Rafał Wojszczyk
Zakład Podstaw Zarządzania i Informatyki
Wydział Elektroniki i Informatyki
Politechnika Koszalińska
rafal.wojszczyk@tu.koszalin.pl

Włodzimierz Khadzhynov
Katedra Inżynierii Komputerowej
Wydział Elektroniki i Informatyki
Politechnika Koszalińska

Wykorzystanie modeli danych do weryfikacji implementacji wzorców projektowych

Słowa kluczowe: wzorce projektowe, model danych, ERD, weryfikacja oprogramowania

1. Wstęp

Ward Cunningham to znana postać w społeczeństwie programistów. Cunnigham uzasadniając potrzebę refaktoryzacji oprogramowania wprowadził metaforę finansową nazywaną długiem konstrukcyjnym [11] [6]: niespłacany dług prowadzi do narastających odsetek, im dłużej nie jest spłacany tym większe szanse, że urośnie do przytłaczającej kwoty, z którą przedsiębiorstwo sobie nie poradzi. Analizując tą metaforę można wywnioskować, że niespłacony kredyt jest niepożądaną sytuacją w przedsiębiorstwie. Podobnie oprogramowanie zawierające zbyt skomplikowany kod prowadzi do niepożądanych konsekwencji zarówno dla użytkowników programu jak również dla przedsiębiorstwa odpowiedzialnego za jego wytworzenie. Istnieje wiele metod i ogólnych zaleceń mających na celu ulepszenie każdego z etapów procesu wytwórczego oprogramowania, są to: przygotowanie specyfikacji wymagań systemowych zidentyfikowanych we wstępnej analizie, następnie podczas projektowania niezwykle popularne jest zamodelowanie najważniejszych fragmentów lub całego systemu za pomocą diagramów UML, w implementacji wykorzystywane są odpowiednie narzędzia i dobre praktyki programowania, ostatecznie procedury testowania i konserwacji oprogramowania.

Dobre praktyki w programowaniu obiektowym to m. in. różnego rodzaju wzorce, a w tym wzorce [22]:

- architektoniczne (np. MVC) łączące różne warstwy aplikacji,
- projektowe [7] obejmujące swoim zakresem poziom klas,
- implementacyjne, nazywane również idiomami, występujące na poziomie wierszy kodu.

Jedne z najpopularniejszych wzorców projektowych zostały przedstawione w [7], są to za [16]: szkielety gotowych mechanizmów, które można wykorzystywać przy rozwiązywaniu typowych problemów projektowania i programowania. Opracowany w 1995 roku katalog wzorców [7] powstał na podstawie doświadczenia praktyków programowania [11] i do dziś wykorzystywany jest przez wielu programistów. Ze względu na dużą popularność i brak formalnej kontroli pojawia się potrzeba weryfikacji implementacji wzorców projektowych w powstającym oraz istniejącym oprogramowaniu.

Celem artykułu jest prezentacja modeli danych wykorzystywanych w procesie weryfikacji implementacji wzorców projektowych. Wynik weryfikacji jest niezbędny do podjęcia próby oceny jakości implementacji wzorców projektowych, jest to dalekosiężny cel prowadzonych prac. Kontekst zagadnienia oraz problem weryfikacji przedstawia rozdział 2. W rozdziałach 3 i 4 zawarto prezentację wykorzystywanych modeli danych, natomiast 5 rozdział to podsumowanie artykułu.

2. Weryfikacja implementacji wzorców projektowych

2.1. Kontekst problemu

Forma opisu wzorców projektowych przedstawiona w [7] jest z założenia przeznaczona do celów dydaktycznych, zawiera: opis słowny w języku naturalnym, diagramy klas w notacji OMT oraz przykładowy kod implementacji oparty o proste przykłady. Wzorce opisane w ten sposób przedstawiają zazwyczaj jeden z wariantów implementacji, który nie jest optymalnym rozwiązaniem [11]. Dlatego programista bazując na informacjach przedstawionych w literaturze powinien przy każdej implementacji danego wzorca wzbogacić wytwarzany przez siebie kod o wiele czynników związanych m.in. z logiką biznesową, aby lepiej dopasować implementację wzorca do kontekstu programu. Te czynniki zwiększają złożoność i różnorodność implementacji wzorców, czego efektem jest występowanie wielu wariantów implementacji oraz zacieranie się granic wzorców projektowych. Ostatecznie prowadzi to do wzrostu pracochłonności podczas inspekcji oraz konserwacji kodu.

