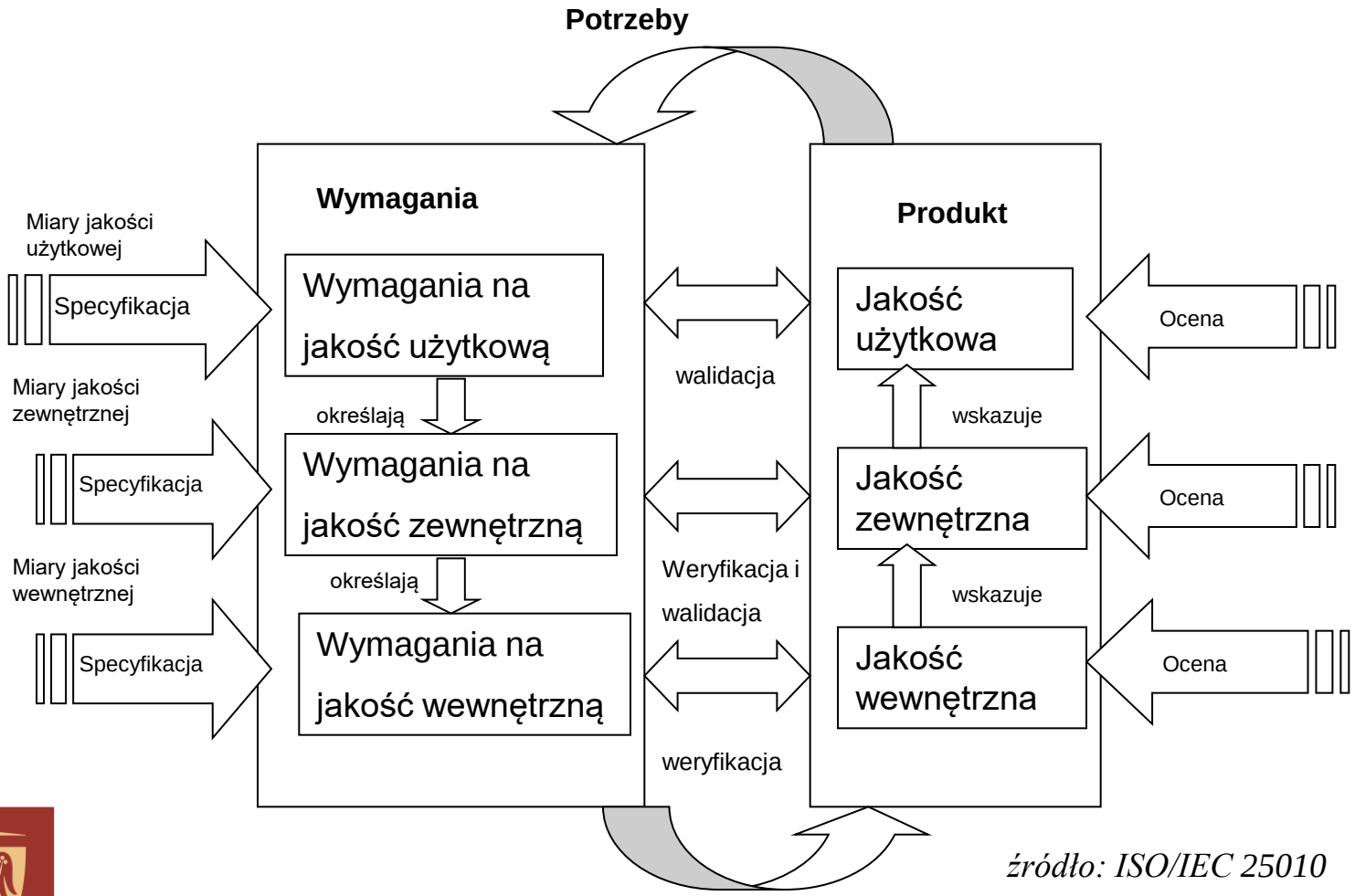
Wyróżnia się:

- **jakość zewnętrzną** (punkt widzenia klienta: własności dotyczące oprogramowania lub jego oferty),

- **jakość wewnętrzną** (punkt widzenia dostawcy oprogramowania: metody narzędzia, reguły, procedury, standardy, normy).

