STUDIA INFORMATICA 2015
Volume 36 Number 3 (121)

Tomasz Lewandowski
Politechnika Śląska, Instytut Informatyki

ZASADY, HEURYSTYKI ORAZ SKAZY W PROJEKTOWANIU ZORIENTOWANYM OBIEKTOWO

Streszczenie. Artykuł omawia różne odmiany dobrych praktyk, które mają zastosowanie w projektowaniu zorientowanym obiektowo – heurystyki, zasady, skazy oraz wzorce. Na tej podstawie przedstawione zostają główne modele jakościowe projektu, tzw. modele FCM (ang. Factor-Criteria-Metric). W pracy zawarte są również metody pomiaru i identyfikacji problemów związanych z wymienionymi praktykami.

Słowa kluczowe: heurystyki obiektowe, zasady projektowe, skazy projektowe, wzorce projektowe, zapachy kodu, projektowanie zorientowane obiektowo, modele jakościowe

SOFTWARE PRINCIPLES, HEURISTICS AND FLAWS IN OBJECT ORIENTED DESIGN

Summary. Article describes different kinds of best practices which are used in the object oriented software design – heuristics, principles, smells and patterns. Based on that, there are presented main project quality models the so-called FCM (Factor-Criteria-Metric) models. Additionally the measurement and problem identification connected with the mentioned best practices is discussed as well.

Keywords: software heuristic, design principles, design flaws, code smells, design pattern, object oriented design, quality models

1. Wstęp

W pierwszej części artykułu przedstawione zostały pojęcia związane z różnymi odmianami dobrych praktyk, które mają zastosowanie w projektowaniu zorientowanym obiektowo – heurystykami, zasadami, skazami oraz wzorcami. Bazują one głównie na doświadczeniu realizacyjnym i mają znaczący wpływ na strukturę kodu. Dodatkowo odnoszą się one do

różnego poziomu abstrakcji, mając pewne ograniczenia związane z ich zastosowaniem i pomiarem. Proces uporządkowania wyżej wymienionych pojęć jest utrudniony przez brak ustandaryzowanych definicji, co często prowadzi do rozbieżności w literaturze omówionej dokładniej w rozdziale 4. Dlatego dla każdej wymienionej w artykule metody została przytoczona definicja, lub, w przypadku gdy nie istnieje, wyprowadzona na podstawie zgrupowanych właściwości. Zgodnie z nowymi badaniami liczba błędów jest jednoznacznie związana z architekturą kodu [1, 2], więc tym bardziej programista powinien umiejętnie wykorzystywać dobre praktyki [3].

Celem drugiej części pracy jest przegląd obecnych metod pomiaru i diagnozowania problemów związanych z projektowaniem oprogramowania. Potencjalne podejścia zostały rozpatrzone pod kątem rozpoznanych wcześniej problemów interpretacyjnych. Następnie przedstawione zostały główne modele jakościowe projektu, tzw. modele FCM (ang. *Factor-Criteria-Metric* [4]). Ich zadaniem jest rozkład i przyporządkowanie odpowiednich charakterystyk jakościowych do właściwości obiektowych, co w końcowej fazie umożliwia łatwiejsze zidentyfikowanie metryk, które je opisują. W przeglądzie zostały również uwzględnione wyznaczniki, jakimi powinien odznaczać się dany model, np. subiektywność oceny, łatwość zrozumienia, łatwość adaptacji.

2. Najważniejsze pojęcia

W rozdziale zaprezentowane są najważniejsze pojęcia związane z dobrymi praktykami w obiektowym projektowaniu oprogramowania, wraz z ich definicją. Ujęte zostały również charakterystyki, jakie można wymienić na podstawie istniejących zbiorów, opartych na praktycznej wiedzy. Każdy rodzaj dobrych praktyk zawiera opracowania, które można uznać za fundamentalne. To właśnie na ich podstawie nastąpiła identyfikacja różnic pomiędzy nimi, które często są subtelne. Kolejną rzeczą, którą należy uwzględnić, jest zasięg, jakim charakteryzują się metody. Skutkuje to kolejnością podrozdziałów – od praktyk abstrakcyjnych, jak zasady, do konkretnych, jak wzorce projektowe.

2.1. Projektowanie oprogramowania

Zgodnie ze standardem IEEE610.12-90 [5] projektowanie oprogramowania jest procesem definiującym strukturę, interfejsy i inne charakterystyki systemu lub komponentu. Po zakończeniu specyfikacji lub w przypadku technik zwinnych, gdy wymagania co do oprogramowania zostały jasno określone, projektowanie jako proces narzuca pewne rozwiązania na najniższym poziomie, jakim jest struktura komponentów, klas, a nawet implementacji poszczególnych algorytmów. Musi ona uwzględniać architekturę oraz realizować wymagania każdego

interesanta systemu, np. klienta lub dyrektora technicznego. Jedną z metod projektowania jest projektowanie zorientowane obiektowo:

Definicja: *Projektowanie zorientowane obiektowo jest metodą projektowania obejmującą proces obiektowej dekompozycji i notacji. Rezultatem jej zastosowania jest przedstawienie logicznych, fizycznych, jak również statycznych i dynamicznych modeli projektowanego systemu* [5].

W przypadku skomplikowanych systemów informatycznych, normy i standardy pomagają w ich poprawnej realizacji, a następnie funkcjonowaniu. Jedną z nich jest międzynarodowy standard ISO/IEC 2500 Software engineering-Software Product Quality Requirements and Evaluation (SQuaRE). Składa się z serii standardów ulepszających oraz łączących dwa poprzednie standardy ISO/IEC 9126 Software product quality oraz ISO/IEC 14598 Software product evaluation. SQuaRE podkreśla potrzebę używania modeli definiujących jakość oprogramowania w postaci charakterystyk oraz ich podgrup. Umożliwia to ustanowienie wymagań systemu, ich kryteriów oceny oraz pomiaru.

Sam standard nie obejmuje tylko projektu oprogramowania, ale dotyczy również architektury, działającej aplikacji oraz kodu źródłowego. Podczas tworzenia systemu informatycznego należy uwzględnić atrybuty jakościowe, mogące mieć wpływ na inne niż docelowe wymagania, np. atrybuty wewnętrzne dotyczące kodu źródłowego. Wszystkie rodzaje modeli mają zagwarantować, że wszystkie charakterystyki są brane pod uwagę. Rozpatrują problemy z punktu widzenia wielu interesariuszy systemu: twórców, właścicieli, użytkowników. Często niezbędny jest kompromis, co wymusza ustanowienie odpowiednich priorytetów w danym projekcie.

Dla każdego atrybutu jakościowego istnieje możliwość dobrania odpowiednich metryk walidujących poprawność architektury kodu, jednak ich automatyzacja nie jest trywialna [6, 7]. W wielu przypadkach wartości metryk zależne są od danego projektu jak i również od częściowo subiektywnej oceny, np. testera w przypadku użyteczności.