Wzorce projektowe bardzo często wiązane są wyłącznie z etapem projektowania oprogramowania a nie z kodem programu [11]. Jest to wąski punkt

widzenia, szczególnie gdy za implementację oprogramowania odpowiedzialny jest mały zespół pracujący według metodyki Scrum [15]. W metodykach zwinnych, z których wywodzi się Scrum, projektowanie oprogramowania oraz dokumentowanie jest mniej ważne niż dostarczenie działającego oprogramowania. Często decyzje o rozwiązaniu pewnego problemu poprzez implementację wzorca są podejmowane na szybko podczas porannych spotkań (tzw. poranny Scrum). Następnie praca nad implementacją może być oddelegowana do członka zespołu, który nie miał wcześniej związku z danym wzorcem. Efektem jego pracy może być implementacja, w której wyłącznie pozornie występuje wzorzec, tj. oprogramowanie może działać prawidłowo i przechodzić wymagane testy, jednakże implementacja samego wzorca będzie niezgodna z zaleceniami i nie będzie spełniała postawionego celu. Problemy wynikające z tego faktu będą najbardziej zauważalne przy próbie rozbudowy lub modyfikacji danego fragmentu oprogramowania. Opisany przykład jest tylko jednym z wielu, które wskazują potrzebę weryfikacji implementacji wzorców projektowych.

Istnieją różne metody i narzędzia przeznaczone do testowania oraz analizy kodu źródłowego oprogramowania. Często dostarczone są razem ze zintegrowanym środowiskiem programistycznym, które zadba o zgodność elementarnych rzeczy z syntaktyką języka i założeniami danego paradygmatu programowania. Możliwości środowisk programistycznych można rozszerzyć dzięki wielu zautomatyzowanym narzędziom, które nie uwzględniają wzorców projektowych [14]. Programista wykorzystując elementy bazowe tworzy własne artefakty oprogramowania (np. biblioteki.dll) zawierające wyspecyfikowaną logikę biznesową, która nie może być sprawdzona przez uniwersalny algorytm. Większość artefaktów wymaga indywidualnego podejścia i przygotowania niezbędnych danych oraz funkcji testujących, jest to często realizowane przez testy jednostkowe. Pozytywny wynik testu jednostkowego dla danego artefaktu nie jest równoznaczny z prawidłową implementacją wzorców projektowych w tym artefakcie.

W metodach związanych z wzorcami projektowymi bardzo często podejmowany jest problem wyszukiwania wystąpień wzorców projektowych w oprogramowaniu [20] [18] [1]. Miarą wymienionych metod jest przedstawienie ilości wystąpień wzorców, jest to niewystarczająca informacja do realizacji stawianej potrzeby. Inne metody [2] skupiają się głównie na wykazaniu poprawności strukturalnej, niestety wymaga to nauki dedykowanego języka programowania do opisu danych implementacyjnych. Podejmowane są również próby stosowania znanych metryk oprogramowania [12] do implementacji wzorców projektowych, jednakże w [9] wykazano, że występowanie wzorców w oprogramowaniu może niekorzystnie wpływać na wyniki metryk.

W przypadku praktyków programowania, np. opisanych wcześniej członkach zespołach Scrumowych, wybór odpowiedniego rozwiązania do weryfikacji wzorców projektowych podyktowany jest odpowiednimi walorami użytkowymi.

Jest to niezbędne, żeby nowe rozwiązanie zostało pozytywnie przyjęte. Rozwiązanie takie musi być dostosowane do umiejętności potencjalnych odbiorców oraz nie powinno wymagać nauki nowych koncepcji. Formalne reprezentacje danych, takie jak wykorzystanie ontologii w informatyce, są stosunkowo nowe i wydają się być mało popularne. Powszechnie znane UML (oraz podobne, np. OMT) w zastosowaniu do wzorców projektowych, jest uznawane za pół-formalną reprezentację [17]. Wynika to z pominięcia w diagramach UML niektórych aspektów implementacji wzorców [17], ograniczonych możliwości weryfikacji implementacji. W [18] [20] autorzy bazując na diagramach klas wykazali, że w takim podejściu nie możliwe jest odróżnienie wzorca strategii od stanu [20] oraz identyfikacja wzorca Singleton [18].

