Zachodniopomorski Uniwersytet Technologiczny
Wydział Informatyki

PRACA DOKTORSKA

Metoda estymacji długości kodu źródłowego na podstawie diagramów statycznych UML

Autor:
mgr inż. Maciej Stachurski

Promotor pracy:
prof. dr hab. inż. Ryszard Budziński

Szczecin, 2009

SPIS TREŚCI

1 WSTĘP

Cywilizacja ludzka współcześnie w coraz większym stopniu opiera się na rozwiązaniach informatycznych. Stanowią one niemal niewidzialną osnowę, na której konstruowanie są nierzadko skomplikowane systemy wspomagające wszelkie przejawy działalności człowieka. Trudno dzisiaj wyobrazić sobie sprawne działanie w tak odległych pojęciowo dziedzinach jak medycyna i bankowość czy giełda i produkcja samochodów w sytuacji, gdyby nie było rozwiązań informatycznych wspierających pracę człowieka. Komputery wraz z działającym na nim oprogramowaniem wyręczają nas w tych działaniach, które z natury jest nam, ludziom, najtrudniej wykonywać: zadaniach powtarzalnych i żmudnych, bo komputer się nie męczy i jest zawsze tak samo dokładny. W tym kontekście pojawia się potrzeba, aby konstruowane rozwiązania informatyczne jak najbardziej ułatwiały codzienną pracę człowieka w dowolnej dziedzinie życia. Zadanie to jest jednym z celów informatyki (ang. *computer science*), dziedziny nauki i techniki zajmującej się przetwarzaniem informacji, w tym technologiami wytwarzania systemów przetwarzających informacje i ich aplikowaniem na platformę sprzętową jaką jest komputer (Denning, 2000).

Rosnący wpływ systemów informatycznych na codzienne życie człowieka sprawia, że wzrasta zapotrzebowanie na nie. Mają one być coraz bardziej funkcjonalne i złożone, aby w maksymalnym stopniu przyczyniać się do poprawy życia człowieka w najróżniejszych jego przejawach. Wdrażanie systemów informatycznych ma również aspekt ekonomiczny: stosowanie nowoczesnych technologii skutkuje usprawnianiem procesów produkcji i dostarczania usług, co sprawia, że organizacje mogą być bardziej konkurencyjne, zwiększając swoje szanse przetrwania na rynku.

Podobną pozycję na rynku zajmują organizacje zajmujące się wytwarzaniem systemów informatycznych. Ich podstawowym celem biznesowym jest przetrwanie. Dlatego też muszą one działać w taki sposób, żeby ich procesy produkcyjne były na tyle sprawne, aby zapewnić im pożądaną pozycję na rynku. Zadanie to jest trudne ze względu na specyfikę wytwarzania systemów informatycznych, a w szczególności specyfikę wytwarzania oprogramowania. Istnieje wiele czynników sprawiających, że realizacja oprogramowania jest trudna, przy czym większość jest rezultatem abstrakcyjności tego produktu. Oprogramowanie jest bytem o typowo myślowej charakterystyce. Jest w swej istocie teoretycznym rozwiązaniem realizowanym na platformie sprzętowej komputera. Tworzenie oprogramo-

wania można porównać do badań naukowych: wyznaczając jakiś cel dociera się do niego zwiększając swoją wiedzę na badany temat, poprzez zapoznawanie się z wiedzą innych (w postaci koncepcji rozwiązań i możliwych do zastosowania technologii) oraz eksperymentowanie. W rezultacie proces wytwarzania oprogramowania cechuje się wysoką nieprzewidywalnością.

Głównym sposobem na zwiększenie konkurencyjności organizacji wytwarzających oprogramowanie jest redukcja nieprzewidywalności w procesie wytwórczym. Głównym sposobem na redukcję nieprzewidywalności jest wprowadzenie dobrej organizacji prac projektowych (CMMI Product Team, 2006)(Szyjewski, 2001), których rezultatem jest zwiększanie wiedzy o realizowanym przedsięwzięciu na wszystkich jego etapach. Wiedza ta powinna opierać się na faktach, jednoznacznie i obiektywnie opisujących przebieg procesu. Korzystając z takiej wiedzy można skutecznie i sprawnie zarządzać procesem rozwoju, na bieżąco planując zadania, przydzielając zasoby i kontrolując postęp realizacji. Ważne jest, aby ta wiedza była aktualizowana przez cały czas życia procesu wytwarzania.

Możliwie najpełniejsza wiedza o procesie wytwarzania znacząco ułatwia szacowanie (estymację) zasobów potrzebnych przy realizacji oprogramowania. Szacowanie, czyli przybliżone określanie wartości pewnych wielkości na podstawie niepełnych danych stanowi jeden z najważniejszych elementów zarządzania procesem wytwarzania. Jej znaczenie jest odwrotnie proporcjonalne do stopnia zaawansowania prac nad realizacją oprogramowanie. Najbardziej istotne jest na początkowych etapach, kiedy podejmowane są tak fundamentalne decyzje, jak koszt realizowanego oprogramowania czy też czas jego realizacji. Szacowanie na kolejnych etapach produkcji nie jest już aż tak istotne dla przedsięwzięcia jako całości, ale stanowi podstawę dla planowania zasobów dla konkretnych zadań, których realizacja jest niezbędna do zakończenia procesu wytwarzania. Szacowanie powinno się odbywać na wielu poziomach hierarchicznej struktury procesu, przyczyniając się do skutecznego zarządzania pracą jednostek kolejnych szczebli organizacji wytwórczej.

Istnieje wiele metod szacowania zasobów procesu wytwarzania oprogramowania. Różnią się one między innymi przedmiotem szacowania, przedziałem czasu kiedy mogą być stosowane, charakterystyką danych na których opierają swoje rezultaty, sposobami wyznaczania rezultatów, dobrocią uzyskiwanych rezultatów. Stosując je należy pamiętać, że uzyskiwane za ich pomocą oszacowania cechują się pewną niepewnością. Skala tej niepewności w dużym stopniu jest uzależniona od wiedzy o procesie wytwarzania i opraco-

wywanym produkcie: im późniejszy etap rozwoju, tym dostępna wiedza jest bogatsza, a co za tym idzie błąd uzyskiwanych oszacowań jest potencjalnie mniejszy.

Szczególne znaczenie w szacowaniu zasobów procesu wytwarzania oprogramowania ma rozmiar. Może być on rozumiany na wiele sposobów, w tym jako zakres dostarczonej funkcjonalności oraz długość kodu źródłowego stanowiącego implementację tejże funkcjonalności. Rozmiar oprogramowania, jakkolwiek rozumiany, bezpośrednio wpływa na inne zasoby procesu wytwórczego, wliczając w to koszt i czas realizacji. Stanowi on podstawę, której szacowanie jest najistotniejsze, bo definiuje ona samą istotę realizowanego produktu, tak w sensie funkcjonalnym, jak i strukturalnym.

Istnieje wiele metod szacujących rozmiar oprogramowania, różniących się przede wszystkim okresem czasu w cyklu życia produktu, kiedy mogą być stosowane. Konsekwencją tego kryterium jest potencjalny błąd oszacowania, którego wielkość maleje wraz ze wzrostem wiedzy o realizowanym oprogramowaniu. Metody działające na wczesnych etapach życia produktu dostarczają bardziej niepewnych wyników niż metody aplikowane później. Bardzo często różnią się one także jednostką uzyskiwanych oszacowań: metody działające na wcześniejszych etapach stosują zwykle bardziej abstrakcyjne (np. punkty funkcyjne) niż metody aplikowane później (np. ilość linii kodu źródłowego).

Niedobór wiedzy o realizowanym produkcie sprawia, że dane wejściowe w wielu metodach muszą być oceniane *a priori* przez człowieka. Taki mechanizm wymaga od osoby dokonującej estymacji doświadczenia w szacowaniu za pomocą konkretnej metody, a ponadto wnosi element subiektywności. Stosowanie tego typu metod do szacowania na wczesnych etapach rozwoju oprogramowania jest jak najbardziej akceptowalne ze względu na wielokrotnie już wspominaną ograniczoną ilość wiedzy o realizowanym produkcie. Jednakże estymacja na późniejszych etapach rozwoju powinna się już w większości opierać na obiektywnych danych, eliminując do minimum subiektywny wpływ oceny człowieka. Danymi na których mogłyby opierać się metody estymacji mogą być diagramy UML. Służą one do graficznego opisu rozmaitych aspektów oprogramowania realizowanego z wykorzystaniem paradygmatu obiektowego. Stanowią doskonałe źródło informacji o realizowanym produkcie, szczególnie jeśli są tworzone za pomocą edytorów diagramów UML. Aplikacje takie umożliwiają eksport diagramów do formatu bazującego na formacie XML, który można w łatwy sposób przetwarzać. Pozwala to na stosunkowo prostą automatyczną

ekstrakcję potrzebnych danych i zastosowanie ich na przykład w procesie szacowania rozmiaru realizowanego oprogramowania.

Wedle dzisiejszej wiedzy dotyczącej problemu estymacji rozmiaru oprogramowania istnieje niewiele metod opierających się na danych uzyskiwanych z diagramów UML. Brak jest spopularyzowanych metod, które w istotny sposób redukują subiektywną ocenę człowieka w procesie estymacji. ***W związku z tym autor niniejszej pracy przyjął sobie za cel opracowanie metody estymacji długości kodu źródłowego stanowiącego implementację realizowanego oprogramowania w oparciu o dane opisujące jego strukturę statyczną w postaci diagramów klas UML***.

Ponadto założono, że zaproponowana metoda będzie wyznaczać oszacowania w sposób całkowicie autonomiczny bez udziału człowieka.

Tezą pracy jest wykazanie, iż metoda algorytmiczna działająca w oparciu o dane opisujące statyczną strukturę realizowanego oprogramowania w postaci diagramów UML pozwoli na szacowanie długości kodu źródłowego stanowiącego jego implementację z błędem mniejszym niż się zazwyczaj spotyka w tego typu zastosowaniach[1].

W pracy zostanie zaprezentowana metoda estymacji, która jako wejście będzie przyjmować wybrany podzbiór danych możliwych do uzyskania z diagramów UML realizowanego oprogramowania, a na wyjściu będzie wyznaczać oszacowanie jego rozmiaru wyrażone w liczbie leksemów konkretnego języka oprogramowania użytego do implementacji. Opracowany zostanie model szacujący, na podstawie którego będą wyznaczane wartości estymat. Model ów powstanie w oparciu o dane doświadczalne pochodzące z rzeczywiście istniejących i zrealizowanych programach. Opracowana zostanie także metoda kalibracji modelu, dopasowująca go do konkretnego środowiska rozwoju oprogramowania. Jako przykład zastosowania procesu kalibracji zaprezentowany zostanie bazowy model estymacji, skalibrowany na potrzeby badań opisanych w niniejszej pracy. Model ten może posłużyć za punkt wyjściowy w przypadku stosowania proponowanej metody estymacji w środowisku rozwojowym nieposiadającym danych historycznych o wcześniej opracowywanych projektach informatycznych. Tak skonstruowany model zostanie oceniony w środowisku bazującym na danych z rzeczywiście istniejących i zrealizowanych przypadków oprogramowania za pomocą ogólnie uznanych kryteriów oceny jakości uzyskiwanych

[1] Typowa wartość graniczna błędu względnego powszechnie stosowana w badaniach to ±25% (Conte, Dunsmore, & Shen, 1986).

oszacowań. Proponowana metoda ze względu na możliwość jej zautomatyzowania może stanowić podstawę do opracowania rozszerzeń istniejących edytorów UML, pozwalających na estymację projektu oprogramowania z poziomu aplikacji.

Charakter i wynik rozważań podjętych w pracy można uznać za metodyczno-eksperymentalny oraz praktyczny.

Struktura pracy

Niniejsza praca została podzielona na sześć rozdziałów. Niniejszy rozdział pierwszy stanowi wprowadzenie do poruszanej tematyki.

Rozdział drugi stanowi wstęp do problematyki estymacji projektów informatycznych. Zawiera opis podstawowych problemów i pojęć odnoszących się do procesu wytwarzania oprogramowania ze szczególnym uwzględnieniem szacowania jako nieusuwalnego elementu inżynierii oprogramowania. Omówione zostaną najważniejsze standardy procesu wytwarzania oprogramowania oraz role, jakie pełni w nich szacowanie. Przedstawiony zostanie cykl życia oprogramowania oraz najważniejsze modele służące do zarządzania nim. W kolejnym podrozdziale opisane zostanie pojęcie pomiaru i jego znaczenie w inżynierii oprogramowania. Zakończenie stanowić będzie przedstawienie estymacji w inżynierii oprogramowania, jej zakresu, podstawowych technik oraz sposobów ich oceny.

Rozdział trzeci stanowi krótkie przedstawienie koncepcji paradygmatu obiektowego i jego zastosowań. Przedstawione zostaną diagramy UML jako wiodąca koncepcja graficznej prezentacji różnorakich aspektów modelowania obiektowego. Zaprezentowane zostanie koncepcja wzorców projektowych, będąca niejako skutkiem stosowania diagramów UML w praktyce.

Czwarty rozdział stanowi prezentację aktualnego stanu wiedzy na temat estymacji rozmiaru oprogramowania. Przedstawione zostanie pojęcie rozmiaru oraz jego trzy podstawowe aspekty. W tym kontekście opisane zostaną podstawowe metryki służące do wyrażania wartości rozmiaru w konkretnym aspekcie oraz możliwe do zastosowania metody estymacji.

Rozdział piąty został poświęcony na zaprezentowanie nowej metody estymacji. Omówiony zostanie aparat matematyczny przygotowany na jej potrzeby, przedstawiona zostanie specjalna metryka służąca do wyrażania rozmiaru oprogramowania. Przedstawione zostaną dane eksperymentalne, wykorzystane na potrzeby stworzenia modelu szacującego oraz do jego weryfikacji. Wreszcie zaprezentowany zostanie model szacujący, jego założenia, budowa oraz wykorzystywane metryki modelu obiektowego. Następnie zaprezentowana zostanie właściwa metoda, opisująca sposób użycia modelu szacującego, wraz z metodą jego kalibracji. Rozdział zostanie zakończony oceną jakości uzyskiwanych przez metodę oszacowań.

Rozdział szósty stanowi zakończenie niniejszej dysertacji. Zawiera on podsumowanie całości poprzez sformułowanie wniosków. Synteza studium została uzupełniona potencjalnymi zastosowaniami proponowanego rozwiązania oraz wskazaniem możliwości dalszych badań.

2 ESTYMACJA W INŻYNIERII OPROGRAMOWANIA

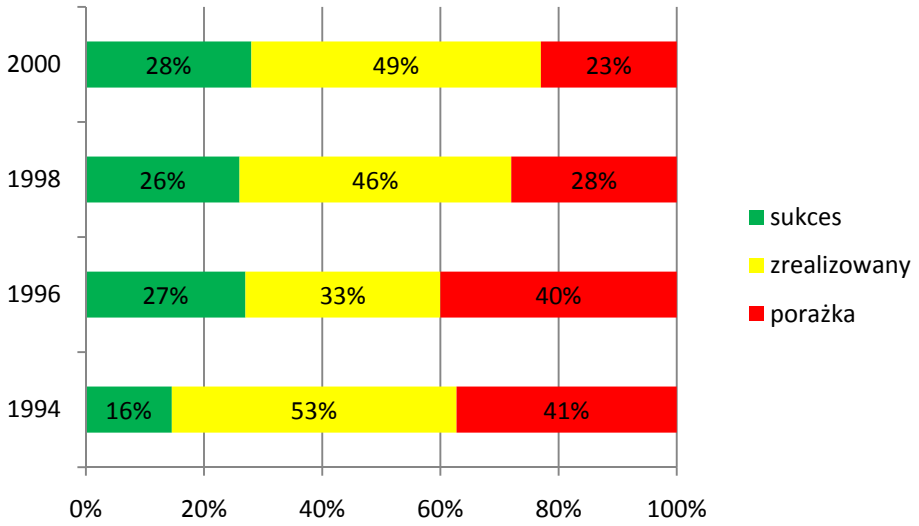
Niniejszy rozdział stanowi przegląd podstawowych problemów dla których rozwiązania powołana została dziedzina nauki zwana inżynierią oprogramowania. Obejmuje on między innymi takie kwestie, jak przebieg cyklu wytwarzania oprogramowania, sposoby zarządzania procesem wytwarzania oprogramowania, a także obecne trendy mające na celu jego usprawnienie. Na tej podstawie przedstawiony zostanie cel i sposoby realizacji procesu zbierania danych (pomiarów) stanowiących podstawę estymacji. Treść rozdziału zostanie zakończona przedstawieniem problemu estymacji w inżynierii oprogramowania wraz z określeniem jej zakresu i sposobów realizacji.

2.1 PRZEDMIOT INŻYNIERII OPROGRAMOWANIA

Inżynieria oprogramowania (ang. *software engineering*) to dziedzina inżynierii systemów zajmująca się wszelkimi aspektami produkcji oprogramowania. Według innych źródeł inżynieria oprogramowania to stosowanie systematycznego, zdyscyplinowanego i kwantyfikowalnego podejścia do wszelkich aspektów wytwarzania oprogramowania. Dokładne określenie czym jest inżynieria oprogramowania jest od wielu lat (a konkretnie od 1969 roku, kiedy to po raz pierwszy oficjalnie użyto tego terminu na konferencji NATO Software Engineering Conference w Garmisch) przedmiotem licznych polemik: David Parnas stwierdził, że jest ona w swej istocie rodzajem inżynierii. Polemizujący z nim Steve McConnell przewrotnie stwierdził, że to nieprawda, ale chciałby żeby stwierdzenie to było prawdziwe (McConnell, 2003).

Potrzeba zdyscyplinowanego podejścia do procesu wytwarzania oprogramowania pojawiła się jako rezultat wielu niepowodzeń na tym polu. Zgodnie z danymi opublikowanymi przez The Standish Group (The Standish Group International, Inc., 2001) w roku 2000 tylko 28% projektów informatycznych zostało zakończonych pełnym sukcesem. Realizacja pozostałych projektów oprogramowania została albo zawieszona (23% ogółu) albo została zrealizowana z przekroczeniem budżetu, zaplanowanego czasu realizacji lub z niepełnym zestawem funkcjonalności (49%). Można tutaj zaobserwować pewną tendencję wzrostową na przestrzeni ostatnich lat, wykazującą poprawę na polu pełnej realizacji projektów oprogramowania: jest ona interpretowana przez twórców raportu jako rezultat coraz

szerszego stosowania zdyscyplinowanych metod zarządzania procesem wytwarzania oprogramowania.



Rysunek 1 Historia realizacji projektów informatycznych w USA w latach 1994-2000. (The Standish Group International, Inc., 2001)

2.2 TRUDNOŚCI TWORZENIA OPROGRAMOWANIA

Istnieje wiele przyczyn niepowodzeń przedsięwzięć wytwarzania oprogramowania. Wszystkie są rezultatem tego, że oprogramowanie komputerowe jest czymś nowym dla człowieka. Istnieje zaledwie od kilkudziesięciu lat, który to okres jest zupełnie nieporównywalny przykładowo do tysiącleci, w czasie których rozwijało się budownictwo. Oprogramowanie charakteryzuje się również swego rodzaju „innością", nie pozwalającą na bezpośrednie jego porównywanie z obiektami realnymi. Podstawowymi cechami wyróżniającymi oprogramowanie są (Stepanek, 2005):

- wysoka złożoność
- abstrakcyjność
- niestałość i niekompletność wymagań
- szybkie zmiany stosowanych technologii
- brak doświadczenia w procesie wytwarzania
- proces wytwarzania jest procesem ciągłych badań
- ostateczna wersja produktu jest formą pełnego projektu

- proces zmian jest łatwy, a zmiany są nieuniknione

Oprogramowanie jest produktem wysoce złożonym. Wyróżnia się w nim komponenty implementowane przy pomocy instrukcji, które można opisać jako najbardziej podstawowe elementy oprogramowania. Komponenty i instrukcje mogą wpływać na siebie nawzajem, tworząc wysoce skomplikowaną sieć współzależności, będącą według niektórych badaczy najbardziej złożonym dziełem ludzkiego umysłu.

Oprogramowanie jest produktem abstrakcyjnym. Nie można go dotknąć czy obejrzeć, a ze względu na jego złożoność bardzo trudno ogarnąć je umysłem. Istnieje wiele metod pozwalających na przedstawienie oprogramowania jako zestawu komponentów istniejących na określonym poziomie abstrakcji, lecz to za mało by pojąć oprogramowanie jako kompletny opis zachowań.

Komputer jest uniwersalnym urządzeniem wspomagającym pracę przedstawicieli dowolnego zawodu, a co za tym idzie, również oprogramowanie posiada taką cechę. Specjaliści danej profesji dla których tworzone są rozwiązania informatyczne bez wątpienia wiedzą o wiele więcej na temat swoich potrzeb (wyrażanych jako wymagania) niż osoby tworzące owo oprogramowanie. Bardzo często jednakże nie są oni w stanie precyzyjnie ich określić na wstępnych etapach realizacji oprogramowania. Jest to rezultatem tego, że praktycznie niezależnie od dziedziny zainteresowań, modelowane procesy w całościowym podejściu do problemu są wysoce skomplikowane, zawierają mnóstwo alternatywnych procedur postępowania i różnych sposobów obsługi sytuacji wyjątkowych. Aby ogarnąć całość takiego procesu należy przeprowadzić bardzo dokładną analizę problemu, a i to (jak wynika z praktyki) nie zawsze rozwiązuje wszystkie problemy. Dlatego najrozsądniejszym rozwiązaniem jest ścisła współpraca między ekspertem w danej dziedzinie a twórcą oprogramowania: w miarę jak oprogramowanie będzie stawało się coraz bardziej realne to użytkownikowi łatwiej będzie wizualizować sobie całość funkcjonalności i dostrzegać oraz modyfikować potencjalne błędy i niedogodności.

Informatyka jest jedną z najszybciej rozwijających się dzisiaj dziedzin nauki. Jeszcze kilkanaście lat temu mało kto słyszał czy używał systemów informatycznych z graficznymi interfejsami użytkownika. Dzisiaj praktycznie każdy może w domu dokonywać edycji materiału wideo czy też poprzez Internet łączyć się z dowolnym miejscem na globie, żeby uzyskać informacje na temat nowego przepisu na ciasto czy też szczegółów konstrukcji

bomby atomowej. Technologie informatyczne, które to umożliwiają, zmieniają się tak szybko, że bardzo trudno jest formułować długofalowe plany tworzenia bardziej rozbudowanych form oprogramowania: po prostu za kilka lat może się okazać, że realizowany skomplikowany system przetwarzania można zastąpić relatywnie dużo prostszym algorytmem wykonującym swoje zadanie dwa razy szybciej i dziesięciokrotnie niższym kosztem.

Stosunkowo krótka historia informatyki oraz wysokie tempo zmian powodują, że niewiele jest praktyk, które można by uznać za dojrzałe. Taka wiedza nie zdążyła się po prostu nagromadzić i usystematyzować. Nie oznacza to oczywiście, że w tej dziedzinie nie ma żadnego postępu: znakomitym tego potwierdzeniem jest chociażby wprowadzenie i ugruntowanie pojęcia wzorców projektowych (ang. *design patterns*) w pracy „Gangu Czterech" (Gamma, Helm, Johnson, & Vlissides, 1995). Podobnie, jeśli nie gorzej, ma się stałość konkretnych technologii informatycznych stosowanych w inżynierii oprogramowania, które podlegają ciągłej aktualizacji, żeby dotrzymać kroku konkurencji, ciągle zmieniającym się parametrom infrastruktury sprzętowej oraz nieustannie rozwijającej się informatycznej wiedzy teoretycznej.

Tworzenie oprogramowania jest procesem rozwoju. W jego skład wchodzą nieustannie przeprowadzane badania, z jednej strony dotyczące funkcjonalności realizowanego przedsięwzięcia, a z drugiej strony dotyczące nowo stosowanych technologii informatycznych. Rzadko który twórca programowania jest specjalistą w dziedzinie, w której tworzone oprogramowanie ma mieć zastosowanie. Jeśli już nawet tak jest, to problemem jest zwykle dostosowanie programu do konkretnych wymagań użytkownika końcowego. Dlatego też konieczna jest (wspomniana już wcześniej) ścisła współpraca między odbiorcą oprogramowania a jego twórcą, sprawiającą, że proces badań wymagań użytkownika jest najsprawniejszy. Badaniom muszą podlegać również kwestie techniczne tworzonego oprogramowania, które nigdy nie istnieje w oderwaniu od innych elementów środowiska sprzętowo-programowego. Jest to szczególnie istotne w dobie tak intensywnego rozwoju technologii informatycznych, jakimi cechują się nasze czasy.

Jak już wcześniej wspomniano, proces wytwarzania oprogramowania jest ciągłym procesem badań, które mają na celu coraz to dokładniejsze dostosowanie oprogramowania do wymagań użytkownika oraz środowiska sprzętowo-programowego. W takich okolicznościach trudno jest utrzymać spójność projektu z jego realizacją (implementacją), bo proces tworzenia oprogramowania jest w swej istocie procesem stopniowego uszczegóławia-

nia (ang. *gradual refinement*). W tym ujęciu (powszechnie przyjmowanym, np. (McConnell, 2006)) kompletnym i maksymalnie dokładnym projektem oprogramowania jest jego realizacja.

Już żyjący w starożytności Heraklit z Efezu twierdził, że jedynym stałym bytem jest zmiana (Wikiquote.org: Heraclitus). Zdanie to jest również prawdziwe dla oprogramowania. W sytuacji nieustannie zmieniających się wymagań użytkownika proces powstawania oprogramowania komputerowego musi przewidywać zmiany i być na nie odporny. Zmiana w realiach procesu wytwarzania jest stosunkowo łatwa: w końcu to tylko kilka linijek kodu źródłowego czy dodatkowe okno graficznego interfejsu użytkownika. Nie należy jednak bagatelizować jej znaczenia, gdyż, za Heraklitem, nie da się jej uniknąć. Ponadto każda zmiana może doprowadzić do poważnych konsekwencji, których waga ściśle się wiąże z poziomem abstrakcji, na którym zostanie wprowadzona: różnica pomiędzy zmianą wprowadzoną do specyfikacji wymagań może być kolosalna w porównaniu do zmiany wprowadzonej w implementacji funkcji realizującej konkretne działanie. Wprowadzając zmianę należy każdorazowo wziąć pod uwagę zależności pomiędzy poszczególnymi komponentami oprogramowania, co potencjalnie może być zadaniem nietrywialnym.

2.3 STANDARDY USPRAWNIANIA PROCESU WYTWARZANIA OPROGRAMOWANIA

Cytowany już raport The Standish Group (The Standish Group International, Inc., 2001) wymienia podstawowe czynniki warunkujące sukces procesów wytwarzania oprogramowania. Zostały one zgromadzone w załączonej tabeli (patrz Tabela 1).

Rezultaty tych badań pokazują, że powodzenie realizacji projektu oprogramowania jest zależne od kilku istotnych czynników. Największe znaczenie zdają się mieć czynniki związane z procesem zarządzania (reprezentowanym tutaj przez wsparcie kierownictwa, doświadczenie kierownika projektu, formalne metodologie), bo to dzięki nim oprogramowanie jest rozwijane w sposób planowy. Sprawne zarządzanie wpływa również pośrednio na inne elementy wymienione w opracowaniu, takie jak ustalanie jednoznacznych celów, niezmienność wymagań i minimalizacja zakresu. Bardzo istotne (jeśli nie kluczowe) znaczenie ma zaangażowanie użytkownika w proces wytwarzania oprogramowania: w ten sposób przyszły odbiorca jest w stanie kontrolować postęp i zakres prac. Warte podkreśle-

nia (z punktu widzenia tematyki niniejszej pracy) jest poczesne miejsce szacowania (estymacji).

Tabela 1 Czynniki sukcesu projektu oprogramowania

czynnik sukcesu	ważony wpływ [%]
Wsparcie kierownictwa	18
Zaangażowanie użytkownika końcowego	16
Doświadczony kierownik projektu	14
Jasne i jednoznaczne cele	12
Minimalizacja zakresu funkcjonalności	10
Standardowa struktura informatyczna	8
Niezmienne podstawowe wymagania	6
Formalna metodologia	6
Wiarygodne oszacowania	5
Inne	5

Źródło: (The Standish Group International, Inc., 2001)

Bazując na rezultatach tych i podobnych badań ośrodki naukowe na całym świecie zaczęły pracować nad sposobami zapewnienia sukcesu przedsięwzięciom realizacji oprogramowania. Wyłaniający się z tych badań obraz dość jednoznacznie wskazuje kierunek, w jakim należy podążać: ulepszanie procesu zarządzania. Sprawne zarządzanie jest istotne przy rozwoju dowolnego typu oprogramowania, a staje się tym bardziej krytyczne, im większy jest realizowany projekt.

W ciągu ostatnich kilkunastu lat na całym świecie zawiązał się cały szereg różnych inicjatyw mających na celu usprawnienie procesu wytwarzania oprogramowania (ang. *software process improvement*). W ich rezultacie powstały dwa główne standardy zapewniania jakości oraz usprawniania procesów produkcyjnych (Persse, 2006):

- standard ISO 9001:2000
- zintegrowany model potencjału i dojrzałości oprogramowania CMM-I

Wszystkie te podejścia zostaną opisane pokrótce w następnych podrozdziałach ze szczególnym naciskiem położonym na uwidocznienie miejsca i charakterystyki szacowania (estymacji).

2.3.1 Standard ISO 9001:2000

Standard ISO 9001:2000 to oficjalna norma Międzynarodowej Organizacji Normalizacyjnej zawierająca wymagania systemu zarządzania jakością dla organizacji potrzebujących wykazać zdolność do ciągłego dostarczania wyrobów zgodnych z wymaganiami klienta i dążenia do zwiększenia zadowolenia klienta (ISO, 2009). Norma ta zastępuje istniejące wcześniej standardy (m.in. ISO 9002, ISO 9003), łącząc to co w nich najistotniejsze i najlepsze w jeden spójny standard. Aby dana organizacja mogła być uznana za spełniającą normę ISO 9001:2000 to musi się ona poddać procesowi certyfikacji realizowanym przez niezależne organizacje certyfikujące.

Standard ISO 9001:2000 wyróżnia 8 podstawowych zasad jakości (ISO, 2009):

- zorientowanie na klienta, gdyż to klient decyduje o pozycji organizacji na rynku
- przywództwo, bo to kierownictwo organizacji wypracowuje kierunki jego rozwoju
- zaangażowanie ludzi, bo to ludzie są najcenniejszą częścią majątku organizacji
- podejście procesowe, gdyż to ono w największej mierze wpływa na skuteczność i efektywność organizacji
- systemowe podejście do zarządzania, bo zarządzanie jakością winno być traktowane jako zarządzanie wzajemnie powiązanymi procesami
- ciągłe doskonalenie się jako stały cel organizacji
- rzeczowe podejście do podejmowania decyzji w oparciu o szczegółowe analizy faktów
- wzajemne korzyści w stosunkach z dostawcami, bo wzajemna korzyść w stosunkach z innymi organizacjami jest gwarantem wysokiej jakości

W ramach standardu zdefiniowano szczegółowo proces jego wdrażania, obejmujący takie zadania, jak przykładowo stworzenie systemu zarządzania jakością, stworzenie księgi jakości, czyli w ogólności przygotowanie infrastruktury pozwalającej organizacji na skuteczne zarządzanie. Finalnym elementem standardu jest opis realizacji wyrobu. Składa się on z następujących elementów (Persse, 2006):

- planowanie realizacji produktu
- procesy związane z klientem
- projektowanie i prace rozwojowe
- zakupy

- operacje produkcyjne i rozwojowe
- nadzorowanie infrastruktury do monitorowania i pomiarów

Standard ISO 9001:2000 podkreśla szczególną rolę planowania prac rozwojowych jako kluczowych elementów procesu wytwarzania produktu. Tworzony w jego ramach plan projektu jest podstawowym narzędziem zarządzania dla kontroli realizacji procesu wytwarzania, monitorowania postępu oraz zapewnia punkty kontrolne dla potwierdzenia poprawności całościowego kierunku realizacji produktu i ewentualnych korekt.

Standard sugeruje tworzenie planów na różnych poziomach zarządczych poprzez dekompozycję procesu realizacji produktu. Rezultatem takiego podziału są najczęściej etapy rozwoju. Standard wymaga, aby w ramach tych etapów realizowane były akcje kontrolne w postaci przeglądów, weryfikacji i walidacji. Elementy te są najczęściej realizowane z wykorzystaniem procesów pomiaru wszelakich wielkości związanych z procesem wytwarzania, jak i z samym produktem. Dane zebrane w pomiarach są kluczowym elementem dla sprzężenia zwrotnego kontroli i stanowią podstawowe źródło wszelkich decyzji związanych z zarządzaniem jakością.

2.3.2 Zintegrowany model potencjału i dojrzałości oprogramowania CMM-I

Historia zintegrowanego modelu potencjału i dojrzałości oprogramowania CMM-I (ang. *Capability Maturity Model Integration*) rozpoczęła się od rozwijanego w latach 1987-1997 modelu dojrzałości oprogramowania CMM, który następnie był kilkukrotnie uaktualniany oraz rozszerzany aż do osiągnięcia w sierpniu 2006 wersji 1.2. Model CMM-I jest aktywnie rozwijany przez Software Engineering Institute na uniwersytecie Carnegie Mellon w Pensylwanii, USA. Może on być swobodnie implementowany przez dowolne organizacje, w przeciwieństwie, przykładowo, do standardów międzynarodowych ISO. W chwili obecnej jest on prawdopodobnie najbardziej popularnym podejściem procesowym do usprawniania wytwarzania oprogramowania (Persse, 2006).

Model CMM-I jest zbudowany jako zbiór obszarów procesowych (ang. *Process Area*) rozumianych jako kolekcja najlepszych praktyk pozwalających organizacji na zarządzanie swoją działalnością i kontrolowanie jakości. Obszary procesowe w pełnym modelu wyszczególnione są w załączonej tabeli (patrz Tabela 2).

Tabela 2 Obszary procesowe modelu CMM-I

kategoria	obszar procesowy
zarządzanie projektem	planowanie projektu
	monitorowanie i kontrola projektu
	zintegrowane zarządzanie projektem
	ilościowe zarządzanie projektem
	zarządzanie ryzykiem
	zarządzanie umowami z dostawcami
inżynieria	zarządzanie wymaganiami
	rozwój wymagań
	weryfikacja
	walidacja
	rozwiązania techniczne
	integracja produktu
wsparcie	zapewnienie jakości produktowi i procesowi
	zarządzanie konfiguracjami
	pomiary i analizy
	analiza i podejmowanie decyzji
	analiza przyczyn
zarządzanie procesem	koncentracja na procesie
	definicje procesów
	Szkolenia
	wydajność procesów
	innowacje i wdrożenia

Źródło: (Persse, 2006)

W ramach każdego z obszarów procesowych stawiane są do osiągnięcia pewne ogólne i specyficzne cele, które są wspierane opisem dobrych praktyk wspomagających użytkownika. Cele ogólne są wspólne dla wszystkich obszarów procesowych, a specyficzne mają sens tylko w kontekście danego obszaru procesowego. Stopień realizacji celów może być charakteryzowany dwojako, albo w sposób ciągły (ang. *continuous representation*) albo w sposób stopniowany (ang. *staged representation*).

Reprezentacja ciągła polega na ocenie każdego obszaru procesowego w sześciostopniowej skali (patrz Tabela 3). Przy jej użyciu można arbitralnie dobrać zestaw obszarów procesowych jakie mają być charakteryzowane, co pozwala na precyzyjne stworzenie profilu danej organizacji.

Tabela 3 Skala reprezentacji ciągłej

poziom	opis
Poziom 0: niekompletny	cele przyporządkowane do obszaru nie są realizowane
Poziom 1: wykonywany	realizacja celów zależy bardziej od pojedynczych osób niż od organizacji jako całości
Poziom 2: zarządzany	realizacja celów opiera się na wcześniej ustalonym planie
Poziom 3: zdefiniowany	wdrożona polityka normalizacji procesów
Poziom 4: zarządzany ilościowo	kontrola za pomocą metod statystycznych i technik ilościowych
Poziom 5: optymalizowany	ciągła adaptacja to zmieniających się celów i strategii organizacji

Źródło: (Clever Age sp. z o.o., 2005)

Reprezentacja stopniowana służy do oceny dojrzałości organizacji jako całości za pomocą pięciostopniowej skali (patrz Tabela 4). W tym podejściu aby móc zadeklarować, że organizacja jest na danym poziomie dojrzałości to jest wymagane, aby realizowała ona wszystkie cele w określonych obszarach procesowych.

Tabela 4 Skala reprezentacji stopniowanej

poziom	opis
Poziom 1: początkowy	procesy są nieprzewidywalne i niekontrolowalne
Poziom 2: zarządzany	ustalono procedury dla każdego projektu osobno
Poziom 3: zdefiniowany	ustalono procedury na poziomie organizacji
Poziom 4: zarządzany ilościowo	ustalono listę celów ilościowych i jakościowych oraz sposoby i narzędzia ich kontroli
Poziom 5: optymalizowany	ciągłe usprawnianie procesów

Źródło: (Clever Age sp. z o.o., 2005)

Zgodnie z ogólnie przyjętymi poglądami rzetelne jakościowo wyniki mogą być uzyskane tylko wtedy, gdy planowanie zostało przeprowadzone jawnie i dokładnie (Persse, 2006). Dlatego też wielu badaczy uważa, że planowanie jest jednym z kluczy powodzenia

przedsięwzięć realizacji oprogramowania. W ramach modelu CMM-I dla planowania projektu należy zrealizować następujące cele (CMMI Product Team, 2006):

- wyznaczyć oszacowania wymaganych parametrów projektu (rozmiar, nakład pracy, koszt), ustalić ostateczny zakres projektu, zdefiniować model cyklu życia oprogramowania.
- stworzyć i udokumentować plan bazując na wcześniej uzyskanych oszacowaniach, ustalić budżet i harmonogram, zidentyfikować potencjalne niebezpieczeństwa
- przeprowadzić konsultacje i zaakceptować plan przez wszystkie zainteresowane strony.
- uaktualniać i utrzymywać plan.

Obszar procesowy planowania projektu nie jest zamykany po ustaleniu początkowej wersji planu. Ze względu na dużą zmienność charakteryzującą proces wytwarzania oprogramowania plan projektu musi być uaktualniany za każdym razem, gdy zmianie ulegną wymagania co do produktu, uzyskane wcześniej oszacowania okażą się być błędne, powstanie potrzeba wprowadzenia akcji korygujących lub zmian procesu (CMMI Product Team, 2006).

Warto podkreślić tutaj, że planowanie projektu (oraz realizowane w jego ramach oszacowania) jest bezwzględnie wymagane, aby móc określić organizację jako zarządzaną (poziom 2, najniższy) w sensie stopniowanej reprezentacji oceny CMM-I.

O skuteczności zintegrowanego modelu potencjału i dojrzałości oprogramowania CMM-I świadczą niedawne badania. Opublikowany w sierpniu 2006 roku raport prezentuje rezultaty wprowadzenia modelu CMM-I w 30 różnych organizacjach (Tabela 5). Wyniki wykazują wyraźnie, że implementacja CMM-I w organizacjach daje pozytywne rezultaty (Gibson, Goldenson, & Kost, 2006).

Tabela 5 Rezultaty implementacji modelu CMM-I

kategoria	mediana zmiany	najmniejsza zmiana	największa zmiana
koszt	34%	3%	87%
harmonogram	50%	2%	95%
produktywność	61%	11%	329%
jakość	48%	2%	132%
zadowolenie klienta	14%	-4%	55%
zwrot z inwestycji	4,0 : 1	1,7 : 1	27,7 : 1

Źródło: (Gibson, Goldenson, & Kost, 2006)

2.4 CYKL ŻYCIA OPROGRAMOWANIA

Realizacja każdego przedsięwzięcia informatycznego składa się z całego szeregu różnych czynności. Ze względu na ich ilość oraz stopień złożoności zarządzanie całością zadań może być bardzo trudnym zadaniem. Dlatego też stosuje się różne metody dekompozycji zadań na prostsze i łatwiejsze do zarządzania składowe, tworząc w ten sposób plan realizacji przedsięwzięcia informatycznego. Niezależnie od sposobu dekompozycji zawsze należy wykonać następujące zadania (Górski, i inni, 2000) (Stachurski, 2002):

- specyfikacja
- analiza
- projektowanie
- implementacja
- testowanie
- konserwacja

Poszczególne fazy zostaną pokrótce opisane w następnych podrozdziałach.