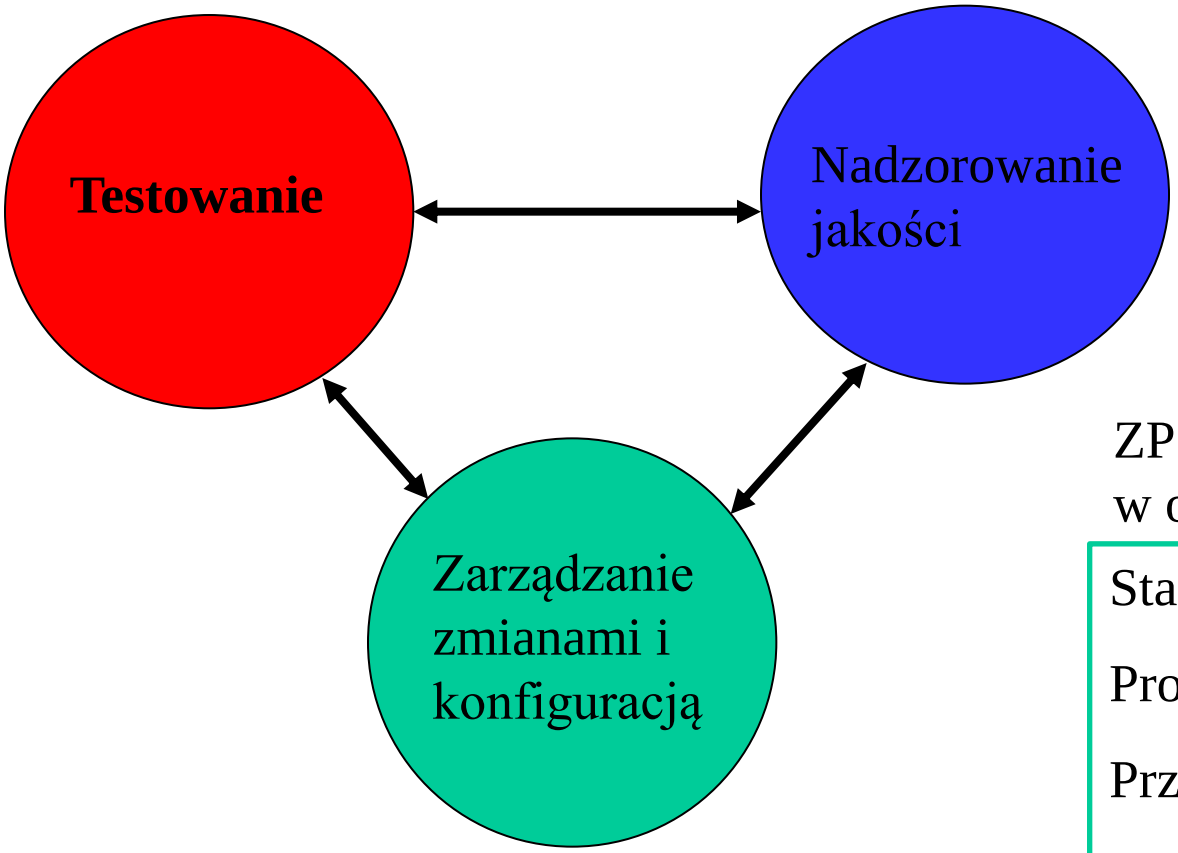


Proces formułowania i implementacji wymagań na jakość



źródło: ISO/IEC 25010

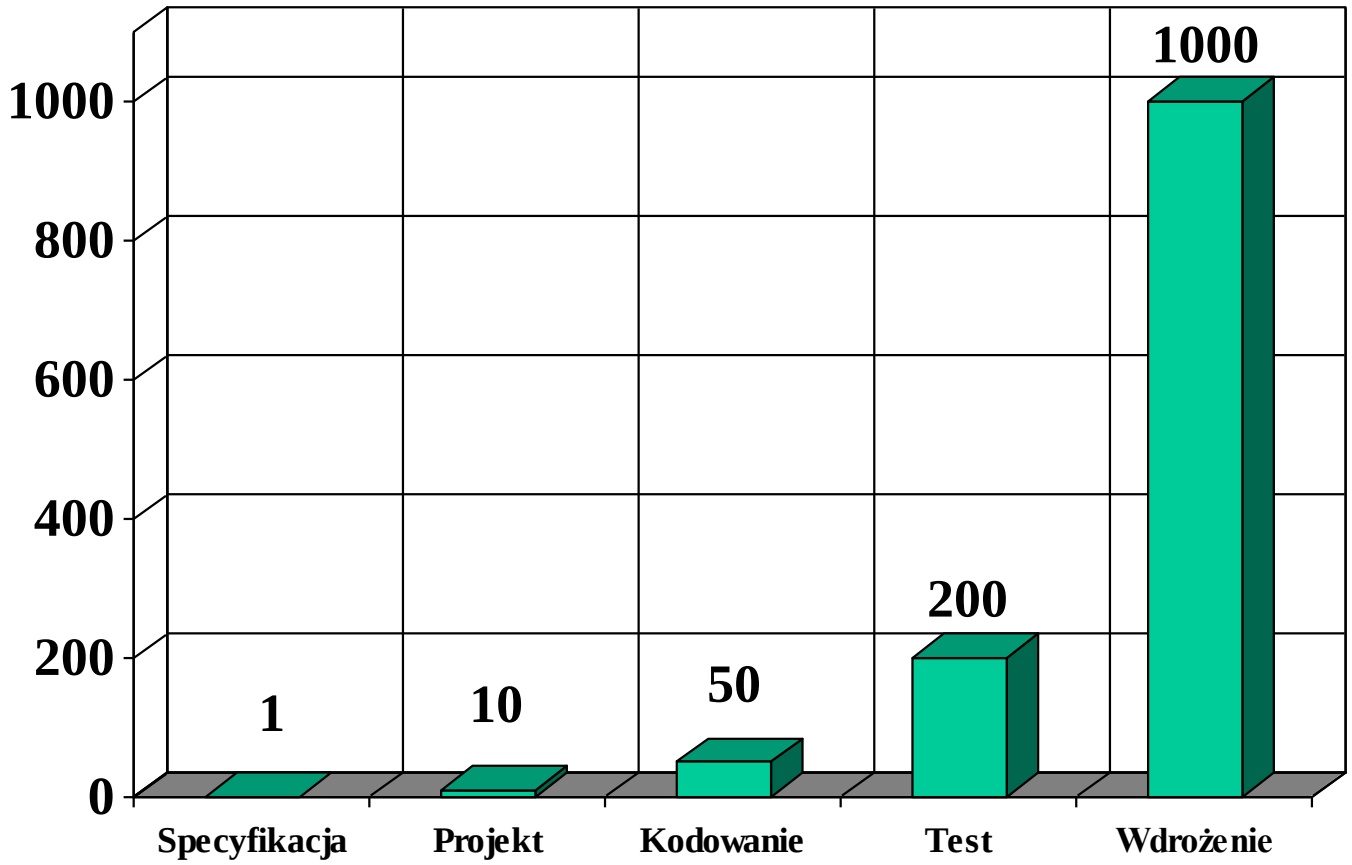
Zapewnienie jakości (ZP)– 3 komponenty



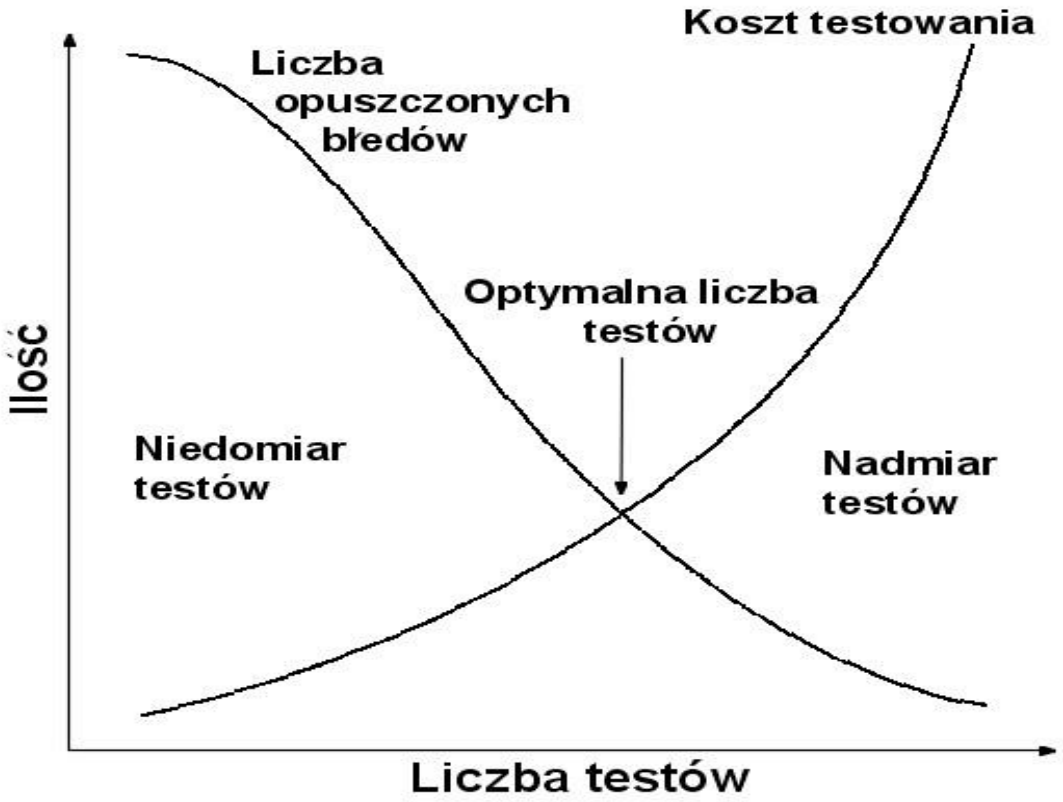
ZP realizowane w oparciu o

- Standardy
- Procedury
- Przewodniki
- Wymagania

Za podsumowanie: koszt usuwania błędów



Testowanie oprogramowania – koszty vs efektywność



Procesy badania jakości - definicje

TESTOWANIE to wykonywanie programu w celu i z intencją znalezienia błędu (na podstawie błędnego wykonywania się programu - ang. fault)

WERYFIKACJA - wykazanie w warunkach „sztucznych” zgodności wytworzonego programu/systemu ze specyfikacją.

WALIDACJA (zatwierdzanie) - wykazanie w warunkach „naturalnych”(np. w środowisku klienta) zgodności wytworzonego programu/systemu ze specyfikacją.

CERTYFIKACJA - autorytatywne potwierdzenie zgodności wytworzonego programu/systemu ze specyfikacją, która jest znormalizowana (np. w postaci normy ISO, ANSI, IEC).

Testowanie - definicje

1. **Testowanie** - ~~proces potwierdzający poprawność programu~~, ~~polegający za wykazaniu, że nie ma on błędów~~
2. **Testowanie** - proces nabywania zaufania, że system wykonuje to, czego się od niego oczekuje. *(Hetzel 1973)*
3. **Testowanie -** proces wykonywania programu z intencją znalezienia błędu. *(Myers 1979)*
4. **Testowanie - jakakolwiek działalność mająca na celu ocenę (obliczenie, oszacowanie) atrybutów lub możliwości systemu i określenie, czy są one zgodne z oczekiwanymi wynikami**. *(Hetzel 1983)*
5. **Testowanie** - mierzenie jakości oprogramowania**.**

Testowanie – definicje pojęć

BŁĘDY są wrodzoną własnością programów

Błąd oprogramowania (ang. fault) - nie są spełnione sensowne oczekiwania użytkownika/klienta; stan rzeczy odbiega od przyjętych założeń/zasad.