2.2. Weryfikacja implementacji

Weryfikacja implementacji danego wzorca projektowego to zespół czynności, które należy wykonać aby wykazać zgodność badanego kodu programu z regułami i zasadami implementacji przyjętymi dla danego wzorca. Proces weryfikacji realizowany jest w oparciu o autorski model oceny jakości implementacji wzorców projektowych [24].

Badany kod programu, dokładniej fragment kodu źródłowego oprogramowania (lub kodu pośredniego) zawiera zaimplementowany konkretny wzorzec projektowy. Kod badanego oprogramowania, w proponowanym podejściu, jest przekształcany do formalnej reprezentacji (ekwiwalent kodu) i wyłącznie ta postać jest wykorzystywana w weryfikacji. Formalna reprezentacja oprogramowania to pierwszy z omawianych modeli danych.

Drugi proponowany model danych to repozytorium implementacji wzorców projektowych. Bazując częściowo na rozwiązaniu przedstawionym w [17] zaproponowano repozytorium, które zakłada opisanie ogólnych definicji wzorców jako zbiór cech tworzących dany wzorzec. Każda z cech jest następnie opisana przez szczegółowe dane implementacyjne, które uwzględniają różne warianty występowania wzorców. Szczegóły implementacyjne w proponowanym podejściu zostały opisane w modelu referencyjnym, jednakże cechy wzorców mogą być uszczegółowione na różne sposoby [21]: poprzez dodatkowy opis w postaci reguł lub odpowiednie operacje wykonywane na kodzie badanego programu.

Przed pierwszym wykonaniem procesu weryfikacji należy jednokrotnie uzupełnić repozytorium opisami wzorców, następnie nowe warianty wzorców mogą być dodawane na podstawie badanego oprogramowania. W uproszczonym ujęciu, dla jednego zaimplementowanego wzorca projektowego, proces weryfikacji rozpoczyna się od przekształcenia kodu programu do formalnej reprezentacji, następnie wykonywana jest weryfikacja zgodnie z występującymi cechami. W przypadku wykorzystania modelu referencyjnego proces weryfikacji sprowadza się do porównania fragmentu ekwiwalentu kodu do danych implementacyjnych,

które wskazywane są przez opis każdej cechy występującej w danym wzorcu projektowym.

Modele danych omówione w dalszej części pracy mogłyby zostać przedstawione za pomocą diagramów klas UML. Jednakże po uwzględnieniu wcześniej wymienionych informacji o zastosowaniu UML do wzorców projektowych, proponowane modele danych zostały opracowane w oparciu o diagramy związków-encji, co dodatkowo pozwoliło na wyróżnienie przewidzianych encji. ERD są jednymi z popularniejszych diagramów przeznaczonych do modelowania danych. Diagramy przedstawione na rysunkach 1, 2, zostały przygotowane z wykorzystaniem programu narzędziowego Power Designer 15 z użyciem notacji Barkera. Wykorzystane narzędzie pozwoliło na przystępną implementację tych modeli jako bazy danych.

3. Formalna reprezentacja kodu źródłowego oprogramowania

3.1. Motywacja

Najwięcej informacji o programowaniu jest zawartych w kodzie źródłowym, którego jednocześnie wadą jest fizyczna reprezentacja – są to odpowiednio skatalogowane pliki tekstowe, które trudniej analizować niż ustrukturyzowaną reprezentację formalną. W kilku podejściach [8], [13], związanych z analizowaniem wzorców projektowych, wykorzystano transformację kodu programu do formalnej reprezentacji. Uzyskany w ten sposób ekwiwalent kodu programu pozwala na zautomatyzowane przetwarzanie danych z występującą implementacją, a dodatkowo umożliwia usunięcie zbędnego kodu i niepotrzebnych informacji (np. komentarzy zapisanych w języku naturalnym, kodu testów jednostkowych). Niestety istniejące sposoby reprezentacji oprogramowania czy też wzorców projektowych nie spełniają postawionych dalej wymagań, co przyczyniło się do opracowania nowych rozwiązań.