2.2. Zasady projektowe

W przeciągu wielu lat udało się ustalić pewne właściwości, jakimi powinna charakteryzować się architektura jego kodu źródłowego, aby reprezentowała wysoką jakość. Ponieważ istnieje szereg technik i narzędzi IT, właściwości te zostały wyekstrahowane do pojęć bardziej abstrakcyjnych – zasad projektowych. Jak zauważył Pescio [8], nie są one nazbyt konstruktywne, nie dostarczają gotowych wyników, lecz ich naruszenie można stwierdzić jednoznacznie, np. poprzez wystąpienie skaz. Znalezienie odpowiedniego rozwiązania dla danego problemu zazwyczaj bazuje na doświadczeniu. Melton [9] podkreślił potrzebę zbadania zasad i przyporządkowania ich do odpowiednich charakterystyk projektowych. Dodatkowo potwierdził, iż mniej znane reguły nie są przestrzegane w środowiskach komercyjnych, a ich

skuteczność, pomimo przeświadczenia, nie została jeszcze dokładnie zbadana. Jego tezy zostały częściowo potwierdzone w późniejszych badaniach [10, 11]. Z drugiej strony respektowanie reguł może zostać wymuszone w zależności od specyfiki projektu i funkcji, jakie oferuje. Wskazują na to początkowe badania, dotyczące projektowania opartego na modułach i wtyczkach [12].

Martin [13] zaznaczył, że wraz z rozwojem systemu programiści mogą stracić z oczu cały jego kształt. Realizatorzy, zwłaszcza w modelu zwinnym, skupiają się na obecnie rozwijanych wymaganiach systemu, nie przykładając szczególnej uwagi do wymagań związanych z przyszłymi wersjami, które w większości nie są dokładnie zdefiniowane. Skutkiem takiego podejścia jest utrudniona ocena wewnętrznych charakterystyk jakości projektu. Martin [13] zdefiniował kilka symptomów umożliwiających zdiagnozowanie psującej się architektury kodu programistycznego, których rozwiązaniem jest zastosowanie jego zasad projektowych S.O.L.I.D. [13]. Nazwa jest akronimem pochodzącym do następujących reguł:

Tabela 1

Zasady S.O.L.I.D

S	SRP	Zasada jednej odpowiedzialności (ang. *Single responsibility principle*)
O	OCP	Zasada otwarte/zamknięte (ang. *Open/closed principle*)
L	LSP	Zasada podstawienia Liskov (ang. *Liskov substitution principle*)
I	ISP	Zasada segregacji interfejsów (ang. *Interface segregation principle*)
D	DIP	Zasada odwrócenia zależności (ang. *Dependency inversion principle*)

Z drugiej strony Lakos [15] kładzie nacisk na reguły związane z fizyczną strukturą komponentów i logiczną klas. Opisuje rozkład systemów, podkreśla ważną rolę, jaką odgrywa acykliczność powiązań i ich wpływ na łatwość utrzymania i testowania takiego systemu. Rozciąga zakres swoich zasad do średnich i dużych systemów. Podobne wnioski przedstawił również Martin w swoim późniejszym artykule [14].

Ponieważ zasady obejmują swoim zakresem wiele poziomów szczegółowości, można je podzielić w następujący sposób:

a) ogólne – zawierające definicję sięgająca poza obiektową kompozycję, a nawet poza proces projektowania oprogramowania, np. „Nie powtarzaj się” (ang. *Don't Repeat Yourself)* w skrócie *DRY* [15] lub „Utrzymuj prostotę, łatwość” (ang. *Keep It Simple, Stupid* w skrócie *KISS)* [16],
b) zorientowane obiektowo – np. reguły *S.O.L.I.D.* Podobnie jak wzorce można je podzielić ze względu na ich przeznaczenie:
   - czynnościowe – mające wpływ na zachowanie obiektów,
   - kreacyjne – mające wpływ na proces tworzenia obiektów,
   - strukturalne – mające wpływ na kompozycję klas lub komponentów.

Po analizie istniejących zasad można wymienić najważniejsze ich własności:

a) duży wpływ na strukturę kodu źródłowego na wyższym, koncepcyjnym poziomie niż skaza lub heurystyka obiektowa. Reguły nie zawierają gotowych rozwiązań, jedynie wytyczne, np. rezultatem zastosowania prawa Demeter jest czytelniejszy i mniej podatny na błędy system [17, 18]. Reguła kompozycji nad dziedziczeniem [16] spowoduje zmniejszenie się hierarchii klas i jej przesunięcie do zewnętrznych struktur,
b) jest często powiązana z daną metodyką tworzenia oprogramowania, np. techniki zwinne bardziej skupiają się na rozwijalności systemu (co wspiera zbiór *S.O.L.I.D.*), z drugiej strony model wodospadu często wymusza trzymanie się kurczowo zdefiniowanych reguł,
c) wpływ na strukturę kodu źródłowego podczas projektowania oprogramowania – zasada jednej odpowiedzialności będzie wymuszała głównie małe klasy, reguła odwrócenia zależności zmieni sposób, w jaki tworzone są klasy, a zasada Hollywood [16] preferuje architekturę kodu zorientowaną na zdarzenia i wywołania zwrotne,
d) wpływ na właściwości projektowe, głównie łatwość utrzymania i rozwijalność. Niektóre z reguł, takie jak *SRP* lub kompozycja nad dziedziczeniem wymuszają struktury, które będzie łatwiej zmienić bez wpływu na powiązane klasy. W wyniku tego istnieje mniejsze prawdopodobieństwo popełnienia błędu,
e) możliwa interpretacja na wiele sposobów, np. reguła hermetyzacji zmienności [16] dotyczy problemu rozwijalności systemu, przez co może być zrealizowana np. za pomocą wzorca fabryki [19] lub zasady segregacji interfejsów,
f) duża uniwersalność - możliwość użycia nie tylko w programowaniu obiektowym, czego przykładem jest brzytwa Ockhama wywodząca się z filozofii i pojawiająca się pierwszy raz w XVII wieku,
g) subiektywność implementacji zasad związana jest z ich ogólnością. Ponieważ istnieje wiele możliwości ich interpretacji, rozwiązania najczęściej są niejednoznaczne.

Bazując na powyższych właściwościach można wyprowadzić następującą definicję:

Definicja: *Reguła lub zasada projektowa w projektowaniu zorientowanym obiektowo to wytyczna obowiązująca podczas projektowania oraz implementacji oprogramowania. Może ona definiować sposób powstawania, zachowania i kompozycji poszczególnych klas, pakietów lub komponentów.*

2.3. Heurystyki obiektowe

Heurystyka jest zbiorem twórczych metod rozwiązywania problemów. Podobnie jak zasada projektowa, wynika ona z wiedzy opartej na doświadczeniu. Najbardziej popularnych jest 60 obiektowych heurystyk Riela [20].

