Zastosowanie metryk obiektowych w technologii Java

Jerzy Kaczmarek, Maciej Kucharski
Katedra Zastosowań Informatyki, Politechnika Gdańska

jkacz@pg.gda.pl, mkuchar@loqi.eti.pg.gda.pl

Streszczenie

Artykuł przedstawia wybrane metryki opisujące rozmiar i strukturę oprogramowania wykonanego w języku Java. Przedstawiono wyniki pomiarów dla 20 programów oraz porównano je z pomiarami oprogramowania napisanego w języku C++. Stwierdzono, że na podstawie ilości klas można szacować ilość linii kodu programu.

1. Wstęp

Metryki stanowią istotny element systemu kontroli jakości i zarządzania procesem wytwarzania oprogramowania. Pozwalają wykrywać i eliminować błędy zanim spowodują one straty oraz pozwalają na planowanie i organizację procesu wytwarzania oprogramowania.

Rozwój nowych narzędzi wspomagających wytwarzanie oprogramowania oraz nowych języków programowania wymusza rozwój technik związanych z opomiarowaniem procesu wytwarzania oprogramowania.

Java jest w pełni obiektowym językiem programowania zorientowanym na wytwarzanie aplikacji sieciowych. Z punktu widzenia konstrukcji składniowej język Java zbliżony jest do C++, jednakże pomiary wartości metryk obiektowych różnią się istotnie.

Opisane w artykule wyniki pomiarów mogą być przydatne przy szacowaniu nakładu i analizie jakości oprogramowania wykonanego w języku Java.

2. Metryki

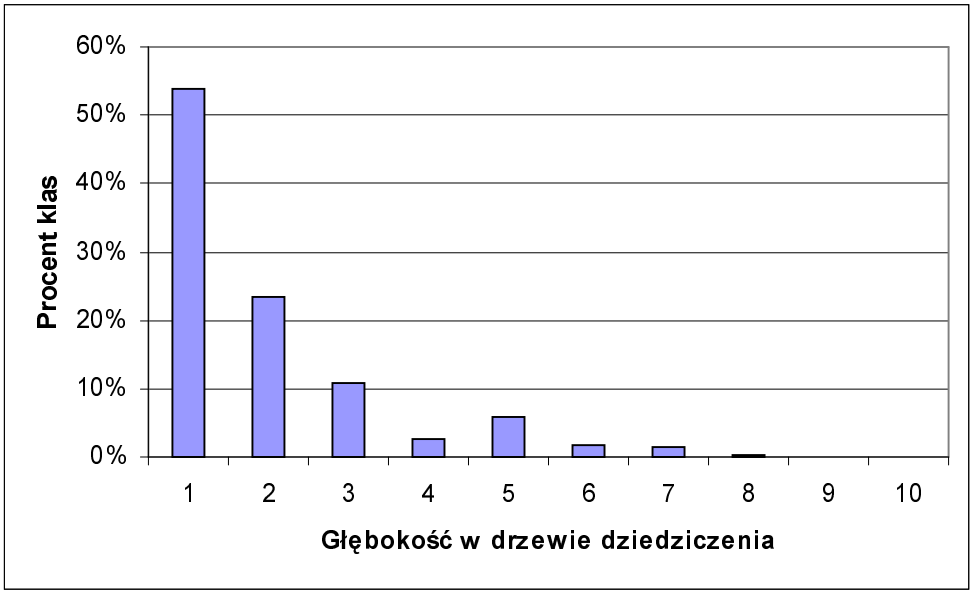
Zagadnienie metryk dla języków obiektowych jest przedmiotem licznych badań w wielu ośrodkach naukowych [1,2,3,4,5,6,7]. W teorii wytwarzania oprogramowania obiektowego (Object Oriented Analysis and Design) zdefiniowano różne metryki. Z praktycznego punktu widzenia najistotniejszymi są:

- **Głębokość w drzewie dziedziczenia** (**DIT-Depth in Inheritace Tree**) - określa jak głęboko w drzewie dziedziczenia znajduje się dana klasa. Metryka ta kontroluje prawidłową konstrukcję drzewa dziedziczenia. W świecie rzeczywistym nie ma potrzeby stosowania zbyt dużej liczby specjalizacji, dlatego wysokość w drzewie dziedziczenia powyżej 6 sygnalizuje, że dziedziczenie nie odbywa się poprzez specjalizację. Zbyt płaskie drzewo dziedziczenia wskazuje na redundancję kodu na skutek braku wykorzystania mechanizmów generalizacji i specjalizacji.
- **Liczba potomków** (**NOC - Number Of Childern**) - metryka ta określa liczbę klas, które w sposób bezpośredni dziedziczą po danej klasy. W przypadku gdy drzewo dziedziczenia jest zbyt płaskie, istnieją klasy mające dużo potomków. Istnieje wtedy duże prawdopodobieństwo, że część z nich posiada wspólną funkcjonalność, która przy prawidłowym procesie specjalizacji oraz generalizacji powinna znaleźć się w klasach pośrednich.
- **Liczba linii kodu na klasę** (**LOC per class**) - metryka ta jest najbardziej naturalnym sposobem wyrażenia rozmiaru, ale jest zależna od stylu pisania programu przez programistę. Metryki LOC oraz SLOC stanowią podstawę szacowania nakładu w metodach COCOMO [8,9,10]. Ilość linii kodu w klasach powinna być zbliżona. W przeciwnym przypadku rozłożenie funkcjonalności na poszczególne klasy nie jest równomierne. Duże klasy powinny być w procesie reenginering'u dzielone na mniejsze.
- **Liczba metod na klasę** - metryka ta oznacza liczbę nowych (nie odziedziczonych) metod wprowadzonych w danej klasie. Nadmierna liczba metod w klasie jest ostrzeżeniem, że zbyt duża funkcjonalność została przeznaczona dla tej klasy. Prawidłowo przygotowany projekt powinien rozdzielać logikę systemu na wiele współpracujących klas. Skupienie znacznej części logiki systemu w jednej klasie jest błędem projektowym.
- **Liczba linii kodu na metodę** (**LOC per method**) - określa rozmiar metody. Metody powinny być małe i w czytelny sposób implementować wyspecyfikowaną funkcjonalność. Istnienie dużych metod w klasie świadczy o nierównomiernym rozłożeniu funkcjonalności wewnątrz klasy i może być traktowane jako błąd projektowy.

3. Wyniki pomiarów

Dokonano pomiarów dla 20 programów napisanych w języku Java. wykorzystując stworzony dla tych celów system automatycznie mierzący liczbę linii kodu, głębokość w drzewie dziedziczenia i liczbę klas potomnych dla wybranych klas i metod.

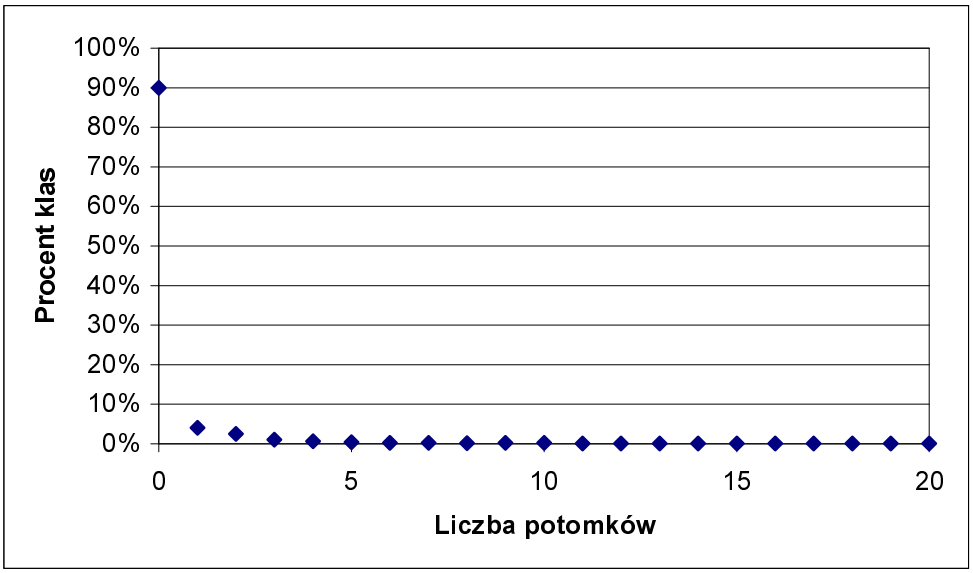
Na rysunku 1 przedstawiono wyniki pomiaru głębokości w drzewie dziedziczenia. Podczas pomiaru jako głębokość klasy bezpośrednio dziedziczącej od klasy należącej do biblioteki języka Java przyjęto wartość jeden.