3.2. Wymagania

Wzorce projektowe [7] implementowane są w oprogramowaniu obiektowym, toteż modele pozwalające na weryfikację implementacji powinny bazować (przynajmniej częściowo) na paradygmacie programowania obiektowego. Zasadniczą zaletą tego podejścia jest intuicyjność i przystępność dla osób dobrze znających paradygmat programowania obiektowego, szczególnie dla programistów ale również dla badaczy zajmujących się dziedziną inżynierii oprogramowania. Koleją zaletą jest niezależność od konkretnego języka programowania, ważne aby był to język zorientowany obiektowo.

Dodatkowo przy opracowywaniu modelu danych formalnej reprezentacji badanego oprogramowania, towarzyszyły następujące wymagania:

- reprezentacja struktury obiektowej oprogramowania, w tym oddzielenie elementów składowych klas (osobne encje dla pól, właściwości, metod itd.),
- reprezentacja instancji i zmiany stanów obiektów,
- zasilenie danymi na podstawie kodu zarządzanego [23] oraz możliwość rozbudowy o inne źródła danych,
- realizacja poprzez relacyjną bazę danych.

Jednocześnie zostały nałożone ograniczenia zmniejszające szczegółowość danych względem kodu źródłowego. W proponowanym modelu danych nie zostały odzwierciedlone wartości pól i zmiennych (np. zawartość łańcuchów znaków, dane binarne). Szczegółowość danych została zmniejszona również w przypadku specyficznych elementów nowoczesnych języków programowania, np. typy anonimowe są reprezentowane jako nowe klasy, w tym opracowaniu jest to zgodne ze standardem ECMA-355 [19]. W konsekwencji postawionych wymagań oraz nałożonych ograniczeń, opracowany model formalnej reprezentacji oprogramowania nie jest bezpośrednim odpowiednikiem żadnego języków programowania i nie mógł powstać automatycznie.

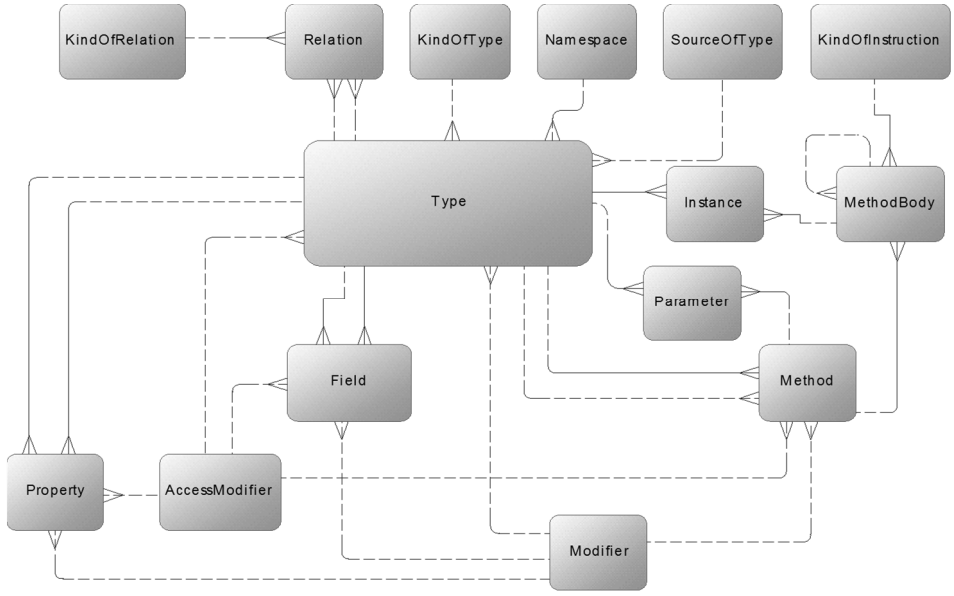
3.3. Model danych

Proponowany model danych oraz algorytm transformacji oprogramowania zostały wstępnie przedstawione w [24]. Rysunek 1 przedstawia rozbudowaną wersję modelu. Elementem o największej odpowiedzialności jest encja *Type*, odpowiada za reprezentację typów (np. konkretna klasa) w paradygmacie programowania obiektowego. Encja ta zawiera:

- kolekcja *Field* – pojedynczy element kolekcji reprezentuje pole (tzw. zmienna globalna) w danym typie, scharakteryzowany jest przez nazwę, typ (określa typ danych przechowywanych przez pole, realizowane przez osobną relację do encji *Type*) oraz encje słownikowe modyfikatora dostępu i modyfikatora,
- kolekcja *Property* – pojedynczy element kolekcji reprezentuje właściwość (tzw. setter lub getter), scharakteryzowany jest przez analogiczne pola i encje jak *Filed*, a dodatkowo określenie funkcji Set i Get tj. zapis i odczyt danych,

- kolekcja *Method* – pojedynczy element kolekcji reprezentuje metodę (funkcję) występującą w danym typie, scharakteryzowany jest przez nazwę, określenie czy metoda jest konstruktorem, typem (określa typ danych zwracanych przez metodę) oraz encje słownikowe modyfikatora dostępu i modyfikatora, dodatkowe encje zostały opisane w dalszej części,
- kolekcja *Relation* – pojedynczy element kolekcji reprezentuję zależność rozpatrywanego typu od innego typu wraz z dookreśleniem rodzaju zależności (np. dziedziczenie, realizacja) poprzez encję słownikową *KindOfRelation*,
- *KindOfType* – określa rodzaj danego typu (np. klasa, interfejs),
- *SourceOfType* – źródło pochodzenia (z badanego oprogramowania, typ systemowy lub zaślepkowy [23]),
- *Namespace* – określenie przestrzeni nazw, w której znajduje się dany typ.

Dodatkowo każdy typ scharakteryzowany jest modyfikatorem i modyfikatorem dostępu.



Rysunek 1. Model danych formalnej reprezentacji oprogramowania

Encja *Method* zawiera kolekcję *Parameter* definiującą parametry przyjmowane przez daną metodę, każdy element kolekcji scharakteryzowany jest przez nazwę oraz oczekiwany typ danych (osobna relacja do encji *Type*). Kolekcja *MethodBody*

określa instrukcje występujące w danej metodzie. Złożone instrukcje są dekomponowane do pojedynczych instrukcji i charakteryzowane odpowiednim rodzajem instrukcji (zgodnie z OpCode [19] kodu zarządzanego, np. tworzenie obiektu) oraz kolejnością występowania. Relacja zwrotna oznacza powiązanie ze sobą poszczególnych instrukcji w instrukcję złożoną. Dodatkowo kolekcja *Instance* grupuje ze sobą instrukcje, które odnoszą się do tej samej instancji danego typu. Instancja danego typu (np. wystąpienie konkretnego egzemplarza danej klasy) scharakteryzowana jest przez nazwę instancji oraz unikatowy Guid (wymagane jeśli nie można uzyskać nazwy instancji).

Encja słownikowa modyfikator (*Modifier*) opisuje modyfikatory (np. abstract, sealed, static, virtual, itd.) występujące przy typach oraz wybranych elementach składowych typów. Encja modyfikator dostępu (*AccessModifier*) określa zasięg widoczności (np. public, private, internal, protected) wybranych elementów. Zawartość encji *Modifier*, *AccessModifier*, *KindOfRelation*, *SourceOfType*, *KindOfInstruction* jest predefiniowana w zależności od języka programowania jaki jest reprezentowany przez model danych (w tym opracowaniu C#). W szczególnym przypadku dopuszczalne jest występowanie wartości „empty" w wymienionych encjach, jeśli pozwala na to dany język programowania.