Interesujące problemy związane z wykorzystaniem tego zbioru przedstawiają Churcher, Frater, Huynh i Irwin [21]:

a) brak jednoznacznych informacji, które heurystyki powinny być zaimplementowane. Heurystyki, które się wzajemnie się wykluczają, często dotyczą konkretnego problemu. Z drugiej strony konflikt może wynikać z filozofii i metodologii zastosowanej w danym projekcie, gdzie programista będzie zainteresowany tylko jednym problematycznym obszarem,
b) nieprecyzyjne definicje – heurystyka H2.8 mówi "*Dana klasa powinna zawierać tylko jedną kluczową abstrakcję*", gdzie określenie „kluczowej abstrakcji" jest problematyczne,
c) subiektywność i dopasowanie. Skazy projektowe wymagają od programisty oceny, gdy pewne subiektywne wartości zostały przekroczone. Herustyki "*large class smell*", "*lazy class smell*" i "*long method smell*" są przykładami, gdzie może obowiązywać wiele standardów. Ważność danych heurystyk (w konflikcie pomiędzy sobą) zależy od projektanta,
d) interpretacja w zależności od kontekstu – wiele heurystyk jest wyrażona ogólnikowo po to, aby pasowały do każdej architektury kodu. Dlatego często wymagana jest ich odpowiednia interpretacja, przykładowo programista musi podjąć decyzję, czy wielkość łańcucha dziedziczenia powinna uwzględniać wymagania danego języka co do wspólnej klasy bazowej,
e) wiele poziomów abstrakcji – niektóre heurystyki mogą zostać zinterpretowane różnorodnie, w zależności od ich poziomu. Heurystyka H2.1 *"Wszystkie dane powinny być ukryte w klasie"*, może zostać zastosowana na poziomie poszczególnych atrybutów albo nawet i właściwości tzw. „getterów",
f) przeładowanie informacjami – heurystyki pomagają radzić sobie ze złożonością danej architektury kodu, lecz ich nieroztropne wykorzystanie może tylko pogłębić problem,
g) pozyskiwanie odpowiednich danych i powiązanie ich z heurystykami. Wiele heurystyk wymaga dużej ilości danych. Heurystyka H4.6 "*Większość metod w danej klasie powinna używać większości pól tej klasy przez większość czasu*" jest tego przykładem. Dodatkowo zależność pomiędzy heurystyką a dostępną informacją nie jest zawsze jasna.

Na podstawie ogólnego pojęcia heurystyki i powyższych cech można wyprowadzić następującą definicję:

Definicja*: Heurystyka obiektowa to praktyczna zasada, reguła lub wydedukowany wniosek, która ogranicza poszukiwanie rozwiązań dla skomplikowanych lub niezrozumianych problemów związanych z obiektowym projektowaniem oprogramowania lub interakcjami do wewnątrz danej klasy.*

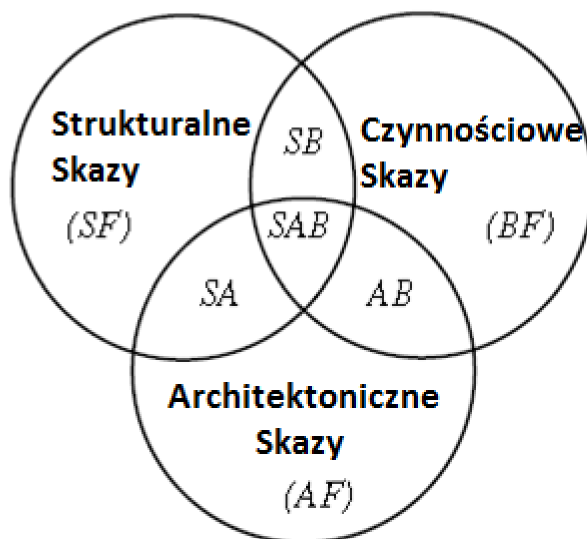
2.4. Skazy projektowe

Skazy projektowe lub, jak inaczej je nazywa Fowler, „zapachy kodu" [22] wskazują na problemy wymagające refaktoryzacji. Pojawiają się, gdy zostanie złamana co najmniej jedna

reguła projektowa lub heurystyka obiektowa. Skazy projektowe są jednoznaczne oraz dobrze zdefiniowane, przez co są możliwe do wykrycia, dobierając odpowiednie metryki. Dodatkowo badania potwierdzają ich przydatność, ponieważ kod, w którym występują, jest bardziej podatny na błędy wykonania [1, 23].

Bazując na pracy Tahvildari i Kontogiannis [24], skazy można podzielić na następujące grupy (rys. 1):

a) strukturalne (SF) – kategoria, do której należą skazy dotyczące wewnętrznej struktury danej klasy zarówno składniowej, jak i stylu,
b) architektoniczne (AF) – kategoria, do której należą skazy, które dotyczą zewnętrznej struktury klas (i ich interfejsów). Wynikają one ze złego zaprojektowania systemu,
c) czynnościowe (BF), zawierają wszystkie skazy związane z semantyką aplikacji,
d) strukturalno-architektoniczne (SA) – kategoria, do której należą skazy, które zmieniają wewnętrzną strukturę klas, powodując zmianę architektury kodu,
e) strukturalno-czynnościowe (SB) - kategoria, do której należą skazy, które po zmianie ich wewnętrznej struktury zmieniają swoje zachowanie,
f) architektoniczno-czynnościowe (AB) – kategoria, do której należą skazy, w której zmiana architektury kodu zmienia zachowanie klasy.



Rys. 1. Podział skaz projektowych
Fig. 1. Classification of code smells

Podział skaz stworzony przez Mohe et al. [25] bazuje na roli, jaką spełniają dane skazy:

- strukturalne, które dotyczą relacji i struktury poszczególnych składowych systemu, np. ilości odwołań,
- leksykalne, które dotyczą nazw części składowych systemu,
- mierzalne, które dotyczą mierzalnych składowych systemu, np. długość metody.

Opierając się na powyższych podziałach można wyprowadzić następującą definicję:

Definicja**:** *Skaza projektowa jest problemem związanym ze złym zaprojektowaniem systemu lub nieprawidłową interakcją do wewnątrz klas wynikającą ze złamania co najmniej jednej zasady projektowej lub nieprzestrzegania heurystyk projektowych.*

2.5. Wzorce projektowe

Wzorzec projektowy nie jest gotowym kodem zaimplementowanym w określonym języku, a jedynie szablonem określającym, w jaki sposób rozwiązać dany problem w pewnych warunkach. W obiektowym oprogramowaniu często przedstawia się powiązania i zależności pomiędzy klasami [26]. W porównaniu do zasad projektowych, które mówią, „co należy zrobić”, odpowiadają za część praktyczną, „jak to zrobić”.

Wzorce projektowe są sprawdzonymi praktykami przyśpieszającymi tworzenie systemu. Muszą one zostać za każdym razem zaimplementowane od nowa, co w dużym stopniu ogranicza ich automatyzację w kontekście refaktoryzacji. Ich zastosowanie poprawia czytelność, wydajność i rozwijalność oprogramowania przez co i jakość kodu [27]. Mogą one zostać wielokrotnie użyte i wymagają niewielkiego doświadczenia od programisty. Jedno z najbardziej znaczących opracowań w tej dziedzinie [19] opisuje główne wzorce projektowe, które wykorzystywane są w oprogramowaniu obiektowym oraz podaje następującą ich definicję:

Definicja*: Wzorzec projektowy opisuje komunikację pomiędzy obiektami i klasami, która została dostosowana po to, aby rozwiązać ogólne problemy projektowe w danym kontekście* [19].