2.4.1 Faza specyfikacji

Faza specyfikacji ma na celu zgromadzenie, zorganizowanie i opracowanie wszystkich dokumentów potrzebnych do opisu fragmentu dziedziny wiedzy, na potrzeby którego realizowane jest oprogramowanie. Na podstawie tak zgromadzonego materiału generowana jest lista wymagań i oczekiwań klienta w stosunku do nowo powstającej aplikacji. Ponadto wszystkie dokumenty zebrane na tym etapie będą również wykorzystywane w etapach

późniejszych, a szczególnie istotne są w fazie następnej (fazie analizy). W ramach tego etapu wykonywane są następujące zadania (Subieta, 2002):

- określenie celów przedsięwzięcia z punktu widzenia klienta
- określenie zakresu oraz kontekstu przedsięwzięcia
- ogólne określenie wymagań
- wykonanie zgrubnej analizy i projektu systemu
- oszacowanie kosztów realizowanego systemu
- określenie wstępnego harmonogramu przedsięwzięcia oraz określenie struktury zespołu go realizującego
- określenie standardów, zgodnie z którymi przedsięwzięcie będzie realizowane

Wyniki fazy specyfikacji to (Subieta, 2002):

- raport obejmujący definicję celów przedsięwzięcia, opis zakresu, opis systemów zewnętrznych, ogólny opis wymagań, ogólny model systemu, opis proponowanego rozwiązania, oszacowanie kosztów, wstępny harmonogram prac
- opis wymaganych zasobów
- definicja wykorzystywanych standardów
- harmonogram realizacji fazy analizy

2.4.2 Faza analizy

Faza analizy ma na celu dokładne rozpoznanie wszystkich aspektów rzeczywistości, które mogą mieć wpływ na postać, organizację lub wynik realizacji przedsięwzięcia informatycznego. W szczególności wykonywane są następujące zadania (Subieta, 2001):

- rozpoznanie, wyjaśnienie, modelowanie, specyfikowanie i dokumentowanie dziedziny realizowanego oprogramowania
- ustalenie kontekstu realizowanego przedsięwzięcia informatycznego
- ustalenie wymagań organizacyjnych

Rezultatem przeprowadzenia etapu analizy powinny być (Subieta, 2002):

- dokument zawierający logiczny model systemu, opisujący sposób realizacji wymagań

- dokument zawierający szczegółowe wymagania względem realizowanego systemu informatycznego
- wstępny plan testów
- harmonogram realizacji fazy projektowania

2.4.3 Faza projektowania

Celem fazy projektowania jest opracowanie szczegółowego planu implementacji systemu. W odróżnieniu od wcześniejszych etapów, tutaj szczególnego znaczenia nabiera środowisko implementacyjne, co bezpośrednio rzutuje na zakres wiedzy wymaganej od projektanta systemu informacyjnego. W ramach fazy projektowania podejmowane są następujące czynności (Subieta, 2002):

- uszczegółowienie wyników analizy
- projektowanie składników systemu nie związanych bezpośrednio z dziedziną problemu
- wysokopoziomowa optymalizacja systemu
- dostosowanie do możliwości i ograniczeń środowiska implementacji
- określenie fizycznej struktury systemu

Podstawowe rezultaty fazy projektowania to (Subieta, 2002):

- poprawiony i uszczegółowiony dokument opisujący wymagania
- poprawiony i uszczegółowiony model systemu (opis struktury, schemat bazy danych, opis interfejsów użytkownika itd.)
- poprawiony plan testów
- harmonogram fazy implementacji

2.4.4 Faza implementacji

Celem fazy implementacji jest stworzenie kodu źródłowego realizującego zachowanie systemu informatycznego, a następnie sprawdzenie jego bezbłędności oraz zgodności ze specyfikacją. Rezultatem działań podjętych w ramach fazy implementacji są (Subieta, 2002):

- poprawiony i uszczegółowiony dokument opisujący wymagania

- poprawiony i uszczegółowiony model analityczny
- poprawiony projekt (stanowiący odtąd dokumentację techniczną)
- wstępnie przetestowany kod źródłowy wraz z raportem z testów
- harmonogram fazy testowania

2.4.5 Faza testowania

Podstawowymi celami fazy testowania są wykrycie i usunięcie błędów w systemie oraz ocena niezawodności systemu. Osiąga się je poprzez wykonanie następujących czynności (Subieta, 2002):

- przeglądy techniczne
- sprawdzenie, czy zrealizowane są wymagania klienta
- testowanie jednostek (modułów) systemu informatycznego
- testowanie systemu po integracji modułów
- testy akceptacyjne realizowane przez klienta
- kontrola jakości systemu informatycznego

W wyniku działań podejmowanych w fazie testowania generowane są (Subieta, 2002):

- poprawiony kod źródłowy, projekt, model, specyfikacja wymagań
- raport przebiegu testów zawierający informację o przeprowadzonych testach i ich rezultatach
- oszacowanie niezawodności oprogramowania i kosztów konserwacji

2.4.6 Faza konserwacji

Głównym celem fazy konserwacji jest wprowadzenie modyfikacji do już istniejącego i działającego systemu informatycznego. Modyfikacje te można podzielić na trzy podstawowe klasy (Subieta, 2002):

- modyfikacje poprawiające polegające na usuwaniu z oprogramowania błędów popełnionych na wcześniejszych etapach rozwoju oprogramowania
- modyfikacje poprawiające polegające na poprawie jakości oprogramowania

- modyfikacje dostosowujące polegające na adaptacji oprogramowania do zmian zachodzących w wymaganiach użytkownika lub w środowisku sprzętowym

Rezultatem fazy konserwacji są poprawiony kod źródłowy, projekt, model i specyfikacja wymagań.

2.5 MODELE CYKLU ŻYCIA OPROGRAMOWANIA

W ciągu minionych kilkudziesięciu lat rozwoju inżynierii oprogramowania stworzono wiele różnych modeli cyklu życia oprogramowania. Wszystkie z nich opisują jakie czynności należy podjąć, aby oprogramowanie zostało zrealizowane (czynności te zostały opisane w poprzednim podrozdziale) oraz charakterystykę przebiegu tych czynności, a w szczególności porządek w jakim powinny być realizowane te czynności oraz kryteria decyzji, czy dana czynność może być zakończona (McConnell, 1996).

W poniższych podrozdziałach pokrótce opisano najważniejsze i najbardziej popularne modele cyklu życia oprogramowania.

2.5.1 Model kaskadowy

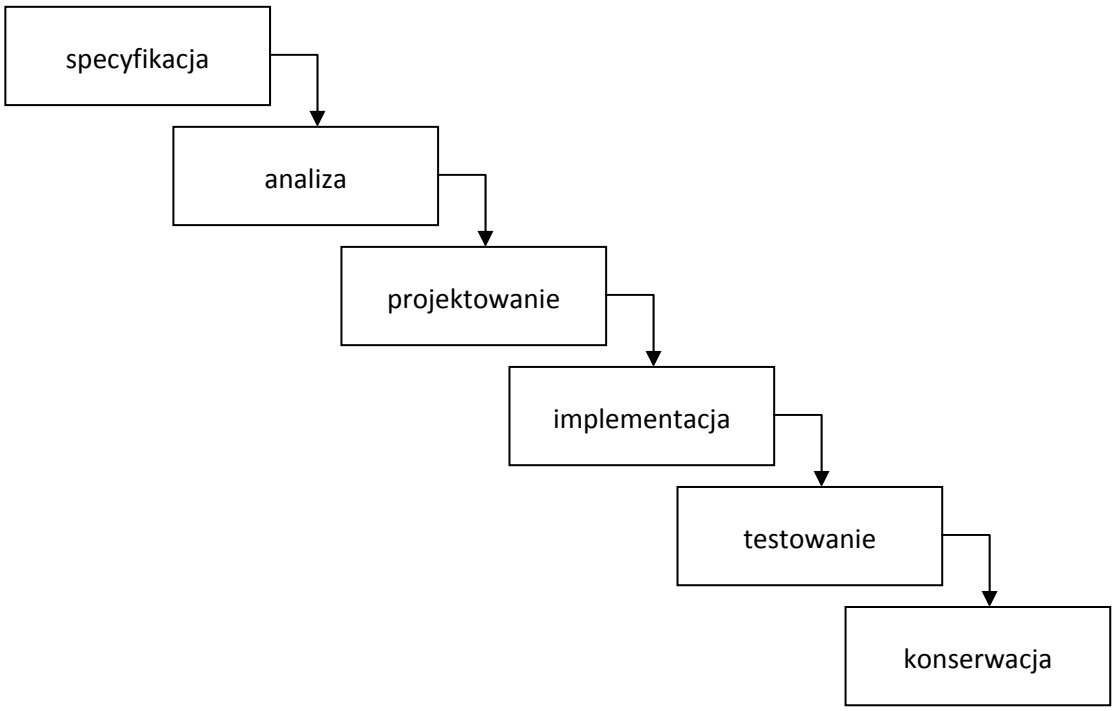
Model kaskadowy (ang. *waterfall model*) jest najstarszym modelem cyklu życia oprogramowania. Został on po raz pierwszy przedstawiony w 1970 roku przez Royce'a (Royce, 1970) jako koncepcja modelu cyklu życia oprogramowania służąca jako podstawa do przedstawianych w tym samym artykule ulepszeń. Ciekawostką jest to, że większość ówczesnych badaczy tematu zignorowała rozszerzenie modelu kaskadowego autorstwa Royce'a (które w swej istocie było modelem spiralnym) i uznała niedoskonały model kaskadowy jako główne przesłanie jego publikacji (McConnell, 1996). Ten podrozdział opisuje model kaskadowy w jego ogólnie przyjętym znaczeniu.

Podstawową ideą klasycznego modelu kaskadowego jest realizacja oprogramowania w sekwencyjnym procesie składającym się z następujących etapów (Górski, i inni, 2000)(McConnell, 1996):

- specyfikacja
- analiza

- projektowanie
- implementacja
- testowanie
- konserwacja

Istotą modelu kaskadowego jest to, że przejście do kolejnego etapu może być realizowane tylko i wyłącznie wtedy, gdy poprzedni etap został zakończony (fazy są rozłączne, patrz Rysunek 2).



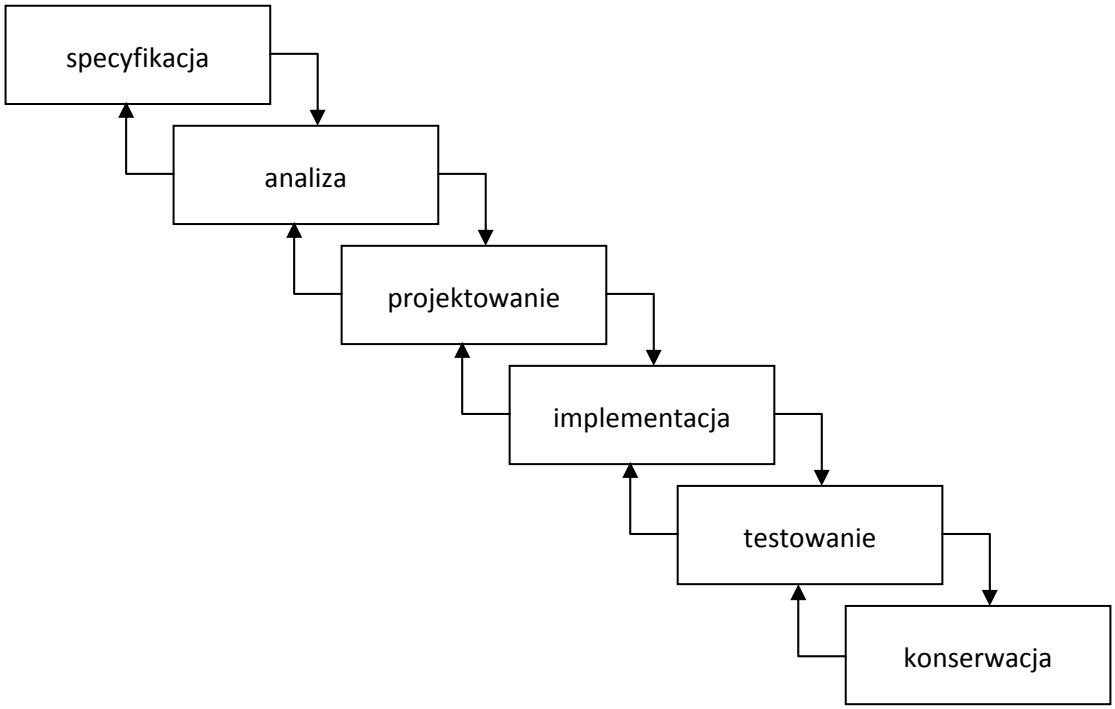
Rysunek 2 Model kaskadowy (opracowanie własne)

Model kaskadowy w dużym stopniu opiera się i jest kierowany dokumentami (ang. *dokument driven*) – rezultatem każdej fazy są przede wszystkim wszelkiego rodzaju dokumenty opisujące działania w ramach konkretnej fazy.

Klasyczny model kaskadowy sprawdza się przede wszystkim w przypadkach, gdy powstający produkt informatyczny ma stabilną definicję wymagań i jest realizowany dobrze zrozumianymi metodologiami. W przypadku takich projektów model kaskadowy ułatwia wyszukanie błędów na wczesnych (a przez to mniej kosztownych) etapach rozwoju oprogramowania. Jest również sporą pomocą w przypadku niedoświadczonego zespołu realizatorów, bo wszystkie prace są przeprowadzane w sposób wysoce zorganizowany i zdyscyplinowany, co pozwala na minimalizację marnotrawienia nakładów pracy.

Wady modelu kaskadowego wynikają przede wszystkim z trudności dokładnego ustalenia wymagań na samym początku procesu wytwarzania. Sytuacja, gdy wymagania są kompletne już na samym początku realizacji przedsięwzięcia informatycznego jest bardzo rzadka i dotyczyć może tylko niewielkich projektów. Ponadto charakterystyczną cechą modelu kaskadowego jest długi okres czasu pomiędzy rozpoczęciem prac a pojawieniem się interesujących z punktu widzenia klienta efektów: duża część nakładów pracy zespołu realizatorskiego dotyczy etapów koncepcyjnych, które dla klienta są mało interesujące.

W wyniku pojawiającej się krytyki pojawiły się zmodyfikowane wersje klasycznego modelu kaskadowego eliminujące konieczność realizacji etapów rozwoju oprogramowania w tylko jednym kierunku (*vide* Rysunek 2). Model kaskadowy z iteracjami (finalny model Royce'a) pozwalał na dwukierunkowe poruszanie się na ścieżce etapów rozwoju (patrz Rysunek 3), co znacząco poprawiało elastyczność klasycznego modelu kaskadowego.



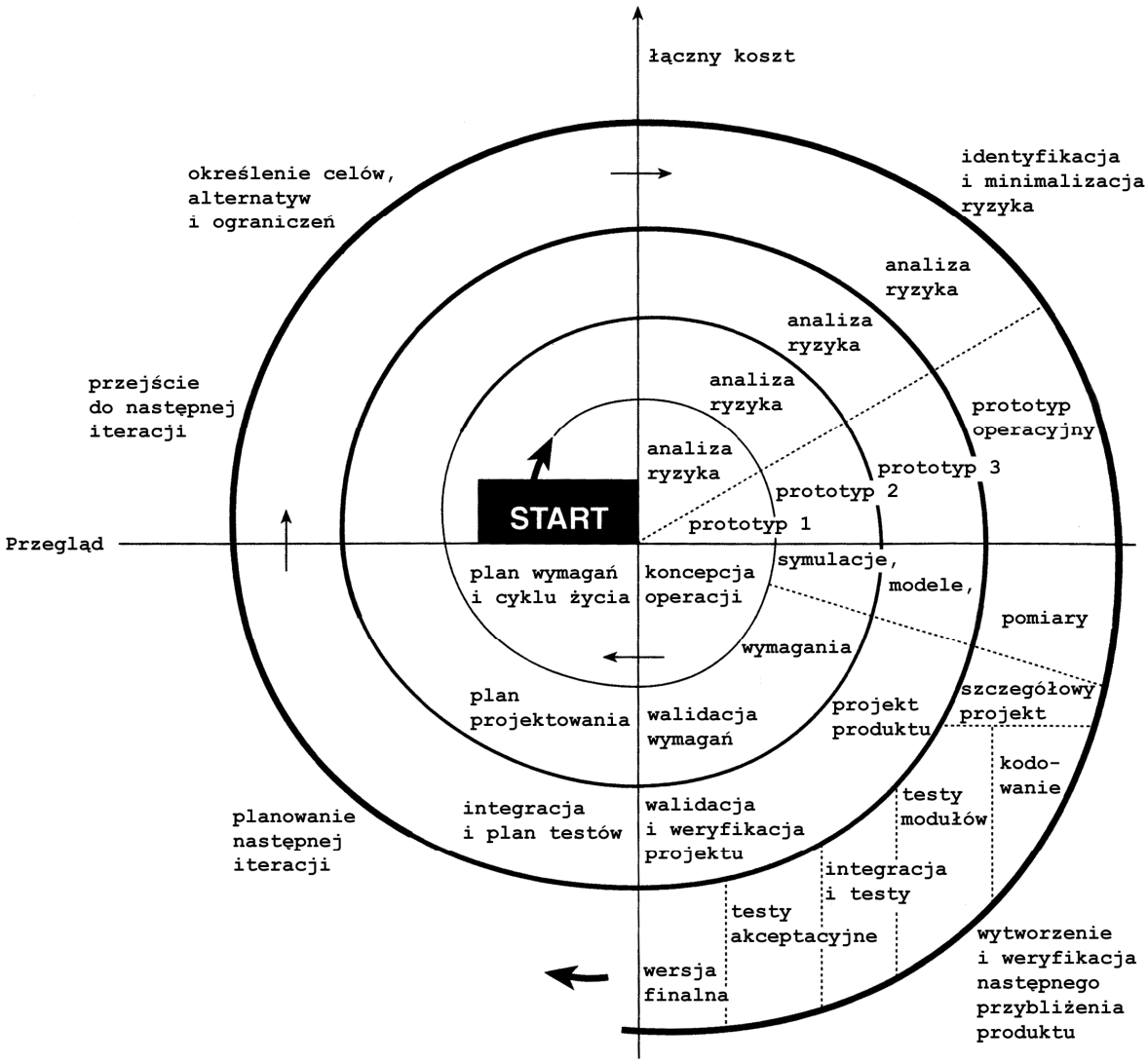
Rysunek 3 Model kaskadowy z iteracjami (opracowanie własne)

2.5.2 Model spiralny

Model spiralny (ang. *spiral model*) został zdefiniowany przez Boehma w 1988 roku (Boehm, 1988). Nie był to pierwszy model z rodziny modeli iteracyjnych (już zmodyfiko-

wany model Royce'a opisywany wcześniej należał do tej kategorii), ale był to pierwszy model, który opisywał znaczenie iteracji w procesie rozwoju oprogramowania.

Model spiralny jest modelem cyklu życia oprogramowania zorientowanym na ryzyko, w ramach którego realizowany projekt oprogramowania jest dzielony na podprojekty. Celem każdego z podprojektów jest zminimalizowanie bądź usunięcie pewnego rodzaju ryzyka. Pojęcie ryzyka jest tutaj rozumiane wieloznacznie, na przykład jako słabo zdefiniowane wymagania, architektura, potencjalne problemy wydajnościowe, problemy wynikające z zastosowanej technologii. Każdy podprojekt może być zobrazowany jako jedna iteracja (patrz Rysunek 4).



Rysunek 4 Model spiralny (McConnell, 1996)

W ramach każdego podprojektu (iteracji) realizowane są następujące kroki (McConnell, 1996)(Górski, i inni, 2000):

- określenie celów, alternatyw i ograniczeń
- identyfikacja i eliminacja ryzyka
- ocena rozwiązań alternatywnych
- wytworzenie i weryfikacja kolejnego przybliżenia produktu
- planowanie i przejście do następnej iteracji (jeśli takowa ma wystąpić)

Im wcześniejsza jest iteracja, tym mniejszy jest koszt związany z jej realizacją: definiowanie wymagań jest mniej kosztowne niż tworzenie modelu, tworzenie modelu jest mniej kosztowne niż projektowanie oprogramowania i tak dalej.

Jedną z najbardziej istotnych cech modelu spiralnego jest to, że wraz ze wzrostem kosztów tworzenia systemu zmniejsza się poziom ryzyka. Ponadto zapewnia on co najmniej taki sam poziom kontroli realizacji projektu jak to jest w przypadku modelu kaskadowego. Z kolei jego zorientowanie na ryzyko pozwala na dostrzeżenie i eliminację poważnych przeszkód uniemożliwiających realizację projektu na bardzo wczesnych etapach jego rozwoju. Model spiralny jest bardziej odporny na zmiany pojawiające się w trakcie realizacji projektu; na przykład jeśli pojawią się nowe wymagania to zostaną one po prostu zrealizowane w jednej z następnych iteracji procesu. Niewątpliwą zaletą jest również to, że pierwsze efekty pracy w postaci działających fragmentów realizowanego systemu pojawiają się stosunkowo wcześnie.

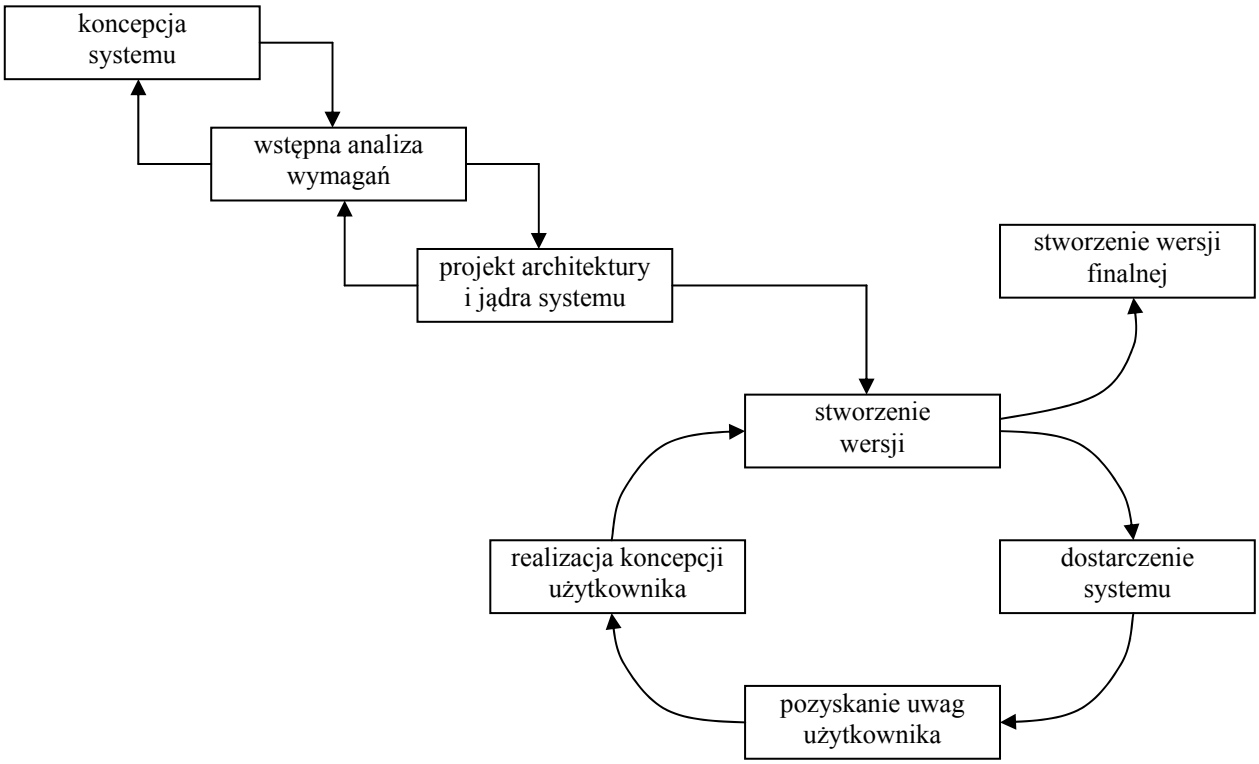
Najpoważniejszą wadą modelu spiralnego jest spory poziom jego komplikacji. Wymaga on bardzo uważnego, sumiennego i starannego zarządzania czynnościami projektowymi.

2.5.3 Model ewolucyjny

Jednym z podstawowych problemów tradycyjnego kaskadowego modelu cyklu życia systemu informatycznego jest znaczne opóźnienie między stworzeniem specyfikacji a dostarczeniem finalnego produktu. Rozwiązaniem tej niedogodności może być dostarczanie produktu w porcjach, będącymi kolejnymi przyrostami jego funkcjonalności, co jest podstawą modelu ewolucyjnego (ang. *evolutionary delivery*, patrz Rysunek 5).

Podstawową zaletą tego modelu jest bardzo wczesne wprowadzenie użytkownika w obsługę projektowanego systemu. Pozwala to na jego stały udział w tworzeniu produktu, umożliwiając mu proponowanie zmian w specyfikacji, mających na celu dostarczenie rozwiązania jak najbardziej bliskiego ideału.

Ceną tego udogodnienia jest szczególnie trudne zarządzania tworzonym projektem. Jednym z zadań zarządzania jest zatwierdzanie propozycji zmian w powstającym systemie. Pojawia się tutaj realne zagrożenie wystąpienia braku możliwości realizacji zaakceptowanych zmian ze względu na ich nadmiar. Dlatego też zarządzanie w przypadku modelu ewolucyjnego jest kwestią kluczową dla sprawnej realizacji zadania wytwarzania systemu.



Rysunek 5 Model ewolucyjny (McConnell, 1996)

2.5.4 Montaż z gotowych elementów

Metoda ta (ang. *design-to-tools*) wiąże się ściśle z postępem w tworzeniu coraz to bardziej uniwersalnych i elastycznych narzędzi, takich jak kompletne struktury aplikacji, wizualne środowiska programistyczne, rozbudowane środowiska programowania baz danych.

Polega ona na dostarczaniu w ramach tworzonego produktu tylko tych funkcji, które są zawarte w dostępnych na rynku narzędziach czy bibliotekach. Implikuje to fakt, że nie będzie możliwe dostarczenie pełnej funkcjonalności wymaganej przez przyszłego użytkownika systemu.

Podstawową zaletą tego podejścia jest bardzo krótki czas tworzenia systemu w porównaniu z sytuacją, gdy funkcjonalność byłaby zapewniana na drodze opracowywania jej od podstaw.

Wadą tego rozwiązania (obok wspomnianej już wcześniej niemożności dostarczenia pełnego wymaganego zakresu funkcji) jest niewielki stopień kontroli nad tworzonym produktem. Korzystanie z rozwiązań komercyjnych prowadzi do uzależnienia od ich producentów, ich strategii dotyczących danego produktu oraz finansowej stabilności.

2.5.5 Rozważania dotyczące wyboru modelu cyklu życia

Różne projekty mają różne wymagania, nawet wtedy, gdy wszystkie mają być wykonane tak szybko jak to możliwe. Implikacją tego faktu jest niemożność jednoznacznego wskazania modelu, który cechowałby się pełną uniwersalnością zastosowania. Wybierając model cyklu życia dla określonego projektu należy najpierw odpowiedzieć na szereg pytań go dotyczących (McConnell, 1996):

- Jak dobrze producent i klient rozumieją wymagania względem projektu na początku jego realizacji? W jakim stopniu możliwe jest, że wymagania te ulegną zmianie w trakcie realizacji?
- Jak dobrze producent systemu rozumie jego architekturę? Czy możliwe jest, że w trakcie realizacji systemu ulegnie ona poważnym zmianom?
- Jaki poziom niezawodności jest wymagany?
- Jak dużo planowania i projektowania wymaga dany systemu w celu wytworzenia jego przyszłych wersji?
- Jak wielkie ryzyko pociąga za sobą realizacja danego systemu?
- Czy producent jest ograniczony predefiniowanym harmonogramem?
- Czy wymagana jest możliwość producenta do dokonywania poprawek w trakcie realizacji projektu?

- Czy od wytwórcy wymagana jest zdolność do zaprezentowania klientowi stanu zaawansowania prac nad projektowanym systemem?
- Czy od wytwórcy wymagana jest zdolność do zaprezentowania kierownictwu stanu zaawansowania prac nad projektowanym systemem?
- Jak wiele skomplikowanych kwestii teoretycznych wymaganych jest do sprawnego wykorzystania danego modelu cyklu życia?

Po udzieleniu odpowiedzi na te pytania należy skonfrontować je z poniższą tabelą (Tabela 6) w celu określenia, który model cyklu życia systemu informatycznego będzie najbardziej adekwatny do zastosowania w przypadku konkretnego projektu.

Tabela 6 Kryteria stosowalności modeli cyklu życia oprogramowania

cecha modelu cyklu życia	**model kaskadowy**	**model spiralny**	**model ewolucyjny**	**montaż z gotowych elementów**
działa ze słabo zrozumianymi wymaganiami	słabo	doskonale	przeciętnie do doskonale	przeciętnie
działa ze słabo zrozumianą architekturą	słabo	doskonale	słabo	słabo do doskonale
wytwarza system o wysokiej niezawodności	doskonale	doskonale	przeciętnie do doskonale	słabo do doskonale
tworzy system o wysokiej elastyczności względem zmian w jego strukturze	doskonale	doskonale	doskonale	słabo
zarządza ryzykiem	słabo	doskonale	przeciętnie	słabo do przeciętnie
może być przystosowany do predefiniowanego harmonogramu	przeciętnie	przeciętnie	przeciętnie	doskonale
ma niski poziom prac nadmiarowych	słabo	przeciętnie	przeciętnie	przeciętnie do doskonale
pozwala na wprowadzanie poprawek w trakcie realizacji	słabo	przeciętnie	przeciętnie do doskonale	doskonale
umożliwia przedstawienie postępów klientowi	słabo	doskonale	doskonale	doskonale
umożliwia przedstawienie postępów kierownictwu	przeciętnie	doskonale	doskonale	doskonale
wymaga posiadania niebyt rozległej wiedzy specjalistycznej	przeciętnie	słabo	przeciętnie	przeciętnie

Źródło: opracowanie własne na podstawie (McConnell, 1996)

2.6 POMIARY W INŻYNIERII OPROGRAMOWANIA

Pomiary są jednym z najistotniejszych elementów w badaniach dotyczących nauk przyrodniczych. Są one w powszechnym mniemaniu źródłem wszelkich faktów. Już lord Kelvin w 1900 twierdził, że „kiedy potrafisz coś zmierzyć i wyrazić to przy pomocy liczb to wiesz coś o tym, ale kiedy nie potrafisz tego zmierzyć i wyrazić przy pomocy liczb to twoja wiedza jest skąpa i niewystarczająca" (Laird & Brennan, 2006).

Pomiary są również jednym z najistotniejszych elementów wszystkich formalnych metod zarządzania realizacją przedsięwzięcia informatycznego. We wspomnianym już wcześniej zintegrowanym modelu potencjału i dojrzałości oprogramowania CMM-I pełnią one kluczową rolę będąc podstawą procesów planowania i kontroli już na najniższym poziomie oceny przedsiębiorstwa informatycznego. Im wyższy poziom oceny tym znaczenie pomiarów wzrasta aż do poziomu krytycznego, kiedy to wszystkie decyzje są podejmowane na ich podstawie. Bardzo dobrze zostało to ujęte w (Fenton & Pfleeger, 1997), gdzie autorzy twierdzą, że nie można przewidywać ani kontrolować czegoś, czego nie można zmierzyć.

Zgodnie ze współcześnie przyjętymi rezultatami badań pomiary w inżynierii oprogramowania powinny być zgodne z teorią pomiaru (Fenton & Pfleeger, 1998)(Zuse, 1991). Zasadniczym elementem tych prac jest definicja pomiaru (patrz Definicja 1) oraz definicja metryki (patrz **Definicja 2**).

Definicja 1. Pomiar (ang. *measurement*) to proces, w którym atrybutom elementów świata rzeczywistego przypisywane są liczby lub symbole w taki sposób, aby charakteryzowały one te atrybuty według jasno określonych reguł. Jednostki przydzielane atrybutom w ramach tego procesu nazywane są miarą (ang. *measure*) danego atrybutu (Górski, i inni, 2000)(Laird & Brennan, 2006)(Fenton & Pfleeger, 1998)(Munson, 2003).

Definicja 2. Metryka jest to proponowana (postulowana) miara.

O ile identyfikacja atrybutów mierzalnych w świecie rzeczywistym jest stosunkowo łatwa (np. wielkość człowieka może być definiowana przy pomocy miary długości jego wzrostu), o tyle w inżynierii oprogramowania jest to bardziej skomplikowane. Podstawową przyczyną tego jest to, że większość bytów w inżynierii oprogramowania jest bytami abstrakcyjnymi (np. specyfikacje wymagań, projekt, program komputerowy, proces rozwoju

oprogramowania) przez co trudno jest zidentyfikować atrybuty i dopasować do nich odpowiednie miary. Uznaje się, że wszystkie elementy w inżynierii oprogramowania które podlegają pomiarowi można zaklasyfikować jako (Górski, i inni, 2000):

- proces: określone działanie, zbiór działań lub okres czasu w ramach okresu wytwarzania lub eksploatacji oprogramowania (np. formułowanie wymagań, projektowanie, implementacja, testowanie)
- produkt: każdy przedmiot powstały w wyniku procesu (np. specyfikacja wymagań, specyfikacja projektowa, kod źródłowy, plan testów)
- zasób: każdy element niezbędny do realizacji procesu (np. osoby, zespoły ludzi, kompilatory, komputery)

Ponadto zaproponowano rozróżnienie na atrybuty wewnętrzne i zewnętrzne dla powyższych elementów (Górski, i inni, 2000):

- atrybuty wewnętrzne procesu, produktu lub zasobu to takie, które można zmierzyć odnosząc się wyłącznie do tego elementu
- atrybuty zewnętrzne procesu, produktu lub zasobu to takie, które można zmierzyć wyłącznie z uwzględnieniem jak dany element odnosi się do innych elementów ze swego środowiska

Tabela 7 podaje kilka przykładów jak wprowadzone wcześniej pojęcia stosują się do działań związanych z pomiarem oprogramowania.

Bardzo popularna metoda tworzenia modeli pomiarowych została zaproponowana w (Basili, Caldiera, & Rombach, 2002). Metoda ta została nazwana metodą Cel Pytanie Metryka (ang. *Goal Question Metric* – GQM) i składa się z trzech podstawowych poziomów:

- poziom koncepcyjny (cel): definicja celu dla produktu, procesu lub zasobu
- poziom operacyjny (pytanie): definicja zbioru pytań charakteryzujących sposób osiągnięcia celu i charakteryzujących sam obiekt podlegający pomiarowi (produkt, proces lub zasób)
- poziom ilościowy (metryka): każdemu pytaniu przypisywany jest zbiór danych stanowiący ilościową odpowiedź

Metoda ta pozwala w prosty i usystematyzowany sposób określić jakie metryki powinny być używane do mierzenia określonych atrybutów oprogramowania.

Tabela 7 Elementy pomiaru oprogramowania

kategoria	obiekty	atrybuty	
		wewnętrzne	zewnętrzne
produkty	specyfikacje	rozmiar, wielokrotne użycie, modularność, nadmiarowość, funkcjonalność	zrozumiałość, pielęgnowalność
	projekty	rozmiar, wielokrotne użycie, modularność, spójność	jakość, złożoność, pielęgnowalność
	kod źródłowy	rozmiar, wielokrotne użycie, modularność, spójność, złożoność, strukturalność	niezawodność, używalność, pielęgnowalność
	dane testowe	rozmiar, poziom pokrycia	jakość
procesy	model	czas, nakład pracy, liczba zmian wymagań	jakość, koszt, stabilność
	projekt	czas, nakład pracy, liczba znalezionych błędów specyfikacji	koszt, opłacalność
	testowanie	czas, nakład pracy, liczba znalezionych błędów	koszt, opłacalność, stabilność
zasoby	personel	wiek	wydajność, doświadczenie, inteligencja
	zespoły	wielkość, poziom komunikacji, struktura	wydajność
	oprogramowanie	cena, wielkość	używalność, niezawodność
	sprzęt	cena, szybkość, wielkość pamięci	niezawodność, jakość

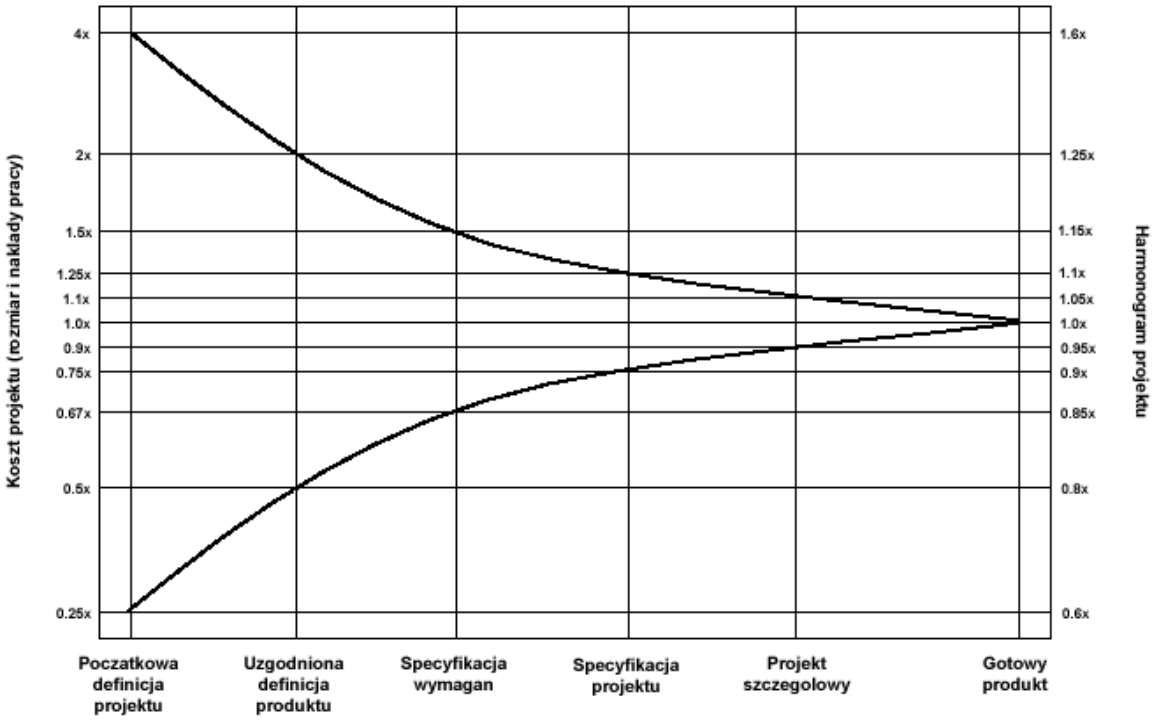
Źródło: (Fenton & Neil, 2000)

2.7 ESTYMACJA W INŻYNIERII OPROGRAMOWANIA

Zgodnie z ogólnie przyjętą definicją estymacja to proces obliczania przybliżenia pewnej wielkości, która jest użyteczna również wtedy, gdy dane wejściowe dla tego procesu są niekompletne, niepewne i zaszumione. Przenosząc tą definicję na grunt inżynierii oprogramowania estymację można zdefiniować jako procedurę przewidywania wartości miar lub metryk procesu, produktu lub zasobów wchodzących w skład realizowanego przedsięwzięcia informatycznego na podstawie wartości miar lub metryk innych procesów, produktów lub zasobów dostępnych wcześniej.

Estymacja parametrów przedsięwzięcia informatycznego jest w ogólności bardzo trudna, a nawet niemożliwa. Podstawowym faktem na poparcie wspomnianej już wcześniej tezy jest charakterystyka procesu budowania oprogramowania, który jest procesem stopniowego uszczegóławiania. Bardzo dobrze zostało to przedstawione w pracy (Boehm, Clark, Horowitz, Westland, Madachy, & Selby, 1995) jako „stożek niepewności" (ang. *cone of uncertainty*; patrz Rysunek 6). Zgodnie z tym modelem zespół w trakcie realizowania przedsięwzięcia informatycznego jest w stanie dostarczyć następujących oszacowań:

- pod koniec fazy studium wykonalności oszacowanie, które może różnić się od rzeczywistości o około 50 procent
- pod koniec fazy analizy oszacowanie o dokładności ±25 procent
- pod koniec fazy projektowania oszacowanie zmieniające się w granicach 10 procent
- pod koniec fazy programowania, gdy do wykonania pozostaje jeszcze testowanie – oszacowanie ostateczne, które powinno się zmieniać nie więcej niż ±5 procent



Rysunek 6 Stożek niepewności (Boehm, Clark, Horowitz, Westland, Madachy, & Selby, 1995)

Do liczb obrazujących potencjalny błąd oszacowania na poszczególnych etapach rozwoju oprogramowania należy podchodzić z dużą ostrożnością. Najważniejsze jest zrozu-

mienie, że przedstawione liczby obrazują najlepszy możliwy do uzyskania przypadek błędu oszacowania. W większości typowych aplikacji błąd ten będzie większy.