Awaria (ang. failure) – efekt 'wykonania' błędu

Przypadek testowy

- specyfikacja sposobu testowania systemu, łącznie z określeniem co testować i pod jakim warunkiem można testować

- wykorzystuje się zbiór par

<dane wejściowe(testy), oczekiwane wyniki>

Procedura testowa – kroki opisujące kolejność wykonywania przypadku testowego

Skrypt testowy – zautomatyzowana procedura testowa

TESTOWANIE – klasyfikacja błędów

Klasyfikacja błędów w systemach informatycznych:

-**wg przyczyny**: uszkodzenie sprzętu, błąd danych, nieadekwatność oprogramowania

- **wg czasu trwania**: trwały, chwilowy
- **wg skutków dla użytkownika**: krytyczny, niekrytyczny
- **wg miejsca powstania** : wewnętrzny, zewnętrzny

TESTOWANIE - kategorie

wg kryterium: faza cyklu życia oprogramowania:

- testowanie jednostki (procedury,modułu, programu)
- testowanie integracyjne
- testowanie systemowe
- testowanie akceptacyjne

- wg kryterium 'wymagania niefunkcjonalne':

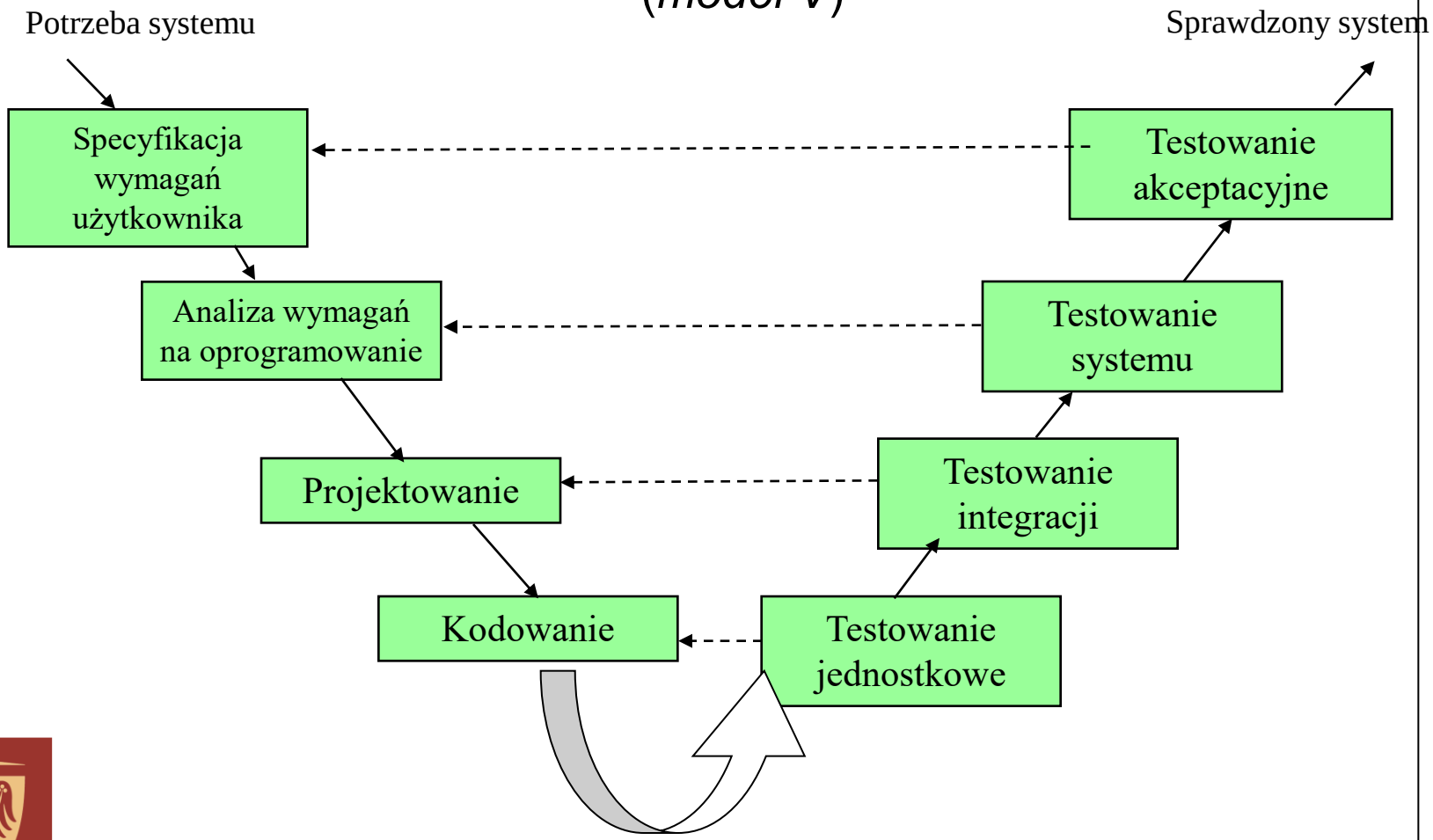
- testowanie instalacyjne
- testowanie wydajnościowe
- testowanie niezawodności

- wg kryterium : technika testowania:

- testowanie statyczne (dotyczy artefaktów 'niewykonywalnych'; inspekcja kodu/projektu)
- testowanie dynamiczne (wykonywanie programu dla danych testowych)

TESTOWANIE a fazy wytwarzania (1)

(*model V*)



TESTOWANIE a fazy wytwarzania (2)

Testy integracyjne dotyczą łączenia modułow/podsystemów (powstawanie kolejnych przyrostów/„build”ów).

Testy systemowe - testowanie powstałej edycji („release”) oprogramowania; wykorzystuje się scenariusze z przypadków użycia (realizacja projektowa) - **weryfikacja wymagań co do systemu.**

Testy akceptacyjne dotyczą zainstalowanego produktu i są generowane na podstawie przypadków użycia.

Poprawne wykonanie każdego z tych testów - rozumiane jako spełnienie warunku (-ów) *post* przypadku użycia - oznacza osiągnięcie celu tego przypadku i jest **walidacją wymagania użytkownika.**

TESTOWANIE – rodzaje testów systemowych

Testowanie systemowe - zalecane typy testów systemowych

Test	*Czynność*	*Najczęściej stosuje się do systemów*
użyteczności	sprawdzenie funkcji systemu	systemy o rozbudowanej funkcjonaln.
wydajności	pomiar parametrów, wyznaczenie charakterystyk	systemy o narzuconych wymaganiach wydajnościowych
obciążenia	praca systemu w ekstremalnych warunkach (dużo danych, wielu użytkowników)	systemy sieciowe, wielodostępne
pamięci	pomiary zapotrzebowania na pamięć	
konfiguracji	próba pracy w różnych konfiguracjach	
instalacji	testowanie procedur instalacyjnych	systemy o skomplikowanej instalacji
poufności	próba włamania	systemy z poufnymi danymi

Testowanie akceptacyjne

- akceptacja fazowa - prototypy, kolejne wersje produktu
- akceptacja ostateczna

Rodzaje testowania

Testowanie można klasyfikować z różnych punktów widzenia:

- **Wykrywanie błędów**, czyli testowanie, którego głównym celem jest **wykrycie** jak największej liczby **błędów** w programie

- **Testowanie statystyczne**, których celem jest **wykrycie przyczyn** najczęstszych błędnych wykonań oraz ocena niezawodności systemu.

Z punktu widzenia techniki wykonywania testów można je podzielić na:

- **Testowanie dynamiczne**, które polega na wykonywaniu (fragmentów) programu i porównywaniu uzyskanych wyników z wynikami poprawnymi.

- **Testowanie statyczne**, oparte na analizie kodu

Wykrywanie błędów

Dynamiczne testowanie zorientowane na wykrywanie błędów dzieli się na:

Testowanie funkcjonalne (*czarnej skrzynki*), które zakłada znajomość jedynie wymagań wobec testowanej funkcji.
System jest traktowany jako czarna skrzynka, która w nieznany sposób realizuje wykonywane funkcje. Przygotowuje się przypadki testowe i porównuje z wynikiem uzyskanym po wykonaniu.

Testowanie powinny wykonywać osoby, które nie były zaangażowane w realizację testowanych fragmentów systemu (**z wyjątkiem testów jednostkowych!)**
Testowanie strukturalne (*szklanej skrzynki*), które zakłada znajomość sposobu implementacji testowanych funkcji.

Testowanie funkcjonalne - założenia

Zwykle zakłada się, że jeżeli dana funkcja działa poprawnie dla **kilku** danych wejściowych, to działa także poprawnie dla **całej klasy** danych wejściowych.

Jest to wnioskowanie czysto heurystyczne. Fakt poprawnego działania w kilku przebiegach nie gwarantuje zazwyczaj, ze błędne wykonanie nie pojawi się dla innych danych z tej samej klasy.