Rys. 1 Rozkład głębokości dziedziczenia

Na rysunku 1 można zauważyć, że większość ze zbadanych klas jest blisko korzenia drzewa dziedziczenia. Otrzymane wyniki wskazują, że mało wykorzystany był mechanizm generalizacji i specjalizacji. Przy interpretacji wyników należy również uwzględnić wpływ klas interfejsu użytkownika. Zazwyczaj klasy interfejsu dziedziczą bezpośrednio od klasy bibliotecznej, co oznacza, że ich głębokość jest równa jeden.

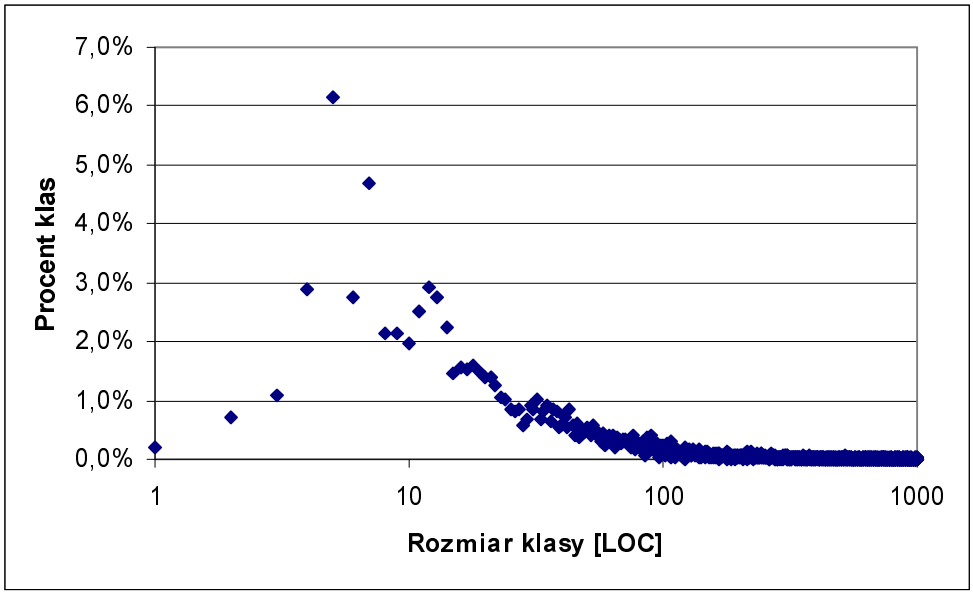
Na rysunku 2 przedstawiono rozkład liczby potomków w badanych programach. W języku Java często deklaruje się klasy tylko w celu wykonania pewnej czynności. Wykorzystywane są wówczas wyspecjalizowane klasy biblioteczne, których głębokość w drzewie dziedziczenia jest równa jeden i nie mają one potomków.



Rys.2 Rozkład liczby potomków

Jak wynika z rysunku 2 niemal 90 % zbadanych klas nie posiada potomków. Może to oznaczać, że są to klasy biblioteczne. W badanych klasach zaledwie 0,7% klas posiada więcej niż 10 potomków. Świadczy to o dobrej hierarchii klas w badanych programach.