Należy tutaj zaznaczyć, że stożek niepewności w przedstawionej postaci jest tylko zgrubnym modelem, ogólnie obrazującym zmianę niepewności w trakcie cyklu życia produktu informatycznego. Niedawno przeprowadzone badania wskazują między innymi na to, że stożek wcale nie musi być regularny ani symetryczny względem osi poziomej (Galorath & Evans, 2006)(McConnell, 2006).

Problem ograniczeń estymacji jest często ignorowany przez osoby nieświadome charakterystyki estymacji przedsięwzięć informatycznych. Przykładem takich osób są klienci, którzy zwykle wymagają estymacji kosztów i czasu realizacji jeszcze zanim realizacja się rozpocznie. W takich przypadkach szczególnie istotna jest „psychologiczna" strona sposobu przedstawienia wartości estymat w kategoriach prawdopodobieństwa (sposoby przedstawiania estymat zostały opisane w rozdziale 2.7.3).

2.7.1 Zakres estymacji

Estymacji może podlegać praktycznie każdy atrybut procesu, produktu lub zasobów istotnych z punktu widzenia zarządzającego przedsięwzięciem wytwarzania oprogramowania. Zgodnie z definicją estymacji oprogramowania zaproponowanej w (Jones, 1994), według której polega ona na próbie przewidywania przyszłego rezultatu projektu w kategoriach różnych czynników (Tabela 7 zawiera ich przykłady), przy czym najczęściej estymowane to:

- rozmiar
- nakład pracy
- czas realizacji
- koszt

Należy tutaj podkreślić szczególną rolę rozmiaru oprogramowania jako podstawę do estymacji pozostałych parametrów procesu wytwarzania oprogramowania (Boehm, Clark, Horowitz, Westland, Madachy, & Selby, 1995)(McConnell, 2006)(ISPA, 2007). Rysunek 7 prezentuje typowy przebieg procesu estymacji.

Rysunek 7 Proces przebiegu estymacji (McConnell, 2006)

Problemy rozmiaru oprogramowania i jego estymacji zostaną dokładniej opisane w jednym z następnych rozdziałów niniejszej pracy.

2.7.2 Podstawowe techniki estymacji

W czasie kilkudziesięciu lat rozwoju inżynierii oprogramowania powstał cały szereg różnorodnych technik i metod estymacji parametrów procesów, produktów i zasobów przedsięwzięcia rozwoju oprogramowania. Na podstawie doświadczeń zebranych w procesie ich tworzenia oraz wykorzystania w praktyce można wysnuć kilka podstawowych technik, które sprawiają, że proces estymacji jest sprawniejszy.

Metoda *licz, przetwarzaj, szacuj* (ang. Count, Compute, Judge)

Podstawową techniką zaproponowaną w pracy (McConnell, 2006) jest, paradoksalnie, unikanie estymacji. Autor słusznie zauważa, że wiele atrybutów procesu wytwarzania oprogramowania pozwala na wyznaczenie istotnych parametrów, które potencjalnie można by było szacować. Dlatego też najważniejszy etap estymacji to liczenie wszystkich tych wielkości, które można następnie poddać procesowi przetwarzania, aby uzyskać wartości wymaganych parametrów.

Liczeniu (ang. *count*) może podlegać praktycznie dowolny znany element wiedzy o zakresie realizowanego oprogramowania i środowiska jego rozwoju. Oczywiście, zgodnie z tym co przedstawiono już wcześniej, im bardziej zaawansowany etap realizacji, tym więcej jest elementów, które można by poddać liczeniu. Liczeniu powinny podlegać wszystkie te atrybuty, co do których istnieje przekonanie (na podstawie analizy danych historycznych), że są one skorelowane z poszukiwanymi parametrami. Przykłady atrybutów, które

mogą być poddane liczeniu to liczba wymagań, liczba zmian wymagań, liczba okien dialogowych i tym podobne.

Kolejnym etapem jest przetwarzanie (ang. *compute*), mające na celu zamianę rezultatów liczenia na estymaty. Stosuje się tutaj przede wszystkim rozmaite metody matematyczne, wykorzystujące istniejące dane historyczne z realizacji poprzednich przedsięwzięć. Etap przetwarzania jest również zwany kalibracją. Dane historyczne to klucz do sprawnej estymacji, bo pozwalają na przystosowanie metod estymacji do działania w realiach konkretnego środowiska rozwoju oprogramowania. Ponadto dane historyczne pozwalają na redukcję (zwykle niepożądanego) wpływu subiektywnych opinii kadry na proces estymacji. Bez obecności tego typu danych praktycznie żadna metoda estymacji nie może zostać uznana za wiarygodną. Dlatego też tak istotny jest proces ich kolekcjonowania, w ramach którego powinny być zbierane dane dotyczące rozmiaru, nakładów pracy na realizację, czasu realizacji, liczby defektów i inne.

Ostatnią i najmniej pożądaną techniką estymacji jest osąd ekspercki (ang. *judgement*). W ramach tej techniki eksperci z zakresu estymacji dokonują określenia oszacowań kierując się tylko i wyłącznie własną wiedzą i doświadczeniem. Technika ta jest najbardziej zawodna, bo nie ma bezpośredniego odniesienia do konkretnej sytuacji. Z praktyki wynika, że najlepsze rezultaty daje połączenie wiedzy i doświadczenia eksperta z realnymi danymi, będącymi rezultatem procesu liczenia.

Osąd ekspercki

Osąd ekspercki jest obecnie najbardziej popularną techniką estymacji. Według (Paynter, 1996) aż 86% organizacji realizujących oprogramowanie komputerowe korzystało z osądu eksperckiego jako podstawowej techniki estymacji. Z kolei w pracy (Kitchenham, Pfleeger, McColl, & Eagan, 2002) przedstawiono dane według których 72% oszacowań było określonych na podstawie opinii ekspertów.

Osąd ekspercki można w ogólności określić jako opracowanie estymat parametrów realizowanego przedsięwzięcia informatycznego przez osoby ekspertów na podstawie ich doświadczenia w realizacji podobnych projektów oraz dostępnej wiedzy na temat projektu obecnie realizowanego. W ramach tej techniki mieszczą się zarówno metody intuicyjne, opierające się na nie popartych faktami „odczuciach" ekspertów, jak i rozmaite metody strukturalne, definiujące sposoby podziału zadań na mniejsze, łatwiejsze do oszacowanie

fragmenty i łączenie ich estymat w całościowe oszacowania. Uznaje się, że najlepszym źródłem skutecznych i wiarygodnych oszacowań są osoby zajmujące się bezpośrednio realizacją opracowywanego programu, gdyż mają one największe pojęcie co do zakresu i charakteru działań, jakie należy wykonać. Jednakże nie jest to rozwiązanie idealne, gdyż zgodnie z obserwacjami w takich przypadkach bardzo często zakres prac rozszerza się tak, aby zająć cały czas dany na realizację zadania (jest to tak zwane prawo Parkinsona (Parkinson, 1955)). Dlatego też tak istotne jest spojrzenie na realizowane przedsięwzięcie z różnych punktów widzenia, przykładowo poprzez opracowywanie oszacowań przez kilka osób, a następnie dokonanie ich uśrednienia za pomocą odpowiedniego przekształcenia matematycznego, np. równanie 2.01 (Galorath & Evans, 2006):

$$ExpectedCase = \frac{BestCase+4*MostLikelyCase+WorstCase}{6} \qquad 2.01$$

gdzie *ExpectedCase* to oczekiwane oszacowanie całościowe, *BestCase* to najlepsze oszacowanie, *MostLikelyCase* to oszacowanie uznane za najbardziej prawdopodobne, *WorstCase* to najgorsze oszacowanie.

Do ustalania wspólnej opinii wielu ekspertów stosuje się również tzw. rozszerzoną metodę delficką (ang. *Wideband Delphi Approach*), polegającą na tym, że najpierw każdy ekspert formułuje niezależnie od innych swoje przewidywania na podstawie dostępnych danych, a następnie w ciągu kolejnych rund grupa ekspertów wzajemnie ocenia indywidualne opinie i stopniowo dochodzi do konsensusu (Stellman & Greene, 2006).

Dekompozycja i rekompozycja

Technika dekompozycji i rekompozycji polega na rozbijaniu problemu na mniejsze, łatwiejsze do przetwarzania składowe, a następnie składanie wyników przetwarzania w całościowy rezultat. Jest ona jedną z podstawowych technik stosowanych w estymacji parametrów przedsięwzięć informatycznych. Podstawą skuteczności tej techniki jest zastosowane w jej ramach prawo wielkich liczb, według którego opracowanie pojedynczej estymaty dla całego analizowanego oprogramowania niesie ze sobą niebezpieczeństwo tego, że jej wartość będzie najprawdopodobniej albo skrajnie niedoszacowana albo skrajnie przeszacowana. Natomiast przeprowadzenie dekompozycji systemu na mniejsze i łatwiejsze do estymacji elementy sprawi, że błędy poszczególnych estymat składowych będą się

nawzajem znosić, sprawiając, że błąd oszacowania całościowego będzie mniejszy (McConnell, 2006).

Estymacja przez analogię

Estymacja przez analogię stanowi rozwinięcie potrzeby kolekcjonowania danych historycznych dla zrealizowanych produktów. W jej ramach estymacji dokonuje się na podstawie założenia, że obecnie realizowany produkt jest podobny do produktów zrealizowanych w przeszłości. Procedura estymacji przez analogię składa się zwykle z następujących kroków (McConnell, 2006):

- ustal dokładne wartości rozmiaru, nakładu pracy i kosztów podobnych projektów realizowanych w przeszłości; najlepiej jeśli są dostępne dane z dekompozycji według cech produktu i realizowanych komponentów
- porównaj rozmiar komponentów aktualnego projektu z komponentami projektów historycznych
- określ rozmiar aktualnego jako ułamek rozmiaru projektów historycznych
- ustal oszacowanie nakładu pracy aktualnego projektu na podstawie estymaty jego rozmiaru w porównaniu do rozmiaru projektów historycznych
- sprawdź spójność założeń w projekcie aktualnym i projektach historycznych

Estymacja przez analogię stanowi podstawę sposobów szacowania stosowanych w wielu „lekkich" (zwanych również „zwinnymi" od angielskiego określenia *agile*) metodykach zarządzania procesem wytwarzania oprogramowania (np. programowanie ekstremalne), zwanych tak w odróżnieniu od metodyk „ciężkich", czyli tych bazujących na modelu kaskadowym, w których duży nacisk kładziony jest na rozwój dokumentacji projektowej (Beck & Fowler, 2000).

Estymacja przez pośredników

Estymacja przez pośredników (ang. *proxy-based*) stanowi naturalne rozwiązanie problemu braku informacji na temat opracowywanego oprogramowania we wczesnych fazach jego rozwoju. Nie mając szczegółowych danych na temat jego komponentów wprowadza się pojęcie pośredników (ang. *proxy*), czyli obiektów zastępczych co do których istnieje przekonanie, że ich parametry są skorelowane z rzeczywistymi parametrami obiektów za-

stępowanych. Dobrymi przykładami estymacji z użyciem pośredników są różne metody określania rozmiaru oprogramowania bazujące na logice rozmytej.

Korzystanie z narzędzi wspomagających estymację

Problemy estymacji nie są niczym nowym w inżynierii oprogramowania i praktycznie od zawsze istniały narzędzia wspomagające kierownika projektu w procesie estymacji. Ich zadaniem jest przede wszystkim standardyzacja przebiegu estymacji i zautomatyzowanie pewnych czynności z nią związanych. Ponadto często narzędzia te pozwalają na dokonywanie hipotetycznych oszacowań i tworzenia warunkowych ścieżek estymacji. Najpopularniejsze rozwiązanie obecnie dostępne na rynku to (McConnell, 2006):

- *Angel* (http://dec.bournemouth.ac.uk/ESERG/ANGEL/): narzędzie do estymacji kosztu wykorzystujące szacowanie przez analogię
- *Construx Estimate* (http://www.construx.com/estimate/): darmowe narzędzie do estymacji nakładu pracy, kosztu i harmonogramu realizacji na podstawie oszacowań rozmiaru
- *COCOMO II* (http://sunset.usc.edu/research/COCOMOII/): klasyczny, oficjalny i darmowy model estymacji nakładu pracy, kosztu i harmonogramu realizacji na podstawie oszacowań rozmiaru autorstwa Barry'ego Boehma
- *Costar* (http://www.softstarsystems.com/): komercyjna implementacja modelu COCOMO II
- *KnowledgePLAN* (http://www.spr.com/): narzędzie do planowania przebiegu realizacji przedsięwzięcia informatycznego z opcjami szacowania rozmiaru i innych parametrów
- *PRICE-S* (http://www.pricesystems.com/): zestaw narzędzi wspomagających proces estymacji kosztu i harmonogramu, a także rozmiaru realizowanego projektu informatycznego
- *SEER* (http://www.galorath.com/): zestaw narzędzi wspomagających proces estymacji kosztu i harmonogramu, a także rozmiaru realizowanego projektu informatycznego
- *SLIM-Estimate* (http://www.qsm.com/): rodzina narzędzi do estymacji nakładu pracy, harmonogramu i jakości realizowanych systemów informatycznych

Wymienione wyżej rozwiązania są przykładami modeli parametrycznych: oszacowania są obliczane na podstawie podanych przez użytkownika wartości parametrów przez model matematyczny.

Stosowanie różnych metod estymacji

Wykorzystanie wielu różnych metod estymacji dla dokonania konkretnego oszacowania ma głęboki sens, gdyż w ten sposób dokonuje się oceny przedsięwzięcia z wielu punktów widzenia. To z kolei zwiększa prawdopodobieństwo tego, że w procesie estymacji nie zostaną pominięte żadne istotne atrybuty. Negatywną stroną wielopłaszczyznowej analizy jest duży nakład pracy, jaki trzeba poświęcić realizacji takiej procedury.

2.7.3 Sposoby prezentacji estymat

Prezentacja wyników estymacji jest zadaniem ryzykownym. Wynika to przede wszystkim z charakterystyki tego procesu, który jest niczym innym jak próbą przewidywania przyszłości. Najczęstszym problemem jest to, że zarówno klienci, jak i wyższe szczeble kierownictwa często mylą oszacowanie ze zobowiązaniem. Estymata nie jest niczym innym, jak po prostu przedstawieniem pewnej wartości, co do której istnieje niezerowe prawdopodobieństwo, że będzie ona w określonej przyszłości prawidłowo opisywała pewną cechę realizowanego systemu. Estymata jest bardziej przypuszczeniem niż zobowiązaniem. Dlatego też odpowiednio prezentując oszacowania można i trzeba uświadomić odbiorcom, że estymacja jest procesem ryzykownym i podatnym na zmiany.

Estymaty dotyczące atrybutów procesu i produktów rozwoju oprogramowania są zwykle rozumiane jako wartości punktowe. Jest to nieco mylne podejście, bo sugeruje, że estymowana wartość jest pewna (tzn. prawdopodobieństwo jej wystąpienia wynosi 100%). Znacznie lepszym rozwiązaniem jest przedstawianie wartości estymat jako rozkładów prawdopodobieństwa (McConnell, 2006). Wówczas komunikaty typu „jesteśmy w 90% pewni, że dostarczymy ostateczną wersję produktu za 24 tygodnie" są bezpieczne z punktu widzenia kierownictwa, a jednocześnie niosą dużą wartość informacyjną dla klienta.

Inne sugerowane sposoby przedstawiania estymat to (McConnell, 2006):

- estymaty wraz zakresem potencjalnej zmienności (np. 6 miesiące, -1 miesiąc, +5 miesięcy)
- estymaty z kwantyfikacją ryzyka (przykład: Tabela 8)

Tabela 8 Przykład prezentacji estymaty z kwantyfikacją ryzyka

estymata: 6 miesięcy, +5 miesięcy, -1 miesiąc	
wpływ	**opis ryzyka**
+1.5 miesiąca	obecna wersja wymaga o 20% więcej cech niż wersja poprzednia
+1 miesiąc	system graficzny dostarzono później niż planowano
+1 miesiąc	nowe narzędzia nie działają tak dobrze jak planowano
+1 miesiąc	80% dawnej bazy danych nie nadaje się do wykorzystania w obecnej wersji
+0.5 miesiąca	średnia oczekiwana długość zwolnień chorobowych członków zespołu
-0.5 miesiąca	100% zespołu realizowało swe zadania do 1 kwietnia
-0.5 miesiąca	nowe narzędzia graficzne działają lepiej niż planowano

Źródło: opracowanie własne na podstawie (McConnell, 2006)

- estymaty ze współczynnikami pewności (przykład: Tabela 9)

Tabela 9 Przykład estymaty ze współczynnikami pewności

data dostarczenia	**prawdopodobieństwo dostarczenia**
15 stycznia	20%
1 marca	50%
1 listopada	80%

Źródło: (McConnell, 2006)

- estymaty z gradacją przypadków (przykład: Tabela 10)

Tabela 10 Przykład estymaty z gradacją przypadków

przypadek	data
najlepszy (estymata)	15 stycznia
planowany (zobowiązanie)	1 marca
stan obecny (estymata)	1 kwietnia
najgorszy (estymata)	1 listopada

Źródło: (McConnell, 2006)

W wielu przypadkach wysoce wskazane jest używanie różnych form graficznych obrazujących zakres i zmienność prawdopodobieństwa wystąpienia określonych wartości oszacowań.

2.7.4 Ocena metod estymacji

Proces tworzenia nowych metod estymacji nie może się obejść bez ich oceny. Ocenie może podlegać praktycznie każdy element metody, jednakże najistotniejszą cechą jest dokładność oszacowań. Proces oceniania dokładności predykcji przeprowadzany jest w postaci szeregu doświadczeń, w ramach których rezultaty szacowania są przeciwstawiane rzeczywistym wartościom parametrów. Dokładność estymat uzyskiwanych poprzez stosowanie metody ocenia się zwykle przy pomocy wskaźników statystycznych. Najczęściej stosowane to (Conte, Dunsmore, & Shen, 1986)(Ferens & Christensen, 2000)(Briand, El Emam, Surmann, Wieczorek, & Maxwell, 1999):

- moduł błędu względnego *MRE*
- średnia modułu błędu względnego *MMRE*
- mediana modułu błędu względnego *MdMRE*
- wskaźnik predykcji *PRED(p)*
- średnia kwadratowa błędu *RMS*
- względna średnia kwadratowa błędu *RRMS*

Przy definicji powyższych wskaźników przyjęto następujące założenia:

- *P* to pojedyncze przedsięwzięcie informatyczne

- x_P to wartość pojedynczej cechy przedsięwzięcia P
- $\widehat{x_P}$ to estymowana wartość cechy x_P
- $\boldsymbol{P}$ to zbiór przedsięwzięć P_i, gdzie $1 \leq i \leq |\boldsymbol{P}|$

Moduł błędu względnego *MRE* służy do oceny dokładności predykcji dla pojedynczego przypadku. Wskaźnik ten jest zdefiniowany następująco:

$$MRE(P) = \frac{|x_P - \widehat{x_P}|}{x_P} \qquad 2.02$$

Wskaźnik ten pozwala na ocenę skuteczności estymacji wykluczając wzajemne znoszenie się efektów niedoszacowań i przeszacowań. Jego wartość prezentuje się zwykle w postaci procentowej; im mniejsza wartość tego wskaźnika tym lepsza metoda.

Średnia modułu błędu względnego *MMRE* jest najczęściej stosowanym wskaźnikiem oceny metod estymacji na podstawie zbioru wyników oszacowań. Jest on zdefiniowany następująco:

$$MMRE(\boldsymbol{P}) = \frac{\sum_{P_i \in \boldsymbol{P}} MRE(P_i)}{|\boldsymbol{P}|} \qquad 2.03$$

Wyraża się go zwykle w postaci procentowej. Za (Conte, Dunsmore, & Shen, 1986) uznaje się powszechnie, że gdy wartość tego wskaźnika jest nie większa niż 25% to metodę estymacji można uznać za satysfakcjonującą.

Mediana (wartość środkowa) modułu błędu względnego *MdMRE* jest używana w sytuacjach, gdy chce się zredukować wpływ wartości ekstremalnych na wyniki oceny.

Wskaźnik predykcji *PRED(p)* to równie popularne co *MMRE* kryterium oceny metod estymacji na podstawie zbioru wyników oszacowań. Zdefiniowany jest następująco:

$$PRED(\boldsymbol{P}, p) = \frac{|\{P_i \in \boldsymbol{P} : MRE(P_i) \leq p\}|}{|\boldsymbol{P}|} \qquad 2.04$$

Jest to po prostu stosunek liczby przedsięwzięć ocenionych przy pomocy wskaźnika *MRE* poniżej wartości granicznej p do liczby wszystkich obserwacji. Za (Conte, Dunsmore, & Shen, 1986) przyjmuje się zwykle, że $p = 25\%$ oraz próg akceptowalności metody, gdy $PRED(25\%) \geq 75\%$.

Średnia kwadratowa błędu *RMS* jest zdefiniowana następująco:

$$RMS(\boldsymbol{P}) = \sqrt{\frac{\sum_{P_i \in \boldsymbol{P}} \left(x_{P_i} - \widehat{x_{P_i}}\right)^2}{|\boldsymbol{P}|}} \qquad 2.05$$

Im mniejsza jest wartość tego wskaźnika, tym lepiej oceniana jest metoda estymacji. Wskaźnik ten jest bardziej konserwatywny niż rodzina *MRE* i dlatego też rzadziej się go używa. Autorzy (Conte, Dunsmore, & Shen, 1986) twierdzą, że nadaje się on tylko do oceny metod bazujących na regresji.

Względna średnia kwadratowa błędu *RRMS* jest wyznaczana na podstawie następującego równania:

$$RRMS(\boldsymbol{P}) = \frac{|\boldsymbol{P}| * RMS(\boldsymbol{P})}{\sum_{P_i \in \boldsymbol{P}} x_{P_i}} \qquad 2.06$$

Mniejsze wartości tego wskaźnika odnoszą się do lepszych metod. Zgodnie z pracą (Conte, Dunsmore, & Shen, 1986) próg akceptowalności dla metod ocenianych według tego wskaźnika to 25%. Wskaźnik ten jest rzadko używany ze względu na jego konserwatyzm.

Niektóre z zaprezentowanych powyżej wskaźników stały się *de facto* standardami w ocenie skuteczności predykcji metod estymacji, mimo że istnieją w literaturze głosy, że ich stosowanie nie jest pozbawione wad. Stosując podejście ekonometryczne, wybór wskaźnika ocenowego powinien być zależny od charakterystyki zbioru danych do których się odnosi. Dlatego też zapewne nie istnieje uniwersalny wskaźnik pozwalający na porównywanie zupełnie różnych metod estymacji.

Zgodnie z pracą (Foss, Stensrud, Kitchenham, & Myrtveit, 2003) wskaźnik *MMRE* zawsze preferuje modele generujące estymaty poniżej średniej od modeli generujących estymaty średniej. Innymi słowy, wskaźnik *MMRE* będzie stale preferował model obciążony ponad model nieobciążony. Dlatego też autorzy uważają, że prace dotyczące oceny metod estymacji używające wskaźnika *MMRE* jako głównego kryterium oceny (a trzeba podkreślić że dotyczy to zdecydowanej większości opracowań) mają ograniczoną wartość.

W pracy (Foss, Stensrud, Kitchenham, & Myrtveit, 2003) sugeruje się, że dobrymi wskaźnikami dobroci metod estymacji może być zwyczajne odchylenie standardowe. W pracach pokrewnych (Kitchenham, Pickard, MacDonell, & Shepperd, 2001) uznaje się, że dobrymi wskaźnikami są również wykresy pudełkowe wyrażenia $\frac{\widehat{x_P}}{x_P}$ oraz reszt $x_P - \widehat{x_P}$.

3 MODELOWANIE OBIEKTOWE

W rozdziale tym zostało opisane modelowanie obiektowe w dzisiejszym rozumieniu tego pojęcia. Opis ten rozpoczyna się od przybliżenia paradygmatu obiektowości, jest kontynuowany przez opis języka UML stosowanego do modelowania obiektowego do zdefiniowania wzorców projektowych, jako podstawowego rezultatu praktyki i badań nad architekturami obiektowymi.

3.1 PARADYGMAT OBIEKTOWY

Pojęcie paradygmatu w inżynierii programowania jest tożsame z pojęciem metodologii (czyli zbioru uporządkowanych praktyk, których można regularnie używać w procesie wytwarzania oprogramowania), z tym, że jego zakres ogranicza się tylko do dziedziny programowania. Paradygmat programowania zapewnia i określa punkt widzenia programisty na proces przetwarzania realizowany w ramach programu komputerowego. Przykładowo, w programowaniu zorientowanym obiektowo program może być określony jako zbiór obiektów pozostających we wzajemnej interakcji, podczas gdy w programowaniu funkcyjnym program to sekwencja bezstanowych wykonań funkcji (Van Roy & Haridi, 2004).

Paradygmat obiektowy w programowaniu jest koncepcją dość starą, gdyż jego początki sięgają roku 1967, kiedy to powstał język *Simula 67*, stworzony na potrzeby prac nad symulacją statków (Jabłonowski & Sroka, 2006). Podstawowym *novum* w tym języku w porównaniu do ówcześnie najbardziej rozpowszechnionego paradygmatu proceduralnego było wprowadzenie możliwości opisu wybranego fragmentu rzeczywistości **jednocześnie** poprzez opis jego stanu (czyli danych) oraz zachowania (czyli procedur, w programowaniu obiektowym zwanych *metodami*) w modułach zwanych *obiektami*. W programowaniu proceduralnym dane i zachowanie nie były bezpośrednio powiązane (Eckel, 2003).

Paradygmat obiektowy został wprowadzony do programowania jako sposób na rozwiązanie problemu wzrastającej złożoności oprogramowania. Łączenie stanu (czyli danych) i zachowania (czyli metod) w obiekty pozwoliło na uzyskanie większej modularności realizowanego oprogramowania (Meyer, 1997). Wprowadzając pojęcia obiektu (czyli materii, instancji) oraz klasy (czyli wzorca obiektów) zapewniono bardzo łatwy, bo natu-

ralny dla ludzkiego mózgu, sposób identyfikacji bytów w procesie modelowania rzeczywistości.

Istnieje pewna różnica zdań co do tego, jakie cechy języka programowania sprawiają, że może on być uznany za język zorientowany obiektowo. W przeglądzie cech obiektowości stworzonym na podstawie analizy literatury z ostatnich 40 lat za najważniejsze pojęcia uznano (Armstrong, 2006):

- klasa
- obiekt
- metoda
- dziedziczenie
- enkapsulacja
- abstrakcja
- polimorfizm

Klasa

Pojęcie klasy (ang. *class*) powinno jest rozumiane jako abstrakcyjna charakterystyka wycinka rzeczywistości opisująca jego cechy (atrybuty) oraz zachowanie (metody). Przykładowo, klasa *Pies* mogłaby służyć do abstrakcyjnego opisu dowolnego rodzaju psa poprzez wyszczególnienie cech, które posiada każdy pies, zarówno tych statycznych (np. imię, rasa, ciężar), jak i dynamicznych (możliwość szczekania, jedzenia).

Obiekt

Obiekt jest konkretnym egzemplarzem (instancją) klasy. Podczas gdy klasa opisuje potencjalnie dostępne cechy bytu to obiekt charakteryzuje konkretny egzemplarz, np. obiekt *Lassie* jest egzemplarzem klasy *Pies*, charakteryzującym się tym, że jest dalmatyńczykiem i waży 15 kilogramów.

Metoda

Metody opisują zachowanie, jakie może być realizowane przez obiekty. Metoda jest elementem klasy, czyli w opisywanym przykładzie *szczekanie* jest cechą klasy *Pies*, ale może być ono wykonywane przez (czy też na rzecz) obiekt *Lassie*.

Dziedziczenie

Dziedziczenie wspomaga i porządkuje proces modelowania bytów bardziej wyspecjalizowanych na podstawie bytów bardziej ogólnych. Stosuje się tutaj hierarchizację opisu bytów, począwszy od bytów najbardziej ogólnych (np. klasy *Zwierzę*, definiującej klasę zwierząt posiadających ciężar i możliwość poruszania się) do bardziej szczegółowych (np. klasa *Pies* będąca rozszerzeniem klasy *Zwierzę*, a dodająca cechę posiadania imienia i możliwości szczekania). Definiując klasy bytów bardziej szczegółowych (np. *Pies, Kot*) nie ma potrzeby definiowania w nich cech i metod charakterystycznych dla wielu klas; lepiej określić klasę bazową (np. *Zwierzę*) definiującą podstawowe i wspólne dla wielu klas cechy i metody, a następnie zaznaczyć, że klasa bardziej specyficzna dziedziczy po klasie bazowej i (ewentualnie) dodaje swoje specyficzne elementy (np. *szczekanie* dla klasy *Pies* i *miauczenie* dla klasy *Kot*).

Enkapsulacja

Enkapsulacja (ang. *encapsulation*), zwana również hermetyzacją, ogranicza dostęp do składowych (atrybutów i metod) klasy. Służy ona do ochrony stanu obiektu przed nieupoważnioną modyfikacją. Definiując klasę jednocześnie definiuje się jej interfejs (ang. *interface*), czyli zestaw możliwych operacji, jakim może podlegać obiekt. W wielu przypadkach ujawnianie dokładnego mechanizmu działania (np. szczekania w przypadku klasy *Pies*) nie jest wskazane, dlatego też istnieją różne sposoby nadawania uprawnień do korzystania z określonych elementów obiektu. Najczęściej nadawanie uprawnień polega na określeniu, które składowe są powszechnie dostępne (składowe publiczne, ang. *public*), które mogą być dostępne tylko dla siebie i swoich potomków w hierarchii dziedziczenia (składowe chronione, ang. *protected*), a które dostępne tylko dla obiektów własnej klasy (składowe prywatne, ang. *private*).

Abstrakcja

Każdy obiekt w systemie służy jako model abstrakcyjnego wykonawcy posiadającego określone cechy (atrybuty) i umiejętności (metody). Nic jednak nie stoi na przeszkodzie, aby w zależności od kontekstu traktować obiekty w różny sposób. Na przykład może się zdarzyć, że w pewnych przypadkach znacznie wygodniej jest traktować *Lassie* jako instancję klasy *Zwierzę*, a w innych lepiej jako egzemplarz klasy *Pies*. W ten sposób można redukować złożoność przetwarzania.

Polimorfizm

Polimorfizm (ang. *polymorphism*) to możliwość ustalenia dokładnego zachowania się obiektu w zależności od tego na jakim obiekcie zostało ono wymuszone. Przykładowo, dziedzicząc klasę *Pies* po klasie *Zwierzę* zawierającą metodę *wydaj odgłos* można by oczekiwać, że wywołanie zachowania *wydaj odgłos* na instancji klasy *Pies* sprawi, że w rzeczywistości wywołane zachowanie będzie *szczekaniem*. Polimorfizm działa w warunkach dziedziczenia (*Pies* jest potomkiem *Zwierzęcia*) i na podstawie abstrakcji (*Pies* może być traktowany jako *Zwierzę* w kwestiach związanych z wydawaniem odgłosu).

Inne ważne pojęcia związane z paradygmatem obiektowym to (Shalloway & Trott, 2001):

- *klasa abstrakcyjna*: klasa definiująca schemat swoich atrybutów i interfejsu, ale nie definiująca implementacji; nie można tworzyć instancji klas abstrakcyjnych
- *atrybut*: jednostka danych składająca się na opis stanu obiektu; jest częścią klasy
- *konstruktor*: specjalna metoda automatycznie uruchamiana w procesie tworzenia obiektu; używana przede wszystkim do inicjowania wartości atrybutów obiektu
- *destruktor*: specjalna metoda automatycznie uruchamiana w procesie niszczenia obiektu
- *instancja*: egzemplarz klasy, obiekt
- *składowa klasy*: zbiorcza nazwa atrybutów i metod klasy
- *metoda*: funkcja powiązana z obiektem
- *atrybut/metoda statyczna*: atrybut lub metoda powiązana bardziej z daną klasą niż obiektami klasy; może być używana niezależnie od tego czy obiekty danej klasy istnieją czy nie

Mimo, że paradygmat obiektowy został po raz pierwszy opisany w 1967 to długo pozostawał w cieniu programowania proceduralnego w głównym nurcie programowania. Programistom trudno było przyswoić sobie nowe techniki, tak różne od dotychczasowych. Poza tym w tamtych czasach mniej było problemów, które wygodnie byłoby opisywać w sposób obiektowy. Prawdziwy przełom w popularności paradygmatu obiektowego przyniosły lata dziewięćdziesiąte XX wieku. Wraz z postępem technicznym w dziedzinie elektroniki i upowszechnianiem się graficznych interfejsów użytkownika odkryto, że obiektowość dobrze odnajduje się w owej rzeczywistości. Języki takie jak C++ i Objective-C uznano za narzędzia idealnie dopasowane do modelowania pojęć związanych z graficznym interfejsem użytkownika. Warto podkreślić tutaj również rolę elektronicznej rozrywki: gry komputerowe bardzo wygodnie realizuje się z użyciem pojęcia obiektowości i to one zostały jednym z głównych motorów popularności języka C++.

Warto tutaj jednak podkreślić, że obiektowość nie jest idealnym rozwiązaniem wszystkich problemów programowania. W 1971 roku powstała pierwsza wersja języka Smalltalk, który dotąd jest uważany za język „najbardziej obiektowy”: niósł on takie nowości (jak np. możliwość dynamicznej modyfikacji definicji klasy), że nie zdołał przekonać do siebie większej liczby programistów i do dzisiaj pozostaje językiem niszowym. Dlatego też największą popularnością cieszą się języki hybrydowe, takie jak wspomniane już C++ i Objective-C, będące w swej istocie proceduralnym językiem C z dodanymi cechami obiektowymi. Wsparcie dla wielu paradygmatów wydaje się być kluczem do popularności języka programowania, także w przypadku języków skryptowych, takich jak Python, który łączy w sobie paradygmaty proceduralny, obiektowy i funkcyjny.

3.2 DIAGRAMY UML

Rosnąca popularność programowania zorientowanego obiektowo sprawiła, że zaistniała potrzeba jego standaryzacji. O ile próba formalizacji paradygmatu obiektowego w praktyce spełzła na niczym, to w kwestiach związanych z analizą i projektowaniem obiektowym zanotowano spory postęp. Dwa najważniejsze osiągnięcia wynikłe z lat doświadczeń i badań nad obiektowością to język UML oraz wzorce projektowe.

UML (ang. *Unified Modeling Language*), czyli zunifikowany język modelowania jest następcą obiektowych metod analizy i projektowania jakie pojawiły się na przełomie lat

osiemdziesiątych i dziewięćdziesiątych XX wieku. Jest on bezpośrednią unifikacją trzech popularnych metod analizy i projektowania obiektowego (Fowler & Scott, 2002):

- techniki modelowania obiektów OMT (ang. *Object Modeling Technique*) opracowanej przez Jamesa Rumbaugh
- metody Grady'ego Boocha
- zorientowanej obiektowo inżynierii oprogramowania OOSE (ang. *Object-Oriented Software Engineering*) opracowanej przez Ivara Jacobsona

Unifikacja została poparta standardyzacją zrealizowaną w organizacji OMG (ang. *Object Management Group*), która zajmuje się rozwojem języka. Na dzień dzisiejszy najnowszą oficjalną wersją języka UML jest 2.0.

UML to formalny język służący do opisu świata obiektów w analizie, projektowaniu i programowaniu obiektowym. Sam w sobie nie jest on metodą, ale został on stworzony, aby być kompatybilny z istniejącymi ówcześnie metodami (np. wspomniane wcześniej metoda Boocha). UML stanowił również inspirację i podstawę dla tworzenia metod właściwych, na przykład metody RUP (ang. *Rational Unified Process*). UML nie jest systemem zamkniętym, przeciwnie, otwiera się on na inne techniki, takie jak na przykład karty CRC (ang. *Class-Responsibility-Collaboration*).

UML definiuje dwie podstawowe składowe: notację graficzną, definiującą elementy graficzne oraz składnię ich używania oraz meta model, czyli definicje pojęć języka i powiązania między nimi (Nawrocki, 2006).