Dlatego warto **przeprowadzać testowanie dla wartości brzegowych** np. min i max. oraz dla takich wartości, które wynikają z opisu wymagań.

TESTOWANIE jednostkowe - pojęcia

(Idealnie) **Jednostka** nie powinna wywoływać innej metody
Testowanie jednostkowe: dla metody, która wykonuje się bez wywoływania **żadnych innych metod**

Testowanie pozwala na potwierdzanie *poprawności jednostki*. Wyrażenie „potwierdzanie poprawności" jest zwykle używane w kontekście prawdziwości obliczeń.

Testowanie, które stwierdza poprawność jednostki zwykle badają jednostkę na dwa sposoby:

- Testowanie, jak jednostka zachowuje się w normalnych warunkach.
- Testowanie, jak jednostka zachowuje się w warunkach nienormalnych

TESTOWANIE jednostkowe - pojęcia

Potwierdzanie poprawności dotyczy trzech głównych kategorii (tylko drugi dotyczy obliczeń):

1. Sprawdzanie, czy **wejścia do obliczenia są prawidłowe** (czy kontrakt dotrzymany).
2. **Sprawdzanie, czy uzyskane wyniki wywołania metody są w pożądanym zakresie obliczeniowym** (aspekt obliczeniowy);
   cztery typowe zakresy badania:
   -transformacja danych
   -redukcja danych
   -zmiana stanu obiektu
   -poprawność stanu obiektu
3. Obsługa zewnętrznych błędów i ich usuwanie

TESTOWANIE jednostkowe – warunki brzegowe

Ważny aspekt testowania: **identyfikacja warunków brzegowych,** które są efektem np:

- wprowadzenia całkowicie błędnych lub niespójnych danych wejściowych, na przykład łańcucha "!*W:X\&Gi/w~>g/h#WQ@"jako nazwy pliku;
- nieprawidłowo sformatowanych danych, takich jak adres e-mail, który nie zawiera głównej domeny ("fred@foobar.");
- niewprowadzenia odpowiednich wartości (wtedy pojawiają się wartości 0, 0.0, "" lub null);
- pojawienia się wartości znacznie przekraczających oczekiwania (na przykład wiek osoby równy 10 000 lat);
- pojawienia się duplikatów na listach, które nie powinny ich zawierać;
- wystąpienia list nieuporządkowanych zamiast uporządkowanych i na odwrót; (można przekazać algorytmowi sortowania listę, która jest już posortowana, albo w odwrotnym porządku)
- zakłócenia przewidywanego porządku zdarzeń, (próba wydrukowania dokumentu przed zalogowaniem się).

TESTOWANIE jednostkowe – warunki brzegowe

Warunki brzegowe należy badać np. w przypadkach/sytuacjach:

- Zgodność — czy wartość jest zgodna z oczekiwanym formatem?
- Uporządkowanie — czy zbiór wartości jest odpowiednio
- uporządkowany?
- Zakres — czy wartość należy do przedziału oczekiwanych wartości?
- Odwołanie — czy kod odwołuje się do zewnętrznych obiektów, które są poza jego kontrolą?
- Istnienie — czy wartość istnieje (czyli jest różna od null, zera, obecna w zbiorze itd.)?
- Liczność — czy występuje dokładnie tyle wartości, ile jest oczekiwanych?
- -Czas (absolutny i względny) — czy wszystkie zdarzenia zachodzą w oczekiwanej kolejności i we właściwym czasie?

TESTOWANIE jednostkowe - rodzaje

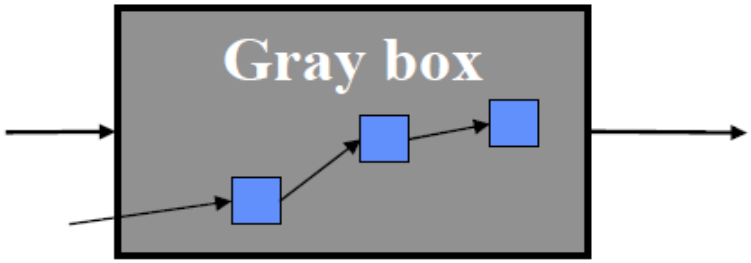
WEJŚCIA określone przez

WYNIKI

wymagania

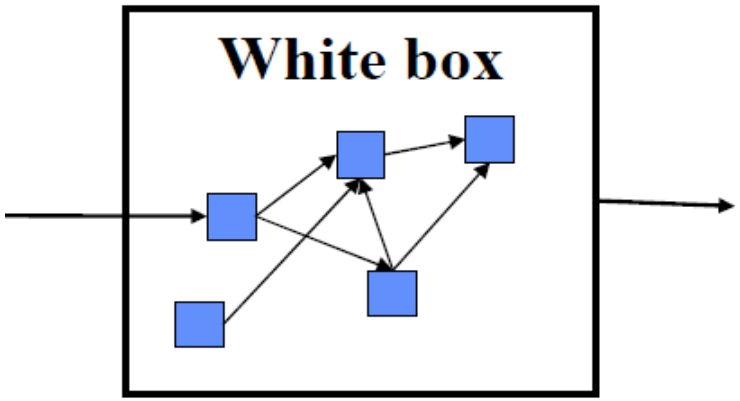
wyjście porównywane z wymaganym wynikiem

wymagania i kluczowe elementy projektowe



jak dla „black” i „white”

elementy projektowe



potwierdzone oczekiwane zachowanie

TESTOWANIE jednostkowe - wymagania

Wymagania pojawiające się podczas implementacji testów jednostkowych:

- Niepowodzenie jednego testu nie powinno przerywać procesu testowania (*assert()* z języka C kończy program)
- Raporty z wykonania testów powinny być generowane automatycznie
- Asercje powinny dawać możliwość porównywania obiektów różnych typów
- Konieczna możliwość organizacji testów w zbiory

TESTOWANIE jednostkowe – JUnit

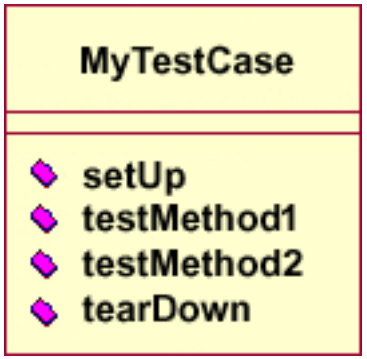
- Framework ułatwiający pisanie testów jednostkowych.
- Idea: Każdej klasie, testowanej odpowiada klasa dziedzicząca po junit.framework.TestCase o odpowiedniej nazwie
  (*Klasa pierwotna KlasaX*, *klasa testowa TestKlasaX*).
- W testowanych klasach tylko są pojedyncze testy, czyli metody; stosowane jest następujące nazewnictwo - jeśli w klasie KlasaX jest metoda add, to w klasie testowej TestKlasaX powinna być metoda testująca "public void testAdd()"
- Testowanie polega na sprawdzaniu asercji.

TESTOWANIE jednostkowe – JUnit

Asercje w metodach testujących (dostępne w Junit):