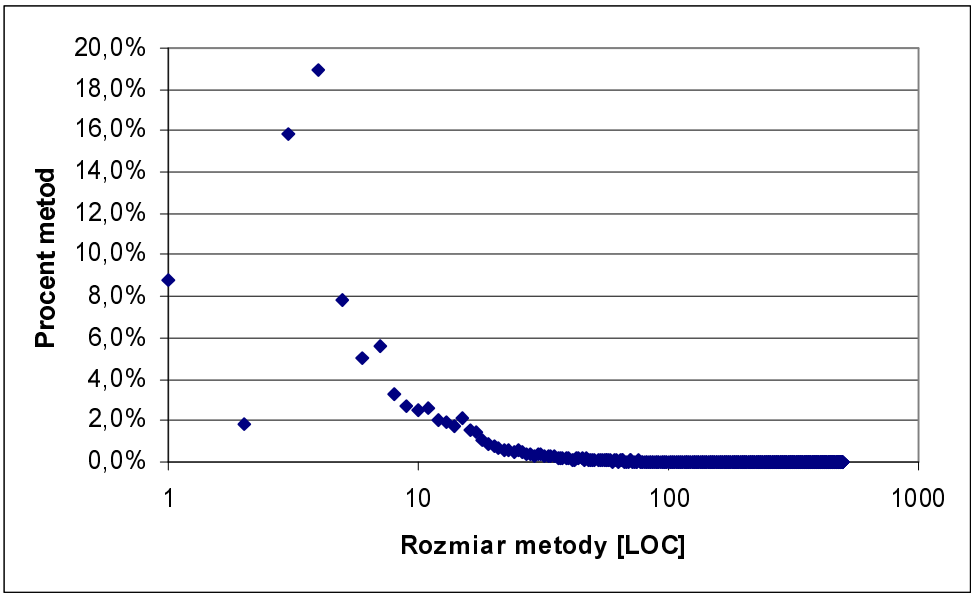
Na rysunku 3 przedstawiono pomiary wielkości klas.



Rys. 3 Rozkład wielkości klas

Na rysunku 3 można zauważyć, że spośród mierzonych klas nieliczne mają rozmiar powyżej 100 linii kodu. Sporadycznie występujące duże klasy związane są na ogół z implementacją obsługi zdarzeń przekazywanych aplikacji. Małe klasy poniżej 10 linii kodu powstają w wyniku odwołań do klas bibliotecznych.

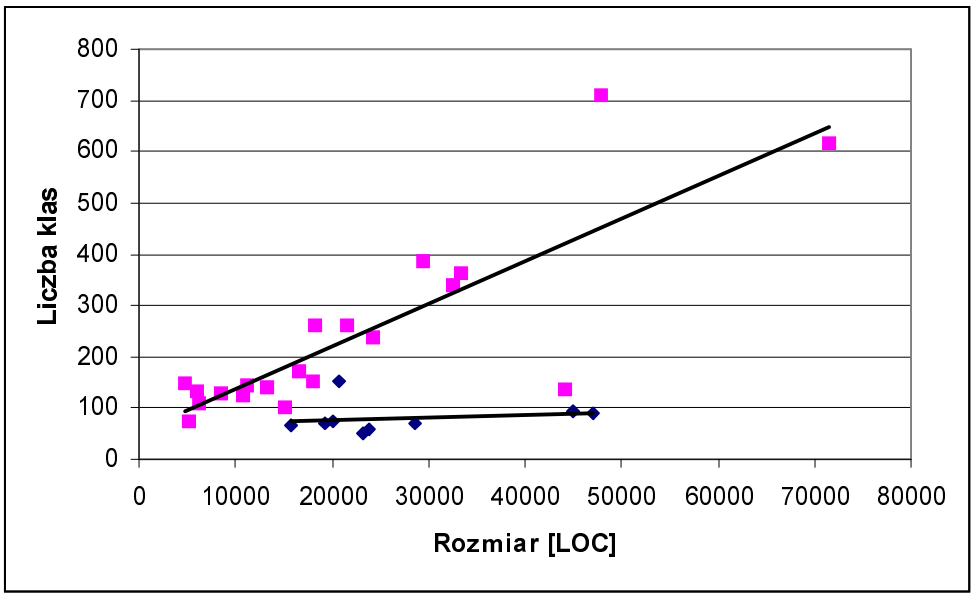
Na rysunku 4 przedstawiono wyniki pomiarów wielkości metod w badanych programach.