Głównym celem języka UML jest ułatwienie komunikacji pomiędzy osobami zajmującymi się rozwojem oprogramowania realizowanego z użyciem paradygmatu obiektowości. Język naturalny jest nieprecyzyjny i może wprowadzać problemy w komunikacji dotyczącej złożonych pojęć. Z drugiej strony kod źródłowy jest bardzo precyzyjny, ale zbyt szczegółowy. Naturalnym rozwiązaniem są takie języki jak UML, pozwalające na wyrażanie skomplikowanych idei, ale jednocześnie pozwalające na stosowanie odpowiedniego poziomu szczegółowości. Poziomy szczegółowości, zwane perspektywami, dobrze są opisane w pracy (Fowler & Scott, 2002):

- perspektywa pojęciowa, obrazująca pojęcia i zależności stosowane w opisie systemu, często niezależne od samej implementacji systemu

- perspektywa specyfikacyjna, dotycząca przede wszystkim interfejsów obiektów stosowanych w implementacji systemu
- perspektywa implementacyjna, dotycząca całej implementacji systemu

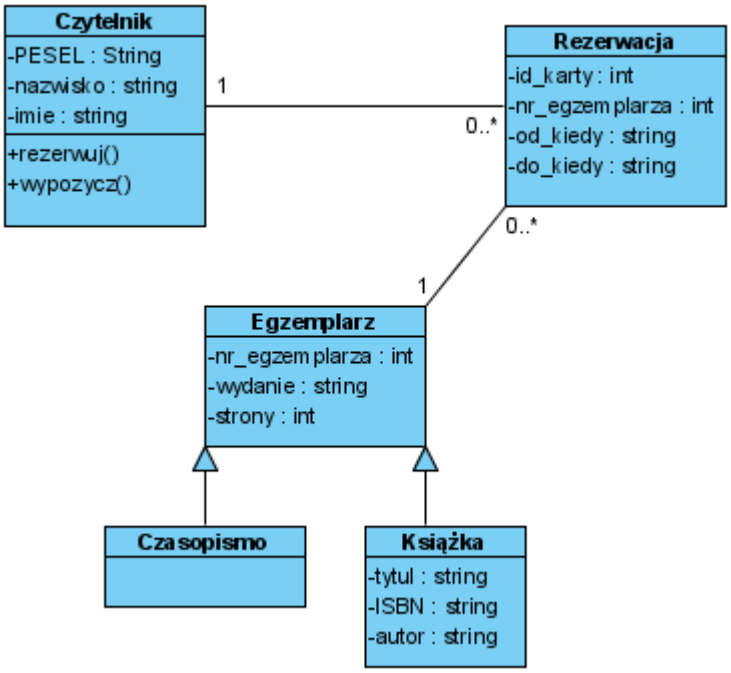
UML w najnowszej wersji 2.0 definiuje pewien zbiór diagramów, jakie można zastosować do opisu realizowanego systemu informatycznego. Kategoryzuje się je zwykle następująco (Nawrocki, 2006):

- diagramy struktury, obrazujące części składowe systemu:
    - diagramy klas (ang. *class diagram*): obrazują klasy, ich strukturę i zależności (patrz Rysunek 8). Jest to jeden z najczęściej stosowanych diagramów (obok diagramów przypadków użycia).
    - diagramy komponentów (ang. *component diagram*): obrazują komponenty (wymienne i wykonywalne fragmenty systemu o hermetyzowanych implementacjach) oraz zależności pomiędzy nimi
    - diagramy struktur połączonych (ang. *composite structure diagram*): obrazują hierarchicznie wewnętrzną strukturę złożonego obiektu z uwzględnieniem punktów interakcji z innymi częściami systemu
    - diagramy obiektów (ang. *object diagram*): prezentują możliwe konfiguracje obiektów (instancji klas) w określonym momencie
    - diagramy pakietów (ang. *packane diagram*): służą do modelowania fizycznego i logicznego podziału systemu (pakiety to element służący do strukturalizacji dowolnych bytów UML)
    - diagramy wdrożenia (ang. *deployment diagram*): odzwierciedlają fizyczną strukturę całego systemu z uwzględnieniem oprogramowania i sprzętu
- diagramy czynnościowe, obrazujące zachowanie systemu:
    - diagramy przypadków użycia (ang. *use case diagram*): służą do modelowania aktorów (użytkowników systemu, odbiorców efektów pracy systemu, systemów zewnętrznych) oraz ich potrzeb (w postaci sekwencji czynności) w stosunku do systemu (patrz Rysunek 9). Jest to jeden z najczęściej stosowanych diagramów (obok diagramów klas).
    - diagramy stanu (ang. *state machine diagram*): reprezentują zachowanie systemu lub jego elementu, a w szczególności zmiany tego zachowania

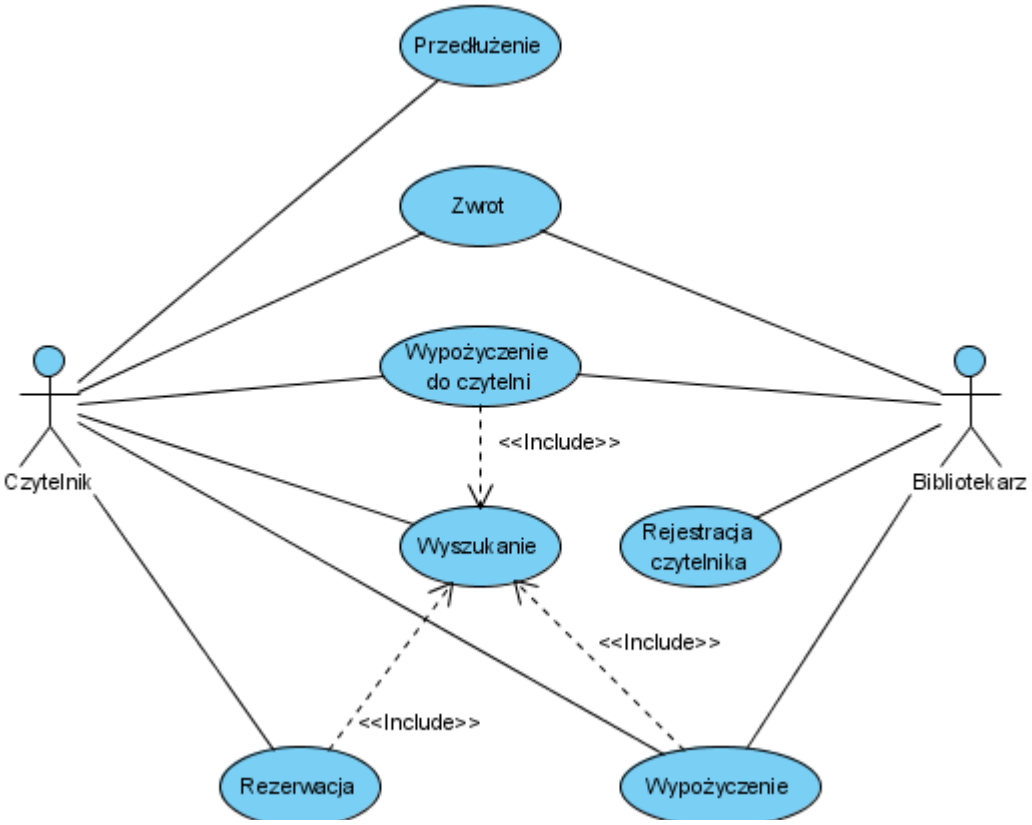
- diagramy czynności (ang. *activity diagrams*): prezentują przepływ sterowania w systemie związany z wykonywaniem pewnej funkcji

- diagramy interakcji, opisujące komunikację między obiektami:

  - diagramy sekwencji (ang. *sequence diagrams*): prezentują kolejność wywołań operacji, przepływ sterowania pomiędzy obiektami oraz szablon realizowanego algorytmu
  - diagramy komunikacji (ang. *communications diagrams*): opisuje obiekty wchodzące w skład interakcji oraz wymieniane przez nie komunikaty
  - diagramy przeglądu interakcji (ang. *interaction overview diagram*): służy do przedstawienia ogólnego przepływu sterowania w interakcjach pomiędzy obiektami, korzystając z uproszczonej notacji diagramu czynności
  - diagramy uwarunkowań czasowych (ang. *timing diagrams*): służą do prezentowania zależności związanych z czasem wykonywania operacji przez obiekt lub grupę obiektów



Rysunek 8 Przykład diagramu klas (źródło: opracowanie własne na podstawie (Nawrocki, 2006))



Rysunek 9 Przykład diagramu przypadków użycia (opracowanie własne na podstawie (Nawrocki, 2006))

Diagramy UML nie są receptą na powodzenie każdego przedsięwzięcia realizacji oprogramowania komputerowego. Są tylko środkiem, nie do końca doskonałym, który może pomóc analitykom, projektantom i programistom w sprawniejszej realizacji ich zadań. Wśród głosów krytyki najważniejsze to (Henderson-Sellers & Gonzalez-Perez, 2006):

- rosnąca złożoność języka: wraz z czasem złożoność języka UML rośnie: w kolejnych wersjach dodawane są coraz to nowe diagramy
- nieprecyzyjna semantyka: ze względu na to, że UML jest kombinacją abstrakcyjnej notacji, języka ograniczeń OCL (ang. *Object Constraint Language*; deklaratywny język definiowania zasad) oraz języka naturalnego to brakuje mu rygoru języka formalnego
- problemy w nauczaniu i stosowaniu wynikające ze złożoności i braku precyzji semantyki

- model a realizacja: jak dotąd nie opracowano skutecznych automatycznych metod pozwalających na pełne generowanie kodu źródłowego systemu na podstawie jego opisu w UML
- narzucanie określonych rozwiązań: jak każdy system notacji, również użycie UML sprawia, że przedstawienie reprezentacji systemu w określonych postaciach jest łatwiejsze i bardziej zwięzłe niż w innych: ogranicza to w pewien sposób używanie tego języka
- uniwersalność UML: język UML został opracowany jako narzędzie pozwalające współpracować z dowolnym językiem programowania, co może potencjalnie sprawić, że opracowany model nie będzie możliwy do zaimplementowania z użyciem konkretnego języka

Charakterystyczne dla języka UML jest to, że poszczególne diagramy są ze sobą luźno powiązane. Dlatego też podczas modelowania systemu istnieje pełna dowolność co do tego, jakie diagramy będą wykorzystane do jego opisu. Właściwość ta może być rozumiana jako zaleta języka, bo w większości przypadków bardziej szczegółowe diagramy potrzebne są tylko tam, gdzie obiekty charakteryzują się bardziej skomplikowaną strukturą lub schematem przetwarzania. Swoboda stosowania diagramów UML może być także rozumiana jako wada, szczególnie w przypadku zastosowania diagramów UML jako źródła danych do zastosowania na przykład w estymacji. W związku z tym, że nie ma żadnych formalnych zobowiązań co do stosowania poszczególnych rodzajów diagramów to implikuje to jednocześnie ograniczenie ilości danych, które potencjalnie można by wykorzystać w procesie pomiarów i szacowania.

Opisane wyżej problemy ze stosowaniem języka UML nie umniejszają znacząco jego roli we współczesnej inżynierii oprogramowania. Na dzień dzisiejszy jest on wiodącą i podstawową techniką wspierająca proces analizy i projektowania systemów realizowanych w paradygmacie obiektowym.

3.3 WZORCE PROJEKTOWE

Wzorce projektowe (ang. *design patterns*) pojawiły się po raz pierwszy w pracy (Gamma, Helm, Johnson, & Vlissides, 1995) jako specyfikacja oraz sposoby rozwiązania pewnych powtarzających się problemów w projektach zorientowanych obiektowo. Naj-

ważniejsze elementy wzorców projektowych to (Gamma, Helm, Johnson, & Vlissides, 1995):

- stosowna nazwa, będąca kluczem do opisu dziedziny problemu i sposobu jego rozwiązania, pozwalająca projektantom i programistom komunikować się na wyższym poziomie abstrakcji
- opis problemu projektowego wraz z jego kontekstem
- rozwiązanie problemu wraz z szkieletem jego projektu, zależnościami, zakresem odpowiedzialności i współpracy
- konsekwencje i kompromisy wynikające z zastosowania danego rozwiązania

Wzorce projektowe klasyfikuje się najczęściej w kategoriach problemów, jakie rozwiązują (Freeman, Freeman, Sierra, & Bates, 2005)(Cooper, 2001):

- wzorce fundamentalne (wzorce delegacji, projektu funkcjonalnego, interfejsu, pośrednika, fasady, kompozytu i inne)
- wzorce konstrukcyjne (wzorce abstrakcyjnej fabryki, metody fabrykującej, budowniczego, leniwej inicjalizacji, prototypu, singletona i inne)
- wzorce strukturalne (wzorce adaptera, agregatu, mostu, kontenera, dekoratora, wagi piórkowej i inne)
- wzorce czynnościowe (wzorce łańcucha odpowiedzialności, komendy, interpretera, iteratora, mediatora, memento, obserwatora, stanu, strategii, szablonu, wizytatora i inne)

Swego rodzaju uzupełnieniem wzorców projektowych są zaprezentowane w pracy (Brown, Malveau, McCormick, & Mowbray, 1998) antywzorce, czyli najczęściej pojawiające się błędne rozwiązania typowych problemów architekturalnych.

Wzorce projektowe są o tyle cenne, że stanowią następny krok ponad zrozumienie podstaw języka czy technik modelowania. Wzorce dają wiele rozwiązań oraz mówią, co jest dobrym modelem i w jaki sposób konstruować model.

4 ESTYMACJA ROZMIARU OPROGRAMOWANIA

Rozdział ten stanowi przegląd pojęć dotyczących rozmiaru oprogramowania i jego podstawowych aspektów: długości, funkcjonalności i złożoności. Zawiera opis podstawowych metryk stosowanych do wyrażania rozmiaru, jak i ogólny opis najpopularniejszych metod estymacji dostępnych w literaturze.

4.1 POJĘCIE ROZMIARU

Rozmiar jest pojęciem szeroko znanym i powszechnie używanym w odniesieniu do obiektów świata codziennego. Praktycznie każdemu elementowi z otoczenia człowieka można przypisać cechę zwaną rozmiarem i określić jej wartość. O ile samo pojęcie rozmiaru jest zupełnie naturalne, to określanie jego wartości nie jest już takie proste. Do oceny rozmiaru można używać najrozmaitszych innych cech, takich jak na przykład wysokość (dla opisu rozmiaru drzewa) i inne wymiary przestrzenne (szerokość, głębokość), powierzchnię (dla opisu wielkości domu) czy masa (dla opisu wielkości zwierzęcia). Rozmiar może również w pewien sposób być predyktorem innych cech obiektów świata rzeczywistego: przykładowo, można uznać, że większy człowiek jest również silniejszy. W technice powszechne jest przekonanie, że realizacja większego przedsięwzięcia (na przykład budowa większego domu) jest kosztowniejsza i wymaga większych nakładów pracy.

Wytwarzanie oprogramowania komputerowego, pomimo jego abstrakcyjności, nosi wiele cech elementów świata rzeczywistego. Pojęcie rozmiaru jest również aktualne dla oprogramowania komputerowego. Ponadto relacja pomiędzy rozmiarem oprogramowania a nakładami pracy i kosztem jego realizacji jest identyczna jak w przypadku przedsięwzięć ze świata rzeczywistego: im większe jest oprogramowanie tym większy jest koszt oraz nakład pracy potrzebny do jego wytworzenia. Zależność ta sprawia, że rozmiar to jedna z najistotniejszych cech oprogramowania, bo pośrednio lub bezpośrednio wpływa na praktycznie wszystkie pozostałe cechy zarówno samego produktu, jak i procesu jego wytwarzania.

Oprogramowanie, jako byt abstrakcyjny nie może być opisywane przez cechy fizyczne używane do charakterystyki świata rzeczywistego, takie jak wysokość czy masa. Dlate-

go też już u zarania informatyki pojawił się problem definicji rozmiaru oprogramowania. Podobnie jak to się ma w przypadku rozmiarów elementów otoczenia człowieka, rozmiar oprogramowania nie może być jednoznacznie zdefiniowany. Obecnie uznaje się (Fenton & Neil, 2000), że rozmiar oprogramowania ma trzy podstawowe aspekty:

- aspekt długości
- aspekt funkcjonalności
- aspekt złożoności

W kolejnych podrozdziałach opisane dokładniej zostaną wszystkie trzy aspekty rozmiaru oprogramowania wraz z metrykami służącymi do ich pomiaru oraz najważniejszymi istniejącymi metodami ich estymacji.

4.2 ESTYMACJA ROZMIARU OPROGRAMOWANIA W UJĘCIU DŁUGOŚCI

Długość kodu źródłowego implementującego dany program jest najstarszą znaną techniką wyrażania rozmiaru oprogramowania. Po raz pierwszy zaczęto go wykorzystywać w połowie lat sześćdziesiątych ubiegłego wieku, głównie do celów przewidywania wysokości nakładów pracy przy realizacji dużych przedsięwzięć informatycznych. Do dzisiaj, mimo pewnego spadku znaczenia długości kodu źródłowego w pomiarach i estymacji, ciągle stosuje się go jako czynnik normalizujący dla porównań różnych metryk i metod estymacji.

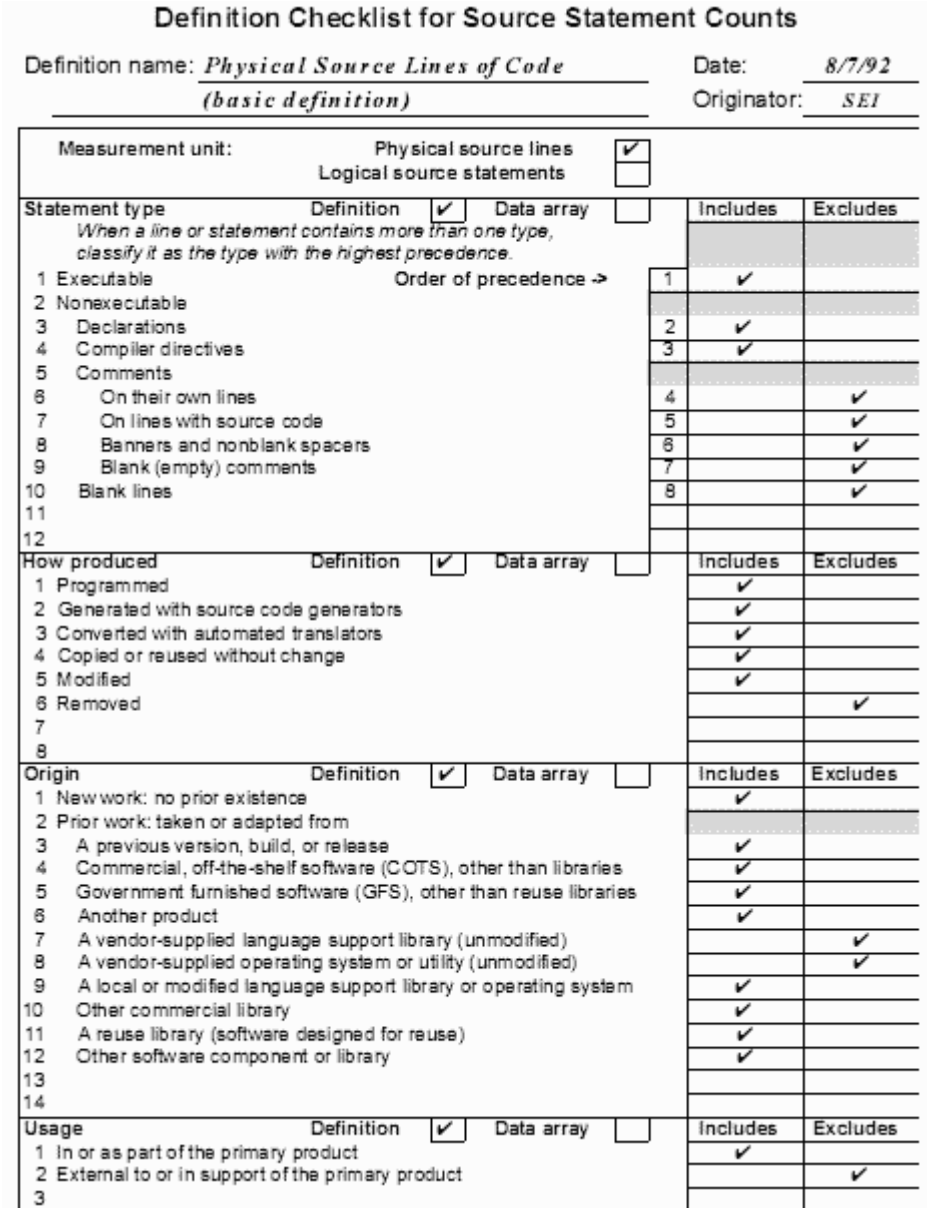
4.2.1 Metryki rozmiaru oprogramowania w ujęciu długości kodu źródłowego

Długość kodu źródłowego mierzy się przy pomocy różnych miar, przy czym wszystkie bazują na pojęciu linii kodu źródłowego. Podstawowym problemem z mierzeniem długości przy pomocy liczby linii kodu źródłowego jest to, że nie ma ustalonej ogólnej definicji tego pojęcia. Większość języków programowania cechuje się tym, że zupełnie nieistotne są w nich takie kwestie jak znak końca linii (poza jego znaczeniem jako separatora leksykalnego i jako literał), co sprawia, że w ramach jednej linii kodu źródłowego można zawrzeć potencjalnie dowolną ilość instrukcji.

Najbardziej znane metryki długości kodu źródłowego to (Galorath & Evans, 2006):

- liczba znaków końca linii
- (*S*)*LOC* (ang. (so*urce*) *lines of code*) – liczba niepustych nie będących komentarzem logicznych linii kodu źródłowego
- logiczna *SLOC* – liczba wyrażeń języka programowania zastosowanych w źródłach (wymaga analizy syntaktycznej źródeł; czasami może się sprowadzić po prostu do liczby terminatorów leksykalnych)
- *DSI* lub *ELOC* (ang. *delivered source instructions*, *executable lines of code*) – liczba niepustych nie będących komentarzem logicznych linii kodu źródłowego z pominięciem deklaracji danych, dyrektyw kompilatora i innych linii nie generujących wykonywalnych instrukcji
- *ESLOC* (ang. *effective source lines of code*) – liczba *SLOC* zaimplementowanych jako dodatkowe (wartość dodana) na istniejącym już systemie informatycznym

Różnorodność definicji pojęcia liczby linii kodu źródłowego, a także ich niekonsekwentne używanie sprawiło, że pojawiła się potrzeba dokładnego definiowania pojęcia do konkretnych zastosowań. Dobrym tego przykładem jest lista wymagań co do definicji linii kodu źródłowego opracowana przez wspominany już instytut SEI (Rysunek 10). Zastosowanie ustalonej definicji linii kodu źródłowego pozwala na stosunkowo proste ich liczenie, także z użyciem narzędzi automatyzujących ten proces.



Definition Checklist for Source Statement Counts

Definition name: *Physical Source Lines of Code (basic definition)* Date: *8/7/92* Originator: *SEI*

Measurement unit: Physical source lines ✔ / Logical source statements

Statement type	Definition ✔	Data array		Includes	Excludes
When a line or statement contains more than one type, classify it as the type with the highest precedence.					
1 Executable	Order of precedence ->		1	✔	
2 Nonexecutable					
3 Declarations			2	✔	
4 Compiler directives			3	✔	
5 Comments					
6 On their own lines			4		✔
7 On lines with source code			5		✔
8 Banners and nonblank spacers			6		✔
9 Blank (empty) comments			7		✔
10 Blank lines			8		✔
11					
12					

How produced	Definition ✔	Data array	Includes	Excludes
1 Programmed			✔	
2 Generated with source code generators			✔	
3 Converted with automated translators			✔	
4 Copied or reused without change			✔	
5 Modified			✔	
6 Removed				✔
7				
8				

Origin	Definition ✔	Data array	Includes	Excludes
1 New work: no prior existence			✔	
2 Prior work: taken or adapted from				
3 A previous version, build, or release			✔	
4 Commercial, off-the-shelf software (COTS), other than libraries			✔	
5 Government furnished software (GFS), other than reuse libraries			✔	
6 Another product			✔	
7 A vendor-supplied language support library (unmodified)				✔
8 A vendor-supplied operating system or utility (unmodified)				✔
9 A local or modified language support library or operating system			✔	
10 Other commercial library			✔	
11 A reuse library (software designed for reuse)			✔	
12 Other software component or library			✔	
13				
14				

Usage	Definition ✔	Data array	Includes	Excludes
1 In or as part of the primary product			✔	
2 External to or in support of the primary product				✔
3				

Rysunek 10 Przykład definicji linii kodu źródłowego (Park, 1992)

Długość kodu źródłowego jest w dużym stopniu zależna od siły ekspresji języka programowania. Języki niskopoziomowe (np. asembler) mają niską siłę ekspresji, bo jedna instrukcja w takich językach wyraża o wiele mniej niż jedna instrukcja w językach wysokopoziomowych (np. Visual Basic). Dobrym przykładem siły ekspresji języka jest instrukcja warunkowa: o ile w językach wysokopoziomowych instrukcja tak składa się zwykle tylko z samej instrukcji oraz warunku logicznego, to w językach niskopoziomowych zawiera kopiowanie wartości zmiennej do rejestru procesora, wywołanie operacji matematycznej i wykonanie skoku w zależności od rezultatu tej ostatniej.

Tabela 11 Porównanie siły ekspresji języków programowania

instrukcja warunkowa w języku Pascal	instrukcja warunkowa w asemblerze x86

`if x > 3 then`
`  -- akcja 1`
`-- akcja 2`
	`mov AX, @x`
	`sub AX, 3`
	`jnz @end_clause`
	`; akcja 1`
	`end_clause:`
	`; akcja 2`

Źródło: opracowanie własne

Siła ekspresji języków uniemożliwia bezpośrednie porównania długości kodu źródłowego zrealizowanych z ich pomocą. Można tego dokonać normalizując uzyskane wyniki, na przykład za pomocą tabeli normalizacji siły ekspresji języków programowania (Tabela 12). Zawartość tej tabeli należy interpretować następująco: im mniejsza wartość współczynnika, tym większa siła ekspresji (mniej linii kodu jest wymagane, żeby zaimplementować daną funkcjonalność).

Tabela 12 Współczynniki normalizacyjne siły ekspresji języków programowania

język programowania	**współczynnik normalizacyjny**
Access (Microsoft)	38
Ada 83	71
Ada 95	49
APL	32
Asembler (Basic)	320
Asembler (Macro)	213
Basic (ANSI)	64
Basic (Visual)	32
C	128
C++	55
Cobol (ANSI 85)	91
Fortran 95	71
HTML 3.0	15
Java	53
Excel (Microsoft)	6

Źródło: (Jones, 1991)

Podstawowe zalety ilości linii kodu źródłowego jako metryki rozmiaru to (Galorath & Evans, 2006):

- jest to pojęcie znane i używane od lat, służące jako podstawowa metryka w wielu modelach (np. *COCOMO*)
- dobrze skorelowane z nakładem pracy na realizację systemu informatycznego
- liczenie ilości linii kodu źródłowego jest stosunkowo łatwe (pod warunkiem zachowania ścisłych definicji pojęć)
- używane jako podstawa do wyznaczania innych metryk (na przykład średniej gęstości błędów)

Podstawowe wady ilości linii kodu źródłowego jako metryki rozmiaru to (Galorath & Evans, 2006):

- nie może być używany jako metryka rozmiaru podczas wszystkich etapów cyklu życia projektu, gdyż na przykład podczas analizy i projektowania ilość kodu źródłowego jest znikoma lub nie ma go wcale
- mierzenie produktywności za pomocą ilości linii kodu źródłowego napisanego w jednostce czasu jest nieetyczne, bo algorytm można zaimplementować na różne sposoby, nawet z użyciem tego samego języka programowania

Oprócz klasycznych miar długości kodu źródłowego bazujących na pojęciu linii używa się również miar o podobnej charakterystyce, ale uwzględniających specyfikę paradygmatu programowania z wykorzystaniem którego systemy informatyczne są tworzone. Na przykład dla oprogramowania realizowanego zgodnie z paradygmatem obiektowości metrykami komplementarnym dla ilości linii kodu źródłowego mogą być ilości podstawowych bytów specyficznych dla świata obiektów, takich jak metody, atrybuty, klasy.

4.2.2 Metody estymacji rozmiaru oprogramowania w ujęciu długości kodu źródłowego

Większość metod estymacji rozmiaru w ujęciu długości kodu źródłowego to rozmaite, często nieformalne, metody bazujące na wiedzy eksperta bądź analogii do zrealizowanych już rozwiązań. Mimo, że od dawna istniała potrzeba estymacji długości kodu źródłowego (na przykład najbardziej znana metoda szacowania kosztów realizacji *COCOMO* używa

oszacowania długości kodu źródłowego jako dane wejściowe) to sformalizowanych metod jest bardzo niewiele. Po części wynikało to z braku standardów analizy i projektowania systemów informatycznych – praktycznie każda organizacja zajmująca się wytwarzaniem oprogramowania opracowywała własne, często niejawne praktyki, co znacząco utrudniało rozwój ogólnoświatowych wspólnych rozwiązań. Dopiero stosunkowo niedawno pojawił się pewien promyk nadziei w postaci języka UML, przy pomocy którego zaproponowano sposób prezentacji struktury i zachowania systemów informatycznych. Jednakże, jak już wcześniej wspomniano, sam język to nie wszystko – potrzebne są jeszcze ustalone praktyki postępowania, a do tego celu droga jest jeszcze daleka. Pojawiają się co prawda pewne popularniejsze rozwiązania (na przykład *Rational Unified Process*), ale nie osiągnęły one jeszcze statusu standardu.

W chwili obecnej większość sformalizowanych metod estymacji rozmiaru w ujęciu długości linii kodu źródłowego opiera się na diagramach UML, które stanowią solidną podstawę pomiarową. Najbardziej znane przykłady takich metod to:

- metoda *PROBE*
- metoda *Fast&&Serious*

Metoda *PROBE*

Metoda *PROBE* (ang. *Proxy-based Estimation*) jest elementem *Personal Software Process*, zbioru najlepszych praktyk wprowadzania dyscypliny do procesów wytwarzania oprogramowania opracowanych w instytucie SEI (stamtąd pochodzi również model potencjału i dojrzałości CMM). Opiera się ona na pojęciu pośrednika (ang. *proxy*), czyli bytu zastępczego służącego jako podstawa do estymacji na etapach planowania. Dobry pośrednik powinien mieć następujące cechy (Pomeroy-Huff, Mullaney, Cannon, & Sebern, 2005):

- być skorelowany z kosztami rozwoju
- być łatwy do identyfikacji na wczesnych etapach rozwoju systemu informatycznego
- być bytem fizycznym, który może być mierzony i automatycznie zliczany

Estymacja rozmiaru przy pomocy metody *PROBE* w dużym stopniu opiera się na historycznych danych z realizacji poprzednich projektów: właśnie na ich podstawie wyznacza się przedziały służące do oceny rozmiaru pośredników. Cała procedura wyznaczania

oszacowania rozmiaru składa się z następujących kroków (Pomeroy-Huff, Mullaney, Cannon, & Sebern, 2005):

- opracowanie projektu konceptualnego (wysokopoziomowy podział systemu na podstawowe elementy)
- identyfikacja i wyznaczenie rozmiaru pośredników (rozmiar jest wyznaczany jako przynależność do przedziałów rozmiaru wyznaczonych na podstawie analizy danych historycznych)
- oszacowanie rozmiaru systemu w liniach kodu źródłowego (za pomocą regresji liniowej)
- wyznaczenie interwałów predykcji (przedziały do których oszacowania rozmiaru wpadają z prawdopodobieństwem większym niż 70%)

Metoda *Personal Software Process* znacząco wpływa na stopniową poprawę skuteczności szacowania. Badania przeprowadzane na studentach biorących udział w kursie *PSP* pokazały, że po zakończeniu kursu są w stanie przewidywać potrzebny nakład pracy (wyliczany na podstawie estymacji rozmiaru) średnio ponad dwukrotnie dokładniej (Humphrey, 2000).

Metoda *PROBE* ma kilka cech, które potencjalnie można uznać za wady. Najistotniejszą z nich jest to, że wymaga istnienia danych historycznych charakteryzujących się dużym stopniem podobieństwa do aktualnie realizowanego systemu, które w wielu przypadkach mogą nie istnieć.

Metoda *Fast&&Serious*

Metoda *Fast&&Serious* została zaprezentowana 2002 roku (Carbone & Santucci, 2002). Składa się z dwóch podmetod (*Fast* i *Serious*) realizowanych w zależności od dostępnego poziomu dokładności modelu systemu. Metoda ta analizuje diagramy UML opisujące projektowane oprogramowanie i na ich podstawie wyznacza oszacowanie jego rozmiaru wyrażonego w ilości linii kodu źródłowego. Zadanie to wykonywane jest w następującej sekwencji kroków (Carbone & Santucci, 2002):

- wyznaczenie podstawowych metryk obiektowych oraz podjęcie decyzji, czy wybrać metodę uproszczoną (*Fast*) czy pełną (*Serious*)
- wyznaczenie złożoności każdej klasy na podstawie diagramu klas

- ocena złożoności klasy w ujęciu diagramów przypadków użycia (krok opcjonalny)
- ocena złożoności klasy w ujęciu diagramów interakcji (krok opcjonalny)
- ocena złożoności klasy w ujęciu diagramów stanów (krok opcjonalny)
- suma złożoności klas uzyskanych w poprzednich krokach i wyznaczenie szacunkowego rozmiaru systemu w kategoriach długości kodu źródłowego

Metryki obiektowe na których operuje metoda to (Carbone & Santucci, 2002):

- głębokość drzewa dziedziczenia (*DIT* – ang. *Depth in Inheritance Tree*)
- liczba metod klasy (*MPC* – ang. *Number of Methods per Class*)
- liczba asocjacji klasy (*NAC* – ang. *Number of Association per Class*)
- odsetek metod z sygnaturami, czyli dokładnymi deklaracjami postaci metod z uwzględnieniem parametrów i rezultatów (*PMS* – ang. *Percentage of Methods with Signatures*)

Autorzy sugerują, żeby metrykę *PMS* stosować jako zmienną decyzyjną w wyborze rodzaju podmetody (*Fast* czy *Serious*).

Złożoność klasy w ujęciu diagramów klas obliczana jest w oparciu o klasyfikację złożoności atrybutów klasy (proste, skomplikowane, importowane), złożoność metod na podstawie analizy stopnia skomplikowania (identycznie jak dla atrybutów) jej parametrów i rezultatów oraz liczbie asocjacji.

Złożoność klasy w ujęciu diagramów przypadków użycia polega na ocenie złożoności aktorów (na podstawie liczby asocjacji i głębokości drzewa dziedziczenia) i przypadków użycia (na podstawie liczby komunikatów wymienianych w jego ramach) z uwzględnieniem tylko tych z nich, które dotyczą konkretnej klasy.

Złożoność klasy w ujęciu diagramów interakcji polega na wyznaczeniu ilości komunikatów wysyłanych i odbieranych przez konkretną klasę i ustaleniu, jaki jest jej udział w całości komunikacji systemu.

Złożoność klasy w ujęciu diagramów stanów polega na wyznaczeniu modyfikatora ostatecznej liczby linii kodu źródłowego na podstawie liczby stanów i akcji.

Wyznaczenie ostatecznego prognozowanego rozmiaru systemu polega na podstawieniu zsumowanych wartości uzyskanych w poprzednich krokach do wzoru

$$Size(c) = 4 + 10 * CP(c)^{0.7} \quad 4.07$$

gdzie *CP(c)* to wspomniana suma oszacowań złożoności klasy *c* oraz zsumowaniu uzyskanych wyników, co prowadzi do wyznaczenia ostatecznego oszacowania całego systemu w kategoriach ilości linii kodu źródłowego.

Metoda *Fast&&Serious* jest stosunkowo nowa i praktycznie brak jakichkolwiek studiów dotyczących jej skuteczności. Jako zaletę w przypadku tej metody można uznać jej wielopłaszczyznową analizę złożoności metody. Wady wynikają przede wszystkim z użycia diagramów UML: w praktyce rzadkością jest tworzenie diagramów innych niż diagramy klas, co może sprawić, że metoda będzie miała potencjalnie mniejszy zakres stosowalności i mniejszą skuteczność, bo pomijać będzie wiele aspektów złożoności metody związanych z diagramami interakcji i stanów.

4.3 ESTYMACJA ROZMIARU OPROGRAMOWANIA W UJĘCIU FUNKCJONALNYM

Potrzeba charakteryzowania rozmiaru oprogramowania w ujęciu funkcjonalności jaką ono implementuje pojawiła się pod koniec lat 70 ubiegłego wieku jako odpowiedź na problemy związane z mierzeniem rozmiaru w ujęciu długości kodu źródłowego. Uznano wówczas (zgodnie z prawdą), że długość linii kodu źródłowego bardziej związana jest z technologiami inżynierii oprogramowania niż z samym problemem, dla którego rozwiązania oprogramowanie jest tworzone. Pojawiła się potrzeba spojrzenia na tworzone systemy informatyczne z punktu widzenia użytkownika, którego interesuje przede wszystkim funkcjonalność za którą płaci, a znacznie mniej technologie użyte do jej dostarczenia.

4.3.1 Metryki rozmiaru oprogramowania w ujęciu funkcjonalnym

Metryki wyrażające rozmiar oprogramowania w ujęciu funkcjonalnym są metrykami pośrednimi i abstrakcyjnymi, w przeciwieństwie do linii kodu źródłowego, które są wartościami bezpośrednio mierzalnymi (na podstawie analizy kodu źródłowego) i fizycznymi (da się je łatwo rozpoznawać). Metryki funkcjonalne są ściśle powiązane z metodami estymacji rozmiaru funkcjonalnego oprogramowania. We wszystkich metodach są one ro-

zumiane jako liczby, które odnoszą się do zakresu funkcjonalności zrealizowanego w ramach systemu informatycznego będącego przedmiotem badania.

Podstawowe zalety metryk funkcjonalności na przykładzie punktów funkcyjnych *IFPUG* (patrz rozdział 4.3.2) to (Galorath & Evans, 2006):

- są powszechnie znane i akceptowane
- dostępnych jest wielu certyfikowanych specjalistów
- nieskorygowana liczba punktów funkcyjnych jest niezależna od technologii użytych przy wytwarzaniu oprogramowania
- punkty funkcyjne mogą być wyznaczane już na etapie analizy cyklu życia oprogramowania
- standardy liczenia punktów funkcyjnych są aktywnie rozwijane przez organizację *IFPUG*

Podstawowe wady metryk funkcjonalności na przykładzie punktów funkcyjnych *IFPUG* to (Galorath & Evans, 2006):

- wymagają treningu
- nie można ich porównywać numerycznie: aplikacja o 1000 punktów funkcyjnych nie jest dwa razy większa, bardziej złożona czy bardziej funkcjonalna niż aplikacja o 500 punktach funkcyjnych; nie jest dwukrotnie lepsza w żadnym znaczącym sensie (Kitchenham, 1997)
- trudności semantyczne: punkty funkcyjne zostały stworzone w latach osiemdziesiątych XX wieku dla potrzeb typowych systemów informatycznych zarządzania (*MIS* – ang. *Management Information Systems*) i od tego czasu nie przeszły procesu gruntownej rewitalizacji, w związku z czym operują na wielu w pewnym sensie archaicznych (z dzisiejszego punktu widzenia) pojęciach
- niekompletność: punkty funkcyjne są określane z punktu widzenia końcowego użytkownika systemu, co może prowadzić do tego, że istotna część funkcjonalności ukryta przed użytkownikiem zostanie pominięta w procesie liczenia
- brak narzędzi wspierających automatyczne zliczanie punktów funkcyjnych, nawet dla systemów już zrealizowanych
- ograniczony zakres zastosowań: metody punktów funkcyjnych uznają ilość przechowywanych danych za istotny czynnik wpływający na funkcjonalność systemu i w

związku z tym systemy informatyczne charakteryzujące się małą ilością przechowywanych danych, ale dużą złożonością procesów przetwarzania danych są zwykle niedoszacowane w sensie funkcjonalnym

Dużym problemem metryk funkcjonalnych jest również to, że są one bezpośrednio związane z wykorzystującymi je metodami i tylko z nimi, co sprawia, że bardzo trudno jest porównywać bezpośrednio uzyskiwane przez nie wyniki. W praktyce w sytuacjach gdy dochodzi do takich porównań wyniki należy przekonwertować na tę samą jednostkę, którą najczęściej jest linia kodu źródłowego.

4.3.2 Metody estymacji rozmiaru oprogramowania w ujęciu funkcjonalnym

Dzisiejszy stan wiedzy na temat metod szacowania funkcjonalności pozwala na wyróżnienie dwóch podstawowych kategorii metod:

- punkty funkcyjne *IFPUG* i metody pokrewne (*Feature Points*, *3D Function Points*, *MkII Function Points*, metoda *COSMIC-FFP*)
- metody bazujące na diagramach przypadków użycia (*Use Case Points*)

Historia metod punktów funkcyjnych zaczęła się w 1979 roku, kiedy to Allan Albrecht zaprezentował nową metodę (Albrecht, 1979) wyznaczania rozmiaru oprogramowania w kategoriach jego funkcjonalności. W latach późniejszych była kilkakrotnie uszczegóławiana przez autora, aż do roku 1986, kiedy to powstała organizacja nazwana *IFPUG* (ang. *International Function Points User Group*), która przejęła zadanie opieki nad metodą. Od tego czasu metoda punktów funkcyjnych rozpowszechniła się i zyskała szersze uznanie. Liczne organizacje zaczęły używać nowej metody i weryfikować jej stosowalność oraz wyniki działania. Rezultatem tego był rozwój nie tylko samej metody, ale i w szerszym ujęciu, podejścia do mierzenia rozmiaru oprogramowania w perspektywie funkcjonalnej: oprócz uaktualniania i usprawniania samej metody punktów funkcyjnych przez *IFPUG* powstały również konkurencyjne metody bazujące na propozycji Allana Albrechta.

Metoda *IFPUG*

Metoda punktów funkcyjnych *IFPUG* jest rozwinięciem metody zaproponowanej przez Allana Albrechta. W ciągu ponad 20 lat swego istnienia była ona wielokrotnie uaktualniana: najnowsza jej wersja zaprezentowana w 2004 roku nosi numer 4.2. Przez te

wszystkie lata sama metoda zasadniczo nie zmieniła się – wszelkie jej modyfikacje dotyczyły przede wszystkim zasad interpretacji i stosowalności pojęć i reguł opisanych w jej ramach.

Ostatecznym celem metody punktów funkcyjnych *IFPUG* jest wyznaczenie skorygowanej liczby punktów funkcyjnych dla systemu będącego przedmiotem stosowania metody. W tym celu wymagane jest wykonanie następujących kroków (Longstreet, 2004):

- określenie rodzaju wyliczanych punktów funkcyjnych
- określenie granic systemu będącego przedmiotem analizy
- identyfikacja i ocena złożoności elementów transakcyjnych
- identyfikacja i ocena złożoności elementów danych
- wyznaczenie wartości czynnika korygującego
- wyznaczenie skorygowanej liczby punktów funkcyjnych

Podręcznik *IFPUG* wymienia trzy rodzaje punktów funkcyjnych (Longstreet, 2004):

- punkty funkcyjne rozwoju projektu (ang. *development project function points*)
- punkty funkcyjne rozszerzania projektu (ang. *enhancement project function points*)
- punkty funkcyjne aplikacji (ang. *application function point*)

Punkty funkcyjne rozwoju projektu służą do mierzenia funkcjonalności systemu będącego na etapie realizacji. Wraz z postępem procesu wytwarzania muszą one być uaktualniane.

Punkty funkcyjne rozszerzania projektu mają na celu pomiar funkcjonalności modyfikowanej w kolejnych wersjach systemu, a więc wszelkiego rodzaju dodawania, usuwania i modyfikacji istniejącej już funkcjonalności.

Punkty funkcyjne aplikacji wykorzystuje się do pomiaru funkcjonalności dostarczonej w ostatecznej produkcyjnej wersji systemu.

Rodzaj wyznaczanych punktów funkcyjnych ma przede wszystkim wpływ na procedurę wyznaczania wartości czynnika korygującego.

Określenie granic systemu będącego przedmiotem analizy ma na celu wyznaczenie zbioru podstawowych elementów systemu, które będą miały wpływ na ostateczną liczbę

punktów funkcyjnych. Owo rozgraniczenie powinno bazować na funkcjonalności, a nie na przesłankach technicznych.

W ramach granic systemu dokonuje się identyfikacji podstawowych elementów wchodzących w jego skład, mianowicie elementów transakcyjnych i elementów danych. Metoda *IFPUG* definiuje trzy podstawowe elementy transakcyjne, z których każdy powinien być procesem elementarnym, samodzielnym oraz zapewniającym spójny stan systemu (Longstreet, 2004):

- wejście danych (ang. *External Input – EI*) – elementarne działanie, w którym dane są wprowadzane w granice systemu i mogą zasilać wewnętrzne pliki danych (*ILF*)
- wyjście danych (ang. *External Output – EO*) – elementarne działanie, w którym przetworzone dane z wewnętrznych plików danych (*ILF*) są wyprowadzane poza granice systemu
- zapytanie (ang. *External Inquiry – EQ*) – elementarne działanie obejmujące operacje wejścia (nie modyfikują wewnętrznych plików danych (*ILF*)) i wyjścia (wyprowadzają nie przetworzone dane z wewnętrznych plików danych (*ILF*) lub zewnętrznych plików danych (*EIF*))

Metoda *IFPUG* definiuje 2 podstawowe elementy danych (Longstreet, 2004):

- wewnętrzny plik danych (ang. *Internal Logical File – ILF*) – identyfikowalny przez użytkownika logicznie powiązany zbiór danych, który znajduje się całkowicie w granicach systemu i jest zasilany przez wejścia danych
- zewnętrzny plik danych (ang. *External Interface File – EIF*) – identyfikowalny przez użytkownika logicznie powiązany zbiór danych, który znajduje się całkowicie poza granicami systemu i może być tylko odczytywany

Zidentyfikowane elementy transakcyjne i elementy danych należy poddać ocenie ich złożoności, która polega na identyfikacji w ich ramach następujących bytów (Longstreet, 2004):

- elementarny rekord (ang. *Record Element Type – RET*) – określona przez użytkownika podgrupa danych w *ILF* lub *EIF*
- elementarne pole danych (ang. *Data Element Type – DET*) – określone przez użytkownika unikalne pole danych

- plik będący operandem (ang. *Filr Type Referenced – FTR*) – plik na którym przeprowadzana jest transakcja

Dla każdego elementu transakcyjnego i elementu danych określa się jego złożoność za pomocą wartości lingwistycznych „mały", średni", „duży" na podstawie ilości odpowiednio:

- *DET* i *FTR* dla elementów transakcyjnych (*EI*, *EO*, *EQ*);
- *DET* i *RET* dla elementów danych (*ILF*, *EIF*).