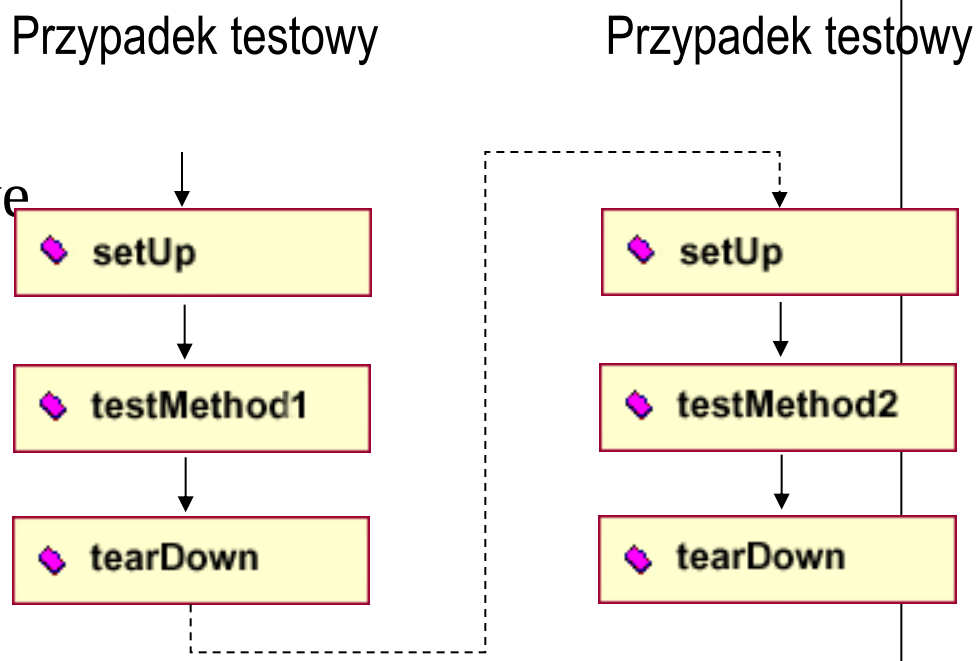
fail([String message])
assertEquals([String message], expected, actual)
assertEquals([String message], expected, actual, tolerance)
assertNull([String message], java.lang.Object object)
assertNotNull([String message], java.lang.Object object)
assertSame([String message], expected, actual)
assertNotSame([String message], expected, actual)
assertTrue([String message], boolean condition)
assertFalse([String message], boolean condition)

TESTOWANIE jednostkowe – idea Junit'a



Ogólna zasada działania JUnit'a

- Dla klasy testowanej tworzymy *„TestCase"*
- Dodajemy metody testowe rozpoczynające się frazą *„test"*
- W miarę potrzeby nadpisujemy metody *setUp()* i *tearDown()*



TESTOWANIE jednostkowe – przykład testu

```
import junit.framework.TestCase;
import junit.framework.Test;
import junit.framework.TestSuite;
public class TestCounter extends TestCase {
public TestCounter(String method)
{
super(method);
}
public void testNext()
{
Counter counter = new Counter();
counter.next();
int prev = counter.getNumber();
counter.next();
assertEquals(prev + 1, counter.getNumber());
}
public void testStopCounting()
{
Counter counter = new Counter();
int prev = counter.getNumber();
counter.stopCounting();
assertEquals(prev, counter.getNumber());
}
public static Test suite() {
TestSuite suite = new TestSuite();
suite.addTest(new TestCounter("testNext"));
suite.addTest(new TestCounter("testStopCounting"));
return suite;
}
}
```

TESTOWANIE jednostkowe – przykład testu

```
public class TestDB extends TestCase {
private Connection dbConn;
protected void setUp() {
dbConn = new Connection("oracle", 1521, "fred", "foobar");
dbConn.connect();
}
protected void tearDown() {
dbConn.disconnect();
dbConn = null;
}
public void testAccountAccess() { // Uses dbConn
xxx xxx xxxxxx xxx xxxxxxxxx;
xx xxx xxx xxxx x xx xxxx;
}
public void testEmployeeAccess() { // Uses dbConn
xxx xxx xxxxxx xxx xxxxxxxxx;
xxxx x x xx xxx xx xxxx;
}
}
```

Testowanie strukturalne

1. Kryterium pokrycia wszystkich instrukcji. Zgodnie z tym kryterium dane wejściowe należy dobierać tak, aby każda instrukcja została wykonana co najmniej raz..

2. Kryterium pokrycia instrukcji warunkowych. Dane wejściowe należy dobierać tak, aby każdy elementarny warunek instrukcji warunkowej został co najmniej raz spełniony i co najmniej raz nie spełniony.

3. Kryteria dla programów „z pętlami”

- Należy dobrać dane wejściowe tak, aby nie została wykonana żadna iteracja pętli, lub, jeżeli to niemożliwe, została wykonana minimalna liczba iteracji.
- Należy dobrać dane wejściowe tak, aby została wykonana maksymalna liczba iteracji.
- Należy dobrać dane wejściowe tak, aby została wykonana „przeciętna” liczba iteracji.

Testowanie statyczne - przeglądy

Przegląd jest procesem lub spotkaniem, podczas którego produkt roboczy/artefakt (lub zbiór produktów roboczych) jest prezentowany personelowi projektu, kierownictwu, użytkownikom, klientom lub innych zainteresowanych stron celem uzyskania komentarzy, opinii i **akceptacji.**

Przeglądy mogą być **formalne i nieformalne**.

Przejście (*walkthrough*). Wczesna, **nieformalna ocena** dokumentów, modeli, projektów i kodu. Celem jest zidentyfikowanie defektów i rozważenie możliwych rozwiązań. Wtórnym celem jest szkolenie i rozwiązanie problemów stylistycznych (np. z formą kodu, dokumentacji, interfejsów użytkownika).

Formalne przeglądy mogą być inspekcją bądź audytem

Testowanie statyczne - Inspekcja

formalna technika oceny, w której wymagania na oprogramowanie, projekt lub kod **są szczegółowo badane przez osobę lub grupę osób nie będących autorami**, w celu identyfikacji błędów, naruszenia standardów i innych problemów *[IEEE Std. 729:1983]*

Korzyści z inspekcji:

1. Wzrost produktywności od 30% do 100%
2. Skrócenie czasu projektu od 10% do 30%
3. Skrócenie kosztu i czasu wykonywania testów od 5 do 10 razy (mniej błędów, mniej testów regresyjnych)
4. 10 razy mniejsze koszty pielęgnacji (naprawczej)
5. Poprawa procesu programowego

Inspekcja jakości wymagań - przykład

Lista kontrolna jakości wymagań

1. Czy żadne wymaganie nie jest w konflikcie z innym ?
2. Czy wymaganie unika specyfikowania projektu?
3. Czy wymagania są wyrażone na spójnym, odpowiednim stopniu szczegółowości? Czy którekolwiek wymaganie powinno być podane bardziej bądź mniej szczegółowo?
4. Czy wymagania są wystarczająco jasne, by przekazać je niezależnej grupie do implementacji i będą ciągle zrozumiałe?
5. Czy każdy element jest odpowiedni do problemu i jego rozwiązania?
6. Czy każdy element może być śledzony ze swoim oryginałem w środowisku problemu?

Testowanie statyczne - Audyt

niezależny przegląd i ocena jakości oprogramowania, która zapewnia, ze osiągnięto zgodność z wymaganiami na oprogramowanie, a także ze specyfikacją, generalnymi założeniami, standardami, procedurami, instrukcjami, kodami oraz kontraktowymi i licencyjnymi wymaganiami.

Celem audytu projektu informatycznego jest dostarczenie odbiorcy i dostawcy obiektywnych, aktualnych i syntetycznych informacji o stanie całego projektu lub produktu końcowego.

Przedmioty audytu:
stan projektu, proces wytwórczy lub jego dowolny podproces
system jakości, produkt końcowego

TESTOWANIE - podsumowanie

przydatne dla prowadzenia dowolnego rodzaju testowania.

*[ISTQB®, **Certyfikowany tester** Plan poziomu podstawowego]*

Zasada 1 - Testowanie ujawnia usterki

testowanie może pokazać usterki - nie może dowodzić braku defektów;

zmniejsza prawdopodobieństwo występowania w oprogramowaniu niewykrytych defektów.

TESTOWANIE - podsumowanie

Zasada 2 - Testowanie gruntowne (kompletne)jest niewykonalne

przetestowanie wszystkiego (wszystkich kombinacji wejść i warunków początkowych) jest wykonalne tylko w trywialnych przypadkach.

Zasada 3 - Wczesne testowanie

od startu projektu - w początkowych fazach wytwarzania

TESTOWANIE - podsumowanie

Zasada 4 - Kumulowanie się błędów

Niewielka liczba modułów zwykle zawiera większość usterek znalezionych przez wydaniem lub jest odpowiedzialna za większość awarii na produkcji.
Pracochłonność testowania jest dzielona proporcjonalnie do spodziewanej oraz zaobserwowanej gęstości błędów w modułach.

Zasada 5 - Paradoks pestycydów

te same (niezmieniane) testy, przestają znajdować nowe usterki=> testy muszą być regularnie przeglądane i uaktualniane.

TESTOWANIE - podsumowanie

Zasada 6 - Testowanie zależy od kontekstu

Testowanie wykonuje się w różny sposób w różnym kontekście. Np. oprogramowanie krytyczne ze względu na bezpieczeństwo a sklep internetowy.