Rys. 4 Rozkład wielkości metod

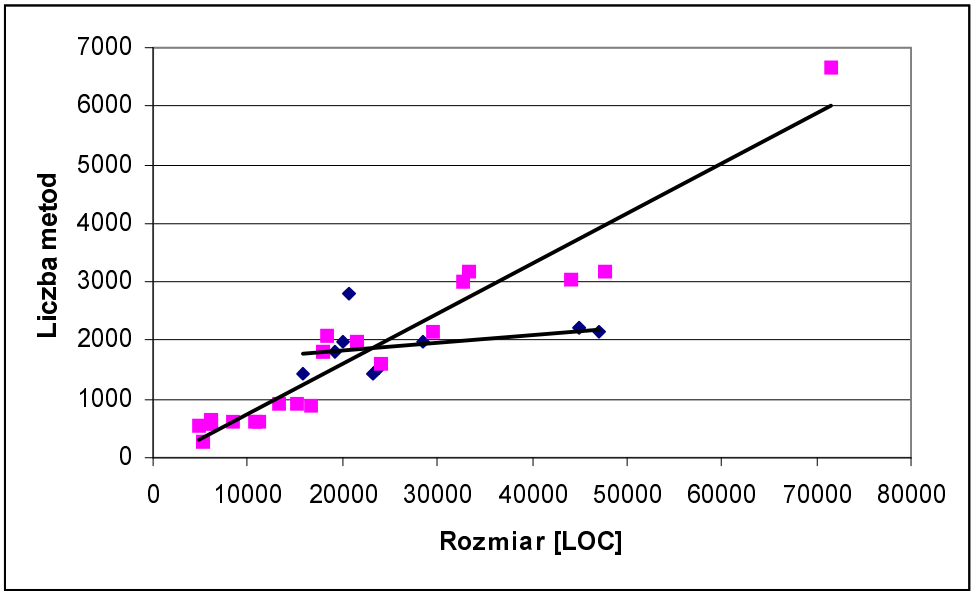
Z otrzymanych rezultatów przedstawionych na rysunku 4 wynika, że większość metod ma poniżej 10 linii kodu, a tylko nieliczne przekraczają 50. Duża liczba małych metod pozytywnie świadczy o jakości projektu i implementacji. Wnikliwej kontroli wymagają jedynie duże metody. Zapewne część z nich wymuszona jest przez konstrukcję bibliotek języka Java, podczas gdy pozostałe duże klasy mogą być wynikiem niewłaściwej dekompozycji systemu na etapie projektu klas.

Na rysunku 5 przedstawiono pomiary zależności pomiędzy liczbą klas a ilością linii kodu w programie. Dla porównania przedstawiono również wyniki programów napisanych w C++ opublikowane przez K. Moriss'a [4]. Wyniki pomiarów dla programów napisanych w języku Java zaznaczono w postaci kwadratów, a wyniki pomiarów dla C++ w postaci rombów.