Pomimo bardzo dobrej dokumentacji i kategoryzacji największym problemem w przypadku wzorców jest ich ograniczona przydatność podczas projektowania oprogramowania [28]. Jeżeli nie pokrywają one danego rozwiązania, programista pozostaje bez dalszego wsparcia. Dlatego są one tworzone wraz powstawaniem nowych technologii oraz technik tworzenia oprogramowania, np. dla aplikacji internetowych [29] lub najnowszych rozwiązań w obrębie interfejsu użytkownika [30]. Jest to w szczególności przydatne dla początkujących programistów, którzy nie posiadają jeszcze niezbędnego doświadczenia, aby znaleźć własne unikalne rozwiązania. Innym rodzajem wzorców są te wykorzystywane przez architektów systemu informatycznego, tzw. wzorce architektoniczne:

Definicja*: Wzorzec architektoniczny przedstawia fundamentalny, strukturalny, organizacyjny schemat dla danego oprogramowania. Określa zbiór predefiniowanych podsystemów, specyfikuje ich role oraz zawiera reguły i wytyczne, służące do organizacji powiązań pomiędzy nimi* [31].

Wybór tego rodzaju wzorca jest decyzją krytyczną z perspektywy kosztów, jakie mogłyby zostać wygenerowane w przypadku przebudowy systemu informatycznego, który nie

spełnia wcześniej postawionych wymagań, np. odpowiedniego podziału na podsystemy. Bardzo dobrym zbiorem wzorców architektonicznych dla systemów zarządzania przedsiębiorstwem jest pozycja Flowera [32]. Również architektura SOA (ang. *Service Oriented Architecture*) doczekała się swoich własnych wzorców zarówno projektowych, jak i architektonicznych [33].

2.6. Kategoryzacja pojęć

Kategoryzacja opisanych pojęć związanych z dobrymi praktykami w tym rozdziale opiera się na następujących kryteriach:

a) zasięg – od pojęć abstrakcyjnych, które można interpretować na wiele sposobów do pojęć bardziej szczegółowych wskazujących na gotowe rozwiązanie:
   - koncepcyjne, które dotyczy wymagań, jakie powinien spełniać projekt, jednak postawienie tych wymagań jest kwestią subiektywną, definiowaną w każdym przypadku na nowo,
   - projektowe, które dotyczy konkretnych wymagań co do projektu zorientowanego obiektowo, gdzie można jednoznacznie stwierdzić, czy dany projekt je spełnia,
   - implementacyjne – skoncentrowane na poszczególnych rozwiązaniach, jasno wskazujące sposób implementacji,

b) typ – który można podzielić na wskazanie już występującego problemu lub sposobu implementacji kodu programistycznego,

c) koszt wdrożenia, który można podzielić na duży, średni, mały. Często koszt jest wynikiem związanym z zasięgiem i typem, w jakim dane pojęcie się znajduje.

Tabela 2

Kategoryzacja pojęć

Nazwa	Zasięg	Typ	Koszt wdrożenia
Zasada projektowa	Koncepcyjny Projektowy	Kierunek realizacji	Duży
Wzorzec architektoniczny	Projektowy Implementacyjny	Kierunek realizacji	Duży
Heurystyka obiektowa	Projektowy	Kierunek realizacji	Średni
Skazy projektowe	Implementacyjny	Detekcja problemu	Średni / Mały
Wzorzec projektowy	Implementacyjny	Kierunek realizacji	Średni / Mały

W tabeli 2 w naturalny sposób udało się stworzyć pewną hierarchię pojęć związanych z jakością oprogramowania. Zmiana projektu we wszystkich wadliwych miejscach jest większa, gdy wykracza poza dostosowanie jednostkowych problemów, co ma miejsce w przypadku problemów o zasięgu koncepcyjnym. Widoczna jest również zależność pomiędzy kosztem wdrożenia a zasięgiem. Skazy i wzorce projektowe mogą być zaimplementowane błyska-

wicznie przez zgrupowanie kilku parametrów danej metody jako jednego obiektu lub być związane z przebudową większej części systemu, np. wprowadzenie wzorca projektowego fabryki. Większość pojęć opisanych w tym rozdziale dotyczy wytycznych związanych z realizacją projektu obiektowego, jedynie skazy mają na celu detekcję już występujących problemów. Ich zasięg skupiony jest głównie na implementacji, np. skaza długich metod (ang. *Long Method)*. Różnicę w zasięgu pomiędzy skazami a zasadami projektowymi widoczne są, gdy dotyczą tego samego problemu, np. skaza odmówionego spadku ang. *Refused beques*. Różni się ona od zasady projektowej *substytucji Liskov*, gdyż ta druga związana jest ze źle zaprojektowaną klasą (koncepcja). Z drugiej strony skaza *„odmówionego spadku"* źle wykorzystuje mechanizm przesłaniania w programowaniu zorientowanym obiektowo.

3. Detekcja problemów związanych z architekturą kodu

Istnieje wiele prób detekcji skaz projektowych lub złamania heurystyk obiektowych w kodzie. Strategie detekcji, jakie przedstawił Marinescu [34], dotyczą tworzenia odpowiednich reguł oraz dobierania wartości granicznych dla metryk manualnie. Erni et al. [35] stworzył koncept multimetryk, który umożliwia dobór wielu kryteriów, aby zidentyfikować i oszacować odpowiednie wartości.

Moha et al. wprowadziła podejście DECOR [25] (ang. *Detection and Correction*) zaczynając od opisu symptomów używając abstrakcyjnego języka reguł. Za pomocą odpowiednich notacji, np. kart reguł DECOR formalizuje wiele procesów związanych z pomiarem skaz, które następnie transformowane są do algorytmów detekcyjnych. Jednak przy tym podejściu nadal występuje problem wartości granicznych, które często należy dobierać manualnie. Zarówno Moha [25], jak i Luo [36] próbują dopasować odpowiednie antywzorce projektowe [37] do zapachów kodu, argumentując, że te pierwsze umożliwiają łatwiejsze zrozumienie problemu. Podkreślają oni, iż zapachy kodu jedynie wskazują na potencjalne miejsca, gdzie można zastosować refaktoryzację, niekoniecznie informując jakiego typu.

Podobnie jak zasady projektowe, antywzorce występują na koncepcyjnym poziomie, przy czym mogą łamać swoim zasięgiem więcej niż jedną zasadę. Skutkuje to dużą ilością potencjalnych rozwiązań oraz zwiększonymi kosztami wynikającymi z przebudowy systemu. Nie ograniczają one również subiektywności związanej z doborem wartości granicznych dla metryk wykorzystywanych w procesie detekcji skaz.