Odwzorowanie liczby poszczególnych bytów na wartości lingwistyczne odbywa się w oparciu o tabele mapowań, które można znaleźć w podręcznikach metody punktów funkcyjnych.

Każda wartość lingwistyczna dla wszystkich elementów transakcyjnych i danych ma przypisaną odpowiednią wagę liczbową. Suma wartości wag (będących ocenami złożoności poszczególnych instancji elementów transakcyjnych i danych) stanowi nieskorygowaną ilość punktów funkcyjnych dla danego systemu.

Czynnik korygujący (ang. *Value Adjustment Factor – VAF*) został wprowadzony w celu uwzględnienia wpływu czynników technicznych na liczbę punktów funkcyjnych badanego systemu (Albrecht, 1985):

- liczba mechanizmów komunikacji i wymiany danych (ang. *Data Communications*)
- stopień rozproszenia przetwarzania i danych (ang. *Distributed Functions*)
- wymagania co do przepustowości i czasu odpowiedzi (ang. *Performance*)
- obciążenie docelowej platformy (ang. *Heavily Used Configuration*)
- liczba i częstość uruchamiania transakcji (ang. *Transaction Rate*)
- ilość informacji wprowadzanych w trybie on-line (ang. *On-line Data Entry*)
- efektywność (ang. *End-User Efficiency*)
- ilość informacji modyfikowanych w trybie on-line (ang. *On-line Update*)
- złożoność przetwarzania (ang. *Complex Processing*)
- łatwość wykorzystania w nowych wdrożeniach (ang. *Reusability*)
- łatwość instalacji (ang. *Installation Ease*)
- łatwość utrzymania (ang. *Operational Ease*)
- liczba niezależnych instalacji (ang. *Multiple Sites*)

- łatwość modyfikacji systemu (ang. *Facilitate Change*)

Dla każdej z charakterystyk powinno ocenić się jej wpływ na kształt systemu w liczbowej skali całkowitej z zakresu od 0 (brak wpływu) do 6 (silny wpływ). Oceny te należy zsumować, uzyskując w ten sposób współczynnik złożoności technicznej (ang. *Technical Complexity Factor – TCF*), który podstawiony do równania 4.08 daje w rezultacie poszukiwany czynnik korygujący *VAF*.

$$VAF = 0.65 + 0.01 * TCF \qquad 4.08$$

Ostatecznie, liczba skorygowanych punktów funkcyjnych to iloczyn nieskorygowanej liczby punktów funkcyjnych i czynnika korygującego. W zależności od rodzaju liczonych punktów funkcyjnych (rozwoju, rozszerzania, aplikacji) stosuje się drobne modyfikacje do formuły (szczegóły są dostępne na przykład w (Longstreet, 2004)).

Feature Points

Metoda *Feature Points* została opracowana w Software Productivity Research w 1986 roku (Jones, 1986). Przyczyną jej opracowania była krytyka metody *IFPUG* dotycząca możliwych obszarów zastosowań. Metoda *IFPUG* została opracowana do stosowania w odniesieniu do projektów z dziedziny informatycznych systemów zarządzania (ang. *Management Information System – MIS*), które mają inną charakterystykę przetwarzania danych niż np. systemy czasu rzeczywistego (ang. *Real-Time – RT*) czy aplikacje naukowe (ang. *Algorithmic/Scientific – A/S*). Podstawowym rozszerzeniem w stosunku do metody *IFPUG* było wprowadzenie nowego elementu podlegającego identyfikacji i ocenie – algorytmów. Jednakże największa zmiana stała się zarazem najbardziej problematyczna – okazało się, że ujednolicenie sposobu oceny złożoności algorytmów rodzi tyle sporów wśród ekspertów, że jest niemożliwe do wykonania. W rezultacie metoda ta została zarzucona i nie jest obecnie już rozwijana.

3D Function Points

Metoda trójwymiarowych punktów funkcyjnych została opracowana przez firmę Boeing w 1991 roku jako sposób na rozszerzenie potencjalnego pola zastosowania oryginalnej metody *IFPUG* na obszar systemów czasu rzeczywistego (*RT*). Nazwa metody wywodzi się od trzech wymiarów oprogramowania: danych, funkcji i kontroli. Metoda te definiuje dane i funkcje tak jak to było w metodzie *IFPUG* (jako odpowiednio elementy da-

nych i elementy transakcyjne), natomiast pod pojęciem kontroli kryje się charakterystyka wewnętrznych stanów systemu oraz przejść między nimi. Metoda ta jest rozwiązaniem wewnętrznym stosowanym obecnie przez firmę Boeing i w związku z tym nie wiadomo zbyt wiele na jej temat.

MkII Function Points

Metoda punktów funkcyjnych MkII została zaprezentowana przez Charlesa Symonsa w 1988 roku. Uzasadnieniem dla jej opracowania były (Lother & Dumke, 2001):

- potrzeba zredukowania poziomu subiektywności przy ocenie operacji na danych charakterystycznej dla metody *IFPUG*
- potrzeba zapewnienia identyczności rozmiaru systemu niezależnie od tego, czy mierzony jest on całościowo czy jako suma podsystemów
- potrzeba skupienia się bardziej wielkości nakładu pracy potrzebnego do dostarczenia danej funkcjonalności niż na wielkości funkcjonalności

Według metody punktów funkcyjnych MkII system należy opisywać w kategoriach logicznych transakcji przez niego przeprowadzanych. Logiczna transakcja jest zdefiniowana jako proces najniższego poziomu składający się z trzech podstawowych elementów (Lother & Dumke, 2001):

- wejścia danych spoza granic systemu
- przetwarzania danych
- wyjścia danych poza zakres granic systemu

Wyznaczanie liczby punktów funkcyjnych według metody MkII polega (analogicznie do metody *IFPUG*) na wyznaczeniu sumy wag kategorii dla wszystkich transakcji logicznych, a następnie na przemnożeniu tak uzyskanego wyniku przez współczynnik korygujący. Przy wyznaczaniu współczynnika korygującego w metodzie MkII oprócz czynników technicznych znanych z metody *IFPUG* uwzględnia się dodatkowe (Lother & Dumke, 2001):

- wymagania innych aplikacji dotyczące interfejsów (ang. *Requirements of Other Applications*)
- zapewnienie bezpieczeństwa (ang. *Security, Privacy, Auditability*)

- wymagania co do potrzeby przeszkolenia użytkownika (ang. *User Training Needs*)
- stopień współpracy z innymi aplikacjami (ang. *Direct Use by Third Parties*)
- dokumentacja (ang. *Documentation*)

Przypadki zastosowania metody MkII ograniczają się niemal wyłącznie do Wielkiej Brytanii (skąd pochodzi jej autor), gdzie zyskała ona sporą popularność, niemalże całkowicie wypierając inne metody. Jest ona ciągle rozwijana przez organizację *UKSMA* (ang. *United Kingdom Software Metrics Association*).

Metoda pełnych punktów funkcyjnych *COSMIC-FFP*

Metoda pełnych punktów funkcyjnych (ang. *Full Function Points – FFP*) została po raz pierwszy zaprezentowana w 1996 roku przez swych twórców z Uniwersytetu Quebec w Montrealu. Celem jej opracowania była potrzeba stworzenia możliwości zastosowania punktów funkcyjnych w określaniu rozmiaru systemów czasu rzeczywistego. W wersji 1.0 metoda ta po prostu rozszerzała metodę *IFPUG* o elementy pozwalające na zastosowanie jej w dziedzinie systemów czasu rzeczywistego (*RT*). W 1998 roku została ona opracowana od podstaw przez specjalnie w tym celu utworzoną organizację *COSMIC* (ang. *Common Software Measurement International Consortium*) zrzeszającą specjalistów pomiarów oprogramowania z całego świata. Podczas jej tworzenia wzięto pod uwagę całość dotychczasowych doświadczeń z zakresu metod określania funkcjonalnego rozmiaru zaprezentowanych wcześniej. Najnowsza wersja metody *COSMIC-FFP* nosi numer 2.2 i została zdefiniowana w 2003 roku.

Wersja 2.2 metody przewiduje wsparcie dla określania rozmiaru funkcjonalnego dla systemów dziedzin aplikacji biznesowych (*MIS*) i aplikacji czasu rzeczywistego (*RT*). Brak jest natomiast wsparcia dla systemów charakteryzujących się wykorzystaniem skomplikowanych algorytmów (*A/S* – na przykład systemy eksperckie, aplikacje symulacyjne) oraz przetwarzaniem ciągłych zmiennych (na przykład gry komputerowe).

Z perspektywy proponowanej w metodzie *COSMIC-FFP* oprogramowanie jest produktem stworzonym w celu zaspokojenia funkcjonalnych wymagań użytkownika, które mogą być przypisywane do jednostek oprogramowania na różnych poziomach funkcjonalnych. Wszystkie wymagania funkcjonalne przypisane do dowolnej jednostki oprogramowania mogą być poddane dekompozycji na procesy funkcjonalne, które z kolei można podzielić na podprocesy. Podproces może reprezentować albo przesunięcie danych albo

transformację danych, przy czym metoda *COSMIC-FFP* bierze pod uwagę tylko przesunięcia danych, zakładając, że każde przesunięcie danych wiąże się ze stałą liczbą transformacji danych.

Metoda *COSMIC-FFP* definiuje 4 podstawowe typy podprocesów przesunięć danych (COSMIC, 2003):

- wejście (ang. *Entry – E*) – przesunięcie grupy danych od użytkownika poprzez granicę do procesu funkcjonalnego
- wyjście (ang. Ex*it – X*) – przesunięcie grupy danych z procesu funkcjonalnego poprzez granicę do użytkownika
- odczyt (ang. *Read – R*) – przesunięcie grupy danych z miejsca składowania do procesu funkcjonalnego
- zapis (ang. *Write – W*) – przesunięcie grupy danych z procesu funkcjonalnego do miejsca składowania

Pomiar rozmiaru funkcjonalnego w metodzie *COSMIC-FPP* opiera się na identyfikacji procesów funkcjonalnych danej jednostki oprogramowania, następnie na identyfikacji i zliczeniu wszystkich podprocesów przesunięć danych występującej w jej ramach. Suma ilości wszystkich podprocesów we wszystkich procesach funkcjonalnych dla danej jednostki oprogramowania stanowi ostateczną wartość rozmiaru funkcjonalnego oprogramowania wyrażonego w jednostce zwanej *Cfsu* (ang. *Cosmic functional size unit*).

Metoda *COSMIC-FFP* jest stosunkowo młoda i w związku z tym nie ma zbyt wiele studiów weryfikujących jej poprawność. Mimo to, wydaje się, że ma ona spore szanse na to, żeby zyskać na popularności w najbliższej przyszłości, chociażby ze względu na zespół, który zajmuje się jej rozwojem.

Use Case Points

Metoda punktów przypadków użycia powstała w 1993 roku jako praca dyplomowa Gustava Karnera; potem była rozwijana w ramach firmy Rational. Jest ona zastosowaniem idei punktów funkcyjnych w realiach przypadków użycia rozpowszechnionych wraz z rosnącą popularnością diagramów UML. Składa się ona z następujących kroków (Laird & Brennan, 2006):

- klasyfikacja złożoności aktorów przypadków użycia
- klasyfikacja złożoności przypadków użycia
- ocena wartości czynników technicznych i środowiskowych

Zastosowane w tej metodzie wzory oraz czynniki techniczne i środowiskowe pokrywają się w dużym stopniu z tymi stosowanymi w klasycznej metodzie *IFPUG*.

Metoda wydaje się być generalnie obiecująca; autorzy (Laird & Brennan, 2006) na podstawie innych badań oraz swoich doświadczeń twierdzą, że metoda jest stosunkowo łatwa, szybka i przejrzysta, generuje dobre rezultaty pod warunkiem, że proces oceny złożoności aktorów i przypadków użycia zostanie ustandaryzowany oraz przypadki użycia będą opisywane na stałym poziomie szczegółowości. Jest to wymaganie fundamentalne, bo żadne diagramy UML, a w szczególności również diagramy przypadków użycia, nie definiują sposobów modelowania, lecz jedynie udostępniają ich notację.

Standardyzacja metod mierzenia funkcjonalności

W 1996 roku w ramach organizacji ISO (ang. *International Organization for Standardization*) utworzono grupę roboczą (ISO/IEC JTC1 SC7 WG12), której zadaniem było opracowanie zestawu powszechnie akceptowalnych standardów opisujących koncepcję i praktykę wyznaczania funkcjonalnego rozmiaru oprogramowania. Standard ten nie opisuje żadnej konkretnej metody, lecz dostarcza definicji cech charakterystycznych jakie musi spełniać kandydat, aby mógł być ona uznany za metodę wyznaczania funkcjonalnego rozmiaru oprogramowania. W chwili obecnej istnieje standard funkcjonalnego mierzenia rozmiaru – ISO 14143. Składa się on z 5 części, opisujących koncepcje pomiaru funkcjonalnego, metody sprawdzania zgodności ze standardem i z zamierzonym celem użycia, metody oceny względem określonych punktów odniesienia oraz sposobów sprawdzania możliwości zastosowania w określonej dziedzinie.

W chwili obecnej cztery metody przeszły przez proces standaryzacji. Są to: metoda *IFPUG*, metoda MkII, metoda *COSMIC-FFP* i metoda *NESMA* (uproszczenie metody *IFPUG*). Warty podkreślenia jest fakt, że standaryzacja powyższych metod dotyczy tylko procesu wyznaczania nieskorygowanej liczby punktów funkcyjnych (bez uwzględnienia wpływu czynników technicznych i środowiskowych).

4.4 ESTYMACJA ROZMIARU OPROGRAMOWANIA W UJĘCIU ZŁOŻONOŚCI

Złożoność oprogramowania jest jednym z podstawowych problemów z jakim muszą sobie radzić projektanci niemal wszystkich systemów informatycznych. Wzrost złożoności niesie ze sobą dużo problemów, między innymi utrudnienie percepcji struktury i zachowania systemu, a co za tym idzie wzrost awaryjności i zmniejszenie produktywności programistów.

W ramach pojęcia złożoności oprogramowania można wyróżnić dwie podstawowe kategorie:

- złożoność przetwarzania danych
- złożoność struktury

4.4.1 Metryki rozmiaru oprogramowania w ujęciu złożoności

Najpopularniejsze metryki złożoności przetwarzania danych to (Laird & Brennan, 2006):

- liczba cyklomatyczna McCabe'a
- metryki Halsteada
- metryka przepływu informacji

Liczba cyklomatyczna McCabe'a

Liczba cyklomatyczna McCabe'a (McCabe, 1976) jest jedną z najpopularniejszych metryk złożoności przetwarzania danych. Metryka ta wyraża liczbę niezależnych ścieżek wykonania zawartych w pojedynczym module oprogramowania, rozumianym jako zbiór wykonywalnych instrukcji, przy czym ów zbiór posiada tylko jedno wejście i jedno wyjście. Bazuje ona na klasycznej teorii grafów zaaplikowanej do schematu blokowego programu: węzły odpowiadają tutaj instrukcjom, a skierowane krawędzie przejściom między instrukcjami. Liczbę cyklomatyczną $V(g)$ można wyznaczać dwojako (oba sposoby dają te same rezultaty):

$$V(g) = e - n + 2 \quad (4.09)$$

gdzie *g* oznacza graf dla którego jest obliczana liczba cyklomatyczna, *e* to liczba krawędzi, *n* to liczba węzłów grafu.

$$V(g) = bd + 1 \quad (4.10)$$

gdzie *g* oznacza graf dla którego jest obliczana liczba cyklomatyczna, *bd* to liczba binarnych decyzji (rozgałęzień grafu). Decyzje *n*-elementowe traktuje się jako *n* – 1 decyzji binarnych.

Liczba cyklomatyczna McCabe'a może być traktowana jako miara testowalności – im jest większa, tym więcej testów jest potrzebnych, bo algorytm przetwarzania danych staje się coraz bardziej skomplikowany. Uznaje się (Van Doren, 2000), że moduły o $V(g) > 20$ stanowią powód do obaw jeśli chodzi o ich złożoność i wynikające z tego konsekwencje.

Należy zwrócić uwagę, że liczba cyklomatyczna McCabe'a nie może służyć do mierzenia złożoności danych. Ponadto jest w niektórych aspektach nieintuicyjna: zagnieżdżone instrukcje warunkowe traktuje tak samo jak równoległe, podczas gdy znaczny stopień zagnieżdżenia skutecznie utrudnia człowiekowi analizę kodu.

Metryki Halsteada

W 1977 roku Maurice Halstead opublikował (Halstead, 1977) zestaw równań, które umożliwiają wyznaczenie podstawowych metryk programu. Jako wejście przyjmują one liczbę niepowtarzalnych operatorów n_1, liczbę niepowtarzalnych operandów, całkowitą liczbę użytych operatorów N_1, całkowitą liczbę użytych operandów N_2. Wówczas prawdziwe są między innymi następujące równania:

- długość programu

$$N = N_1 + N_2 \quad (4.11)$$

- wielkości słownika

$$n = n_1 + n_2 \quad (4.12)$$

- nakład pracy potrzebny na implementację programu

$$E = \frac{n_1 * N_2 * N * log_2 n}{2 * n_2} \quad 4.13$$

Metryki Halsteada wyznacza się na podstawie analizy kodu źródłowego, co ogranicza pola jej zastosowania tylko do etapów implementacji i analizy istniejących rozwiązań.

Podstawowe zalety metryk Halsteada to (Sultanoğlu & Karakaţ, 1998):

- nie wymagają dokładnej analizy struktury kodu źródłowego
- mogą być wykorzystywana do przewidywania nakładu pracy na utrzymanie oprogramowania
- proste do wyznaczania (również automatycznie)

Metryka przepływu informacji

Metryka przepływu informacji (*IFC* – ang. *Information Flow Complexity*) służy jako miara przepływu informacji wchodzącej i wychodzącej z modułu. Im jest większa, tym bardziej skomplikowany jest sposób przetwarzania zawarty w module (Henry & Kafura, 1981). Pierwotna metryka *IFC* miała następującą definicję:

$$IFC = (fanin * fanout)^2 \quad 4.14$$

gdzie *fanin* to liczba informacji przychodzących do modułu, a *fanout* to liczba informacji wychodzących z modułu. Istnieje spora grupa wywiedzionych metryk, w których definicje *fanin* i *fanout* są bardziej rozbudowane oraz korzysta się dodatkowo z długości programu (*length*), na przykład IEEE Standard 982.2 (Laird & Brennan, 2006):

$$IFC = length * (fanin * fanout)^2 \quad 4.15$$

gdzie *fanin* to liczba informacji przychodzących do modułu oraz liczba struktur danych wykorzystywanych przy przetwarzaniu, a *fanout* to liczba informacji wychodzących z modułu oraz liczba struktur danych uaktualnianych w wyniku przetwarzania.

Metryka przepływu danych dobrze nadaje się do określania złożoności programów napędzanych danymi (ang. *data-driven*). Może być wykorzystywana już na etapie projektowania aplikacji.

Metryki złożoności struktury

Metryki złożoności struktury mają duże znaczenie w odniesieniu do paradygmatu obiektowego, który wprowadza takie pojęcia, jak klasa, dziedziczenie, inkapsulacja, abstrakcja. Metryki te odnoszą się przede wszystkim do etapu projektowania. Najpopularniejszym zestawem obiektowych metryk złożoności struktury jest rezultat badań Chidambera i Kemerera (Chidamber & Kemerer, 1994), na który składają się następujące metryki:

- ważona liczba metod w klasie (*WMC* – ang. *Weighted Methods per Class*): jest to suma złożoności metod w klasie, przy czym złożoność metody może być obliczana dowolnie (na przykład przy pomocy liczby cyklomatycznej McCabe'a czy ilości linii kodu źródłowego), a nawet ustalona z góry (najczęściej na 1). Wysokie wartości tej metryki sugerują, że klasa powinna być rozbita na mniejsze podklasy.
- głębokość drzewa dziedziczenia (*DIT* – ang. *Depth of Inheritance Tree*): jest to ilość poziomów w danej hierarchii dziedziczenia. Duże wartości sygnalizują większą złożoność projektu, ale jednocześnie większy stopień ponownego wykorzystania kodu (ang. *reuse*).
- ilość potomków (*NOC* – ang. *Number Of Children*): jest to ilość bezpośrednich potomków klasy w drzewie dziedziczenia. Duże wartości oznaczają większą złożoność projektu i zbyt dużą abstrakcję klasy rodzica.
- liczba powiązań między klasami (*CBO* – ang. *Coupling Between Objects*): jest to miara powiązań danej klasy z innymi klasami w sposób inny niż poprzez dziedziczenie. Im wartość ta jest niższa, tym klasy bardziej niezależne, a przez to łatwiej je utrzymywać.
- odpowiedź klasy (*RFC* – ang. *Response for Class*): jest to wielkość zbioru metod jakie mogą być wywołane w odpowiedzi na komunikat wysłany do obiektu danej klasy. Jest ona obliczana jako suma liczby metod danej klasy i liczby metod innych klas bezpośrednio wywoływanych w metodach danej klasy. Im wartość tej metryki jest większa, tym kod jest bardziej skomplikowany, a testowanie utrudnione.
- brak spójności metod (*LCOM* – ang. *Lack of Cohesion of Methods*): jest to miara powiązania metod z innymi metodami i atrybutami klasy. Jeśli wartość tej metryki wzrasta to oznacza to, że metody są coraz mniej spójne, co sugeruje potrzebę rozbicia klasy.

W 1999 w jednostce organizacyjnej NASA zajmującej się metrykami oprogramowania opracowano zestaw reguł, przy pomocy których ustalano, czy dana klasa wymaga refaktoryzacji (czyli modyfikacji struktury). Jeśli dla danej klasy co najmniej dwa kryteria z poniższych zostają spełnione, to klasa ta powinna zostać poddana przebudowie (Rosenberg, Stapko, & Gallo, 1999):

- *RFC* > 100
- *CBO* > 5
- *RFC* > 5*liczba metod w klasie
- *WMC* > 100
- liczba metod > 40

4.4.2 Metody estymacji rozmiaru oprogramowania w ujęciu złożoności

Estymacja rozmiaru oprogramowania w ujęciu jego złożoności nie doczekała się zbyt wielu przykładów zastosowania jej w postaci konkretnej metody, a już z pewnością żadna z nich nie zyskała takiej popularności jak metoda punktów funkcyjnych *IFPUG*. Najważniejsze metody estymacji złożoności oprogramowania to:

- metoda *Predictive Object Points*

Metoda *Predictive Object Points*

Metoda *Predictive Object Points* (*POP*) została opracowana w 2000 roku (Minkiewicz, 2000) jako odpowiedź na problemy związane z metodami punktów funkcyjnych, w ramach których jedynym aspektem rozmiaru jaki był mierzony była funkcjonalność. W dobie oprogramowania zorientowanego obiektowo uznano to za niedopuszczalne, gdyż pomijany był aspekt złożoności.

Metoda *Predictive Object Points* opiera się na następujących metrykach obiektowych (Minkiewicz, 2000):

- liczba klas najwyższego poziomu (*TLC – Number of Top Level Classes*)
- średnia liczba ważonych metod klasy (*WMC* – ang. *Number of Weighted Methods per Class*)
- średnia głębokość drzewa dziedziczenia (*DIT* – ang. *Depth of Inheritance Tree*)

- średnia ilość potomków klasy w drzewie dziedziczenia (*NOC* – ang. *Number of Children*)

Sercem metody jest odpowiednie wyliczanie metryki *WMC*, która obejmuje zarówno funkcjonalność, jak i międzyobiektową komunikację. Funkcjonalność metody klasy jest wyznaczana na podstawie analizy jej przynależności do określonej kategorii (Booch, 1994):

- konstruktorów, czyli metod wywoływanych w celu inicjacji obiektu
- destruktorów, czyli metod wywoływanych w celu destrukcji obiektu
- modyfikatorów, czyli metod modyfikujących stan obiektu
- selektorów, czyli metod zwracających stan obiektu
- iteratorów, czyli metod pozwalających na dostęp do atrybutów obiektu w ściśle określonym porządku

Oprócz określenia klasy funkcjonalności w skład metryki *WMC* wchodzą również ocena złożoności (wyrażana jako mała, średnia lub duża) metody wyznaczana na podstawie liczby odpowiedzi metody oraz liczby atrybutów na których metoda operuje.

Metoda *Predictive Object Points* jest stosunkowo nowa i brak jest szerszych opracowań na jej temat. Jako jej wady na pewno można wymienić to, że wymaga dość szczegółowych danych na temat implementacji metody, które nie są dostępne na etapach projektowania. Ponadto metoda ta nie została jak dotąd zautomatyzowana.

5 METODA ESTYMACJI DŁUGOŚCI KODU ŹRÓDŁOWEGO NA PODSTAWIE DIAGRAMÓW STATYCZNYCH UML

Niniejszy rozdział zawiera opis proponowanej metody estymacji rozmiaru oprogramowania w aspekcie długości kodu źródłowego implementacji na podstawie diagramów UML opisujących jego statyczną strukturę.

5.1 PODSTAWOWE POJĘCIA I ZAŁOŻENIA

Niniejszy podrozdział służy ścisłemu zdefiniowaniu podstawowych pojęć z zakresu obiektowości, które będą później służyły jako podstawa bardziej rozbudowanych pojęć. Pojęcia obiektowe tutaj zdefiniowane to:

- modyfikator stałości
- modyfikator abstrakcyjności
- modyfikator statyczności
- modyfikator dostępu
- pole danych
- rezultat metody
- parametr metody
- atrybut klasy
- metoda klasy
- relacja dziedziczenia
- klasa
- system

Modyfikator stałości

Modyfikator stałości *const* może być przypisywany do niektórych bytów obiektowych i określa cechę ich niemodyfikowalności:

$$const = \begin{cases} 0, & gdy\ byt\ jest\ modyfikowalny \\ 1, & gdy\ byt\ jest\ niemodyfikowalny \end{cases} \quad 5.16$$

Modyfikator abstrakcyjności

Modyfikator abstrakcyjności *abstract* może być przypisywany do niektórych bytów obiektowych i określa, czy byt jest deklaracją (brak implementacji) czy definicją (implementacja istnieje):

$$abstract = \begin{cases} 0, & gdy\ byt\ ma\ definicję \\ 1, & gdy\ byt\ ma\ tylko\ deklarację \end{cases} \quad 5.17$$

Modyfikator statyczności

Modyfikator statyczności *static* może być przypisywany do niektórych bytów obiektowych i określa, czy istnienie bytu powiązane jest z istnieniem instancji klasy (obiektu) czy z klasą samą w sobie:

$$static = \begin{cases} 0, & gdy\ byt\ istnieje\ w\ ramach\ obiektu \\ 1, & gdy\ byt\ istnieje\ w\ ramach\ klasy\ (nie\ obiektu) \end{cases} \quad 5.18$$

Modyfikator dostępu

Modyfikator dostępu *access* może być przypisywany do niektórych bytów obiektowych i określa, czy dany byt jest osiągalny dla innych elementów środowiska obiektowego:

$$access = \begin{cases} 0, & gdy\ byt\ nie\ jest\ dostępny\ spoza\ klasy \\ 1, & gdy\ byt\ jest\ dostępny\ dla\ potomków \\ 2, & gdy\ byt\ jest\ dostępny\ dla\ wszystkich \end{cases} \quad 5.19$$

Pole danych

Pole danych *D* to podstawowa jednostka służąca do przechowywania danych. Jest zdefiniowana następująco:

$$D = (type, dim) \quad 5.20$$

gdzie *type* to tekstowy identyfikator typu danych jaki może być zawarty w polu danych (inaczej to sposób interpretacji ciągu bajtów o określonej długości skojarzonej z typem), *dim* to liczba całkowita będąca modyfikatorem typu i określająca, czy pole danych

ma być interpretowane jako pojedyncza instancja typu *type* czy jako *n*-wymiarowa struktura:

$$dim = \begin{cases} 0, & gdy\ D\ jest\ pojedynczą\ instancją \\ n, & gdy\ D\ jest\ strukturą\ o\ n\ wymiarach \end{cases} \quad 5.21$$

Na przykład dla wektora instancji *dim* = 1, dla macierzy instancji *dim* = 2.

Rezultat metody

Rezultat metody *R* to pole danych zwracana w wyniku wykonania metody:

$$R = D \quad 5.22$$

Parametr metody

Parametr metody *P* to rozszerzone pole danych służące do przechowywania poje dyn czego egzemplarza danych służących jako wejście dla przetwarzania realizowanego w ramach metody:

$$P = (name, const, D) \quad 5.23$$

gdzie *name* to tekstowy identyfikator parametru, *const* to modyfikator stałości, *D* to pole danych.

Atrybut klasy

Atrybut klasy *A* to rozszerzone pole danych służące do przechowywania pojedynczego atomu informacji o bieżącym stanie obiektu:

$$A = (name, const, abstract, static, access, D) \quad 5.24$$

gdzie *name* to tekstowy identyfikator atrybutu, *const* to modyfikator stałości, *abstract* to modyfikator abstrakcyjności, *static* to modyfikator statyczności, *access* to modyfikator dostępu, *D* to pole danych.

Metoda klasy

Metoda klasy to podstawowy element definiujący zachowanie klasy:

$$M = (name, const, abstract, static, access, \boldsymbol{P}, \boldsymbol{R}) \quad 5.25$$

$$\boldsymbol{P} = \{P_1, P_2, \dots\} \quad 5.26$$

$$\boldsymbol{R} = \{R_1, R_2, \dots\} \quad 5.27$$

gdzie *name* to tekstowy identyfikator metody, *const* to modyfikator stałości, *abstract* to modyfikator anstrakcyjności, *static* to modyfikator statyczności, *access* to modyfikator dostępu, ***P*** to zbiór parametrów metody, ***R*** to zbiór rezultatów metody.

Relacja dziedziczenia

Relacja dziedziczenia dotyczy klas i oznacza, że klasa dziedzicząca jest rozszerzeniem klas-rodziców o swoje własne elementy:

$$I = (name_C, access) \quad 5.28$$

gdzie $name_C$ to tekstowy identyfikator klasy rodzica, *access* oznacza sposób dziedziczenia, oznaczający sposób konwersji dostępów do bytów klas-rodziców według następującego schematu:

Tabela 13 Dostęp do elementów klasy bazowej z klasy dziedziczącej

		dostęp w klasie rodzica		
		private	protected	public
rodzaj dziedziczenia	private	brak	private	private
	protected	brak	protected	protected
	public	brak	protected	public

Źródło: opracowanie na podstawie (Grębosz, 1996)

Klasa

Klasa to podstawowy element strukturalny w programowaniu obiektowym:

$$C = (name, \boldsymbol{I}, \boldsymbol{A}, \boldsymbol{M}) \quad 5.29$$

$$\boldsymbol{I} = \{I_1, I_2, \dots\} \quad 5.30$$

$$\boldsymbol{A} = \{A_1, A_2, \dots\} \quad 5.31$$

$$\boldsymbol{M} = \{M_1, M_2, \dots\} \quad 5.32$$

gdzie *name* to tekstowy identyfikator klasy, $\boldsymbol{I}$ to zbiór relacji dziedziczenia klasy, $\boldsymbol{A}$ to zbiór atrybutów klasy, $\boldsymbol{M}$ to zbiór metod klasy.

System

System, rozumiany tutaj jako realizowane oprogramowanie, to zbiór klas tworzących program komputerowy:

$$S = (name, \boldsymbol{C}) \quad 5.33$$

$$\boldsymbol{C} = \{C_1, C_2, \dots\} \quad 5.34$$

gdzie *name* to tekstowy identyfikator systemu, $\boldsymbol{C}$ to zbiór klas wchodzących w skład systemu.

5.2 PROPONOWANA METRYKA DŁUGOŚCI KODU ŹRÓDŁOWEGO

Podstawową i najpopularniejszą metryką długości kodu źródłowego jest linia kodu źródłowego. Posiada ona cały szereg wad (szczegóły są opisane w rozdziale 4.2.1), dlatego też na potrzeby nowej metody zaproponowano nową metrykę długości kodu źródłowego o nazwie *leksem*.

Pojęcie leksemu wywodzi z analizy leksykalnej, będącej częścią teorii tworzenia translatorów. Analiza leksykalna jest pierwszym etapem procesu translacji (czyli tłumaczenia programu z danego języka programowania na kod maszynowy) i polega na analizie strumienia danych wejściowych oraz rozpoznawaniu w nim ciągów zbudowanych zgodnie z pewnymi zasadami. Właśnie te rozpoznane jako istotne dla języka ciągi noszą nazwę leksemów, czyli symboli leksykalnych (Complak & Bogacki, 2006).

Zastosowanie leksemu jako metryki długości kodu źródłowego przynosi wiele korzyści w porównaniu z metryką ilości linii kodu źródłowego:

- pojęcie leksemu jest ściśle zdefiniowane w kontekście danego języka programowania, nie ma tutaj żadnej swobody interpretacji
- wyznaczanie liczby leksemów jest stosunkowo proste
- liczba leksemów jest niezależna od komentarzy ani sposobu formatowania tekstu źródeł programu

Porównanie sposobów liczenia liczby linii kodu źródłowego oraz liczby leksemów zostało zaprezentowane poniżej (patrz Tabela 14). Dla zwiększenia czytelności procesu liczenia leksemów przykładowy kod źródłowy został sformatowany w ten sposób, że kolejne leksemy został oznaczone naprzemiennie ustawianym kolorem.

Tabela 14 Porównanie metryk długości kodu źródłowego LOC i liczby leksemów w języku Java

kod źródłowy	długość kodu [LOC]	długość linii [leksem]	długość kodu [leksem]

`for (int i=0; i<10; ++i) {`
`  System.out.println(i);`
`}`
	3	15
		9
		1
			25

`for (int i=0; i<10; ++i)`
`{`
`  System.out.println(i);`
`}`
	4	14
		1
		9
		1
			25

`int i=0;`
`for (; i<10; ++i)`
`{`
`  System.out.println(i);`
`}`
	5	5
		10
		1
		9
		1
			26

Źródło: opracowanie własne

Istotną cechą liczby leksemów jako metryki długości kodu źródłowego bezpośrednio związaną z jej charakterystyką jest to, że można ją wyznaczać wyłącznie na podstawie istniejącego kodu źródłowego, co ogranicza zakres jej stosowania tylko do etapów implementacji i późniejszych.

5.3 ANALIZA DANYCH DOŚWIADCZALNYCH

Podrozdział ten stanowi opis analizy danych doświadczalnych wykorzystanych jako wycinek populacji generalnej do badań nad nową metodą estymacji długości kodu źródłowego.

5.3.1 Opis danych eksperymentalnych

Metoda estymacji opisana w niniejszej pracy postuluje użycie diagramów statycznych UML jako źródła danych do estymacji długości kodu źródłowego implementującego program. Niestety, dostęp do tego typu danych rzeczywistych jest w większości przypadków niemożliwy, gdyż albo diagramy UML stanowią część dokumentacji technicznej komercyjnych systemów informatycznych, a co za tym idzie są często niejawne, albo w ogóle nie są tworzone.

Problem dostępności danych eksperymentalnych może być rozwiązany przy pomocy oprogramowania *open source* (otwarte oprogramowanie). Oprogramowanie *open source* to takie, którego kod źródłowy jest dostępny w ramach licencji, która pozwala na używanie, zmianę, ulepszanie, dystrybucję w postaci zmodyfikowanej i niezmodyfikowanej (Open Source Initiative, 2009). Oprogramowanie tego typu bardzo często jest rozwijane przez członków społeczności komunikującej się za pośrednictwem sieci Internet.

Mając do dyspozycji kod źródłowy można dokonać jego konwersji na diagramy klas UML, co może być zrealizowane przy pomocy funkcji zaimplementowanej w wielu istniejących edytorach UML, mianowicie inżynierii wstecznej (ang. *reverse engineering*). Należy tutaj zauważyć, że taka konwersja jest możliwa tylko dla kilku rodzajów diagramów (klas, pakietów); dla większości diagramów taka operacja jest niewykonalna. Na podstawie kodu źródłowego można również przeprowadzić wyznaczenie długości kodu źródłowego implementującego poszczególne elementy systemu informatycznego.

Kod źródłowy systemów informatycznych wykorzystanych w eksperymentach zrealizowanych na potrzeby niniejszej pracy został pozyskany z:

- portalu SourceForge (http://sourceforge.net/)
- portalu BerliOS (http://developer.berlios.de/)

- portalu The Apache Software Foundation (http://www.apache.org/)
- portalu Tigris.org (http://www.tigris.org/)

Podstawowe zasady doboru kodu źródłowego oprogramowania na potrzeby analiz wykonanych w niniejszej pracy to:

- implementacja w języku Java
- duży stopień zaawansowania prac (co najmniej późny etap implementacji)

Tabela 15 Przegląd danych doświadczalnych – podzbiór uczący

system	rozmiar [leksem]	liczba klas	liczba metod
antlr_275	249 626	216	2 603
azureus_2302	1 097 371	3 201	16 326
columba_1	421 848	1 319	6 789
gantt_20	276 535	792	4 220
hibernate_305	641 736	1 409	13 666
hsqldb_1733	371 769	331	4 149
jameleon_302	106 531	241	1 831
jasperreports_100	285 735	640	4 171
jboss_402	3 036 230	7 819	52 744
jetty_514	322 246	566	5 157
jtds_11	246 187	153	2 246
liferay_361	2 050 766	3 445	26 439
mantaray_18	218 813	480	3 288
phex_26489	493 382	1 120	6 978
pmd_34	311 996	700	4 482
rssowl_113	325 652	655	2 307
saxon_84	498 907	819	7 177
spring_121	856 398	3 118	13 776
track_310	815 644	659	10 640
weka_350	1 060 276	1 281	11 028
wicket_102	137 049	559	3 001
xdoclet_123	227 953	603	4 967
argouml_0196	680 395	1 489	10 825
axion_10	139 025	261	2 978
commons-collections_32	151 426	449	3 874
commons-math_11	52 551	128	1 229
commons-primitives_10	54 293	259	2 173
commons-vfs_10	88 147	244	1 839
db4o_60	872 305	2 387	15 349
glassfish_1b14	5 647 199	10 384	91 699
jedit_42	444 492	799	5 154
xmlgui_102	86 231	258	1 170
jruby_092	507 160	689	6 206
jsvn_08	40 540	135	486
ant_170	543 606	1 132	9 272
tomcat_607	931 683	1 437	14 751
derby_10220	1 579 214	1 601	22 749
dom4j_161	100 060	189	2 711
groovy_10	439 156	622	6 600
jacl_140	469 319	756	4 278
junit_41	9 851	65	263
openjms_077	177 794	564	3 616
ozone_12	406 513	1 022	9 573
snmp4j_18	80 353	175	1 616
snmp4jagent_11	90 011	305	1 769
struts_135	257 576	504	5 188
velocity_14	110 244	222	1 781
xercesj_290	630 474	882	8 452
h2_20070102	366 273	384	5 285
resin_310	2 434 820	5 240	48 436
ŚREDNIA	**628 867**	**1 254**	**9 946**
ODCHYLENIE STANDARDOWE	**945 248**	**1 916**	**15 689**

Źródło: opracowanie własne

Tabela 16 Przegląd danych doświadczalnych – podzbiór testowy

system	rozmiar [leksem]	liczba klas	liczba metod
exist_111	1 151 949	1 533	13 316
aoi_251	823 454	753	6 343
jython_21	681 015	1 280	10 070
nice_0913	234 114	375	4 071
cobra_0974	237 191	608	5 166
antelope_341	173 161	386	2 108
aopmetrics_400	18 139	80	455
drjava_20070524	513 061	1 670	8 379
ermodeller_192	109 857	386	2 140
ftp4che_14	29 166	49	376
itext_204	568 781	472	5 937
jfreechart_106	458 258	534	6 961
jftp_120	51 496	90	697
jgap_32	116 880	325	2 396
jmt_09	99 858	210	1 599
jped_13	294 790	504	4 263
jpox_118	15 608	49	57
prism_010	12 982	73	330
rapidminer_40	788 710	1 739	9 953
smallsql_018	115 987	151	2 054
telnetd_20	29 485	55	508
tinyuml_025	61 114	189	1 433
umlet_71	90 511	112	661
apacheds_400	812 786	1 769	10 454
cayenne_203	528 442	1 430	9 549
james_231	213 418	458	3 189
lucene_210	286 389	545	3 551
xalanj_270	790 714	1 197	10 032
log4j_128	82 868	258	1 563
maven_207	182 205	346	2 678
myfaces_115	211 413	464	4 263
xmlsecurity_141	132 764	318	2 190
areca_554	228 647	440	3 685

system	rozmiar [leksem]	liczba klas	liczba metod
robocode_151	171 865	303	2 704
findbugs_131	594 004	1 519	8 866
freemind_080	434 113	801	5 898
borg_161	247 382	516	2 406
sdedit_212	79 534	219	1 301
testng_57	129 997	285	2 217
barcode4j_20	85 211	186	989
umleditor_310	507 587	606	9 475
plandora_100	174 645	290	2 693
jeti_077	415 037	1 195	5 517
xholon_07	304 479	616	4 182
smith_13	194 685	329	3 055
jparsec_12	58 533	284	1 299
jaskell_10	119 944	593	2 504
xfire_126	337 495	1 184	6 402
jaxen_111	107 820	317	2 060
plexus_10	575 298	2 282	11 685
tapestry_507	339 867	1 698	7 049
jackrabbit_133	758 218	1 567	11 385
geronimo_202	1 148 184	2 899	18 532
turbine_232	214 805	489	4 331
activemq_500	864 302	2 321	16 968
jgroups_262	337 631	583	5 776
enhydra_651	297 522	375	6 119
datacrow_3130	337 128	650	4 614
openswing_149	293 639	663	3 808
squirrel_264	293 179	1 176	4 239
jalisto_10	120 905	278	2 947
beehive_102	479 829	1 356	8 422
lenya_20	4 741 204	4 261	25 309
ode_111	359 530	1 335	6 774
jetspeed_213	684 456	1 593	12 296
ŚREDNIA	**399 281**	**794**	**5 419**
ODCHYLENIE STANDARDOWE	**612 451**	**775**	**4 791**

Źródło: opracowanie własne

Podstawowe kategorie aplikacji, których kod źródłowy został użyty do eksperymentów to między innymi:

- bazy danych (relacyjne i obiektowe)
- szkielety wytwarzania aplikacji (ang. *application framework*)

- implementacje języków programowania
- narzędzia XML
- narzędzia automatyzacji procesu budowania oprogramowania
- narzędzia modelowania UML
- klienty poczty elektronicznej
- biblioteki programistyczne
- programy graficzne
- serwery aplikacji
- edytory tekstu
- systemy zarządzania treścią (ang. *content management system – CMS*)
- klienty różnych usług sieciowych

5.3.2 Opis procesu ekstrakcji danych

Dla każdego badanego programu przeprowadzono analizę leksykalną i syntaktyczną kodu źródłowego w celu zrekonstruowania wybranych danych dostępnych w ramach diagramów klas UML:

- struktury i typów atrybutów klas
- struktury, parametrów i rezultatów metod klas
- struktury klas

W ramach powyższej analizy dokonano również pomiarów długości kodu źródłowego wyrażonej w ilości leksemów dla:

- metod
- klas
- całego systemu

Analiza leksykalna i syntaktyczna została przeprowadzona przy pomocy aplikacji *Metrician* opracowanej przez autora w oparciu o narzędzie generacji kompilatorów ANTLR 2.7.5 (http://www.antlr.org/). Aplikacja ta przetwarza kod źródłowy w języku Java generując na jej podstawie pliki XML opisujące jego strukturę. Owe pliki stanowiły podstawowe

źródło danych dla pakietów statystycznych wykorzystanych do analizy zależności między metrykami opisującymi poszczególne byty kodu źródłowego.