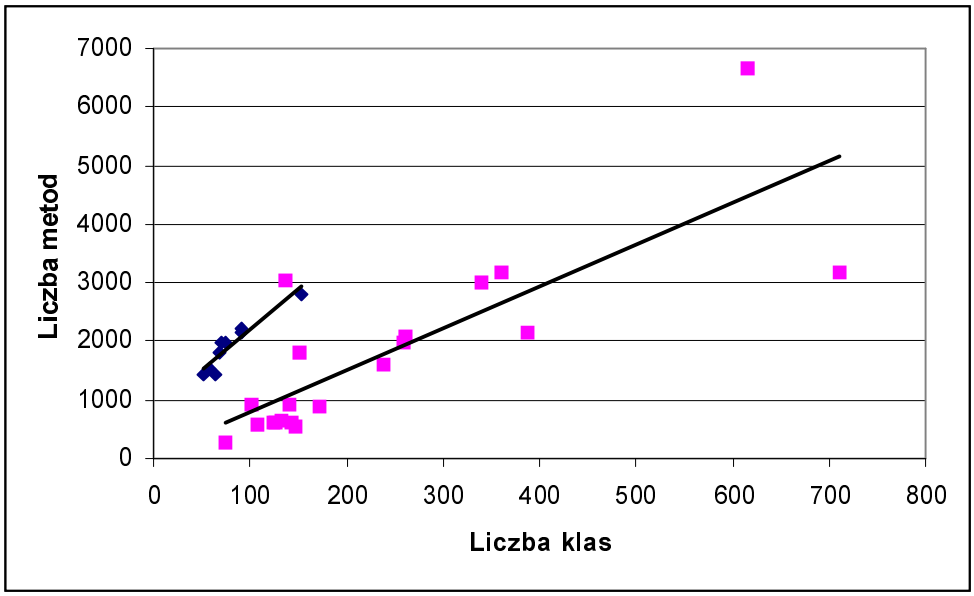
Rys. 5 Zależność rozmiaru programu od liczby klas

Jak wynika z rysunku 5, liniowa zależność pomiędzy rozmiarem programu a liczbą klas w przypadku języka Java ma nachylenie znacznie większe. Ponadto można zauważyć, że programy napisane w języku Java cechują się większą liczbą klas niż programy napisane w C++ .



Rys. 6 Zależność liczby metod od rozmiaru aplikacji

Na rysunku 6 przedstawiono zależność liczby metod od rozmiaru aplikacji. Zarówno w przypadku języka Java jak i C++ liczba metod w programie jest liniowo zależna od ilości linii kodu, przy czym w przypadku języka Java nachylenie jest większe.



Rys. 7 Zależność liczby metod od liczby klas

Na rysunku 7 przedstawiono zależność pomiędzy liczbą klas i liczbą metod w badanych programach. Można zaobserwować, że nachylenie prostych jest podobne.

4. Podsumowanie

Na podstawie pomiarów przeprowadzonych dla programów napisanych w językach Java i C++ można zauważyć pewne istotne tendencje. W przypadku C++ większy program ma większe metody, podczas gdy ilość klas i metod zwiększa się nieznacznie. Natomiast w przypadku języka Java przyrost programu następuje głównie poprzez zwiększanie się liczby klas i metod. Można jednak zauważyć, że w języku Java średnia ilość linii kodu w klasach i metodach jest niezależna od wielkości aplikacji.

Na podstawie przeprowadzonych badań można zatem postawić tezę, że znając ilości klas i metod na wczesnym etapie cyklu życia oprogramowania można szacować z pewnym przybliżeniem końcową ilość linii kodu projektowanej w języku Java aplikacji.

Bibliografia

[1]. *Object-Oriented Metrics an Annotated Bibligraphy*, Bournemouth University, UK, 1998.

[2]. S.R., Chidamber Kemerer C.F., *A Metrics Suit for Object Oriented Design,* IEEE Tran. On Software Engineering, Vol.20. No.6, June 1994.

[3]. K. Morris, *Metrics for object oriented software development*, Master thesis, Cambridge, 1988.

[4]. M. Lorenz, *Object-Oriented Software Development*, Prentice Hall, 1993.

[5]. M. Lorenz, *Real Word Reuse*, Journal of Object-Oriented Programming, 11/12 1991.

[6]. C.F. Kermerer, *An Empirical Validation of Software Cost Estimation* Models.Comm. ACM, volume 30 pages 416-429, 1987.

[7]. B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[8]. J. Kaczmarek, M. Kucharski, *Metody oceny nakładu w zarządzaniu projektem informatycznym*, II Krajowa Konferencja Inżynierii Oprogramowania KKIO, Zakopane 2000.

[9]. J. Kaczmarek, M. Kucharski, *Application of Object-Oriented Metrics for Java Programs*, Project Control for Software Quality. Proceedings of ESCOM-SCOPE99, Herstmonceux Castle, England 1999.