Maneerat i Muenchaisri [38] dostrzegli potencjał, jaki reprezentują algorytmy uczące się do wykrywania skaz. Sprawdzili skuteczność kilku metod, takich jak: *losowy las, naiwny klasyfikator beyesowski, regresje logistyczną, IB1, IBk, VFI* oraz *drzewo decyzyjne J48*. Wyniki badań pozwalają wierzyć, że użycie technik uczących ma dużą przyszłość – spośród 7

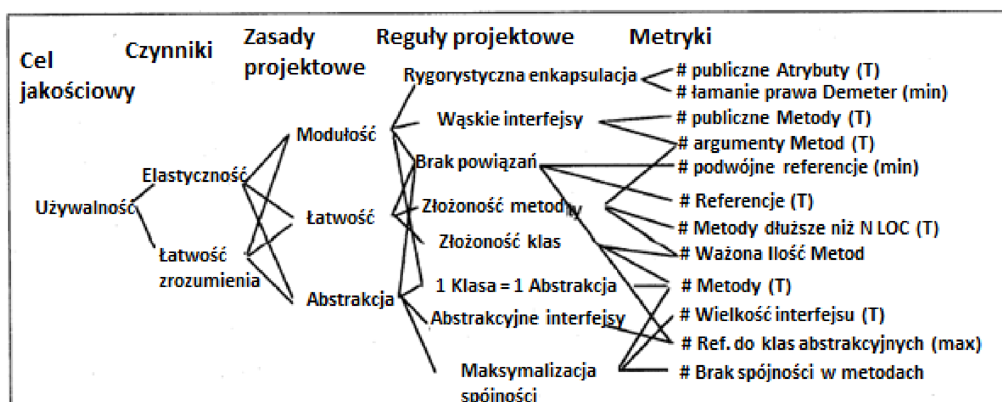
metod 5 uzyskało swoistość większą niż 88%, 6 czułość większą niż 70%, a we wszystkich dokładność była większa niż 72%. Khomh [39] ulepszył podejście DECOR, wprowadzając sieci beyesowskie. Umożliwiają one oszacowanie prawdopodobieństwa wystąpienia danej skazy, jednak ich wadą jest wymóg zdefiniowanych reguł detekcji w DECOR, przed wywołaniem algorytmu. Badania Khomha wykazały, że kalibracja sieci na obecnym systemie informatycznym może zająć średnio o 32% więcej czasu niż użycie przygotowanego modelu. Liu et al. [40] rozpatrzyli problem zależności wzorców przez wykorzystanie manualnej analizy. Jej wynikiem jest priorytetyzacja, w której w pierwszej kolejności znajdują się skazy niezależne. Skutkuje to o wiele wydajniejszym rozwiązywaniem tego rodzaju problemów, np. usunięcie lub przeniesienie części kodu może spowodować automatyczne pozbycie się skaz od niej zależnych. Liu et al. zauważyli również, że powstałe w wyniku manualnej analizy grafy powiązań skaz mają skomplikowaną i nieczytelną strukturę. Jako rozwiązanie tego problemu stworzyli algorytm, mający na celu usunięcie zduplikowanych ścieżek w diagramie, jednocześnie nie zmieniając ich topologicznej kolejności. Kessentini et al. wykorzystali programowanie genetyczne do wyszukiwania występujących skaz, które w większości wypadków jest skuteczniejsze niż podejście detekcji DECOR – średnio, precyzja była większa o 32%. Powyższe podejście, które można określić jako półautomatyczne, wymaga jednak dużej ilości danych podczas fazy trenowania, co oznacza manualną inspekcję nawet kilku systemów [41]. Serban [42] zaadoptowała logikę rozmytą połączoną z klastrowaniem partycyjnym w celu znalezienia skaz projektowych, mianowicie algorytm FDHC (ang. *Fuzzy Divisive Hierarchic Clustering*). Dostrzegła również potrzebę przyporządkowania skaz do bardziej abstrakcyjnych reguł – heurystyk, zasad projektowych. Niestety, nie przedstawiła wyników swoich badań oraz skuteczności tego podejścia. Bardzo dobre wyniki zostały osiągnięte przez wykorzystanie algorytmu klasyfikacyjnego AIS (ang. *Artifical Immune System*) zaczerpniętego z medycyny, odpowiednio adaptując go do podejścia związanego z architekturą kodu – IDS (ang. *Immune-based Detection Strategy*) [43]. W tym przypadku klasy reprezentują komórki, które występują w ciele (projekcie systemu) oraz zawierają ciała obce (zapachy kodu). Palomba et al. wykorzystali systemy kontroli wersji w celu detekcji zapachów kodu, tzw. podejście HIST (ang. *Historical Information for Smell deTection)* [44]. Założyli, że sama natura niektórych skaz wynika ze zmiany. Na tej podstawie, dla każdej z omawianych pięciu skaz stworzona została odpowiednia strategia ich wykrycia, oparta na różnicy pomiędzy dwoma wpisami w systemie kontroli wersji. Precyzja detekcji dla czterech skaz okazała się większa niż 70%, a dla trzech czułość przekroczyła poziom 75%. Minusem tego rozwiązania jest manualne tworzenie odpowiednich strategii wykrywania problemów bazujących na zmianach w strukturze kodu.

Z drugiej strony nastąpiła próba pomiaru heurystyk obiektowych, charakteryzujących się dużą swobodą interpretacyjną. Jak udowodnili Bar i Ciupke [45], większość heurystyk Riela

da się przedstawić w postaci odpowiednich warunków. W tym przypadku został użyty specjalny język programowania Prolog, umożliwiający definiowanie reguł. Jednocześnie przedstawili oni ograniczenia w swoim pomiarze, klasyfikując niemierzalne zasady, będące zbyt abstrakcyjne lub subiektywne. Selehie, Li oraz Tahvildari [46] swój model wiedzy skonstruowali bazując na perspektywie danej heurystyki, asocjując ją do danej skazy, co pozwala jedynie na częściowe jej wykorzystanie w prostych, jednoznacznych przypadkach. Dobór wartości granicznych dla metryk występujących w ich regułach nie został jasno zdefiniowany. Churcher, Farater, Huynh i Irwin [21] dostrzegli problemy związane z heurystykami. Stworzyli metamodel, który może zostać wykorzystany w projektowaniu zorientowanym obiektowo. Bazuje on na predefiniowanych, konfigurowalnych regułach, które są następnie sprawdzane na niezależnym od danego języka modelu semantycznym.

4. Aktualne modele jakości projektu

Pomimo dużej ilości podejść detekcji problemów związanych z jakością architektury kodu przedstawionych w poprzednim rozdziale, istnieje pewien problem dotyczący skutecznego wykorzystania całego doświadczenia projektowego, jakie zostało zebrane. Autorzy prac koncentrują się głównie na skazach, które tylko częściowo wykrywają problemy strukturalne. Lukę tę próbował uzupełnić Erni i Lewerentz [35] swoim modelem *multimetryk*:

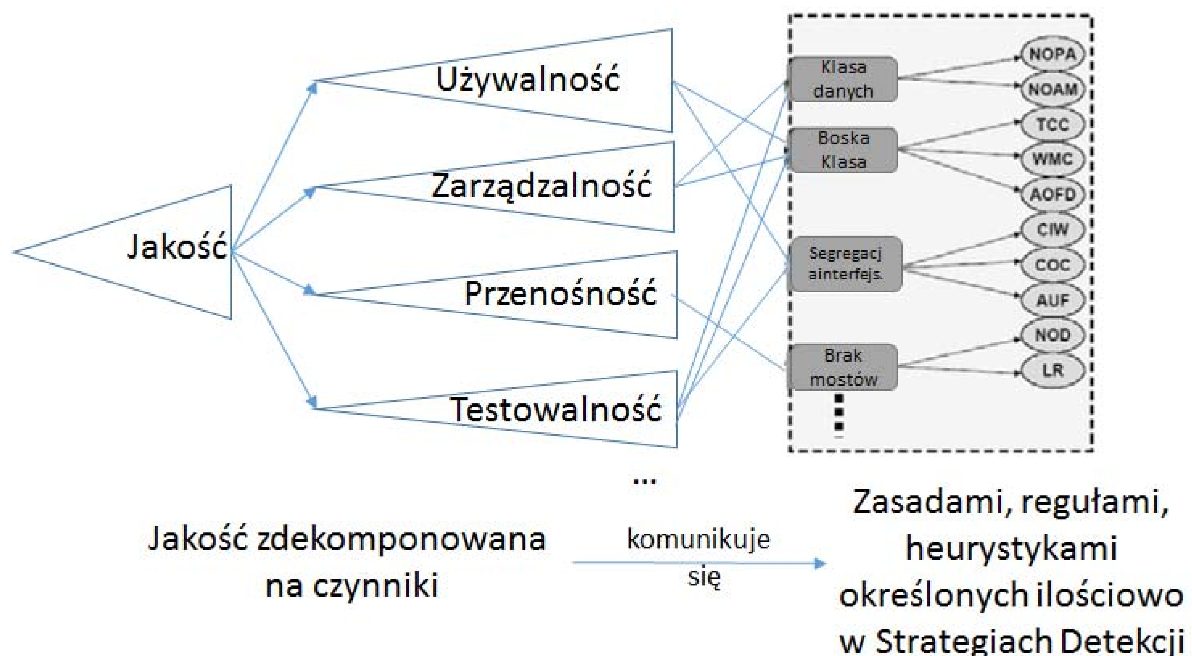


Rys. 2. Model multimetryk
Fig. 2. Multi-metric model

Jednak idee projektowe w tym przypadku występują w postaci podstawowych mechanizmów projektowania obiektowego, np. abstrakcji lub ogólnych pojęć, np. modularności. Następnie w hierarchii występują zasady, które są narzędziami przełożenia tych mechanizmów. Taki podejście może się okazać skuteczne, ale nie wskazuje na potencjalne rozwiązanie danego problemu. Bardziej precyzyjny model zdefiniował Bansiya et al. [7], w tym przypadku metryki zostały przyporządkowane do mechanizmów projektowania zorientowanego obiektowo, np. hermetyzacji, a te z kolei do charakterystyk jakościowych, np. funkcjonalności.

Bansiya ogranicza zbiór metryk do takich, które nie wymagają dużego nakładu implementacyjnego, w ten sposób jego model nabiera możliwości częściowo predykcyjnych.

Marinescu [34] dostrzegł, że poprzednie rozwiązania nie są w stanie przyporządkować metryk do obowiązujących dobrych praktyk w projektowaniu zorientowanym obiektowo. Dodatkowo zakwestionował skuteczność stosowania poszczególnych metryk w izolacji, a nie jako zbioru reguł. Wprowadził model *Factory-Strategy*, składający się z dwóch części: czynników oraz strategii detekcji (rys. 3). Czynniki reprezentują charakterystyki projektowe i łącza się regułami, heurystykami oraz ideami. Marinescu nie uwzględnia jednak ich różnorodnego poziomu abstrakcyjności, w wyniku czego przedstawia reguły metryczne powiązane głównie ze skazami projektowymi.



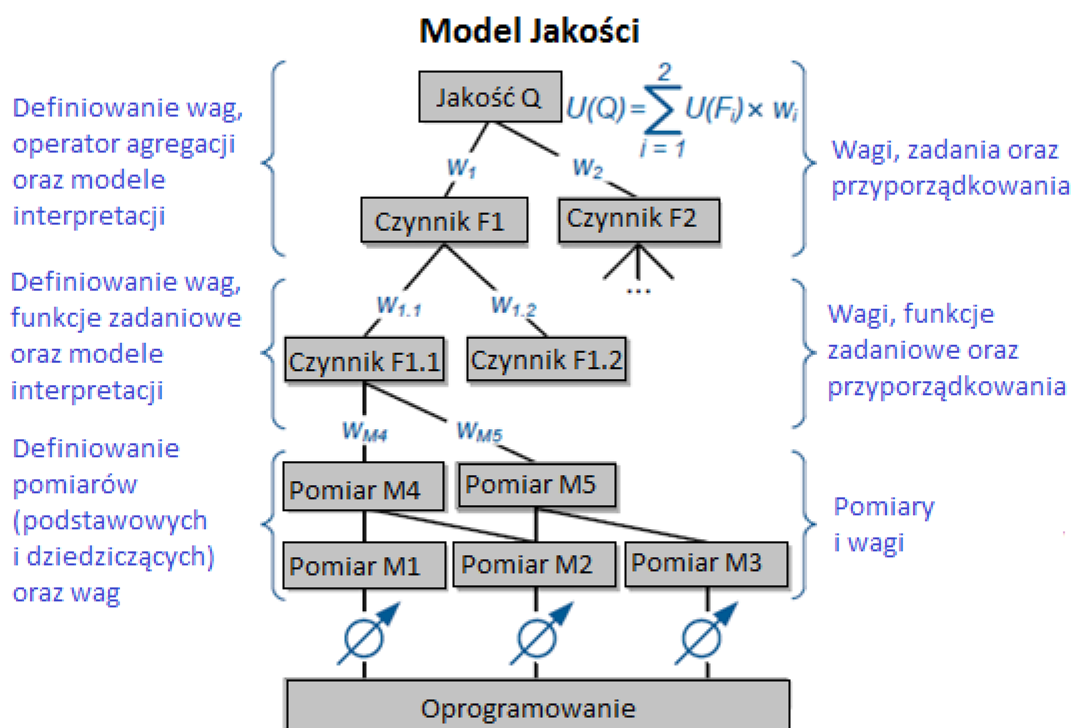
Rys. 3. Factor-Strategy model
Fig. 3. The Factor-Strategy model

Podobne problemy zawiera model pojęciowy Serban [42], w którym na najwyższym poziomie hierarchii znajdują się również maksymy. Detekcja odbywa się przez dobór najoptymalniejszych zbiorów metryk dla skaz projektowych.

Selehie, Li oraz Tahvildari [46] w swym podejściu skoncentrowali się na heurystykach. W skonstruowanym generycznym modelu wiedzy przyporządkowali zasady i heurystyki do jednego poziomu abstrakcji (rys. 4). Na podstawie tego można wnioskować, że heurystyka lub zasada projektowa powiązana jest z jedną skazą, co nie obowiązuje w każdym przypadku.