5.4 MODEL SZACUJĄCY

Centralnym elementem proponowanej metody jest model szacujący służący do estymacji rozmiaru oprogramowania. Niniejszy podrozdział opisuje szczegóły zastosowanego podejścia, poczynając od charakterystyki danych wejściowych, a kończąc na właściwym modelu.

5.4.1 Ogólny model klasy

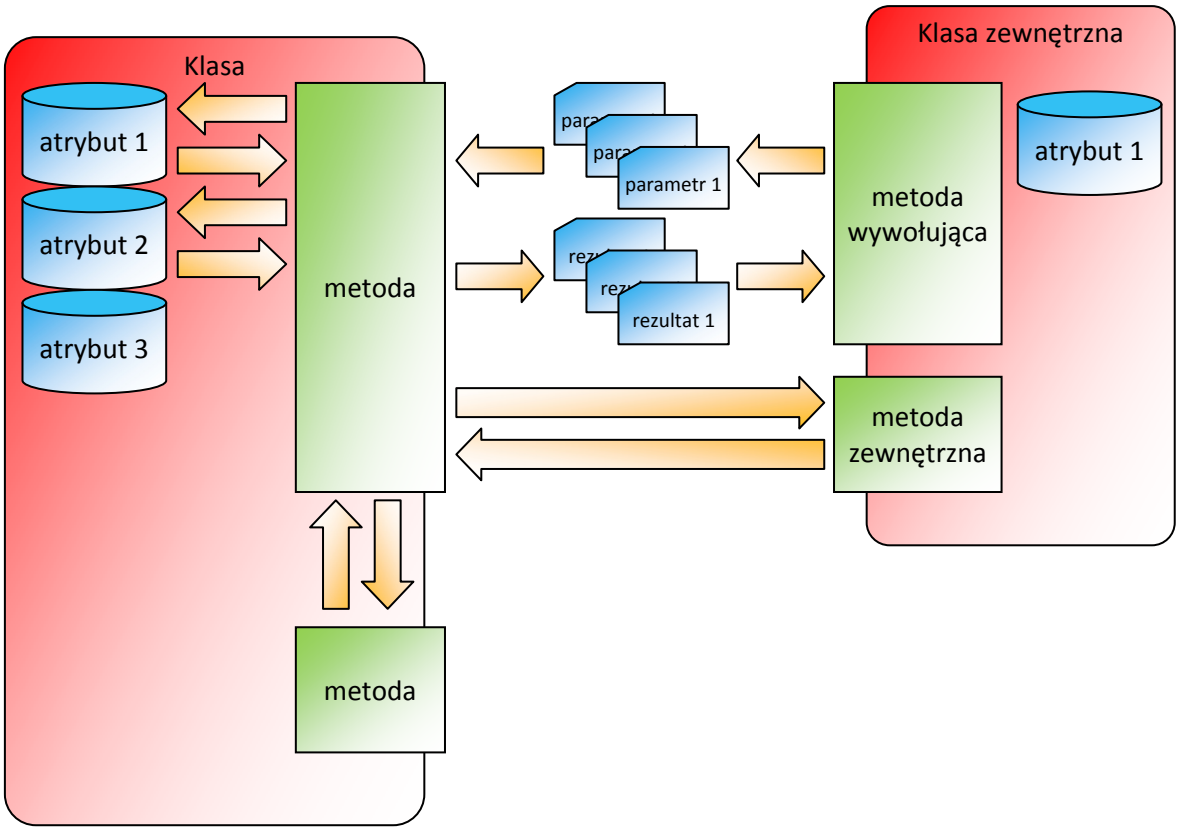
Oprogramowanie budowane z użyciem paradygmatu obiektowego jest konstruowane z podstawowych elementów konstrukcyjnych, jakim są klasy. Klasa to rodzaj kontenera, łączący opis swego stanu (przy pomocy danych zapisanych w atrybutach) oraz opis dozwolonego zachowaniu (poprzez algorytmy realizujące zmiany tego stanu w ramach metod).

Przetwarzanie w środowisku obiektowym to ciąg wywołań metod realizowanych w środowisku definiowanym przez klasę do której metoda należy oraz dostępnych metod innych klas (patrz Rysunek 11). Uruchomienie metody jest równoznaczne z rozpoczęciem procesu przetwarzania zgodnie z zaimplementowanym w jej ramach algorytmem. Algorytm wykonuje operacje na danych pochodzących z następujących źródeł:

- parametrów wywołania metody
- dostępnych atrybutów statycznych i niestatycznych klasy do której metoda należy
- rezultatów działania dowolnych dostępnych metod (z klasy macierzystej i dostępnych klas zewnętrznych) wywołanych w trakcie procesu przetwarzania

Wyniki działania algorytmu przetwarzania danych są zapisywane w następujących lokalizacjach:

- rezultatach wywołania metody
- dostępnych atrybutach statycznych i niestatycznych klasy do której metoda należy
- parametrach dowolnych dostępnych metod (z klasy macierzystej i dostępnych klas zewnętrznych) wywoływanych w trakcie procesu przetwarzania



Rysunek 11 Ogólny model przetwarzania danych w modelu obiektowym (opracowanie własne)

Taka kwalifikacja źródeł i celów danych jest prawdziwa w przypadku ścisłego zachowania zasady enkapsulacji, czyli całkowitego uniemożliwiania dostępu do atrybutów bytom zewnętrznym. Jednakże takie postępowanie w wielu przypadkach może okazać się niewygodne, dlatego też dość powszechne są przypadki, gdy atrybuty których nie można zmieniać są dostępne publicznie (na przykład wszelkiego rodzaju stałe używane w różnych klasach).

Ważnym odstępstwem od modelu obiektowego są atrybuty i metody statyczne, czyli takie, które nie są związane z egzemplarzem klasy (obiektem), lecz z samą klasą. O ile atrybuty i metody statyczne mogą być używane przez metody regularne (niestatyczne), to relacja odwrotna nie jest możliwa: w ramach metod statycznych nie można wywoływać metod niestatycznych bezpośrednio, bez kontekstu konkretnego egzemplarza klasy (obiektu). Dlatego też źródła danych metod statycznych to:

- parametry wywołania metody

- atrybuty statyczne klasy, do której metoda należy
- rezultaty działania dowolnych dostępnych metod statycznych (z klasy macierzystej i dostępnych klas zewnętrznych) wywołanych w trakcie procesu przetwarzania

Rezultaty działania metod statycznych mogą być przesyłane do:

- rezultatów wywołania metody
- dostępnych atrybutów statycznych klasy do której metoda należy
- parametrów dowolnych dostępnych metod statycznych (z klasy macierzystej i dostępnych klas zewnętrznych) wywoływanych w trakcie procesu przetwarzania

5.4.2 Koncepcja określania funkcjonalności na podstawie nazwy metody

Zgodnie z zaprezentowanymi wcześniej rozważaniami rozmiar oprogramowania może być postrzegany w trzech podstawowych aspektach: funkcjonalności, złożoności i długości kodu źródłowego. Wszystkie trzy aspekty są ze sobą wzajemnie powiązane i przez to wpływają na siebie. Dlatego też na potrzeby niniejszej pracy uznano, że estymacja długości kodu może być przynajmniej częściowo wyznaczana na podstawie funkcjonalności.

Wszystkie podstawowe elementy modelu obiektowego, takie jak klasy, metody, atrybuty posiadają własne nazwy. Ich podstawowym celem jest identyfikacja bloków pamięci (w przypadku klas i atrybutów) oraz bloków wykonywalnych instrukcji procesora (w przypadku metod). Patrząc wysokopoziomowo na nazewnictwo klas, atrybutów i metod można przyjąć, że nazwy oprócz swej podstawowej funkcji identyfikacji służą również do opisu pewnych abstrakcyjnych pojęć z dziedziny na której potrzeby system jest realizowany. Jest to rezultatem czysto ludzkiego spojrzenia na modelowanie obiektowe: tworząc byty w świecie obiektowości projektanci i programiści tworzą je na obraz i podobieństwo konkretnych pojęć z dziedziny określanej przez funkcjonalność programu.

O ile nazewnictwo elementów modelu obiektowego nie jest w żaden ścisły sposób usankcjonowane (składnie języków obiektowych praktycznie nie narzucają żadnych wymagań) to w praktyce zwykle stosuje się pewne konwencje nazewnictwa. Są one opracowywane przez firmy zajmujące się wytwarzaniem oprogramowania, zwykle przede wszystkim na własne potrzeby. Posiadanie i stosowanie określonej konwencji tworzenia kodu źródłowego, a w szczególności również ustalonego sposobu nazewnictwa bytów obiektowych ma szereg zalet (Sun Microsystems, Inc., 1997):

- 80% kosztu rozwoju oprogramowania przypada na fazę konserwacji
- rzadko które systemy informatyczne są utrzymywane przez swych autorów
- konwencje kodu źródłowego zwiększają znacząco czytelność kodu źródłowego, pozwalając programistom zrozumieć kod szybciej i dogłębniej
- utrzymanie konwencji kodu źródłowego zapewnia, że czytelność kodu źródłowego wszystkich produktów powstających z biegiem czasu będzie stała zawsze na tym samym poziomie

Głównym zadaniem konwencji nazewnictwa jest zwiększenie czytelności kodu źródłowego. Dokonuje się to na dwóch płaszczyznach:

- wysokopoziomowej, ponieważ nazwa powinna odzwierciedlać funkcję jaką dany byt pełni w abstrakcji dziedziny pojęciowej modelowanego wycinka rzeczywistości
- niskopoziomową, ponieważ nazwa może określać charakterystykę bytu istotną z punktu widzenia programisty (na przykład typ danych, określenie stałości, wielowymiarowości)

Współcześnie najpopularniejsze konwencje nazewnictwa bytów obiektowych sugerują aby (Sun Microsystems, Inc., 1997):

- podstawą nazewnictwa klas były rzeczowniki z ewentualnymi innymi częściami mowy doprecyzowującymi ich opis
- podstawą nazewnictwa metod były czasowniki z ewentualnymi innymi częściami mowy doprecyzowującymi ich opis

Takie podejście ma głęboki sens: klasy zwykle reprezentują konkretne byty z dziedziny pojęciowej, które w naturalny sposób opisuje się przy pomocy rzeczowników (na przykład dom, transakcja, drukowanie), natomiast metody jako elementy wyrażające czynności najwygodniej reprezentować przy pomocy czasowników (na przykład przelicz, wydrukuj, zapisz).

Opisane wyżej podejście do sposobu nazywanie bytów obiektowych implikuje jedno z kluczowych założeń metody opisywanej w niniejszej pracy, a mianowicie, że nazwa metody jest powiązana z jej funkcjonalnością.

Powiązanie to jest do pewnego stopnia niedokładne, gdyż opierając się na nazewnictwie opracowanym w ramach modelowania na podstawie ludzkiego postrzegania świata nosi ono typową cechę charakterystyczną dla tego podejścia, a mianowicie silną zależność od kontekstu. Dla przykładu metoda o nazwie „pobierzDane" wykonywana w kontekście pliku może znacznie się różnić od metody o tej samej wykonywanej w ramach zapytania o konkretne rekordy bazy danych.

Świadomość tej niedokładności spowodowała, że opracowano koncepcje kategorii metody (Booch, 1994)(Riehle, 2000)(Dragan, Collard, & Maletic, 2006). Typowe przykłady kategorii metod to wspominane już wcześniej selektory czy modyfikatory. Należy tutaj zwrócić uwagę, że koncepcja kategorii metody zakłada, że każda metoda ma ściśle określone i pojedyncze zadanie do wykonania. Podejście to ma poparcie w postaci szeroko akceptowanych koncepcji projektowania obiektowego (Beck, 1997) oraz zdrowego rozsądku: łatwiej jest zrozumieć i konserwować metody charakteryzujące się prostotą (zasada *KISS* – ang. *Keep It Simple, Stupid*, mająca swoje początki w programie Apollo realizowanym w USA w latach sześćdziesiątych XX wieku).

Wprowadzając kategorie typów metod należałoby się zastanowić nad sposobem klasyfikacji metody. Podejściem gwarantującym najlepsze rezultaty byłoby określane kategorii metody przez jej autora w procesie rozwoju oprogramowania. Niestety, praktyka pokazuje, że podejścia tego nigdzie się nie stosuje. Pozostają zatem różnorakie metody w sposób automatyczny dokonujące klasyfikacji kategorii metody na podstawie dostępnych danych na jej temat. Dysponując kompletnym kodem źródłowym można by dokonać analizy algorytmu przetwarzania zaimplementowanych w jej ramach i na tej podstawie określić przynależność metody do określonej kategorii, co samo w sobie nie byłoby zadaniem łatwym, ale możliwym do wykonania. Niestety, podejścia takiego nie można zastosować przy estymacji, gdyż na tym etapie kod źródłowy nie jest dostępny.

Dokonując estymacji na podstawie diagramów klas UML jedyną informacją o metodach jaką można uzyskać są:

- nazwa metody
- lista parametrów (danych wejściowych) metody
- lista rezultatów (danych wyjściowych) metody

Podejście do kategoryzacji metod zaprezentowane w niniejszej pracy opiera się na analizie nazwy metody. Jest to prosta metoda opierająca się na analizie nazwy metody w ramach której poszukiwane są w niej określone ciągi znaków (zwykle czasowniki), na podstawie których określana jest kategoria metody. Przyjęto tutaj założenie, że nazewnictwo metod oparte jest na języku angielskim.

Tabela 17 Przykładowe mapowania ciągów znakowych na kategorie metod

kategoria metody	frazy
GETTER	is, get, can, has, accepts, enabled, …
SETTER	set, enable, allow, assign, …
COMPARATOR	compare, equals, less, …
CONVERTER	as, to, convert, …
FINDER	index, choose, find, …

Źródło: opracowanie własne

Kategorie metod wymienione powyżej nie są jedynymi wykorzystywanymi w metodzie opisywanej w niniejszej pracy. Oprócz nich zastosowano również inne, których charakterystyka i wyznaczanie opiera się na specyficznym znaczeniu w modelowaniu obiektowym.

5.4.3 Metryki modelu obiektowego

W celu opracowania modelu szacującego dokonano analizy kodu źródłowego zbioru zrealizowanych programów, wyznaczając dla każdego bytu obiektowego zestaw metryk, służących jako wejście dla modelu szacującego. Metryki te zostały opisane pokrótce poniżej.

Rozmiar projektu *projectSize*

Metryka *projectSize* to rozmiar oprogramowania wyrażony w liczbie leksemów. Jest to zmienna niezależna w modelu szacującym.

Rozmiar klasy *classSize*

Metryka *classSize* jest wyznaczana dla poszczególnych klas i jest zdefiniowana jako rozmiar danej klasy wyrażony w liczbie leksemów. Na rozmiar klasy składa się rozmiar zawartych w niej atrybutów (opis stanu klasy) oraz metod.

Rozmiar opisu stanu klasy *attSize*

Metryka *attSize* to rozmiar opisu stanu klasy wyrażony w liczbie leksemów. Opis stanu klasy to definicja wszystkich (statycznych i niestatycznych) atrybutów wchodzących w skład klasy.

Rozmiar metody *mthSize*

Metryka *mthSize* to rozmiar metody wyrażony w liczbie leksemów, wliczając w to zarówno nagłówek, jak i ciało metody.

Kategoria metody *mthType*

Metryka *mthType* to opis przynależności metody do określonej kategorii strukturalno-funkcjonalnej. Kategorie te to:

- *ABSTRACT* – kategoria metod abstrakcyjnych, czyli takich, które nie posiadają implementacji, a jedynie deklarację (sygnaturę)
- *CONSTRUCTOR* – kategoria metod wywoływanych automatycznie przy tworzeniu instancji klas (obiektów), a służących do ich inicjacji
- *GETTER* – kategoria metod zwracających niestatyczne atrybuty obiektu
- *GETTER_STATIC* – kategoria metod statycznych zwracających statyczne atrybuty klas
- *SETTER* – kategoria metod ustawiających niestateczne atrybuty obiektu
- *SETTER_STATIC* – kategoria metod statycznych ustawiających statyczne atrybuty klas
- *OTHER* – kategoria metod nie zakwalifikowanych do żadnej z powyższych kategorii
- *OTHER_STATIC* – kategoria metod statycznych nie zakwalifikowanych do żadnej z powyższych kategorii

$$methodCategories = \begin{Bmatrix} ABSTRACT, CONSTRUCTOR, \\ GETTER, GETTER\_STATIC, \\ SETTER, SETTER\_STATIC, \\ OTHER, OTHER\_STATIC \end{Bmatrix} \quad 5.35$$

Przynależność do kategorii jest określana na podstawie analizy deklaracji metody (kategoria *ABSTRACT*, *CONSTRUCTOR*) lub analizy nazwy metody (pozostałe kategorie) zgodnie z opisem w podrozdziale 5.4.2).

Łączna liczba atrybutów w klasie *attributes*

Metryka *attributes* oznacza łączną liczbę atrybutów klasy, niezależnie od ich złożoności i statyczności.

$$\boldsymbol{A}_C = \{A_1, A_2, \cdots\} \in C \quad 5.36$$

$$attributes(C) = |\boldsymbol{A}_C| \quad 5.37$$

Łączna liczba metod w klasie *methods*

Metryka *methods* oznacza łączną liczbę metod klasy, niezależnie od ich cech.

$$\boldsymbol{M}_C = \{M_1, M_2, \cdots, M_n\} \in C \quad 5.38$$

$$methods(C) = |\boldsymbol{M}_C| \quad 5.39$$

Łączna liczba rozszerzeń (dziedziczeń) klasy *extends*

Metryka *extends* oznacza łączną liczbę klas po których dana klasa dziedziczy. Dla języka Java dopuszczalne wartości to 0 lub 1.

$$\boldsymbol{C}_P = \{C_1, C_2, \cdots\} \in P \quad 5.40$$

$$\boldsymbol{C}_C^{inherited} \subset \boldsymbol{C}_P \quad 5.41$$

$$extends(C) = \left|\boldsymbol{C}_C^{inherited}\right| \quad 5.42$$

Łączna liczba zaimplementowanych interfejsów w klasie *implements*

Metryka *implements* oznacza łączną interfejsów (klas czysto abstrakcyjnych, czyli bez ciał metod) zaimplementowanych w klasie.

$$\boldsymbol{C}_P^{abstract} = \{C : C.abstract = 1 \cap C \in \boldsymbol{C}_P\} \quad 5.43$$

$$\boldsymbol{C}_C^{implements} \subset \mathbf{C}_{\mathrm{P}}^{\mathrm{abstract}} \quad 5.44$$

$$extends(C) = \left|\boldsymbol{C}_C^{implements}\right| \quad 5.45$$

Proponowane metryki charakteryzują projekt *P* będący elementem zbioru projektów $\boldsymbol{P}$ na podstawie charakterystyki klas *C* wchodzących w skład zbioru klas projektu $\boldsymbol{C}_P$. Każda klasa *C* zawiera zbiór atrybutów $\boldsymbol{A}_C$ oraz zbiór metod $\boldsymbol{M}_C$. Atrybut *A* jest charakteryzowany za pomocą następujących cech:

- oceny złożoności
- statyczności (patrz podrozdział 5.1)
- stałości (patrz podrozdział 5.1)
- skalarności

Ocena złożoności polega na sprawdzeniu czy typ atrybutu należy do określonego zbioru typów podstawowych języka Java:

- `boolean` (typ logiczny)
- `byte` (8-bitowa liczba całkowita ze znakiem)
- `char` (16-bitowy znak Unicode)
- `double` (liczba zmiennoprzecinkowa podwójnej precyzji)
- `float` (liczba zmiennoprzecinkowa pojedynczej precyzji)
- `int` (32-bitowa liczba całkowita ze znakiem)
- `long` (64-bitowa liczba całkowita ze znakiem)
- `short` (16-bitowa liczba całkowita ze znakiem)

Mając świadomość, że pole danych *D* zawiera tekstowy identyfikator typu danych można zdefiniować:

$$typeSimple = \{boolean, byte, char, double, float, int, long, short\} \quad 5.46$$

$$isSimple(D) = \begin{cases} 1, D.type \in typeSimple \\ 0, D.type \in typeSimple' \end{cases} \quad 5.47$$

Skalarność pola danych *D* definiuje, czy jest ono pojedynczą instancją (skalarem) czy też wielowymiarową strukturą (wektorem). Innymi słowy:

$$isScalar(D) = \begin{cases} 1, D.dim = 0 \\ 0, D.dim \neq 0 \end{cases} \quad 5.48$$

Atrybuty klasy są oceniane przy pomocy metryk będących liczbami atrybutów o ustalonej kombinacji oceny złożoności (*simple*), statyczności (*static*), stałości (*const*) i skalarności (*scalar*):

$$\boldsymbol{A}^{static,const}_{simple,scalar}(C) = \left\{ A(C): A \in \boldsymbol{A}_C \cap \begin{array}{c} A.static = static \\ \cap \\ A.const = const \\ \cap \\ isSimple(A.D) = simple \\ \cap \\ isScalar(A.D) = scalar \end{array} \right\} \quad 5.49$$

$$att^{static,const}_{simple,scalar}(C) = \left| \boldsymbol{A}^{static,const}_{simple,scalar}(C) \right| \quad 5.50$$

Taka definicja atrybutów klasy oznacza, że atrybut może być opisywany przy pomocy 16 (2^4 – 4 cechy o wartościach binarnych) metryk tego typu. Spis wszystkich metryk atrybutów zamieszczony został poniżej (Tabela 18).

Tabela 18 Metryki atrybutów

metryka		*static*	*const*	*simple*	*scalar*
attCpl_0	$att^{0,0}_{0,0}$	0	0	0	0
attCpl_1	$att^{0,0}_{0,1}$	0	0	0	1
attSim_0	$att^{0,0}_{1,0}$	0	0	1	0
attSim_1	$att^{0,0}_{1,1}$	0	0	1	1
attCplConst_0	$att^{0,1}_{0,0}$	0	1	0	0
attCplConst_1	$att^{0,1}_{0,1}$	0	1	0	1
attSimConst_0	$att^{0,1}_{1,0}$	0	1	1	0
attSimConst_1	$att^{0,1}_{1,1}$	0	1	1	1
attCplSta_0	$att^{1,0}_{0,0}$	1	0	0	0
attCplSta_1	$att^{1,0}_{0,1}$	1	0	0	1
attSimSta_0	$att^{1,0}_{1,0}$	1	0	1	0
attSimSta_1	$att^{1,0}_{1,1}$	1	0	1	1
attCplStaConst_0	$att^{1,1}_{0,0}$	1	1	0	0
attCplStaConst_1	$att^{1,1}_{0,1}$	1	1	0	1

metryka		*static*	*const*	*simple*	*scalar*
attSimStaConst_0	$att_{1,0}^{1,1}$	1	1	1	0
attSimStaConst_1	$att_{1,1}^{1,1}$	1	1	1	1

Źródło: opracowanie własne

Metody *M* klasy *C* opisywane są za pomocą następujących cech:

- charakterystyki parametrów *P* ze względu na ocenę ich złożoności, stałości, bezwymiarowości i kategorii metody
- charakterystyki rezultatów *R* ze względu na ocenę ich złożoności, skalarności i kategorii metody
- kategorii funkcjonalnej metody

$$category \in methodCategories \tag{5.51}$$

$$methodCategory: M \rightarrow category \tag{5.52}$$

Parametry metody są oceniane przy pomocy metryk będących liczbami parametrów o ustalonej kombinacji oceny złożoności (*simple*), stałości (*const*), skalarności (*scalar*) i kategorii metody (*category*):

$$\boldsymbol{P}_{simple,scalar}^{category,const}(M) = \left\{ P(M): P \in \boldsymbol{R}_M \cap \begin{array}{c} methodCategory(M) = category \\ \cap \\ P.const = const \\ \cap \\ isSimple(P.D) = simple \\ \cap \\ isScalar(P.D) = scalar \end{array} \right\} \tag{5.53}$$

$$p_{simple,scalar}^{category,const}(M) = \left|\boldsymbol{P}_{simple,scalar}^{category,const}(M)\right| \tag{5.54}$$

Zgodnie z tą definicją parametry metody mogą być charakteryzowane przy pomocy 64 metryk (8×2^3 – 3 cechy o wartościach binarnych przemnożone przez 8 kategorii metod). Spis wszystkich metryk parametrów metody został zamieszczony poniżej (Tabela 19).

Tabela 19 Metryki parametrów metod

metryka		*category*	*const*	*simple*	*scalar*
pCpl_ABSTRACT_0	$p_{0,0}^{ABSTRACT,0}$	*ABSTRACT*	0	0	0
pCpl_ABSTRACT_1	$p_{0,1}^{ABSTRACT,0}$	*ABSTRACT*	0	0	1
pSim_ABSTRACT_0	$p_{1,0}^{ABSTRACT,0}$	*ABSTRACT*	0	1	0

metryka		*category*	*const*	*simple*	*scalar*
pSim_ABSTRACT_1	$p_{1,1}^{ABSTRACT,0}$	*ABSTRACT*	0	1	1
pCplConst_ABSTRACT_0	$p_{0,0}^{ABSTRACT,1}$	*ABSTRACT*	1	0	0
pCplConst_ABSTRACT_1	$p_{0,1}^{ABSTRACT,1}$	*ABSTRACT*	1	0	1
pSimConst_ABSTRACT_0	$p_{1,0}^{ABSTRACT,1}$	*ABSTRACT*	1	1	0
pSimConst_ABSTRACT_1	$p_{1,1}^{ABSTRACT,1}$	*ABSTRACT*	1	1	1
pCpl_CONSTRUCTOR_0	$p_{0,0}^{CONSTRUCTOR,0}$	*CONSTRUCTOR*	0	0	0
pCpl_CONSTRUCTOR_1	$p_{0,1}^{CONSTRUCTOR,0}$	*CONSTRUCTOR*	0	0	1
pSim_CONSTRUCTOR_0	$p_{1,0}^{CONSTRUCTOR,0}$	*CONSTRUCTOR*	0	1	0
pSim_CONSTRUCTOR_1	$p_{1,1}^{CONSTRUCTOR,0}$	*CONSTRUCTOR*	0	1	1
pCplConst_CONSTRUCTOR_0	$p_{0,0}^{CONSTRUCTOR,1}$	*CONSTRUCTOR*	1	0	0
pCplConst_CONSTRUCTOR_1	$p_{0,1}^{CONSTRUCTOR,1}$	*CONSTRUCTOR*	1	0	1
pSimConst_CONSTRUCTOR_0	$p_{1,0}^{CONSTRUCTOR,1}$	*CONSTRUCTOR*	1	1	0
pSimConst_CONSTRUCTOR_1	$p_{1,1}^{CONSTRUCTOR,1}$	*CONSTRUCTOR*	1	1	1
pCpl_GETTER_0	$p_{0,0}^{GETTER,0}$	*GETTER*	0	0	0
pCpl_GETTER_1	$p_{0,1}^{GETTER,0}$	*GETTER*	0	0	1
pSim_GETTER_0	$p_{1,0}^{GETTER,0}$	*GETTER*	0	1	0
pSim_GETTER_1	$p_{1,1}^{GETTER,0}$	*GETTER*	0	1	1
pCplConst_GETTER_0	$p_{0,0}^{GETTER,1}$	*GETTER*	1	0	0
pCplConst_GETTER_1	$p_{0,1}^{GETTER,1}$	*GETTER*	1	0	1
pSimConst_GETTER_0	$p_{1,0}^{GETTER,1}$	*GETTER*	1	1	0
pSimConst_GETTER_1	$p_{1,1}^{GETTER,1}$	*GETTER*	1	1	1
pCpl_GETTER_STATIC_0	$p_{0,0}^{GETTER\_STATIC,0}$	*GETTER_STATIC*	0	0	0
pCpl_GETTER_STATIC_1	$p_{0,1}^{GETTER\_STATIC,0}$	*GETTER_STATIC*	0	0	1
pSim_GETTER_STATIC_0	$p_{1,0}^{GETTER\_STATIC,0}$	*GETTER_STATIC*	0	1	0
pSim_GETTER_STATIC_1	$p_{1,0}^{GETTER\_STATIC,0}$	*GETTER_STATIC*	0	1	1
pCplConst_GETTER_STATIC_0	$p_{0,0}^{GETTER\_STATIC,1}$	*GETTER_STATIC*	1	0	0
pCplConst_GETTER_STATIC_1	$p_{0,1}^{GETTER\_STATIC,1}$	*GETTER_STATIC*	1	0	1
pSimConst_GETTER_STATIC_0	$p_{1,0}^{GETTER\_STATIC,1}$	*GETTER_STATIC*	1	1	0
pSimConst_GETTER_STATIC_1	$p_{1,1}^{GETTER\_STATIC,1}$	*GETTER_STATIC*	1	1	1
pCpl_SETTER_0	$p_{0,0}^{SETTER,0}$	*SETTER*	0	0	0
pCpl_SETTER_1	$p_{0,1}^{SETTER,0}$	*SETTER*	0	0	1
pSim_SETTER_0	$p_{1,0}^{SETTER,0}$	*SETTER*	0	1	0
pSim_SETTER_1	$p_{1,1}^{SETTER,0}$	*SETTER*	0	1	1
pCplConst_SETTER_0	$p_{0,0}^{SETTER,1}$	*SETTER*	1	0	0
pCplConst_SETTER_1	$p_{0,1}^{SETTER,1}$	*SETTER*	1	0	1
pSimConst_SETTER_0	$p_{1,0}^{SETTER,1}$	*SETTER*	1	1	0
pSimConst_SETTER_1	$p_{1,1}^{SETTER,1}$	*SETTER*	1	1	1
pCpl_SETTER_STATIC_0	$p_{0,0}^{SETTER\_STATIC,0}$	*SETTER_STATIC*	0	0	0
pCpl_SETTER_STATIC_1	$p_{0,1}^{SETTER\_STATIC,0}$	*SETTER_STATIC*	0	0	1
pSim_SETTER_STATIC_0	$p_{1,0}^{SETTER\_STATIC,0}$	*SETTER_STATIC*	0	1	0
pSim_SETTER_STATIC_1	$p_{1,1}^{SETTER\_STATIC,0}$	*SETTER_STATIC*	0	1	1

metryka		*category*	*const*	*simple*	*scalar*
pCplConst_SETTER_STATIC_0	$p_{0,0}^{SETTER\_STATIC,1}$	*SETTER_STATIC*	1	0	0
pCplConst_SETTER_STATIC_1	$p_{0,1}^{SETTER\_STATIC,1}$	*SETTER_STATIC*	1	0	1
pSimConst_SETTER_STATIC_0	$p_{1,0}^{SETTER\_STATIC,1}$	*SETTER_STATIC*	1	1	0
pSimConst_SETTER_STATIC_1	$p_{1,1}^{SETTER\_STATIC,1}$	*SETTER_STATIC*	1	1	1
pCpl_OTHER_0	$p_{0,0}^{OTHER,0}$	*OTHER*	0	0	0
pCpl_OTHER_1	$p_{0,1}^{OTHER,0}$	*OTHER*	0	0	1
pSim_OTHER_0	$p_{1,0}^{OTHER,0}$	*OTHER*	0	1	0
pSim_OTHER_1	$p_{1,1}^{OTHER,0}$	*OTHER*	0	1	1
pCplConst_OTHER_0	$p_{0,0}^{OTHER,1}$	*OTHER*	1	0	0
pCplConst_OTHER_1	$p_{0,1}^{OTHER,1}$	*OTHER*	1	0	1
pSimConst_OTHER_0	$p_{1,0}^{OTHER,1}$	*OTHER*	1	1	0
pSimConst_OTHER_1	$p_{1,1}^{OTHER,1}$	*OTHER*	1	1	1
pCpl_OTHER_STATIC_0	$p_{0,0}^{OTHER\_STATIC,0}$	*OTHER_STATIC*	0	0	0
pCpl_OTHER_STATIC_1	$p_{0,1}^{OTHER\_STATIC,0}$	*OTHER_STATIC*	0	0	1
pSim_OTHER_STATIC_0	$p_{1,0}^{OTHER\_STATIC,0}$	*OTHER_STATIC*	0	1	0
pSim_OTHER_STATIC_1	$p_{1,1}^{OTHER\_STATIC,0}$	*OTHER_STATIC*	0	1	1
pCplConst_OTHER_STATIC_0	$p_{0,0}^{OTHER\_STATIC,1}$	*OTHER_STATIC*	1	0	0
pCplConst_OTHER_STATIC_1	$p_{0,1}^{OTHER\_STATIC,1}$	*OTHER_STATIC*	1	0	1
pSimConst_OTHER_STATIC_0	$p_{1,0}^{OTHER\_STATIC,1}$	*OTHER_STATIC*	1	1	0
pSimConst_OTHER_STATIC_1	$p_{1,1}^{OTHER\_STATIC,1}$	*OTHER_STATIC*	1	1	1

Źródło: opracowanie własne

Rezultaty metody są oceniane przy pomocy metryk będących liczbami parametrów o ustalonej kombinacji oceny złożoności (*simple*), skalarności (*scalar*) i kategorii metody (*category*):

$$\boldsymbol{R}_{simple,scalar}^{category}(M) = \left\{ R(M): R \in \boldsymbol{R}_M \cap \begin{array}{c} methodCategory(M) = category \\ \cap \\ isSimple(R.D) = simple \\ \cap \\ isScalar(R.D) = scalar \end{array} \right\} \quad 5.55$$

$$r_{simple,scalar}^{category}(M) = \left| \boldsymbol{R}_{simple,scalar}^{category}(M) \right| \quad 5.56$$

Zgodnie z tą definicją rezultaty metody mogą być charakteryzowane przy pomocy 32 metryk (8×2^2 – 2 cechy o wartościach binarnych przemnożone przez 8 kategorii metod). Spis wszystkich metryk rezultatów metody został zamieszczony poniżej (Tabela 20).

Tabela 20 Metryki rezultatów metod

metryka		*category*	*simple*	*scalar*
rCpl_ABSTRACT_0	$r_{0,0}^{ABSTRACT}$	*ABSTRACT*	0	0
rCpl_ABSTRACT_1	$r_{0,1}^{ABSTRACT}$	*ABSTRACT*	0	1
rSim_ABSTRACT_0	$r_{1,0}^{ABSTRACT}$	*ABSTRACT*	1	0
rSim_ABSTRACT_1	$r_{1,0}^{ABSTRACT}$	*ABSTRACT*	1	1
rCpl_CONSTRUCTOR_0	$r_{0,0}^{CONSTRUCTOR}$	*CONSTRUCTOR*	0	0
rCpl_CONSTRUCTOR_1	$r_{0,1}^{CONSTRUCTOR}$	*CONSTRUCTOR*	0	1
rSim_CONSTRUCTOR_0	$r_{1,0}^{CONSTRUCTOR}$	*CONSTRUCTOR*	1	0
rSim_CONSTRUCTOR_1	$r_{1,0}^{CONSTRUCTOR}$	*CONSTRUCTOR*	1	1
rCpl_GETTER_0	$r_{0,0}^{GETTER}$	*GETTER*	0	0
rCpl_GETTER_1	$r_{0,1}^{GETTER}$	*GETTER*	0	1
rSim_GETTER_0	$r_{1,0}^{GETTER}$	*GETTER*	1	0
rSim_GETTER_1	$r_{1,0}^{GETTER}$	*GETTER*	1	1
rCpl_GETTER_STATIC_0	$r_{0,0}^{GETTER\_STATIC}$	*GETTER_STATIC*	0	0
rCpl_GETTER_STATIC_1	$r_{0,1}^{GETTER\_STATIC}$	*GETTER_STATIC*	0	1
rSim_GETTER_STATIC_0	$r_{1,0}^{GETTER\_STATIC}$	*GETTER_STATIC*	1	0
rSim_GETTER_STATIC_1	$r_{1,0}^{GETTER\_STATIC}$	*GETTER_STATIC*	1	1
rCpl_SETTER_0	$r_{0,0}^{SETTER}$	*SETTER*	0	0
rCpl_SETTER_1	$r_{0,1}^{SETTER}$	*SETTER*	0	1
rSim_SETTER_0	$r_{1,0}^{SETTER}$	*SETTER*	1	0
rSim_SETTER_1	$r_{1,0}^{SETTER}$	*SETTER*	1	1
rCpl_SETTER_STATIC_0	$r_{0,0}^{SETTER\_STATIC}$	*SETTER_STATIC*	0	0
rCpl_SETTER_STATIC_1	$r_{0,1}^{SETTER\_STATIC}$	*SETTER_STATIC*	0	1
rSim_SETTER_STATIC_0	$r_{1,0}^{SETTER\_STATIC}$	*SETTER_STATIC*	1	0
rSim_SETTER_STATIC_1	$r_{1,0}^{SETTER\_STATIC}$	*SETTER_STATIC*	1	1
rCpl_OTHER_0	$r_{0,0}^{OTHER}$	*OTHER*	0	0
rCpl_OTHER_1	$r_{0,1}^{OTHER}$	*OTHER*	0	1
rSim_OTHER_0	$r_{1,0}^{OTHER}$	*OTHER*	1	0
rSim_OTHER_1	$r_{1,0}^{OTHER}$	*OTHER*	1	1
rCpl_OTHER_STATIC_0	$r_{0,0}^{OTHER\_STATIC}$	*OTHER_STATIC*	0	0
rCpl_OTHER_STATIC_1	$r_{0,1}^{OTHER\_STATIC}$	*OTHER_STATIC*	0	1
rSim_OTHER_STATIC_0	$r_{1,0}^{OTHER\_STATIC}$	*OTHER_STATIC*	1	0
rSim_OTHER_STATIC_1	$r_{1,0}^{OTHER\_STATIC}$	*OTHER_STATIC*	1	1

Źródło: opracowanie własne

Poniżej zamieszczony został schemat obrazujący zastosowania metryk dla bezpośredniego opisu wybranych elementów modelu obiektowego. Czcionką pogrubioną oznaczone zostały metryki będące zmiennymi niezależnymi.