Zasada 7 - Mylne przekonanie o braku błędów

Znajdowanie i naprawa usterek nie pomoże, produkowany system nie nadaje się do użytkowania lub nie spełnia wymagań i oczekiwań użytkownika.

Jakość oprogramowania - pomiary

Metryka oprogramowania:

- *funkcja odwzorowująca jednostkę oprogramowania w wartość liczbową; ta wartość jest interpretowalna jako stopień spełnienia pewnej własności(atrybutu,charakterystyki) jakości jednostki oprogramowania* *[IEEE1061-1998]*

- miara pewnej własności oprogramowania lub jego specyfikacji (nie ma precyzyjnej definicji i może oznaczać właściwie dowolną wartość liczbową charakteryzującą oprogramowanie).

Metryki związane z analizą jakości oprogramowania dzieli się na **metryki statyczne** oraz **metryki dynamiczne**,

Metryki statyczne pozwalają m.in. na ocenę jakości kodu źródłowego i łączą się ściśle z analizą statyczną - badanie struktury kodu źródłowego.

Elementy pomiaru oprogramowania

Obiekty	Atrybuty bezpośrednio mierzalne	Wskaźniki syntetyczne = metryki
	Produkty	
Specyfikacje	rozmiar, ponowne użycie, modularność, nadmiarowość, funkcjonalność, ...	zrozumiałość, pielęgnacyjność, ...
Projekty	rozmiar, ponowne użycie, modularność, spójność, funkcjonalność,...	jakość, złożoność, pielęgnacyjność, ...
Kod	rozmiar, ponowne użycie, modularność, spójność, złożoność, strukturalność, ...	niezawodność, używalność, pielęgnacyjność, ...
Dane testowe	rozmiar, poziom pokrycia,...	jakość,...
	Procesy	
Specyfikacja arch.	czas, nakład pracy, liczba zmian wymagań, ...	jakość, koszt, stabilność, ...
Projekt szczegółowe	czas, nakład pracy, liczba usterek w specyfik.	koszt, opłacalność, ...
Testowanie	czas, nakład pracy, liczba błędów kodu, ...	koszt, opłacalność, stabilność, ...
	Zasoby	
Personel	wiek, cena, ...	wydajność, doświadczenie, inteligencja, ...
Zespoły	wielkość, poziom komunikacji, struktura,...	wydajność, jakość, ...
Oprogramowanie	cena, wielkość, ...	używalność, niezawodność, ...
Sprzęt	cena, szybkość, wielkość pamięci	niezawodność, ...
Biura	wielkość, temperatura, oświetlenie,...	wygoda, jakość,...
....	....	....

Definiowanie metryk- zasady

Zasady definiowania metryk analizy statycznej (związane z obiektowymi paradygmatami programowania):

· sposób w jaki wyznaczane są metryki powinien być formalnie określony tzn. wynik pomiaru tego samego systemu powinien być zawsze taki sam,

· metryki nie należące do kategorii metryk rozmiaru powinny być niezależne od rozmiaru systemu,

· metryki powinny być bezwymiarowe lub wyrażone w pewnej spójnej jednostce systemowej,

· konstrukcja metryk powinna być taka, aby można było zbierać ich wartości w jak najwcześniejszej fazie projektu,

· metryki powinny być **skalowalne w dół** – możliwość stosowania dla całego systemu, jak i dla wybranych modułów,

· metryki powinny być łatwo obliczalne – umożliwi to opracowanie automatycznych narzędzi,

· metryki powinny być **niezależne od języka programowania**.

Podstawowe prawa dot. jakości oprogramowania

Prawo Demeter: metoda danego obiektu może odwoływać się jedynie do metod:

- należących do tego samego obiektu,
- metod dowolnego parametru przekazanego do niej,
- dowolnego obiektu przez nią stworzonego,
- dowolnego składnika klasy, do której należała dana metoda.

Popularnie jest ono wyrażane: „Nigdy nie rozmawiaj z obcymi

Zalety stosowania:
Ogranicza długie łańcuchy wywołań; ułatwia pielęgnację, zmniejsza współczynnik błędów, ogranicza powiązania między obiektami, wzmacnia hermetyzację i abstrakcję obiektów

Podstawowe prawa dot. jakości oprogramowania

Zasady SOLID: dotyczą projektu na poziomie klas

Zasada jednej odpowiedzialności - SRP *(ang. single responsibility principle)* - odnosi się do spójności klasy

Zasada otwarte/zamknięte - OCP *(ang. open/closed principle).* różnego rodzaju typy (np. klasy albo pakiety) powinny być otwarte na rozszerzanie, ale zamknięte na modyfikacje. Oznacza to, że można zmienić zachowanie takiego komponentu bez zmiany istniejącego kodu; jest realizowana przy użyciu typów abstrakcyjnych.

Zasada podstawienia Liskov - LSP *(ang. Liskov substitution principle)*: funkcje, które używają wskaźników bądź referencji do klas bazowych, muszą być w stanie używać obiektów klas dziedziczących po klasach bazowych bez dokładnej znajomości tych obiektów. Oznacza to w praktyce, że jeśli obiekt S dziedziczy po obiekcie T, to możliwe jest podstawienie za obiekt T obiektu S.

Podstawowe prawa dot. jakości oprogramowania

Zasady SOLID *cd.*

Zasada segregacji interfejsów - ISP *(ang. interface segregation principle).:* klient powinien mieć dostęp przez interfejs jedynie do metod klasy, które używa- klient nie powinien zależeć od metod, których nie używa. W praktyce -wprowadzenie kilku interfejsów dla jednej klasy. Zaleta: zmniejszenie liczby powiązań w systemie.

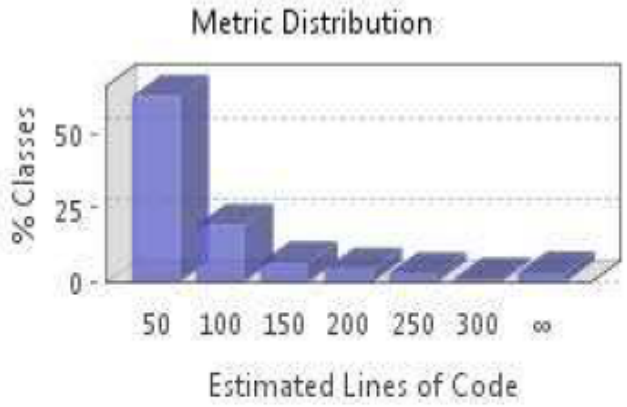
Zasada odwrotnych zależności - DIP *(ang. dependency inversion principle).* wyrażona jest w dwóch sformułowaniach:

· moduły wysokiego poziomu nie powinny zależeć od modułów niższego poziomu: obydwa powinny zależeć od typów abstrakcyjnych,

· typy abstrakcyjne nie powinny zależeć od szczegółowych ale typy szczegółowe powinny zależeć od typów abstrakcyjnych.

Metryki rozmiaru oprogramowania (1)

LOC (*lines of code*) lub SLOC (*source lines of code*)- liczba linii kodu źródłowego; najprostsza metryka rozmiaru oprogramowania; opisuje ‚skalę' programu; nie wystarcza do innych szczegółowych analiz kodu.



Rysunek 6. Metryka LOC dla klas. 11 klas przekracza rekomendację Stan'a dla LOC równą 400. 80% klas ma LOC poniżej 100.

Tabela 4. 5 klas o najwyższych wartościach LOC. Zalecany podział klas.