Rys. 4. Generyczny model wiedzy w programowaniu zorientowanym obiektowo
Fig. 4. A Generic OO Design Knowledge-Base Model

Kolejnym podejściem oceny jakości projektu są modele konfigurowalne. Często powstają one w wyniku współpracy środowisk komercyjnych wraz naukowymi. Przykładami takich modeli są Quamoco [47] oraz Squale [48]. Umożliwiają indywidualne połączenie charakterystyk jakościowych z dobrymi praktykami, a w końcu z metrykami. Squale wprowadza jako warstwę pośrednią pojęcie dobrych praktyk do modelu FCM, opierając się na standardzie ISO 9126. Przez zastosowanie wag i normalizacji każda praktyka może zostać dostosowana do norm panujących w danej firmie. Z drugiej strony Quamoco zawiera dodatkowo model produktu, umożliwiając podział związany z pojęciami i czynnikami abstrakcyjnymi oraz czynnikami zależnymi od danej technologii lub języka programowania. W ten sposób autorzy uniknęli problemów związanych z przełożeniem charakterystyk jakościowych na dobre praktyki. Quamoco ma na celu udoskonalenie standardu ISO 25010, jeżeli chodzi o oszacowanie jakości oprogramowania. W tym celu powstała hierarchia czynników jakościowych oraz pomiarów, np. na rys. 5 metryka wynikowa M4 może składać się z dwóch metryk: M1 - metryki rozmiaru odpowiadającej za normalizację oraz metryki M2 odpowiadającej za wartości, np. ilość komentarzy.



Rys. 5. Model Quamoco
Fig. 5. The Quamoco model

W modelu Quamoco (rys. 5) na najniższym poziomie znajdują się pomiary, które następnie połączone są z funkcjami zadaniowymi (ang. *utility functions*). Na podstawie analizy systemów oraz manualnej walidacji zostają określone wartości graniczne dla pomiaru. Zaletą powyższego modelu jest zmienna obliczana za pomocą wag oraz funkcji zadaniowych, która umożliwia ocenianie jakości i jej porównanie pomiędzy projektami.

Niektórzy autorzy próbują klasyfikować podobne problemy występujące w projektowaniu obiektowym w postaci konceptów [49, 50]. W ten sposób naturalnie udaje się stworzyć grupy, które jest łatwiej interpretować oraz rozwiązać. Często takie podejście umożliwia odkrycie prawdziwej przyczyny występującej skazy oraz potrafi oddzielić problem, symptom, oraz rozwiązanie.

Większość modeli nie rozróżnia dobrych praktyk, umiejscawiając je w tej samej warstwie. Co więcej, występuje niejednoznaczność ich standardowej definicji, przykładowo model multimetryk jako zasady projektowe uznaje podstawowe mechanizmy programowania obiektowego. Również perspektywa, z jakiej model został zbudowany, wpływa na jego użyteczność, co jest widoczne dla modelu generycznego kładącego nacisk głównie na heurystyki. Z drugiej strony, modele konfigurowalne wymagają dobrego zrozumienia i dużego nakładu pracy, przez co solidnej bazy wiedzy ekspertów. Przez hierarchizację czynników jakościowych rozwiązują problem przełożenia abstrakcyjnych charakterystyk na ich pomiar, jednak większość pracy z ich konstrukcją musi zostać wykonana manualnie.

5. Konkluzje

Pomimo powstania przez lata wielu heurystyk, zasad oraz skaz projektowych, obecne modele mające na celu oszacowanie wysokiej jakości architektury kodu często nie wykorzystują ich potencjału, skupiając się na problemach jednostkowych. Prowadzi to do braku możliwości diagnozy odchyleń od charakterystyk jakościowych o większym spektrum, np. przez łamanie zasad projektowych, które wpływają na budowę, i powiązań pomiędzy klasami w całym systemie informatycznym. Takie zagadnienia często dotyczą metodyki, która jest używana podczas tworzenia systemu, a to z kolei wymusza dopasowanie odpowiednich reguł detekcji, więc zmianę wartości granicznych dla metryk występujących w najniższych warstwach modeli jakościowych. Pomimo użycia wielu metod algorytmów uczących się, które radzą sobie z indywidualną oceną, adaptacja wyuczonych modeli do wymogów obwiązujących w danej firmie nie jest łatwa. Badania problemów dotyczących naruszenia kilku dobrych praktyk powinny uwzględnić konflikty, np. na poziomie jednej metryki w zależności od perspektywy, która jest obecnie badana. Należy również uwzględnić subiektywność projektantów, w wyniku czego sam proces uczący będzie konfigurowalny. Co więcej, taka adaptacja powinna być w miarę szybka, ponieważ reguły metryczne powinny zostać ustalone podczas rozpoczęcia danego projektu. Kolejnym problemem jest rodzaj metryk, na jakich autorzy się skupiają podczas detekcji problemów projektowych. Są to często metryki, które wymagają zaimplementowanej funkcjonalności, np. CBO (ang. *Coupling Beetwen Objects*) wymaga całej struktury klas. Zbyt mało badań zostało przeprowadzonych w celu wyszukania reguł detekcji skaz, które bazują na alternatywnych pomiarach. Z drugiej strony, powinny one być w dalszym ciągu jak najprostsze, umożliwiając ich łatwe zrozumienie przez programistę.

BIBLIOGRAFIA

1. D'Ambros M., Bacchelli A., Lanza M.: On the Impact of Design Flaws on Software Defects. 10th International Conference on Quality Software QSIC, Zhangjiajie 2010.
2. Khomh F., Penta M. D., Guéhéneuc Y.-G. L.: An Exploratory Study of the Impact of Code Smells on Software Change-proneness.16th Working Conference on Reverse Engineering WCRE, Lille 2009.
3. Yamashita A., Mesan A.S., Moonen L.: Do developers care about code smells? An exploratory survey. 20th Working Conference on Reverse Engineering WCRE, Koblenz 2013.

4. McCall J. A., Richards P. K., Walters G. F.: Factors in Software Quality. NTIS, t. 1, Springfield 1977.
5. 610.12-1990 – IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society, 1990 (Reaffirmed 2002).
6. Ragab S. R., Ammar H. H.: Object oriented design metrics and tools a survey. The 7th International Conference on Informatics and Systems INFOS, Kair 2010.
7. Bansiya J., Davis C. G.: A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, t. 28, nr 1, 2002, p. 4÷17.
8. Pescio C.: Principles Versus Patterns. Computer, t. 30, nr 9, Los Alamitos 1997.
9. Melton H.: On the Usage and Usefulness of OO Design Principles. OOPSLA 2006, Portland 2006.
10. Wermelinger M., Yu Y., Lozano A., Capiluppi A.: Assessing architectural evolution: a case study. Empirical Software Engineering, Wyd. Springer, t. 16, nr 5, 2011, s. 623÷666.
11. Wermelinger M., Yu Y., Lozano A.: Design principles in architectural evolution: A case study. 24th IEEE International Conference on Software Maintenance ICSM, Beijing 2008.
12. Wermelinger M., Yu Y., Lozano A., Capiluppi A.: Assessing architectural evolution: a case study. Empirical Software Engineering, t. 16, nr 5, s. 623÷666.
13. Martin R. C.: Agile Principles, Patterns, and Practices in C#. Wyd. Prentice Hall, 2006.
14. Martin R. C.: Design Principles and Design Patterns. 01 2000. [Online]. Available: www.objectmentor.com.
15. Thomas D., Hunt A.: The Pragmatic Programmer: From Journeyman to Master. Wyd. Addison-Wesley Professional, 1999.
16. Freeman E., Freeman E., Bates B., Sierra K., Robson E.: Head First Design Patterns. Wyd. O'Reilly Media, 2004.
17. Wuersch M., Giger E., Gall H.: An Empirical Validation of the Benefits of Adhering to the Law of Demeter. 18th Working Conference on Reverse Engineering WCRE, Lero 2011.
18. Lieberherr K. J.: Controlling the complexity of software designs. 26th International Conference on Software Engineering ICSE, Edinburgh 2004.
19. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Wyd. USA: Addison-Wesley, 1995.
20. Riel A. J.: Object-Oriented Design Heuristics, Wyd. Addison Wesley, 1996.
21. Churcher N., Frater S., Huynh C. P., Irwin W.: Suporting OO Desgin Heuristics. Engineering: Reports, Wyd. Uniwersytetu Canterbury, Canterbury 2006.
22. Martin Fowler K. B.: Refactoring: Improving the Design of Existing Code. Wyd. Addison-Wesley Professional, 1 edycja, 1999.