Tabela 21 Zastosowanie metryk do bezpośredniego opisu wybranych elementów modelu obiektowego

	system	klasa	metoda
projectSize	tak		
classSize		tak	
attSize		tak	
attributes		tak	
methods		tak	
extends		tak	
implements		tak	
$att_{simple,scalar}^{static,const}$		tak	
mthSize			tak
mthType			tak
$p_{simple,scalar}^{category,const}$			tak
$r_{simple,scalar}^{category}$			tak

Źródło: opracowanie własne

Zgodnie z zaprezentowanym powyżej schematem (Tabela 21) do bezpośredniego opisu klas wykorzystywane są przede wszystkim metryki powiązane z atrybutami i samej klasy. Jednakże pośredni wpływ na klasę mają również metody: jest on uwzględniony poprzez uwzględnienie metryk specyficznych dla metod dla każdej metody składowej klasy.

Rozmiar metody zależy również od atrybutów klasy do której metoda należy. Jest to oczywiste w świetle ogólnego modelu przetwarzania w środowisku obiektowym – metoda zawsze działa w kontekście swojej klasy, a więc jej atrybuty wpływają pośrednio na jej charakterystykę.

W badaniach przeprowadzonych na potrzeby niniejszej pracy wszystkie metryki były wyznaczane na podstawie istniejącego kodu źródłowego. Jednakże nic nie stoi na przeszkodzie, aby takie metryki wyznaczać na podstawie diagramów klas UML, co jest bardziej naturalne w środowisku wytwarzania oprogramowania, kiedy nie ma jeszcze kodu źródłowego. Proces wyznaczania metryk można stosunkowo niewielkim kosztem zautomatyzować, korzystając z dostępnej praktycznie w każdym edytorze diagramów UML opcji eksportu modelu do formatu XMI.

Format XMI jest oficjalnym standardem wymiany danych opracowanym przez organizację Object Management Group, która zajmuje się rozwijaniem języka UML. Format ten jest dialektem XML, co pozwala na łatwe przetwarzanie modeli w nim zapisanych.

Format XMI jest również oficjalnym międzynarodowym standardem wymiany metadanych (ISO/IEC 19503:2005 Information Technology – XML Metadata Interchange (XMI)).

5.4.4 Postać i proces budowy modelu szacującego

Głównym elementem proponowanej metody estymacji jest model szacujący. Model ten został stworzony z wykorzystaniem metody regresji wielorakiej. Jako podstawę dla stworzenia modelu szacującego wykorzystano 50-elementowy podzbiór uczący. Na model składa się 8 podmodeli stworzonych w optymalizującym procesie dopasowywania wyników regresji wielorakiej do podzbiorów reprezentujących podobne elementy zbioru uczącego.

Proces dopasowywania polegał na próbie wydzielenia podzbioru spełniającego założenia regresji wielorakiej, czyli (Welfe, 2003):

- niezmienniczości modelu ze względu na obserwacje
- liniowości modelu względem parametrów
- nielosowości wartości zmiennych objaśniających
- równości rzędu macierzy zmiennych objaśniających i liczby szacowanych parametrów
- posiadania wielowymiarowego rozkładu normalnego przez składnik losowy
- zerowej wartości oczekiwanej składnika losowego
- sferyczności składnika losowego
- opieraniu się wyłącznie na informacjach zawartych w modelu

Autorski program działający w środowisku pakietu statystycznego *Stata 10* (http://www.stata.com/) eliminował skrajne przypadki z podzbioru testującego, doprowadzając do wydzielenia podzbioru spełniającego kryteria regresji wielorakiej. Jednocześnie stworzony został model szacujący dopasowany do tego konkretnego podzbioru. Proces ten był powtarzany ośmiokrotnie dla pozostałych (nie włączonych do modelu) przypadków podzbioru uczącego.

Statystyki rozmiarów klas, ilości atrybutów, ilości metod podzbiorów klas na bazie których zostały zbudowane podmodele zostały zamieszczone poniżej.

Tabela 22 Przegląd statystyk rozmiaru klasy podzbiorów bazowych podmodeli

	liczba obserwacji	średnia	odchylenie standardowe	minimum	maksimum
podmodel 1	15904	59,39072	289,932	2	16467
podmodel 2	16332	138,508	417,751	7	42687
podmodel 3	7852	317,9003	448,5614	27	16001
podmodel 4	10441	544,808	819,0063	11	60252
podmodel 5	5237	730,435	836,5126	19	14797
podmodel 6	1637	1327,27	1073,269	95	12490
podmodel 7	1425	1929,683	1577,361	194	15685
podmodel 8	1894	1767,859	1815,666	49	36061
pozostałe	1986	4006,069	4742,624	45	81465

Źródło: opracowanie własne

Analizując statystyki rozmiaru klasy podzbiorów bazowych można zauważyć, że typowy rozmiar klasy zwiększa się wraz z numerem porządkowym podmodelu. Oznacza to, że im „późniejszy" podmodel tym większy średni rozmiar klasy (z pojedynczym wyjątkiem w przypadku podmodeli 7 oraz 8). Podobną charakterystykę ma zmienność rozmiaru klasy: podmodele wcześniejsze charakteryzują się mniejszą zmiennością.

Tabela 23 Przegląd statystyk ilości atrybutów klasy podzbiorów bazowych podmodeli

	liczba obserwacji	średnia	odchylenie standardowe	minimum	maksimum
podmodel 1	15904	0,833312	2,832727	0	71
podmodel 2	16332	1,517205	2,723969	0	45
podmodel 3	7852	2,616021	7,543491	0	358
podmodel 4	10441	4,008716	6,791152	0	161
podmodel 5	5237	5,513462	11,15332	0	321
podmodel 6	1637	7,83201	21,63123	0	617
podmodel 7	1425	8,700351	21,84208	0	448
podmodel 8	1894	10,48205	15,92506	0	246
pozostałe	1986	14,36707	32,89575	0	1095

Źródło: opracowanie własne

Statystyki ilości atrybutów klasy w zależności od podzbioru pokazują, że podmodele o wyższym numerze porządkowym dotyczą klas o bardziej rozbudowanym opisie stanu. Potwierdza to związek zaobserwowany na podstawie wyników poprzedniej tablicy: podmodele o większym numerze porządkowym odnoszą się do klas o większym rozmiarze.

Tabela 24 Przegląd statystyk ilości metod klasy podzbiorów bazowych podmodeli

	liczba obserwacji	średnia	odchylenie standardowe	minimum	maksimum
podmodel 1	15904	2,890342	4,526701	0	160
podmodel 2	16332	4,610397	8,497583	0	708
podmodel 3	7852	6,128502	7,884607	0	258
podmodel 4	10441	9,442391	11,96687	0	454
podmodel 5	5237	13,11552	14,34818	0	230
podmodel 6	1637	15,26634	18,98693	0	189
podmodel 7	1425	18,07579	23,14779	0	224
podmodel 8	1894	23,07656	24,72945	0	327
pozostałe	1986	33,3288	40,91163	0	707

Źródło: opracowanie własne

Statystyki ilości metod klasy potwierdzają wcześniejsze spostrzeżenie o związku między wartością numeru porządkowego podmodeli a rozmiarem klasy: im „późniejszy" podmodel tym większych i bardziej zmiennych klas on dotyczy.

W poniższej tablicy znajduje się zestawienie parametrów równań regresji wielorakiej dla wszystkich 8 podmodeli składających się na pełny model.

Tabela 25 Wartości parametrów metryk objaśniających w poszczególnych podmodelach

	podmodel 1	podmodel 2	podmodel 3	podmodel 4	podmodel 5	podmodel 6	podmodel 7	podmodel 8
wyraz wolny	5,669857	8,418936	73,52192	45,97138	85,24409	356,7409	644,0016	246,2353
methods	5,363048	7,537074	11,77388	31,97805	11,33564	37,28238	27,17273	27,65834
attributes	25,38593	113,8702	265,1664	8,666136	-39,0971	-189,854	0	-29,4787
extends	3,979077	15,59802	20,82045	88,53971	12,13588	116,7803	89,31837	253,2189
implements	3,706936	2,198578	-4,71062	21,89796	-31,9189	36,18374	0	-76,1941
attCpl_0	-2,18597	-106,306	-211,636	38,45593	0	250,3937	123,5785	0
attCpl_1	-21,4068	-95,6047	-233,97	42,80805	62,37843	236,5636	62,37789	119,8041
attSim_0	-10,5629	-59,4485	-119,69	76,8867	115,6075	339,4835	155,8482	0
attSim_1	-20,7179	-107,747	-260,549	23,41639	46,41817	228,0161	5,472886	147,8604
attCplConst_0	0	-94,213	-219,416	0	0	232,252	0	0
attCplConst_1	-19,5521	-102,084	-247,592	25,02258	81,95764	241,0821	62,20671	0
attSimConst_0	-11,3866	34,27504	-284,695	104,006	0	0	785,9223	540,4463
attSimConst_1	-17,7153	-103,96	-251,134	32,36324	79,66968	238,0861	0	282,5446
attCplSta_0	74,97857	69,13742	-187,596	0	238,9611	342,3206	50,97801	110,8531
attCplSta_1	-10,0574	-85,1803	-230,21	52,53529	61,01728	199,1879	-3,75952	185,3976
attSimSta_0	164,3811	0	0	-75,3171	561,0331	163,9317	96,41529	323,1187
attSimSta_1	63,37712	-37,9084	-222,555	0	80,21597	182,3366	217,0607	110,2488

	podmodel 1	podmodel 2	podmodel 3	podmodel 4	podmodel 5	podmodel 6	podmodel 7	podmodel 8
attCplStaConst_0	2,720917	-85,9375	-219,987	22,40215	364,5847	63,93	189,2093	0
attCplStaConst_1	-17,3471	-102,298	-243,734	0	94,63186	195,0258	6,305516	44,24411
attSimStaConst_0	84,8088	35,48888	-204,742	69,7817	245,3146	177,3124	205,6384	0
attSimStaConst_1	-17,3519	-105,188	-251,171	14,31964	43,80588	200,3601	17,19831	48,70461
pCpl_ABSTRACT_0	5,22092	5,29078	-14,8988	103,1409	-45,8437	0	-213,084	-370,925
pCpl_ABSTRACT_1	2,57685	0,683231	-0,94522	-15,1166	3,611245	4,474906	-23,4865	-12,061
pSim_ABSTRACT_0	4,484074	9,021517	-15,9413	-45,8765	53,13353	-167,639	-540,272	-868,383
pSim_ABSTRACT_1	2,998919	1,799105	3,478471	0	11,8719	9,789426	42,51783	62,03485
pCplConst_ABSTRACT_0	5,916923	14,02957	0	0	0	0	0	0
pCplConst_ABSTRACT_1	3,70714	2,961523	-14,9538	20,58357	0	0	0	0
pSimConst_ABSTRACT_0	10,2878	72,80848	0	0	0	0	0	0
pSimConst_ABSTRACT_1	4,39334	0	-32,403	-525,692	0	0	0	0
pCpl_CONSTRUCTOR_0	7,907716	4,347878	-11,6512	27,34051	24,37647	-29,5906	47,83128	-64,8529
pCpl_CONSTRUCTOR_1	7,05379	3,402255	-3,118	-8,9999	7,568937	-9,15848	-12,6225	19,6774
pSim_CONSTRUCTOR_0	15,21121	-17,916	-31,8063	-12,6184	0	-120,244	186,8955	-159,546
pSim_CONSTRUCTOR_1	4,235385	5,757326	10,71858	-3,54202	6,553523	-38,0817	-15,1256	0
pCplConst_CONSTRUCTOR_0	-4,75084	50,79435	0	0	89,21337	260,5657	-505,725	0
pCplConst_CONSTRUCTOR_1	7,910415	9,181498	3,887954	-41,2904	0	-45,5232	-128,716	0
pSimConst_CONSTRUCTOR_0	76,04044	-85,2793	-95,2031	0	1573,102	-1043,31	0	0
pSimConst_CONSTRUCTOR_1	7,606069	7,053949	12,07778	-69,1706	87,10363	507,6494	0	-478,135
pCpl_GETTER_0	-11,2371	8,437843	60,07916	-117,342	127,5561	66,72301	-313,876	251,6721
pCpl_GETTER_1	31,5661	13,86647	19,38234	49,72718	6,679297	50,33135	60,43369	32,50393
pSim_GETTER_0	52,13178	57,39635	152,668	160,0625	0	325,9332	202,4381	-524,337
pSim_GETTER_1	28,65223	9,318837	7,894433	45,8984	0	33,05968	3,21602	171,3029
pCplConst_GETTER_0	52,29173	-38,424	302,2799	-101,34	469,5236	-553,764	-1187,88	0
pCplConst_GETTER_1	46,78373	74,47861	55,00702	43,46596	0	0	122,0438	261,4281
pSimConst_GETTER_0	588,9362	0	-230,5	2141,451	0	0	0	0
pSimConst_GETTER_1	188,3343	57,78047	-34,3998	144,0756	0	-331,99	119,4305	0
pCpl_GETTER_STATIC_0	92,77814	89,04875	-66,0588	27,43859	-168,619	252,8329	-39,2651	-138,055
pCpl_GETTER_STATIC_1	15,50317	24,78361	25,91382	10,11881	76,81028	-74,5224	-13,8817	36,26349
pSim_GETTER_STATIC_0	281,5671	210,3705	46,9987	-74,6142	82,40608	285,5091	-396,702	620,6271
pSim_GETTER_STATIC_1	4,325952	31,3961	4,662815	0	58,6633	47,09889	265,735	0
pCplConst_GETTER_STATIC_0	40,53164	-293,142	-42,789	0	-981,975	-1630,34	0	0
pCplConst_GETTER_STATIC_1	-7,72163	18,41455	43,77389	-19,0969	126,4085	71,83741	-34,1096	0
pSimConst_GETTER_STATIC_0	-358,479	0	-64,312	1200,299	0	0	0	0
pSimConst_GETTER_STATIC_1	280,3705	43,23754	327,543	0	268,9554	1110,443	4398,715	0
pCpl_SETTER_0	216,013	31,73235	86,7856	126,3047	53,85703	101,5523	0	146,2185
pCpl_SETTER_1	8,600148	8,055962	6,199575	-44,1745	9,805128	-48,1541	-28,1524	-141,936
pSim_SETTER_0	0	15,6066	0	-19,9951	-151,172	271,3571	-499,36	298,094
pSim_SETTER_1	5,540626	13,84626	7,9343	-20,156	56,29171	-37,2453	0	15,4863
pCplConst_SETTER_0	0	133,9069	-233,558	357,5947	0	-1426,94	0	0
pCplConst_SETTER_1	46,2453	10,54643	23,75043	-16,9192	0	72,58311	-265,501	0
pSimConst_SETTER_0	0	0	1154,892	972,4151	-1044,49	0	0	0
pSimConst_SETTER_1	44,78918	95,02666	-15,6972	0	-103,906	-99,3965	503,0092	273,0625

	podmodel 1	podmodel 2	podmodel 3	podmodel 4	podmodel 5	podmodel 6	podmodel 7	podmodel 8
pCpl_SETTER_STATIC_0	186,5367	232,4411	-83,3739	553,2441	1111,857	1586,891	-580,699	2839,496
pCpl_SETTER_STATIC_1	-16,3347	38,29352	-62,5435	61,61354	-37,8436	34,20019	284,6922	-114,282
pSim_SETTER_STATIC_0	995,4871	0	444,2424	1148,3	0	0	0	988,7078
pSim_SETTER_STATIC_1	-31,7476	-15,6018	-30,8048	-79,3963	-107,153	168,9059	-66,7901	157,6149
pCplConst_SETTER_STATIC_0	0	0	0	0	0	0	0	0
pCplConst_SETTER_STATIC_1	129,5373	96,57393	226,9087	1079,701	-454,631	1601,898	0	0
pSimConst_SETTER_STATIC_0	0	0	0	0	0	0	0	0
pSimConst_SETTER_STATIC_1	-92,2044	0	0	0	0	0	0	0
pCpl_OTHER_0	7,94016	24,97708	20,68608	-27,158	145,3621	0	-48,2281	-76,5448
pCpl_OTHER_1	7,161348	15,89433	17,56989	10,18092	10,6652	15,87463	24,80376	18,91003
pSim_OTHER_0	7,90554	-8,10148	0	0	-80,6123	33,14864	-51,9964	207,1781
pSim_OTHER_1	28,06073	14,37557	9,882568	-4,48904	43,39656	5,654428	-5,07847	-68,8569
pCplConst_OTHER_0	70,90285	20,32806	70,73953	138,7939	-77,0254	173,8118	556,9616	0
pCplConst_OTHER_1	23,18732	22,50749	22,35782	4,713296	54,77572	-23,7709	43,70222	-16,4965
pSimConst_OTHER_0	27,92504	0	284,9526	0	58,20756	826,9884	0	3456,442
pSimConst_OTHER_1	20,14987	26,74433	0	0	67,92149	-68,6069	-38,7474	0
pCpl_OTHER_STATIC_0	33,19864	95,28429	152,2944	12,2957	328,6332	78,52884	35,98596	0
pCpl_OTHER_STATIC_1	20,19131	17,89192	23,57247	12,49123	16,69444	3,902921	32,27074	39,74277
pSim_OTHER_STATIC_0	72,23	6,675467	24,78992	95,40708	0	-107,577	115,6987	-244,853
pSim_OTHER_STATIC_1	-3,49534	13,34322	12,19327	3,032338	27,26009	0	-42,4589	92,67324
pCplConst_OTHER_STATIC_0	47,41807	-29,1841	71,72734	208,3177	-157,495	455,6804	0	-755,97
pCplConst_OTHER_STATIC_1	23,62318	30,75994	20,32718	12,14148	19,04829	13,71716	57,65731	39,40571
pSimConst_OTHER_STATIC_0	-50,5766	0	856,3753	198,286	1638,841	-805,151	0	0
pSimConst_OTHER_STATIC_1	61,97161	-3,17634	0	85,78708	-126,862	-131,608	-112,801	908,7309
rCpl_ABSTRACT_0	2,231426	0	-5,78236	-109,555	27,42025	-152,139	-488,017	0
rCpl_ABSTRACT_1	1,204657	1,377358	2,674484	8,617221	-7,90104	-8,02831	18,49112	0
rSim_ABSTRACT_0	1,963269	0	9,717829	-45,0641	0	309,7361	-308,347	1350,679
rSim_ABSTRACT_1	0,250035	-4,25622	-3,77822	-23,4299	-22,2418	11,96105	57,66642	-86,684
rCpl_CONSTRUCTOR_0	0	0	0	0	0	0	0	0
rCpl_CONSTRUCTOR_1	1,532397	14,3882	25,36747	32,8274	0	0	76,13067	0
rSim_CONSTRUCTOR_0	70,62273	75,87945	0	0	0	0	0	0
rSim_CONSTRUCTOR_1	15,32003	24,73891	-27,2713	0	0	0	0	0
rCpl_GETTER_0	13,3335	2,826926	11,92461	44,2787	-12,0776	16,52067	173,3839	46,35541
rCpl_GETTER_1	6,049649	3,500951	0	-28,279	8,652131	-11,3694	-55,4821	6,835836
rSim_GETTER_0	12,6349	0	32,63019	-59,2636	81,40527	37,0165	-155,725	154,3189
rSim_GETTER_1	9,24502	9,324372	2,866855	-44,9503	10,02827	-48,0145	-17,9012	-68,2352
rCpl_GETTER_STATIC_0	41,20681	31,74615	68,17086	0	0	-54,2088	-147,744	218,538
rCpl_GETTER_STATIC_1	33,433	14,89794	21,79073	30,16933	-25,5798	117,3899	45,30159	-59,8526
rSim_GETTER_STATIC_0	-39,6713	-127,062	-42,1404	0	150,7299	-103,406	-537,788	-1079,92
rSim_GETTER_STATIC_1	4,719636	0	45,09065	22,31291	-16,8998	36,96203	-30,1747	226,1814
rCpl_SETTER_0	23,73353	-114,664	420,082	-342,029	1499,351	1130,761	0	5027,273
rCpl_SETTER_1	26,77794	6,371285	5,545497	58,15704	-31,1693	26,52417	45,39325	85,6139
rSim_SETTER_0	1539,408	0	0	0	0	0	0	-770,546
rSim_SETTER_1	80,04432	52,36252	-72,8599	-80,2263	79,93634	0	81,11833	339,885

	podmodel 1	podmodel 2	podmodel 3	podmodel 4	podmodel 5	podmodel 6	podmodel 7	podmodel 8
rCpl_SETTER_STATIC_0	-105,418	236,2689	0	0	0	0	0	0
rCpl_SETTER_STATIC_1	63,58054	-36,3821	28,33784	-138,764	152,6733	-227,245	-245,227	544,076
rSim_SETTER_STATIC_0	302,8357	0	0	0	0	0	0	0
rSim_SETTER_STATIC_1	-185,445	0	173,9216	0	990,4021	-966,935	-873,502	-2243,37
rCpl_OTHER_0	10,03736	52,89283	52,23411	-35,0181	521,6803	-19,7007	156,7348	0
rCpl_OTHER_1	6,991027	5,521074	5,953665	13,73586	0	9,620962	38,37418	3,376031
rSim_OTHER_0	59,39392	-20,6339	81,16701	-52,4917	486,0348	0	-196,368	-210,65
rSim_OTHER_1	9,629023	8,809983	2,988105	-18,187	33,7739	-8,7213	-14,7556	0
rCpl_OTHER_STATIC_0	-11,9062	-52,7294	-55,0984	95,82994	-223,007	-24,3558	-120,416	339,296
rCpl_OTHER_STATIC_1	12,89442	14,68298	6,987242	-3,66718	32,90686	10,01017	21,27316	-47,8648
rSim_OTHER_STATIC_0	536,8548	543,2953	508,1183	390,4483	-65,5125	164,5417	410,3282	586,2244
rSim_OTHER_STATIC_1	18,49198	53,68409	-46,5431	0	66,64194	-14,7991	6,485976	-149,07

Źródło: opracowanie własne

Model szacujący jest stworzony do estymacji rozmiaru klasy. Oszacowanie rozmiaru całego programu następuje poprzez zsumowanie oszacowań klas zawartych w jego ramach.

5.4.5 Model bazowy

Jak wspomniano wcześniej, na kompletny model szacujący w proponowanej metodzie składa się zestaw 8 podmodeli szacujących rozmiar oprogramowania w różnych aspektach rodzajów charakterystyki klas. Oszacowania realizowane przez poszczególne podmodele można łączyć, na przykład przy użyciu średniej ważonej. Sposób łączenia oszacowań podmodeli powinien być eksperymentalnie opracowany dla specyficznego środowiska rozwoju oprogramowania.

Na potrzeby testowania proponowanego modelu stworzono bazowy sposób łączenia oszacowań wyznaczanych przez podmodele. Bazując na oszacowaniach podzbioru uczącego wyznaczono zestaw wag, które następnie aplikowano w formule średniej ważonej do wartości oszacowań wyznaczonych przez poszczególne podmodele (dokładny sposób aplikacji wag znajduje się w następnym podrozdziale). Ostateczny wektor wag modelu bazowego wyznaczony został w następujący sposób:

$$w_U = [0 \quad 2 \quad 1 \quad 16 \quad 4 \quad 0 \quad 1 \quad 0]^T \qquad 5.57$$

Wagi wyznaczono w sposób eksperymentalny, w usystematyzowany sposób podstawiając wartości z zakresu od 0 do 20 do formuły wyznaczającej wartość współczynnika predykcji *PRED*(25%) dla średniej ważonej oszacowań uzyskanych ze wszystkich podmodeli. Celem eksperymentu było wyznaczenie takich wag, które maksymalizują wartość współczynnika predykcji dla średniej ważonej oszacowań.

5.5 OPIS METODY

Dokonywanie oszacowań rozmiarów programów z wykorzystaniem proponowanej metody estymacji składa się z następujących kroków:

- kalibracja modelu szacującego na podstawie dostępnych danych historycznych
- wyznaczenie metryk klas oprogramowania będącego przedmiotem zainteresowania
- podstawienie uzyskanych wartości metryk do modelu szacującego w celu uzyskania ostatecznego oszacowania rozmiaru badanego programu

Kalibracja polega na dostosowaniu istniejącego modelu szacującego do konkretnego środowiska wytwarzania oprogramowania (ISPA, 2007). Proces ten zwiększa szanse na uzyskiwanie dokładniejszych oszacowań, bo bierze pod uwagę historyczne doświadczenia realizacji programów. Na kalibrację modelu szacującego będącego częścią proponowanej metody składają się następujące kroki:

- dla każdego zrealizowanego programu wyznaczyć metryki oraz oszacowania rozmiaru za pomocą każdego z 8 podmodeli wchodzących w skład modelu szacującego
- dla uzyskanych wyników wyznaczyć wagi w ten sposób, aby wskaźniki ocenowe (np. *MMRE* lub *PRED(25%)*) dla średnich ważonych oszacowań uzyskanych z podmodeli dla wszystkich programów używanych w kalibracji dążyła do wartości optymalnych

Sposób wyznaczania metryk klas wchodzących w skład programów zależy od postaci w jakiej dostarczony jest opis klas. Zwykle jest to diagram klas zapisany w formacie XMI, co pozwala na stosunkowo łatwe jego przetwarzanie.

Opis klas wchodzących w skład szacowanego oprogramowania zostaje zapisany za pomocą następującej macierzy:

$$\boldsymbol{C}_{(I\times J)} = \begin{bmatrix} 1 & c_{1(1)} & c_{2(1)} & \cdots & c_{J(1)} \\ 1 & c_{1(2)} & c_{2(2)} & \cdots & c_{J(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_{1(I)} & c_{2(I)} & \cdots & c_{J(I)} \end{bmatrix}_{(I\times(J+1))} \tag{5.58}$$

gdzie kolejne wiersze opisują *I* klasy poprzez wartość 1 oraz wartości *J* metryk:

$$\boldsymbol{c}_{(i)}^{T} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{J-1} \end{bmatrix}_{(J\times 1)} \tag{5.59}$$

Podmodele wchodzące w skład modelu szacującego zapisane są w postaci następującej macierzy:

$$\boldsymbol{M}_{(J\times K)} = \begin{bmatrix} m_{1(1)} & m_{2(1)} & \cdots & m_{K(1)} \\ m_{1(2)} & m_{2(2)} & \cdots & m_{K(2)} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1(J)} & m_{2(J)} & \cdots & m_{K(J)} \end{bmatrix}_{(J\times K)} \tag{5.60}$$

gdzie kolejne kolumny opisują współczynniki *K* równań liniowych podmodeli dla wszystkich *J* metryk klas:

$$\boldsymbol{m}_{(k)} = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_J \end{bmatrix}_{(J\times 1)} \tag{5.61}$$

Wpływ oszacowań poszczególnych podmodeli na końcowe oszacowanie całego modelu jest ustalony za pomocą wektora wag:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_K \end{bmatrix}_{(K\times 1)} \tag{5.62}$$

Oszacowanie wszystkich *I* klas wchodzących w skład oprogramowania jest realizowane w następujący sposób:

$$\boldsymbol{e}_{(I\times 1)} = \boldsymbol{C}_{(I\times J)}\boldsymbol{M}_{(J\times K)}\boldsymbol{w}_{(K\times 1)} \quad 5.63$$

Ostateczne oszacowanie rozmiaru całego programu następuje poprzez zsumowanie uzyskanych oszacowań cząstkowych rozmiarów klas:

$$estimatedSize = \sum_{i=1}^{I} \boldsymbol{e}_i \quad 5.64$$

5.6 OCENA SKUTECZNOŚCI METODY W OPARCIU O MODEL BAZOWY

Dla potwierdzenia skuteczności proponowanej metody estymacji przeprowadzono eksperyment, w ramach którego wykorzystano model bazowy do oszacowania rozmiarów programów wchodzących w skład podzbioru testowego. Dodatkowo, aby uwiarygodnić wyniki eksperymentu podjęto próbę sprawdzenia, czy podzbiór testowy jest reprezentatywną próbką populacji generalnej. Dokonano tego poprzez sprawdzenie losowości podzbioru testowego za pomocą testu serii Stevensa.

Każdy pojedynczy przypadek z 65-elementowego zbioru testowego jest opisany za pomocą 115 zmiennych. Dla każdej zmiennej przeprowadzono test serii Stevensa, stawiając hipotezę zerową H_0 (dobór elementów próby jest losowy) oraz hipotezę alternatywną H_1 (dobór elementów próby nie jest losowy). Opracowane wyniki testu zostały zaprezentowane poniżej.

Tabela 26 Wyniki testu serii dla elementów podzbioru testującego

zmienna	statystyka *z*	wartość *p*	H_0 odrzucona?
classSize	-1,42	0,16	brak podstaw
methods	-0,39	0,70	brak podstaw
attributes	-0,91	0,37	brak podstaw
extends	-1,94	0,05	brak podstaw
implements	-0,91	0,37	brak podstaw
attCpl_0	-0,91	0,37	brak podstaw
attCpl_1	0,09	0,93	brak podstaw
attSim_0	-1,94	0,05	brak podstaw
attSim_1	-0,52	0,60	brak podstaw
attCplConst_0	-0,39	0,70	brak podstaw
attCplConst_1	-0,59	0,56	brak podstaw
attSimConst_0	-0,91	0,37	brak podstaw
attSimConst_1	1,00	0,32	brak podstaw
attCplSta_0	-0,39	0,70	brak podstaw

zmienna	statystyka *z*	wartość *p*	H_0 odrzucona?
pCplConst_SETTER_1	1,00	0,32	brak podstaw
pSimConst_SETTER_0	1,00	0,32	brak podstaw
pSimConst_SETTER_1	1,00	0,32	brak podstaw
pCpl_SETTER_STATIC_0	-1,44	0,15	brak podstaw
pCpl_SETTER_STATIC_1	1,00	0,32	brak podstaw
pSim_SETTER_STATIC_0	1,79	0,07	brak podstaw
pSim_SETTER_STATIC_1	1,00	0,32	brak podstaw
pCplConst_SETTER_STATIC_0	1,00	0,32	brak podstaw
pSimConst_SETTER_STATIC_0	1,00	0,32	brak podstaw
pCpl_OTHER_0	-0,91	0,37	brak podstaw
pCpl_OTHER_1	-1,42	0,16	brak podstaw
pSim_OTHER_0	0,64	0,52	brak podstaw
pSim_OTHER_1	-1,87	0,06	brak podstaw
pCplConst_OTHER_0	-0,39	0,70	brak podstaw

zmienna	statystyka *z*	wartość *p*	H_0 odrzucona?
attCplSta_1	-0,59	0,55	brak podstaw
attSimSta_0	-0,46	0,64	brak podstaw
attSimSta_1	1,00	0,32	brak podstaw
attCplStaConst_0	0,64	0,52	brak podstaw
attCplStaConst_1	-1,31	0,19	brak podstaw
attSimStaConst_0	-1,94	0,05	brak podstaw
attSimStaConst_1	0,13	0,90	brak podstaw
pCpl_ABSTRACT_0	-1,21	0,23	brak podstaw
pCpl_ABSTRACT_1	-0,26	0,79	brak podstaw
pSim_ABSTRACT_0	-1,42	0,16	brak podstaw
pSim_ABSTRACT_1	0,02	0,98	brak podstaw
pCplConst_ABSTRACT_0	1,00	0,32	brak podstaw
pCplConst_ABSTRACT_1	1,00	0,32	brak podstaw
pSimConst_ABSTRACT_0	1,00	0,32	brak podstaw
pSimConst_ABSTRACT_1	1,00	0,32	brak podstaw
pCpl_CONSTRUCTOR_0	-0,67	0,50	brak podstaw
pCpl_CONSTRUCTOR_1	0,13	0,90	brak podstaw
pSim_CONSTRUCTOR_0	-0,39	0,70	brak podstaw
pSim_CONSTRUCTOR_1	0,40	0,69	brak podstaw
pCplConst_CONSTRUCTOR_0	0,69	0,49	brak podstaw
pCplConst_CONSTRUCTOR_1	1,00	0,32	brak podstaw
pSimConst_CONSTRUCTOR_0	1,00	0,32	brak podstaw
pSimConst_CONSTRUCTOR_1	1,00	0,32	brak podstaw
pCpl_GETTER_0	-1,42	0,16	brak podstaw
pCpl_GETTER_1	-1,87	0,06	brak podstaw
pSim_GETTER_0	0,93	0,35	brak podstaw
pSim_GETTER_1	1,00	0,32	brak podstaw
pCplConst_GETTER_0	1,00	0,32	brak podstaw
pCplConst_GETTER_1	1,00	0,32	brak podstaw
pSimConst_GETTER_0	1,00	0,32	brak podstaw
pSimConst_GETTER_1	1,00	0,32	brak podstaw
pCpl_GETTER_STATIC_0	0,13	0,90	brak podstaw
pCpl_GETTER_STATIC_1	1,00	0,32	brak podstaw
pSim_GETTER_STATIC_0	1,22	0,22	brak podstaw
pSim_GETTER_STATIC_1	1,00	0,32	brak podstaw
pCplConst_GETTER_STATIC_0	1,00	0,32	brak podstaw
pCplConst_GETTER_STATIC_1	1,00	0,32	brak podstaw
pSimConst_GETTER_STATIC_0	1,00	0,32	brak podstaw
pSimConst_GETTER_STATIC_1	1,00	0,32	brak podstaw
pCpl_SETTER_0	-1,42	0,16	brak podstaw
pCpl_SETTER_1	-1,87	0,06	brak podstaw
pSim_SETTER_0	-0,91	0,37	brak podstaw
pSim_SETTER_1	-0,46	0,64	brak podstaw
pCplConst_SETTER_0	1,00	0,32	brak podstaw

zmienna	statystyka *z*	wartość *p*	H_0 odrzucona?
pCplConst_OTHER_1	1,00	0,32	brak podstaw
pSimConst_OTHER_0	0,39	0,69	brak podstaw
pSimConst_OTHER_1	1,00	0,32	brak podstaw
pCpl_OTHER_STATIC_0	-1,42	0,16	brak podstaw
pCpl_OTHER_STATIC_1	-0,14	0,89	brak podstaw
pSim_OTHER_STATIC_0	-1,68	0,09	brak podstaw
pSim_OTHER_STATIC_1	0,95	0,34	brak podstaw
pCplConst_OTHER_STATIC_0	1,04	0,30	brak podstaw
pCplConst_OTHER_STATIC_1	1,00	0,32	brak podstaw
pSimConst_OTHER_STATIC_0	1,00	0,32	brak podstaw
pSimConst_OTHER_STATIC_1	1,00	0,32	brak podstaw
rCpl_ABSTRACT_0	-1,53	0,13	brak podstaw
rCpl_ABSTRACT_1	0,49	0,62	brak podstaw
rSim_ABSTRACT_0	-0,91	0,37	brak podstaw
rSim_ABSTRACT_1	-0,26	0,79	brak podstaw
rCpl_CONSTRUCTOR_0	-0,97	0,33	brak podstaw
rCpl_CONSTRUCTOR_1	1,00	0,32	brak podstaw
rSim_CONSTRUCTOR_0	1,00	0,32	brak podstaw
rSim_CONSTRUCTOR_1	1,00	0,32	brak podstaw
rCpl_GETTER_0	-0,91	0,37	brak podstaw
rCpl_GETTER_1	-0,98	0,33	brak podstaw
rSim_GETTER_0	-0,91	0,37	brak podstaw
rSim_GETTER_1	-0,49	0,62	brak podstaw
rCpl_GETTER_STATIC_0	0,13	0,90	brak podstaw
rCpl_GETTER_STATIC_1	1,83	0,07	brak podstaw
rSim_GETTER_STATIC_0	0,67	0,50	brak podstaw
rSim_GETTER_STATIC_1	1,00	0,32	brak podstaw
rCpl_SETTER_0	-1,21	0,23	brak podstaw
rCpl_SETTER_1	1,00	0,32	brak podstaw
rSim_SETTER_0	0,05	0,96	brak podstaw
rSim_SETTER_1	1,00	0,32	brak podstaw
rCpl_SETTER_STATIC_0	1,00	0,32	brak podstaw
rCpl_SETTER_STATIC_1	1,00	0,32	brak podstaw
rSim_SETTER_STATIC_0	1,00	0,32	brak podstaw
rSim_SETTER_STATIC_1	1,00	0,32	brak podstaw
rCpl_OTHER_0	-1,94	0,05	brak podstaw
rCpl_OTHER_1	-1,53	0,13	brak podstaw
rSim_OTHER_0	-1,42	0,16	brak podstaw
rSim_OTHER_1	-0,22	0,83	brak podstaw
rCpl_OTHER_STATIC_0	-1,94	0,05	brak podstaw
rCpl_OTHER_STATIC_1	-1,87	0,06	brak podstaw
rSim_OTHER_STATIC_0	-0,94	0,35	brak podstaw
rSim_OTHER_STATIC_1	0,13	0,90	brak podstaw

Źródło: opracowanie własne

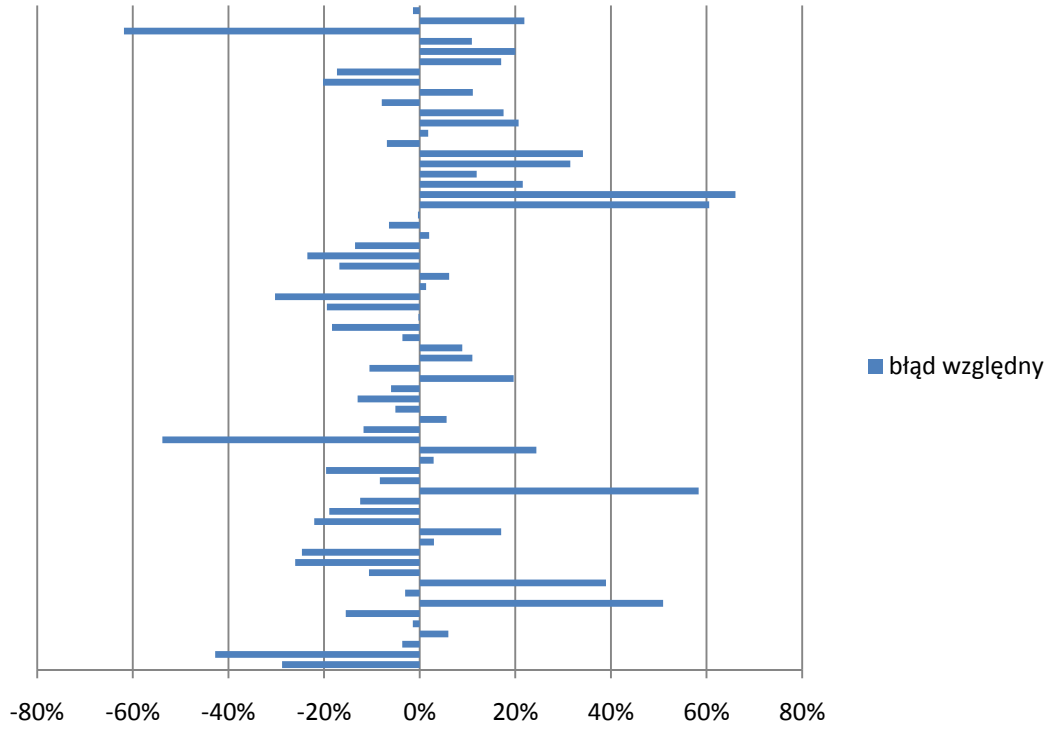
Wyznaczone wartości statystyk *z* znajdują się poza obszarami krytycznymi, co oznacza, że dla każdej zmiennej na poziomie istotności 0,05 brak jest podstaw do odrzucenia hipotezy zerowej o losowym doborze elementów próby eksperymentalnej.

Wyniki eksperymentu szacowania elementów podzbioru testowego za pomocą modelu bazowego zostały zaprezentowane poniżej.