Klasy	LOC
ew.workbench.client.ui.SnippetPanel	951
ew.workbench.client.ui.FileBrowserPanel	771
sshexecutor.SSHFileManager	725
core.execution.ExperimentSessionBase	614
sshexecutor.ModifiedSCPClient	512

*[B.Bodziechowski, **Analiza jakości kodu przy użyciu metryk oprogramowania**..., praca mgr AGH 2012]*

Metryka Halsteada oprogramowania (2)

nie mierzy samego programu, lecz skupia się na pomiarze jego jednostek syntaktycznych (tokenów):

W programie **P** wyróżniono dwa typy tokenów - operatory i operandy:

$\mathbf{n_1}$ – liczba unikalnych operatorów

$\mathbf{n_2}$ – liczba unikalnych operandów

$\mathbf{N_1}$ – całkowita liczba wystąpień operatorów w programie **P**

$\mathbf{N_2}$ – całkowita liczba wystąpień operandów w programie **P**

Słownik **P** zawiera $\mathbf{n} = \mathbf{n_1} + \mathbf{n_2}$ elementów

Długość programu wynosi $\mathbf{N = N_1 + N_2}$

Według Elshoffa, który opracował modyfikację tej metody, długość programu lepiej jest szacować ze wzoru:

$$\mathbf{N = n_1 \log n_1 + n_2 \log n_2}$$

Dodatkowo można określić ‚zwięzłość' implementacji V:

$$\mathbf{V = N \log n}$$

Metryki złożoności oprogramowania

Złożoność na poziomie kodu:

1. liczba McCabe'a,
2. miara McClure (liczba zmiennych sterujących +liczba porównań)

Liczba McCabe'a

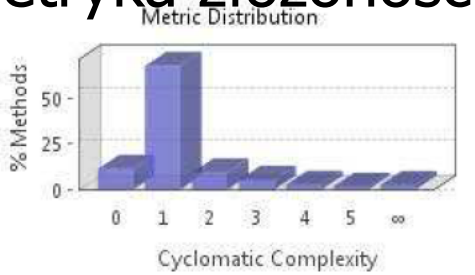
Jeśli G jest schematem blokowym programu P i G posiada e krawędzi i n węzłów to liczba niezależnych ścieżek w G (złożoność cyklomatyczna - CC) wyraża się wzorem:

$$v(P) = e - n + 2$$

lub prościej jeżeli d jest liczbą węzłów decyzyjnych G, wtedy:

$$v(P) = d+1$$

Metryka złożoności McCabe'a (przykład)



Rysunek 9. Złożoność cyklomatyczna CC dla metod. Jedynie 2,5% metod przyjmuje wartości CC wyższe niż 5, co jest bardzo dobrym rezultatem.

W Tabeli 7 pokazano 5 metod z największą wartością metryki CC – kandydaci do refaktoryzacji. W całym systemie wykryto jedynie 11 metod powyżej zalecanego progu CC = 10.

Tabela 7. 5 metod z najwyższą wartością metryki CC. Zaleca się aby wymienione tutaj metody rozbić na mniejsze. W całym systemie wykryto tylko 11 metod przekraczających zalecany próg równy dla CC 10.

Metoda	CC
`core.execution.ExperimentSessionBase.runSnippet(SnippetRun, String)`	36
`ew.webgui.client.richtoolbar.RichTextToolbar$EventHandler.onClick(ClickEvent)`	22
`ew.webgui.client.WebGuiDisplay.createField(FieldType, String, String, Map<String, String>)`	20
`cyfronet.gs2.ew.base.server.OpenerController.getFileContent(HttpServletRequest, WebRequest, String, String, String, HttpServletResponse)`	18
`core.Populator.populate(String, Variables)`	16

*[B.Bodziechowski, **Analiza jakości kodu przy użyciu metryk oprogramowania**..., praca mgr AGH 2012]*

Jakość kodu - zapachy

Zapachy kodu - pewne cechy kodu źródłowego mówiące o złym sposobie implementacji i będące sygnałem do refaktoryzacji

Pięć kategorii [Mika Mäntylä] :

- ang. **The Bloaters** - coś co rozrosło się do tak dużych rozmiarów, iż nie może być poprawnie obsłużone: duża klasa, długa metoda, obsesja typów podstawowych, długa lista parametrów, zbitki danych.
- ang. **The Object-Orientation Abusers**: - rozwiązania które nie wykorzystują w pełni możliwości programowania obiektowego: instrukcje switch, pole tymczasowe, odrzucony spadek, różne klasy z identycznym interfejsem.
- ang. **The Change Preventers** - utrudniają zmiany lub rozwój oprogramowania: poszatkowanie, równoległe hierarchie dziedziczenia, rozbieżna zmiana.
- ang. **The Dispensables** - zawierają coś, co powinno być usunięte z kodu źródłowego: leniwa klasa, klasa danych, powtórzony kod, martwy kod, spekulatywna ogólność.
- ang. **The Couplers** - zapachy związane z powiązaniem kodu: zazdrość o kod, zbytnia intymność, pośrednik, łańcuchy wywołań.

Jakość kodu – zapachy (2)

Przykłady:

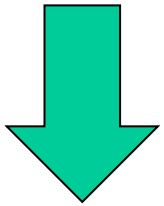
- **Długa metoda** (ang. Large method) – długi kod metody.
- **Duża klasa** (ang. Large class) - klasy posiadające zbyt wiele odpowiedzialności -należy przeorganizować strukturę klas w projekcie.
- **Zazdrość o kod** (ang. Feature envy) - metody intensywnie korzystające z danych innej klasy; metoda taka powinna być przeniesiona do klasy, z której danych korzysta.
- **Zbytnia intymność** (ang. Inappropriate intimacy) - klasy, których działanie jest zależne od implementacji innych klas- sprzeczne z ideą hermetyzacji, gdzie nie należy znać szczegółów implementacyjnych innych klas, a jedynie ich interfejs.
- .

Jakość kodu – zapachy (3)

- **Odrzucony spadek** (ang. Refused bequest) - istnieją klasy pochodne, które przeciążają metodę z nadklasy tak iż naruszają jej kontrakt. Jest to naruszenie zasady podstawienia Liskov (*Funkcje, które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów*.)
- **Leniwa klasa** (ang. Lazy class) - klasy posiadające bardzo mały zakres odpowiedzialności.
- **Powielony kod** (ang. Duplicated code) - ten sam fragment kodu powtarza się w kilku miejscach; utrudnia to wprowadzanie zmian (muszą zostać odnalezione wszystkie miejsca w kodzie, które realizują to samo zadanie)[2].
- **Contrived Complexity** - użyte zostały skomplikowane wzorce projektowe - zastosowanie znacznie prostszych byłoby wystarczające.

Refaktoryzacja kodu

Przykry zapach w kodzie jest sygnałem istnienia złej struktury programu (wymagającej poprawy)



- Refaktoryzacja (kodu) - technika umożliwiająca utrzymywanie odpowiedniej, wysokiej jakości kodu co skutkuje łatwiejszą i tańszą pielęgnację oprogramowania
- Refaktoryzacja **zachowuje funkcjonalność** programu (*poprawa projektu struktury klas i samych klas bez zmiany funkcjonalności)*
- Analiza statyczna i testowanie - metody weryfikacji poprawności refaktoryzacji

Refaktoryzacja kodu

Podstawowe cele refaktoryzacji:

- upraszczanie struktury
- zwiększanie elastyczności
- zmniejszanie zbyt dużych klas
- zmniejszanie zbyt długich metod
- unikanie klas podobnych do struktur (zbyt mało kodu)
- unikanie duplikowania (zasada once and only once - „raz i tylko raz”)
- unikanie konstrukcji „switch” i rozbudowanych „if...else if...else” (wprowadzanie dziedziczenia)
- *Refaktoryzacja jest możliwa dzięki oddzieleniu interfejsu od implementacji*