23. Li W., Shatnawi R.: An empirical study of the badsmells and class error probability in the post-releaseobject-oriented system evolution. Journal of Systems and Software, t. 80, nr 7, 2007, s. 1120÷1128.
24. Tahvildari L., Kogiannis K.: A metric-based approach to enhance design quality through meta-pattern transformations. 7th European Conference on Software Maintenance and Reengineering, Benevento 2003.
25. Moha N. G. Y.-G., Duchien L., Le Meur A.-F.: DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering, t. 36, nr 1, 2010.
26. McNatt W. B., Bieman J. M.: Coupling of design patterns: common practices and their benefits. 25th Annual International Conference on Computer Software and Applications COMMPSAC, Taichung 2001.
27. Prechelt L., Unger-Lamprecht B., Philippsen M., W. F. Tichy: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Transactions on Software Engineering, t. 28, nr 6, 2002.
28. Pescio C.: Principles Versus Patterns. Computer, t. 30, nr 9, 1997, s. 130÷131.
29. Alur D., Malks D., Crupi J., Core J2EE Patterns: Best Practices and Design Strategies. Wyd. Prentice Hall, 2 edycja, 2003.
30. Smith J.: WPF Apps With The Model-View-ViewModel Design Pattern. MSDN Magazine , nr 2/2009, 2009, s. 72÷83.
31. Buschmann F., Meunier R., Rohnert H., Sornmerlad P., M. Stal: Pattern-oriented software architecture. A system of patterns. Wyd. JOHN WlLEY & SONS, 2001.
32. Fowler M.: Patterns of Enterprise Application Architecture, Wyd. Addison Wesley, 2002.
33. Erl T.: SOA Design Patterns. Wyd. Prentice Hall PTR, 2009.
34. Marinescu R.: Measurement and quality in object-oriented design. 21st IEEE International Conference on Software Maintenance ICSM, Budapeszt 2005.
35. Erni K., Lewerentz C.: Applying design-metrics to object-oriented frameworks. 3rd International Software Metrics Symposium, Berlin 1996.
36. Luo Y., Hoss A. M., Carver D. L.: An ontological identification of relationships between anti-patterns and code smells. IEEE Aerospace Conference, Montana 2010.
37. Brown W. J., Malveau R. C., McCormick H. W., Mowbray T. J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wyd. Wiley; 1 edycja, 1998.
38. Maneerat N., Muenchaisri P.: Bad-smell prediction from software design model using machine learning techniques. 8th International Joint Conference on Computer Science and Software Engineering JCSSE, Nakhon Pathom 2011.

39. Khomh F. V. S., Gueheneuc Y.-G., Sahraoui H.: A Bayesian Approach for the Detection of Code and Design Smells. 8th International Conference on Quality Software QSIC, Jeju 2009.
40. Liu H., Ma Z., Shao W., Niu Z.: Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. IEEE Transactions on Software Engineering, t. 38, nr 1, 2012, s. 220÷235.
41. Kessentini M., Kessentini W., Sahraoui H. A., Boukadoum M. A., Ouni A.: Design Defects Detection and Correction by Example. 19th International Conference on Program Comprehension ICPC, Kingston 2011.
42. Serban C.: A Conceptual Framework for Object-oriented Design Assessment. Fourth UKSim European Symposium on Computer Modeling and Simulation (EMS), Pisa 2010.
43. Hassaine S., Khomh F., Guéhéneuc Y.-G. L., Hamel S.: IDS: An Immune-Inspired Approach for the Detection of Software Design Smells. Seventh International Conference on Quality of Information and Communications Technology QUATIC, Porto 2010.
44. Palomba F., Bavota G., Di Penta M., Oliveto R.: Detecting bad smells in source code using change history information. 28th International Conference on Automated Software Engineering (ASE), Palo Alto 2013.
45. Bär H., Ciupke O.: Exploiting Design Heuristics for Automatic Problem Detection. Workshop ion on Object-Oriented Technology ECOOP, Bruksela 1998.
46. Salehie M., Li S., Tahvildari L.: A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. 14th IEEE International Conference on Program Comprehension ICPC, Ateny 2006.
47. Wagner S., Lochmann K., Heinemann L., Klas M.: The Quamoco product quality modelling and assessment approach. 34th International Conference on Software Engineering (ICSE), Zurych 2012.
48. Mordal-Manet K., Balmas F., Denier S., Ducasse S.: The squale model – A practice-based industrial quality. 25th IEEE International Conference on Software Maintenance ICSM, Edmonton 2009.
49. Trifu A., Marinescu R.: Diagnosing design problems in object oriented systems. 20th 12th Working Conference on Reverse Engineering WCRE, Pittsburgh 2005.
50. Yashita A. F., Benestad H. C., Anda B. C. D., Arnstad P. E., Sjoberg D., Moonen L.: Using concept mapping for maintainability assessments. 3rd International Symposium on Empirical Software Engineering and Measurement ESEM, Lake Buena Vista 2009.

Abstract

In the first part the article describes different kinds of best practices which are used in the object oriented software design – heuristics, principles (e.g. S.O.L.I.D. principles listed in table 1), smells and patterns. They are mainly based on the development experience and have a major influence on the code architecture. Additionally they refer to different abstraction levels with some usage and measurement constraints. The classification process of the above-mentioned best practices is hampered by lack of standardized definitions which often leads to interpretation differences in the literature (described more precisely in chapter 4). That is why for each presented in this article method a definition is quoted. In case of its absence it is introduced as a result of earlier grouped characteristics. In line with new scientific studies [1, 2] the count of bugs is clearly connected with the code architecture so the programmer should put more emphasis on the usage of best practices [3].

The purpose of the second article part is the review of current problem diagnosis and measurement methods connected with software design. Potential approaches were examined in terms of earlier identified interpretation differences between best practices. Next the main project quality models so-called FCM (Factor-Criteria-Metric [4]) models are presented like the Factor-Strategy model on fig. 3 or the Quamoco model on fig. 5. Their task is to decompose and assign quality characteristics to object oriented properties, which finally allows the identification of the software metrics describing them. In the article expected factors are examined, which each model ought to have e.g. assessment subjectivity, easiness of understanding, easiness of adaptation.

Adres

Tomasz LEWANDOWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, t.lewandowski84@gmail.com.