Tabela 27 Rezultat szacowania podzbioru testowego za pomocą modelu bazowego

system	rozmiar rzeczywisty [leksem]	rozmiar oszacowany [leksem]	błąd względny	system	rozmiar rzeczywisty [leksem]	rozmiar oszacowany [leksem]	błąd względny
exist_111	1 151 949	820 560	-29%	findbugs_131	594 004	592 431	0%
aoi_251	823 454	471 304	-43%	freemind_080	434 113	349 890	-19%
jython_21	681 015	656 172	-4%	borg_161	247 382	172 512	-30%
nice_0913	234 114	248 121	6%	sdedit_212	79 534	80 600	1%
cobra_0974	237 191	233 744	-1%	testng_57	129 997	138 000	6%
antelope_341	173 161	146 389	-15%	barcode4j_20	85 211	70 886	-17%
aopmetrics_400	18 139	27 373	51%	umleditor_310	507 587	388 397	-23%
drjava_20070524	513 061	497 363	-3%	plandora_100	174 645	151 008	-14%
ermodeller_192	109 857	152 649	39%	jeti_077	415 037	423 203	2%
ftp4che_14	29 166	26 068	-11%	xholon_07	304 479	284 928	-6%
itext_204	568 781	420 652	-26%	smith_13	194 685	193 979	0%
jfreechart_106	458 258	345 357	-25%	jparsec_12	58 533	93 969	61%
jftp_120	51 496	53 032	3%	jaskell_10	119 944	199 161	66%
jgap_32	116 880	136 805	17%	xfire_126	337 495	410 209	22%
jmt_09	99 858	77 843	-22%	jaxen_111	107 820	120 662	12%
jped_13	294 790	239 051	-19%	plexus_10	575 298	756 578	32%
jpox_118	15 608	13 664	-12%	tapestry_507	339 867	455 914	34%
prism_010	12 982	20 558	58%	jackrabbit_133	758 218	706 291	-7%
rapidminer_40	788 710	722 986	-8%	geronimo_202	1 148 184	1 168 509	2%
smallsql_018	115 987	93 294	-20%	turbine_232	214 805	259 233	21%
telnetd_20	29 485	30 341	3%	activemq_500	864 302	1 016 026	18%
tinyuml_025	61 114	76 026	24%	jgroups_262	337 631	310 867	-8%
umlet_71	90 511	41 788	-54%	enhydra_651	297 522	330 629	11%
apacheds_400	812 786	717 447	-12%	datacrow_3130	337 128	268 998	-20%
cayenne_203	528 442	558 116	6%	openswing_149	293 639	242 881	-17%
james_231	213 418	202 558	-5%	squirrel_264	293 179	343 096	17%
lucene_210	286 389	249 184	-13%	jalisto_10	120 905	145 116	20%
xalanj_270	790 714	743 389	-6%	beehive_102	479 829	532 232	11%
log4j_128	82 868	99 149	20%	lenya_20	4 741 204	1 808 387	-62%
maven_207	182 205	163 033	-11%	ode_111	359 530	438 269	22%
myfaces_115	211 413	234 685	11%	jetspeed_213	684 456	674 802	-1%
xmlsecurity_141	132 764	144 599	9%	**średnia MRE (MMRE)**			**18,29%**
areca_554	228 647	220 423	-4%	**odchylenie standardowe MRE**			**16,21%**
robocode_151	171 865	140 318	-18%	**PRED(25%)**			**80,00%**

Źródło: opracowanie własne



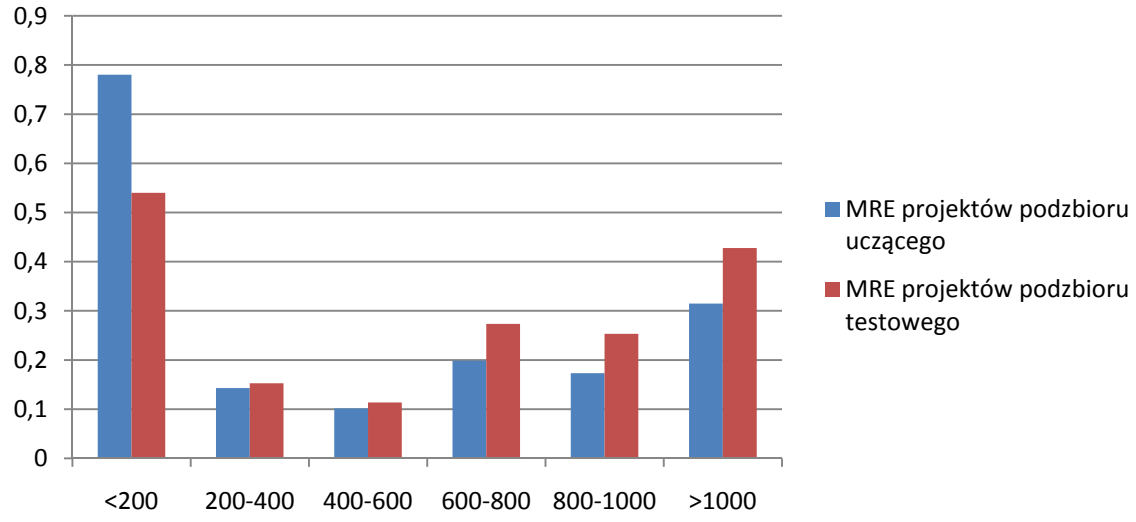
Rysunek 12 Rezultaty oszacowania rozmiarów programów z podzbioru testowego za pomocą modelu bazowego (opracowanie własne)

Wyniki uzyskane podczas eksperymentu potwierdzają skuteczność proponowanej metody estymacji działającej w oparciu o model bazowy. Oceniając wyniki pod kątem najczęściej stosowanych wskaźników ocenowych nie ma wątpliwości, że metoda jest skuteczna, osiągając wyniki w ramach powszechnie uznanych granic: średniej modułu błędu względnego *MMRE* poniżej 25% oraz współczynnika predykcji *PRED(25%)* powyżej 75% (Conte, Dunsmore, & Shen, 1986).

Oprócz potwierdzenia skuteczności metody uzyskane wyniki pokazują, że proponowana metoda nie jest idealna: w niektórych przypadkach uzyskany błąd względny estymacji wynosi nawet powyżej 50%. Jest to wynikiem tego, że nawet pomimo stosunkowo późnej fazy rozwoju oprogramowania (późny etap projektowania), w której aplikowana jest proponowana metoda estymacji, to informacje na temat projektu ciągle są niepełne. O ile znane są ogólne informacje na temat klas, atrybutów i metod to ciągle brakuje elementu, który wnosi najwięcej zmienności: implementacji algorytmów w metodach. Złożoność algorytmów przetwarzania jest wspólną bolączką wszystkich metod estymacji: przykładowo, w specyfikacji metody punktów funkcyjnych znajduje się uwaga, że nie nadaje się ona

do estymacji rozmiaru programów cechujących się dużą złożonością algorytmów przetwarzania danych.

W związku z zaobserwowanym problemem podjęto dodatkowe badania mające na celu stworzenie zbioru zasad, których stosowanie sprawi, że wyniki oszacowań będą bliższe rzeczywistym. Jak już wcześniej wspominano, podstawowym czynnikiem odchylającym oszacowania jest złożoność zaimplementowanych algorytmów. Podając zasady implementacji algorytmów w metodach można w istotny sposób poprawić skuteczność metody. W tym celu zobrazowano zależność uzyskiwanych rezultatów oszacowań modelu bazowego od średniego łącznego rozmiaru metod w klasie dla każdego badanego projektu informatycznego. Uzyskane wyniki zaprezentowano poniżej.



Rysunek 13 Zależność modułu błędu względnego (MRE) projektów od średniego łącznego rozmiaru klasy w projekcie (opracowanie własne)

Analizując wyniki łatwo zauważyć, że najlepsze uzyskane wartości błędu oszacowań dotyczą tych projektów, dla których średni łączny rozmiar metod w klasie pozostaje w zakresie od 200 do 600 leksemów. Co ciekawe, w zakresie tym znajduje się większość z badanych projektów informatycznych (76% elementów podzbioru uczącego oraz 80% podzbioru testowego). Jeżeli by ograniczyć podzbiór testowy do tych projektów, które spełniają warunek pozostawania średniego łącznego rozmiaru metod w klasie w zakresie od 200 do 600 leksemów to uzyskane rezultaty szacowania przy pomocy modelu bazowego poprawiają się:

- średnia MRE z 18,29% na 13,24%
- odchylenie standardowe MRE z 16,21% na 11,15%
- wskaźnik PRED(25%) z 80% na 92,31%

Podsumowując uzyskane powyżej rezultaty można stwierdzić, że zastosowanie wytycznej o ograniczaniu łącznego rozmiaru metod w klasach do zakresu 200-600 leksemów w istotny sposób poprawia uzyskiwane oszacowania.

Innym, bardziej gruntownym sposobem usprawnienia proponowanej metody estymacji w celu zwiększenia dokładności uzyskiwanych oszacowań jest odpowiednie jej przystosowanie do konkretnych warunków środowiska rozwoju oprogramowania, czyli kalibrację metody. Kalibracja bazuje na danych historycznych uzyskanych na podstawie wcześniej zrealizowanych przedsięwzięć informatycznych. Mając do dyspozycji tego typu dane warto zwrócić uwagę na samą charakterystykę realizowanych programów. Planując rozwój określonego typu projektu informatycznego warto kalibrować proponowaną metodę estymacji w oparciu o pulę już zrealizowanych projektów informatycznych cechujących się podobieństwem zarówno funkcjonalnym, jak i strukturalnym.

5.7 POWIĄZANIA Z INNYMI METODAMI

Opracowując jakiekolwiek nowe rozwiązanie dotyczące dowolnej dziedziny życia warto zrobić to tak, aby można je było porównywać z dotychczas istniejącymi. Analogicznie jest w przypadku metod estymacji. Konfrontacja z istniejącymi rozwiązaniami pozwala na ocenę konkretnej metody na tle innych. Metody estymacji porównuje się najczęściej pod kątem uzyskiwanych przez nie wyników. W ogólności jest to trudne zadanie, ze względu na różnorodność istniejących metod estymacji, a w szczególności metryk w jakich wyrażane są oszacowania. Przykładowo, trudno jest porównywać oszacowania wyrażane w punktach funkcyjnych metody *IFPUG* i jednostkach *Cfsu* wyznaczonych za pomocą metody *COSMIC-FFP*, mimo że metody są bardzo do siebie podobne (obie służą do szacowania rozmiaru oprogramowania w aspekcie funkcjonalnym).

Problem konwersji oszacowań wyznaczanych przez metody estymacji pojawił się wraz z opracowaniem metody punktów funkcyjnych *IFPUG*. Metoda ta operowała na zupełnie nowej kategorii metryk w porównaniu do najczęściej wówczas używanej metryki, jaką była liczba linii kodu źródłowego. Aby móc wykorzystywać oszacowania rozmiaru

wyrażane w punktach funkcyjnych w metodach estymacji nakładu pracy (np. wczesne wersje metody *COCOMO*) opracowano tabele, które pozwalały przeliczać punkty funkcyjne na linie kodu źródłowego określonego języka programowania.

Tabela 28 Tabela konwersji linii kodu źródłowego na punkty funkcyjne

język	SLOC/FP			
	średnia	mediana	minimum	maksimum
Ada	154		104	205
ASP	69	62	32	127
Assembler	172	157	86	320
C	148	104	9	704
C++	60	53	29	178
C#	59	59	51	66
COBOL	73	77	8	400
HTML	43	42	35	53
J2EE	61	50	50	100
Java	60	59	14	97
JavaScript	56	54	44	65
JSP	59			
Natural	60	52	22	141
Perl	60			
PL/1	59	58	22	92
PL/SQL	46	31	14	110
REXX	67			
Smalltalk	35	32	17	55
SQL	39	35	15	143
VBScript	45	34	27	50
VisualBasic	50	42	14	276

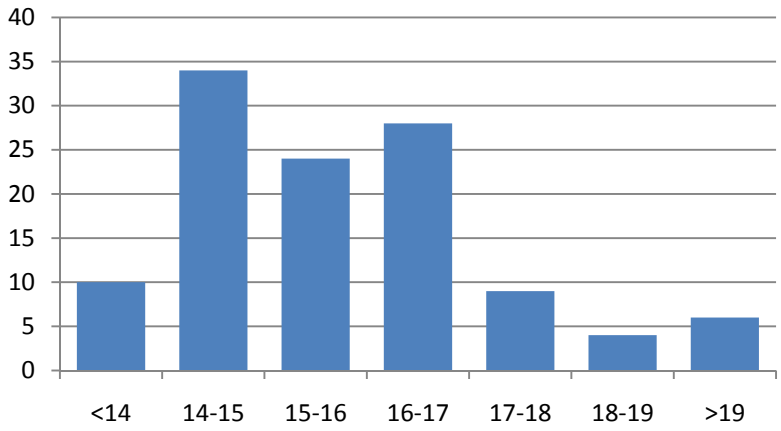
Źródło: (Quantitative Software Management, 2005)

Przedstawiona wyżej tabela dostarcza informacji ile średnio linii kodu źródłowego w danym języku programowania jest wymagane, aby zaimplementować ilość funkcjonalności wyrażoną w jednym punkcie funkcyjnym. Do przedstawionych współczynników należy podchodzić z dużą ostrożnością przede wszystkim ze względu na to, że dla każdego języka potencjalnie mogła być użyta zupełnie inna definicja linii kodu źródłowego.

Również w przypadku metody szacowania proponowanej w niniejszej pracy istnieje potrzeba opracowania sposobu na porównywanie uzyskiwanych oszacowań z rezultatami działania innych metod. Główną przyczyną jest zastosowana metryka, jaką w przypadku proponowanej metody jest leksem.

W procesie analizy całego zbioru danych (podzbiory uczący i testowy) dla każdego projektu wyznaczono średnią liczbę leksemów wchodzących w skład pojedynczej logicznej linii kodu. Logiczna linia kodu została zdefiniowana jako obecność w tekście programu (ale nie w komentarzu) znaków średnika (**;**) lub klamry zamykającej (**}**), czyli znaków używanych w języku Java do oznaczania końca wyrażenia i grupy wyrażeń. Uzyskano następujące rezultaty:

- średnia ilość leksemów w pojedynczej linii kodu źródłowego języka Java dla programu: 16,07
- odchylenie standardowe ilości leksemów w pojedynczej linii kodu źródłowego języka Java dla programu: 3,40



Rysunek 14 Histogram średnich długości linii kodu źródłowego w języku Java w badanych programach (opracowanie własne)

5.8 ZASTOSOWANIA METODY

Jak już wcześniej wspomniano, użyteczność proponowanej metody estymacji ogranicza się do etapów projektowania i implementacji rozwiązań informatycznych rozwijanych w oparciu o paradygmat obiektowy. Zgodnie z założeniami co do potrzeby ciągłości procesu estymacji wymaganych w ramach modelu CMM-I (CMMI Product Team, 2006) proponowana metoda może stanowić skuteczne uzupełnienie innych metod estymacji, a w szczególności tych, które można stosować na wcześniejszych etapach rozwoju oprogramowania. Przykładowe rozwiązanie szacujące mogłoby opierać się na metodzie punktów funkcyjnych podczas faz specyfikacji i wczesnej analizy, metodzie punktów przypadków

użycia na etapach późnej analizy i wczesnego projektowania i proponowanej metodzie estymacji na etapach późnego projektowania i implementacji. Takie rozwiązanie estymacyjne zapewniłoby zwiększenie prawdopodobieństwa uzyskania skutecznego oszacowania ze względu na obecność następujących czynników:

- wielości perspektyw, w których oceniane jest realizowane oprogramowanie
- stopniowego uszczegóławiania oszacowań, będącego wynikiem zwiększania wiedzy o realizowanym oprogramowaniu

Dużą zaletą proponowanej metody jest jej pełna (oprócz kalibracji metody) automatyzacja. Cecha ta pozwala na jej łatwiejsze wdrożenie w środowisku rozwoju oprogramowania. Metodę można by stosunkowo łatwym kosztem przystosować do pracy w konkretnym edytorze UML, na przykład jako zewnętrzna wtyczka (ang. *plugin*). Pozwoliłoby to dokonywanie oszacowań automatycznie w trakcie projektowania klas w edytorze UML. Innym sposobem na integrację proponowanej metody z konkretnym środowiskiem rozwoju oprogramowania może być zamknięcie metody w postaci zewnętrznej aplikacji, która dokonywałaby oszacowań przetwarzając opis projektu UML systemu informatycznego zapisanego w formacie XMI, który bazuje na standardzie XML. Eksport do formatu XMI jest zaimplementowany praktycznie w każdym istniejącym edytorze UML.

Proponowana metoda może być szczególnie przydatna w przypadku programów, których rozwój oparty jest na modelu iteracyjnym. W takim podejściu oprogramowanie jest tworzone w kolejnych etapach (iteracjach), na które składają się wszystkie fazy rozwoju oprogramowania (specyfikacja, analiza, projektowanie, implementacja, testowanie). Iteracje te są stosunkowo krótkie (np. 2 do 4 tygodni w metodyce *SCRUM* należącej do grupy metodyk „zwinnych"), co sprawia, że implementowana funkcjonalność jest stosunkowo nieduża. Estymację przeprowadza się zwykle na początku każdej iteracji, ale powinno się ją powtarzać także w trakcie jej trwania, aby na bieżąco modyfikować plan w zależności od bieżącej sytuacji. Stosunkowo niewielka ilość implementowanej funkcjonalności w ramach pojedynczej iteracji może sprawić, że proponowana metoda może znaleźć zastosowanie bardzo wcześnie, nawet podczas dokonywania początkowej estymacji iteracji. Ponadto możliwość automatyzacji procesu estymacji znakomicie wpisuje się w idee postulowane w ramach „zwinnego" (ang. *agile*) podejścia do rozwoju oprogramowania, zaraz obok ciągłej integracji (ang. *continuous integration*) i automatycznego testowania.

Proponowana metoda może również znaleźć zastosowanie w przypadku zarządzania niewielkimi grupami programistów. Typowe zadania takich grup sprowadzają się zwykle do projektowania i implementacji określonych wcześniej cech realizowanego oprogramowania, co sprawia, że proponowana metoda (jako przeznaczona do stosowania w ramach tych właśnie etapów) mogłaby być stosowana w szerokim zakresie. W grupach programistycznych wyznaczona zostaje zwykle jedna osoba, pełniąca rolę lidera grupy. Lider jest odpowiedzialny za kierowanie pracami realizowanymi przez taką grupę i często w ramach swoich zadań dostarcza oszacowań zakresu zadań implementacyjnych. Korzystając z proponowanej metody lider zespołu dokonując samodzielnie oszacowań może kontrolować tempo i przebieg prac realizowanych przez jego zespół, dostarczając systematycznie kierownictwu informacji o stanie zaawansowania implementacji.

6 PODSUMOWANIE

Podsumowując całość zaprezentowanych wyżej wyników należy zaznaczyć, że wszystkie realizowane zadania zmierzały do osiągnięcia celu postawionego w pierwszym rozdziale niniejszej dysertacji. Jak pokazano, estymacja jest istotnym czynnikiem zwiększającym prawdopodobieństwo zakończenia sukcesem przedsięwzięcia realizacji oprogramowania. Szacowanie jest integralnym elementem modelu CMM-I (rozdział 2.3.2) począwszy od jego najbardziej uproszczonej postaci i powinno być realizowane w sposób ciągły na wszystkich etapach rozwoju przedsięwzięcia realizacji oprogramowania. Szczególnie istotne jest szacowanie rozmiaru projektu ze względu na jego znaczenie dla rozwiązań informatycznych: jest on podstawową cechą, która niejako łączy wszystkie sposoby rozumienia zakresu czynności, jakie realizowane oprogramowanie ma wykonywać oraz wszystkie przejawy istnienia, pod jakimi ma ono być obserwowane. Tak szerokie rozumienie koncepcji rozmiaru sprawia, że jego wartość wpływa na inne cechy realizowanego produktu oraz samego procesu wytwórczego. Wynika stąd, że estymacja dowolnych istotnych zasobów procesu wytwarzania oprogramowania jest pochodną estymacji rozmiaru. Przykładowo, metoda *COCOMO*, będąca jedną z najbardziej rozpowszechnionych metod estymacji kosztów jako wejście przyjmuje oszacowanie rozmiaru oprogramowania wyrażone w ilości linii kodu źródłowego bądź (w nowszych odsłonach) w ilości punktów funkcyjnych.

Istniejące metody estymacji różnią się przede wszystkim okresem w cyklu życia oprogramowania (rozdział 2.4) kiedy mogą być stosowane. Fakt ten jest bezpośrednią konsekwencją charakterystyki procesu wytwarzania, który jest procesem stopniowego uszczegóławiania. Postęp prac nad realizacją oprogramowania wzbogaca wiedzę na jego temat, co prowadzi do zwiększenia skuteczności metod estymacji. Czas stosowania metody szacowania wpływa również na jednostki używane do wyrażania rozmiaru oprogramowania: na wczesnych etapach rozwoju oprogramowania metody szacowania stosują zwykle bardziej abstrakcyjne jednostki (np. punkty funkcyjne). W metodach stosowanych w ramach późniejszych etapów używane są zwykle jednostki bliższe fizycznej stronie produktu (np. ilość linii kodu źródłowego w konkretnym języku oprogramowania).

Tworząc metodę estymacji zawsze należy rozpatrywać w oparciu o charakterystykę danych na podstawie których będzie ona działać. Przewidywania rozmiaru również muszą

się opierać o pewne dostępne fakty opisujące realizowany produkt. Jakość tych faktów jest ściśle powiązana z wiedzą o realizowanym oprogramowaniu, która, jak już wielokrotnie wspomniano, rośnie wraz z postępem prac wytwórczych. Rzutuje ona także na wyniki uzyskiwane przez metodę estymacji. Dane dostępne na początkowych etapach procesu są nielicznie i rozmyte i są stopniowo rozszerzane i wyostrzane w procesie rozwoju. Na etapach projektowania są już na tyle bogate, że można je wydajnie stosować do estymacji. Przykładem takich danych mogą być diagramy UML (rozdział 3), służące do opisu oprogramowania realizowanego przy pomocy paradygmatu obiektowego. Stanowią one doskonałą informację o różnych aspektach realizowanego oprogramowania z perspektywy strukturalnej i behawioralnej.

Analiza istniejących rozwiązań estymacyjnych rozmiaru (rozdział 4) pokazała, że na chwilę obecną brakuje metod szacowania dla późniejszych etapów procesu wytwarzania, które, opierając się na danych uzyskanych z diagramów UML byłyby w stanie w autonomiczny sposób dokonywać estymacji rozmiaru. Większość istniejących metod wykorzystujących diagramy UML dla swojego działania ciągle wymaga interakcji z człowiekiem, którego zadaniem jest zwykle eksperckie ocenianie wielkości pewnych parametrów. I właśnie tutaj pojawia się potrzeba metody, która będąc częścią procesowego podejścia do estymacji realizowałaby zadanie szacowania w sposób jak najbardziej autonomiczny, do minimum redukujący subiektywny wkład oceny dokonywanej przez człowieka.

Proponowana metoda, stanowiąca realizację celu postawionego przez autora niniejszej dysertacji, jest uzupełnieniem istniejących metod estymacji na potrzeby szacowania rozmiaru oprogramowania na etapach projektowania i implementacji. Metoda koncentruje się na koncepcji klasy, stanowiącej jedno z podstawowych pojęć paradygmatu obiektowego. Dane pochodzące z diagramów UML i opisujące klasy projektu informatycznego stanowią wejście dla modelu szacującego będącego centralnym elementem proponowanej metody. Wyjście, czyli wartości oszacowań rozmiaru, jest wyrażane za pomocą liczby użytych leksemów języka programowania (rozdział 5.2). Zastosowanie leksemów zostało podyktowane niską precyzją pojęcia ilości linii kodu źródłowego. Nowatorskim elementem jest z pewnością koncepcja określania kategorii funkcjonalnej algorytmu metody (czyli funkcji powiązanej z klasą obiektu i operującej na właściwościach obiektu) na podstawie jej nazwy oraz wykorzystanie jej w procesie szacowania rozmiaru (rozdział 5.4.2). Model szacujący (rozdział 5.4) został opracowany na podstawie rzeczywistych danych zrealizowanych

programów za pomocą metod klasycznych (regresji wielorakiej). Został on oparty o bogate metryki modelu obiektowego (rozdział 5.4.3), szeroko opisujące różnorakie aspekty struktury klas i wchodzących w ich skład komponentów. Oprócz modelu szacującego opisano także proces kalibracji (rozdział 5.5), który jest jednym z najistotniejszych czynników skutecznego wdrażania metody estymacji do konkretnego środowiska rozwoju oprogramowania.

Proponowane rozwiązanie poddano weryfikacji. Dokonano tego poprzez przetestowanie modelu bazowego (czyli modelu szacującego skalibrowanego na potrzeby środowiska zdefiniowanego poprzez zbiór danych uczących; rozdział 5.4.5) na odrębnym zbiorze danych testowych. Dane testowe, podobnie jak to było w przypadku danych uczących, również pochodziły z rzeczywistych środowisk rozwoju oprogramowania (rozdział 5.3.1). Sama procedura testowania była oparta o zastosowanie współcześnie zalecanych kryteriów ocenowych metod estymacji oraz wartości brzegowych (rozdział 2.7.4). Uzyskane wyniki stanowią dowód potwierdzający słuszność tezy postawionej w rozdziale pierwszym, gdyż średnia wartość uzyskanego błędu mieściła się w założonych granicach. Należy podkreślić, że uzyskany wynik jest akceptowalny nawet pomimo tego, że dane testowe pochodziły w większości z niezwiązanych ze sobą źródeł i stanowiły przykłady aplikacji bardzo różnorodnych funkcjonalności. W przypadku zastosowania metody w konkretnym środowisku rozwoju oprogramowania uzyskane wyniki byłyby najprawdopodobniej jeszcze lepsze.

Oprócz mierzalnych wyników uzyskanych w procesie testowania skuteczności metody autor wskazał również inne, nie mniej istotne zalety. Taką cechą jest przede wszystkim niemal pełna automatyzacja metody, pozwalająca na stosunkowo łatwą jej implementację i zastosowanie w istniejących edytorach UML. Takie rozwiązanie pozwala na wykorzystywanie proponowanej metody niejako w tle głównych zadań związanych z tworzeniem diagramów UML opisujących realizowane oprogramowanie. Zbliża ono metodę estymacji do jej potencjalnych użytkowników, jakimi mogą być programiści i menedżerowie niższego szczebla, znacznie niwelując koszty związane z nauką nowego sposobu estymacji. Problem nauki nowego podejścia jest bardzo poważny w przypadku metod, w których wykorzystuje się ocenę wartości określonych bytów przez człowieka. Takie metody, nie dość, że wymagają specjalistycznego przeszkolenia to jeszcze ich skuteczne używanie w praktyce musi być wspomagane doświadczeniem w estymacji za pomocą konkretnej metody.

Konsekwencją działań podjętych w ramach niniejszego podsumowania są także perspektywy dalszych badań nad rozważanym problemem. Oprócz wspomnianej już wcześniej implementacji metody w postaci wtyczki do któregoś z edytorów UML, warto by również przeprowadzić dokładniejsze badania w ustalonym środowisku rozwoju oprogramowania. Testy przeprowadzone na potrzeby niniejszej pracy dotyczyły aplikacji pochodzących z różnych źródeł, realizowanych przez różnorodne zespoły i implementowały bardzo różną funkcjonalność. Potwierdziły one skuteczność dla szerokiego spektrum kategorii oprogramowania. Dobrym sposobem na potwierdzenie skuteczności proponowanej metody byłoby przetestowanie jej wykorzystanie w homogenicznym środowisku rozwoju przy realizacji podobnych funkcjonalnie i technologicznie rozwiązań informatycznych. Wysoce prawdopodobne jest, że w takim środowisku metoda uzyskiwałaby jeszcze lepsze rezultaty ze względu na podobieństwo w stylu realizowania różnych produktów przez te same zespoły.

Kolejnym elementem, któremu warto byłoby przyjrzeć się bliżej jest wprowadzenie do metody rozpoznawania w diagramach UML wzorców projektowych (rozdział 3.3). Wzorce projektowe to pewne powtarzające się typowe rozwiązania będące w powszechnym użyciu. Ich rozpoznawanie w diagramach UML wpłynęłoby na usprawnienie uzyskiwanych oszacowań ze względu na fakt wcześniejszej znajomości oczekiwanego rozmiaru przedstawiciela konkretnego typu wzorca projektowego. Pozwoliłoby to na zmniejszenie nieprzewidywalności rozmiaru algorytmów w metodach należących do wzorca klas.

SPIS ILUSTRACJI

SPIS TABEL

BIBLIOGRAFIA

Albrecht, A. (1985). *AD/M Productivity measurement and estimate validation.* IBM Corporate Information Systems and Administration.

Albrecht, A. (1979). Measuring Application Development Productivity. *Proceedings of IBM Applications Development Symposium*, (strony 83-92). Monterey.

Armstrong, D. (2006, 02). The Quarks of Object-Oriented Development. *Communications of the ACM 49 (2)* , strony 123–128.

Basili, V., Caldiera, G., & Rombach, D. (2002). The Goal Question Metric Approach. W *Encyclopedia of Software Engineering.* Hoboken: John Wiley & Sons.

Beck, K. (1997). *Smalltalk Best Practice Patterns.* Upper Saddle River: Prentice Hall.

Beck, K., & Fowler, M. (2000). *Planning Extreme Programming.* Boston: Addison-Wesley Professional.

Boehm, B. (1988, 05). A spiral model of software development and enhancement. *Computer* , 61-72.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., & Selby, R. (1995). Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering* (strony 57-94). Springer Netherlands.

Booch, G. (1994). *Object Oriented Analysis with Applications.* Redwood City: Benjamin/Cummings Publishing Co., Inc.

Briand, L., El Emam, K., Surmann, D., Wieczorek, I., & Maxwell, K. (1999). An Assessment and Comparison of Common Software Cost Estimation Modeling Techniques. *International Conference on Software Engineering* (strony 313-322). Los Alamitos: IEEE Computer Society Press.

Brown, W., Malveau, R., McCormick, H., & Mowbray, T. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* New York: John Wiley & Sons.

Carbone, M., & Santucci, G. (2002). Fast&&Serious: a UML based metric for effort estimation. *Quantitative Approaches in Object-Oriented Software Engineering.* Berlin: Springer Verlag.

Chidamber, S., & Kemerer, C. (1994). A Metric Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* , strony 476-493.

Clever Age sp. z o.o. (2005). *CMM-I, czyli jak kontrolować rozwój oprogramowania*. Pobrano 04 09, 2007 z lokalizacji http://www.clever-age.com.pl/cmmi_jak_kotrolowac_rozwoj_oprogramowania

CMMI Product Team. (2006). *CMMI for Development, Version 1.2.* Pittsburgh: Carnegie Mellon University.

Complak, W., & Bogacki, B. (2006, 09 28). *Podstawy kompilatorów.* Pobrano 06 10, 2007 z lokalizacji http://wazniak.mimuw.edu.pl/index.php?title=Podstawy_kompilator%C3%B3w

Conte, S., Dunsmore, H., & Shen, V. (1986). *Software Engineering Metrics and Models.* Menlo Park: Benjamin Cummings Publishings.

Cooper, J. (2001). *Java: Wzorce projektowe.* Gliwice: Wydawnictwo Helion.

COSMIC. (2003). *COSMIC-FFP Measurement Manual.* The Common Software Measurement International Consortium (COSMIC).

Denning, P. (2000). Computer Science: The Discipline. W A. Ralston, E. Reilly, & D. Hemmendinger, *Encyclopedia of Computer Science.* John Wiley & Sons, Inc.

Dragan, N., Collard, M., & Maletic, J. (2006). Reverse Engineering Method Stereotypes. *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (strony 24-34). Washington: IEEE Computer Society.

Eckel, B. (2003). *Thinking in Java.* Helion SA.

Fenton, N., & Neil, M. (2000). The Future of Software Engineering. W A. Finkelstein, *The Future of Software Engineering* (strony 359-370). Limerick: ACM Press.

Fenton, N., & Pfleeger, S. (1997). *Software Metrics.* Boston: PWS Publishing.

Fenton, N., & Pfleeger, S. (1998). *Software Metrics: A Rigorous Approach.* Boston: Course Technology.

Ferens, D., & Christensen, D. (2000, 04). Does Calibration Improve Predictive Accuracy? *CrossTalk. The Journal of Defense Software Engineering* , strony 14-17.

Foss, T., Stensrud, E., Kitchenham, B., & Myrtveit, I. (2003). A Simulation Study of the Model Evaluation Criterion MMRE. *IEEE Transactions on Software Engineering* , 985-995.

Fowler, M., & Scott, K. (2002). *UML w kropelce.* Warszawa: Oficyna Wydawnicza LTP Sp. z o. o.

Freeman, E., Freeman, E., Sierra, K., & Bates, B. (2005). *Head First Design Pattern.* Gliwice: Wydawnictwo Helion.

Galorath, D., & Evans, M. (2006). *Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves.* New York: Auerbach Publications.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley Professional.

Gibson, D., Goldenson, D., & Kost, K. (2006). *Performance Results of CMMI-Based Process Improvement.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Górski, J., Redmill, F., Fenton, N., Burns, J. T., Ezhilchelvan, P., Jurkowlaniec, M., i inni. (2000). *Inżynieria oprogramowania w projekcie informatycznym.* Warszawa: Wydawnictwo Informatyki MIKOM.

Gręboszc, J. (1996). *Symfonia C++. Programowanie w języku C++ orientowane obiektowo.* Kraków: Oficyna Kallimach.

Halstead, M. (1977). *Elements of Software Science.* New York: Elsevier.

Henderson-Sellers, B., & Gonzalez-Perez, C. (2006). Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0. *Lecture Notes in Computer Science Volume 4199/2006* (strony 16-26). Heidelberg: Springer-Verlag Berlin.

Henry, S., & Kafura, D. (1981). Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering* , strony 510-518.

Humphrey, W. (2000). *The Personal Software Process.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

ISO. (2009). *Quality management principles*. Pobrano 03 29, 2009 z lokalizacji ISO - International Organization for Standardization.

ISPA. (2007). *Parametric Estimating Handbook.* International Society of Parametric Analysts. Vienna: ISPA/SCEA Joint Office.

Jabłonowski, J., & Sroka, J. (2006, 10 4). *Materiały dydaktyczne.* Pobrano 05 20, 2007 z lokalizacji Wydział Matematyki, Informatyki i Mechaniki Uniwersytety Warszawskiego: http://wazniak.mimuw.edu.pl/index.php?title=Programowanie_obiektowe

Jones, C. (1991). *Applied Software Measurement.* New York: McGraw-Hill.

Jones, C. (1994). *Assessment and Control of Software Risks.* Indianapolis: Prentice Hall.

Jones, C. (1986). *The SPR Feature Point Method.* Boston: Software Productivity Research, Inc.

Kitchenham, B. (1997, 03-04). The Problem with Function Points. *IEEE Software* , strony 29-31.

Kitchenham, B., Pfleeger, S., McColl, B., & Eagan, s. (2002, 10 15). An empirical study of maintenance and development estimation accuracy. *Journal of Systems and Software* , strony 57-77.

Kitchenham, B., Pickard, L., MacDonell, S., & Shepperd, M. (2001). What accuracy statistics really measure. *IEE Proceedings - Software* , 81-85.

Laird, L., & Brennan, C. (2006). *Software Measurement and Estimation: A Practical Approach.* Hoboken: John Wiley & Sons, Inc.

Longstreet, D. (2004). *Function Point Analysis Training Course.* Blue Springs: Longstreet Consulting, Inc.

Lother, M., & Dumke, R. (2001). Points Metrics: Comparison and Analysis. *Current Trends in Software Measurement - Proceedings of the International Workshop on Software Measurement* (strony 155-172). Montreal: Shaker Verlag.

McCabe, T. (1976, 12). A Complexity Measure. *IEEE Transactions on Software Engineering* , strony 308-320.

McConnell, S. (2003). *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers.* Addison-Wesley.

McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules.* Redmond: Microsoft Press.

McConnell, S. (2006). *Software Estimation: Demystifying the Black Art.* Redmond: Microsoft Press.

Meyer, B. (1997). *Object-Oriented Software Construction.* Englewood Cliffs: Prentice Hall.

Minkiewicz, A. (2000). *Measuring Object Oriented Software with Predictive Object Points.* PRICE Systems, L.L.C.

Munson, J. (2003). *Software Engineering Measurement.* Boca Raton: CRC Press LLC.

Nawrocki, J. (2006, 11 30). *Materiały dydaktyczne.* Pobrano 05 25, 2007 z lokalizacji Wydział Matematyki, Informatyki i Mechaniki Uniwersytety Warszawskiego: http://wazniak.mimuw.edu.pl/images/7/76/Io-5-wyk.pdf

Open Source Initiative. (2009). *The Open Source Definition.* Pobrano 10 25, 2009 z lokalizacji opensource.org: http://opensource.org/docs/osd

Park, R. (1992). *Software Size Measurement: A Framework for Counting Source Statements.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Parkinson, C. N. (1955, 11). Parkinson's Law: The Pursuit of Progress. *The Economist* .

Paynter, J. (1996). Project Estimation Using ScreenflowEngineering. *Internation Conference on Software Engineering: Education and Practice* (strony 150-159). Los Alamitos: IEEE Computer Society Press.

Persse, J. R. (2006). *Process Improvements Essentials.* Sebastopol: O'Reilly.

Pomeroy-Huff, M., Mullaney, J., Cannon, R., & Sebern, M. (2005). *The Personal Software Process Body of Knowledge.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Quantitative Software Management, I. (2005, 04). *QSM Function Point Programming Languages Table.* Pobrano 05 07, 2009 z lokalizacji QSM: http://www.qsm.com/FPGearing.html

Riehle, D. (2000). Method Types in Java. *Java Report* .

Rosenberg, L., Stapko, R., & Gallo, A. (1999). Applying object-oriented metrics. *6th International Symposium on Software Metrics.*

Royce, W. (1970). Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON* (strony 1-9). San Francisco: IEEE.

Shalloway, A., & Trott, J. (2001). *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Boston: Addison-Wesley Professional.

Stachurski, M. (2005). An analysis of dependence between number of code lines and set of preferred parameters in UML diagrams. *International Multiconference ACS-CISIM.* Ełk.

Stachurski, M. (2004). Badanie zależności ilości linii kodu źródłowego od wybranych parametrów diagramów UML. *Materiały VIII Sesji Naukowej Informatyki.* Szczecin: Politechnika Szczecińska Wydział Informatyki.

Stachurski, M. (2002). Estymacja projektów informatycznych w zarządzaniu procesami wytwarzania oprogramowania. *Materiały VII Sesji Naukowej Informatyki.* Szczecin: Politechnika Szczecińska Wydział Informatyki.

Stachurski, M. (2008). Metoda estymacji długości kodu źródłowego na podstawie diagramów statycznych UML. *Studia i Materiały Polskiego Stowarzyszenia Zarządzania Wiedzą.* Warszawa.

Stachurski, M. (2006). Metoda estymacji długości kodu źródłowego na podstawie diagramów UML. *Rozwój i zastosowania metod ilościowych i technik informatycznych wspomagających procesy decyzyjne.* Warszawa.

Stachurski, M. (2005). Rodzina metod punktów funkcyjnych. *Materiały Ogólnopolskiej Konferencji Naukowej Ryzyko Przedsięwzięć Informatycznych.* Szczecin: Politechnika Szczecińska Wydział Informatyki.

Stellman, A., & Greene, J. (2006). *Applied Software Project Management.* Sebastopol: O'Reilly Media, Inc.

Stepanek, G. (2005). *Software Project Secrets: Why Software Projects Fail.* Apress.

Subieta, K. (2002). Budowa i integracja systemów informacyjnych. Warszawa: Polsko-Japońska Wyższa Szkoła Technik Komputerowych.

Subieta, K. (2001). Projektowanie systemów informacyjnych. Warszawa: Polsko-Japońska Wyższa Szkoła Technik Komputerowych.

Sultanođlu, S., & Karakaţ, Ü. (1998, 10 12). *Complexity Metrics and Models*. Pobrano 06 07, 2007 z lokalizacji http://yunus.hun.edu.tr/~sencer/complexity.html

Sun Microsystems, Inc. (1997, 09 12). *Java Code Conventions.* Pobrano 06 17, 2007 z lokalizacji Sun Developer Network: http://java.sun.com/docs/codeconv/CodeConventions.pdf

Szyjewski, Z. (2001). *Zarządzanie projektami informatycznymi.* Warszawa: Agencja Wydawnicza PLACET.

The Standish Group International, Inc. (2001). Extreme Chaos.

Van Doren, E. (2000). *Cyclomatic Complexity.* Pobrano 06 07, 2007 z lokalizacji Software Engineering Institute: http://www.sei.cmu.edu/activities/str/descriptions/cyclomatic.html

Van Roy, P., & Haridi, S. (2004). *Concepts, Techniques and Models of Computer Programming.* MIT Press.

Welfe, A. (2003). *Ekonometria.* Warszawa: Polskie Wydawnictwo Ekonomiczne.

Wikiquote.org: Heraclitus. (brak daty). Pobrano 04 01, 2007 z lokalizacji Wikiquote.org: http://en.wikiquote.org/wiki/Heraclitus

Zuse, H. (1991). *Software Complexity: Measures and Methods.* Berlin: Walter De Gruyter, Inc.