Miary złożoności projektu oprogramowania

Dla podejścia strukturalnego dwie miary:

moc modułu – miara złożoności wewnątrz modułowej:

przypadkowa, klasyczna, logiczna, komunikacyjna,

funkcjonalna, informacyjna

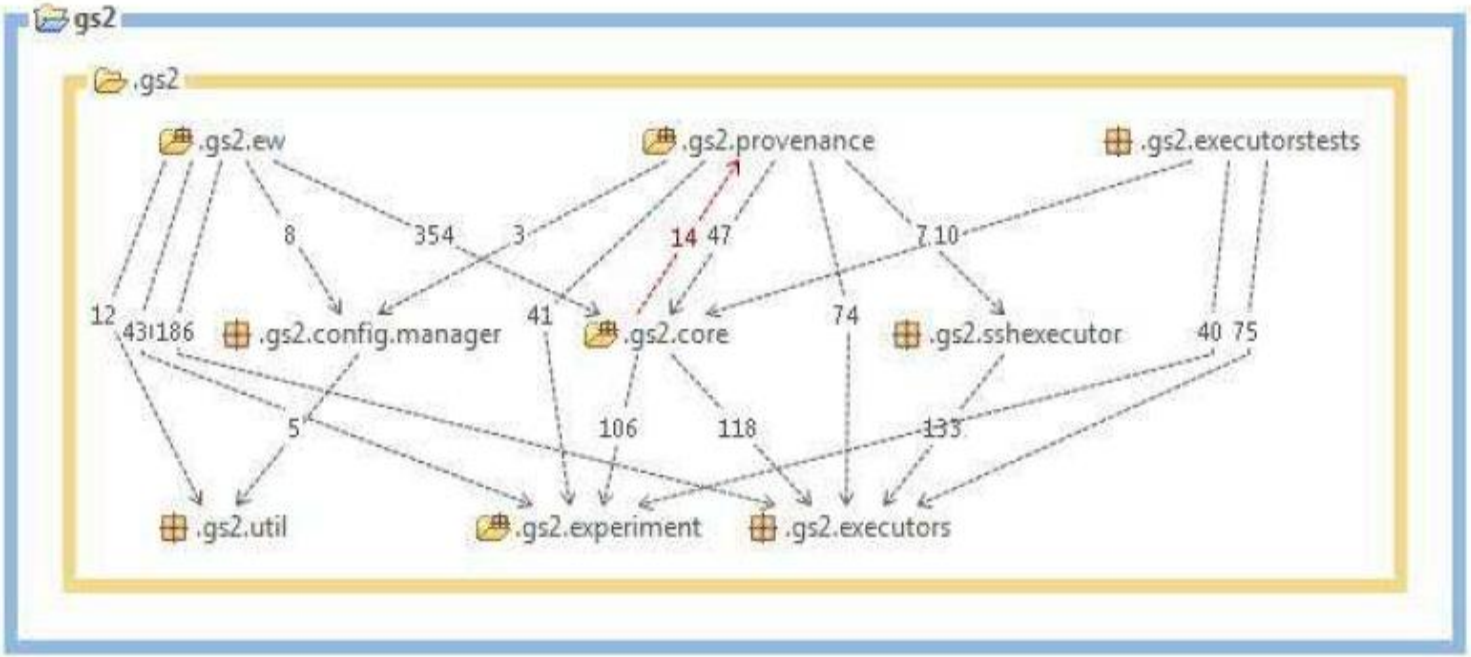
więź modułu - miara złożoności komunikacji międzymodułowej:

treściowa, sterowania, wspólna, zewnętrzna, ***cechy, danych***

Dla podejścia obiektowego miary

dla klasy – zestaw miar Chidamber/Kemerer (1994)
dla projektu - MOOD (Abreu,Goualo,Esteves 1995)

Miary jakości modułów – przykład powiązań



Rysunek 5. Diagram zależności GS2 stworzony przez aplikację Stan. Powiązania między modułami oznaczone są poprzez strzałkę wskazującą kierunek zależności, dodatkowo nad strzałką zaznaczono liczbę powiązań. Pomiędzy modułami provanance i core widoczny jest splot, co oznacza łamanie zasady ADP.

Miary złożoności (podejście obiektowe)

(Chidamber/Kemerer 1994)

Złożoność klasy (WMC)- *to, jeśli złożoność każdej metody klasy jest traktowana jako jeden, to WMC=n, gdzie n jest liczbą metod w klasie*

Złożoność metody *jest obliczana w postaci wariantu złożoności cyklomatycznej liczby McCabe'a; ponieważ ta miara jest addytywna, to można policzyć złożoność klasy.*

Głębokość drzewa dziedziczenia (DIT) *– zlicza poziomy, od 'korzenia' drzewa*

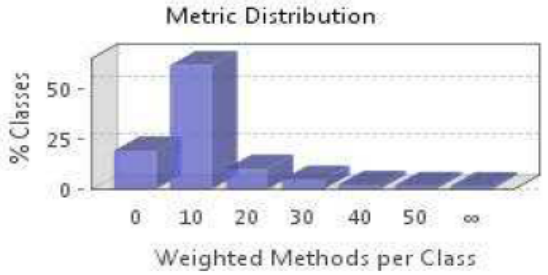
Liczba potomków *(NOC) – zlicza klasy pochodne na sąsiednim poziomie*

Więź międzyobiektowa *(CBO) – zlicza komunikacje nie-dziedziczone*

Żądanie wygenerowane przez klasę *(RFC) – zlicza wszystkie metody stosowane przez klasę.*

Brak spójności w metodach *(LCOM) – identyfikuje metody, które stosują różne atrybuty.*

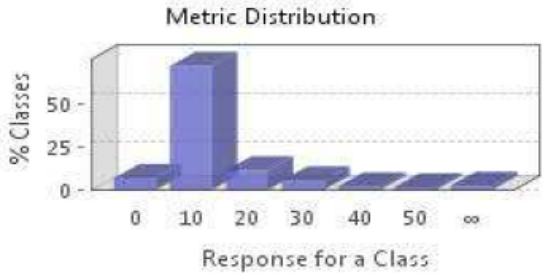
Metryki Chidamber/Kemerer -przykład



Rysunek 10. Metryka WMC. 75% klas ma wartość WMC < 10, co oznacza, że klasy nie są zbyt złożone.

Tabela 9. WMC najwyższe wartości dla 5 klas. Jedynie 4 klasy łamią rekomendację NASA dla WMC równą 100, jako zalecaną do podziału.

Klasy	WMC
ew.workbench.client.ui.SnippetPanel	136
sshexecutor.SSHFileManager	120
sshexecutor.ModifiedSCPClient	118
core.execution.ExperimentSessionBase	109
ew.workbench.client.ui.ExperimentPanel	78



Rysunek 11. Metryka RFC. 80% klas ma RFC poniżej 10, a jedynie 6% przekracza próg 20. Wartości średnio 2 razy niższe niż w NASA.

Tabela 10. RFC najwyższe wartości dla 5 klas. Brak naruszeń wartości progowej NASA ustalonej na 200.

Klasy	RFC
ew.workbench.client.ui.SnippetPanel	195
ew.workbench.client.ui.FileBrowserPanel	159
ew.workbench.client.ui.ExperimentPanel	140
core.execution.ExperimentSessionBase	137
ew.workbench.client.ui.MainSitePanel	126

*[B.Bodziechowski, **Analiza jakości kodu przy użyciu metryk oprogramowania**..., praca mgr AGH 2012]*

Miara stabilności oprogramowania

Indeks dojrzałości oprogramowania:
Ocenia stabilność oprogramowania

$$SMI = \frac{M_T - (F_a + F_C + F_d)}{M_T}$$

gdzie: M_T - liczba funkcji (modułów) w obecnej wersji oprogramowania
Fa - liczba funkcji dodanych
Fc - liczba funkcji zmienionych
Fd - liczba funkcji usuniętych

Miara niezawodności produktu

Indeks defektów (ocena poprawy jakości oprogramowania w kolejnych etapach wytwarzania)

Indeks defektów w i-tej fazie (stosuje się do faz w których wykryto defekty tzn. Di <> 0):

$$PI_i = W_1 \frac{S_i}{D_i} + W_2 \frac{M_i}{D_i} + W_3 \frac{T_i}{D_i}$$

gdzie: Di - liczba defektów wykrytych w i-tej fazie
Si - liczba poważnych defektów wykrytych w i-tej fazie
Mi - liczba defektów o umiarkowanej wadze wykrytych w i-tej fazie
Ti - liczba defektów trywialnych wykrytych w i-tej fazie
W1 - waga poważnych defektów (np. 10)
W2 - waga defektów umiarkowanych (np. 3)
W3 - waga defektów trywialnych (np. 1)

Metryki oprogramowania - podsumowanie

1. Zalecenia i zalety stosowania metryk
- używać od początkowej fazy tworzenia programowania.
- analiza uzyskanych wartości metryk pozwala wychwycić niewłaściwe tendencje w wytwarzaniu oprogramowania, co zmniejszy koszt refaktoryzacji;.
- metryki pomagają lepiej zrozumieć złożoność budowanego systemu.

2. Nie uzyskano (wyrażonej formalnie) korelacji metryk (np. kodu) z takimi zewnętrznymi charakterystykami jakości oprogramowania jak: pielęgnowalność, funkcjonalność, niezawodność, użyteczność, efektywność czy przenośność.