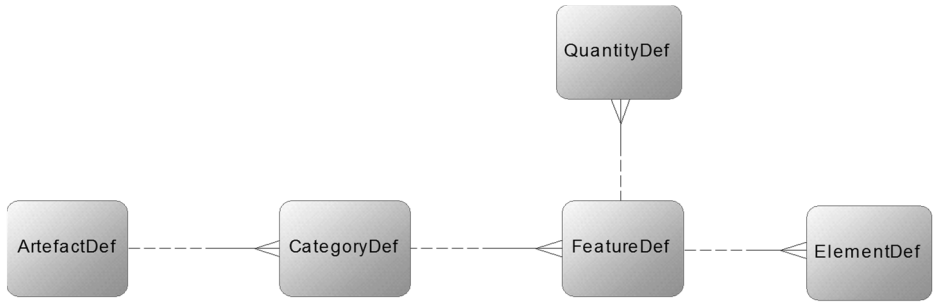
4. Repozytorium implementacji wzorców projektowych

Repozytorium implementacji wzorców projektowych to składnica informacji o możliwych sposobach implementacji rozpatrywanych wzorców projektowych. Z powodu dużej różnorodności implementacyjnej podjęto decyzję o rozdzieleniu ogólnej definicji wzorców od szczegółów implementacyjnych. Definicja wzorca to pewnego rodzaju spis treści, który zawiera odpowiednio dobrane cechy wzorców w oderwaniu od szczegółowej implementacji. Cecha wzorca projektowego to jeden z koniecznych elementów, które charakteryzują dany wzorzec. Bazując na przykładzie wzorca Singleton jedną z cech jest udostępnianie instancji, co można zaimplementować na kilka sposobów (np. pole, właściwość, metoda). Fakt wystąpienia udostępniania instancji jest cechą konieczną tego wzorca, natomiast sposób implementacji jest zależny od różnych czynników (np. kontekst kodu programu, język programowania). Podobne rozdzielenie występuje w normie ISO 9126 określającej jakość produktu programowego. Norma opisana jest przez drzewiastą strukturę, która zawiera za [4] charakterystyki a w nich rekursywnie podcharakterystyki. Charakterystykom liściom przyporządkowane są metryki – funkcje, które wyznaczają pewne wartości na podstawie mierzalnych atrybutów oprogramowania. Definicja charakterystyk jest nieformalna – wyraża pewną intencję, natomiast metryki są sformalizowane. W proponowanym podejściu charakterystykom z ISO 9126 odpowiada model definicji a metrykom szczegółowa implementacja wzorców projektowych.

Odseparowanie cech od implementacji wzorców projektowych pozwala na zwiększenie zakresu weryfikacji. W zależności od badanego aspektu można wykorzystać odpowiednie rozwiązanie: aspekty strukturalne implementacji wzorców projektowych można zweryfikować poprzez porównanie badanego oprogramowania do modelu referencyjnego, natomiast aspekty związane z dynamiką i zachowaniem oprogramowania mogą wymagać weryfikacji opartej o śledzenia zmian stanów obiektów (np. poprzez realizację odpowiednich zapytań na formalnej reprezentacji oprogramowania).

4.1. Model definicji

W proponowanym rozwiązaniu opis cech został zrealizowany jako hierarchiczna struktura danych. Metamodel tej struktury przedstawia rysunek 2. Korzeniem hierarchii i jednocześnie najbardziej ogólna jest encja *ArtefactDef*, która reprezentuje konkretne wzorce projektowe. Jak już zostało wspomniane, definicja każdego wzorca projektowego składa się z odpowiednich cech, które dla ułatwienia zostały zgrupowane w encji *CategoryDef* zawartej w *ArtefactDef*. Cechy (kolekcja encji *FeatureDef*) zgrupowane są w kategoriach według kryterium przynależności do aspektów: struktury tworzącej wzorzec, wykorzystania, zachowania itp. Każda cecha może być zależna od innej cechy, rodzaj zależności określa pole *RelatedType* (niewidoczne na rysunku) będące typem wyliczeniowym. Możliwe rodzaje zależności cech to: koniunkcja, alternatywa oraz zawieranie cech podrzędnych. Cecha scharakteryzowana jest przez nazwę oraz określenie liczebności (encja *QuantityDef*) występowania danej cechy (w szczególnym przypadku liczebność występowania minimum 1 oznacza konieczność wystąpienia cech). Każda cecha zawiera szczegółowe elementy (encja *ElementDef*), które pozwalają na dodatkowe zdekomponowanie cechy np.: cechą struktury tworzącej wzorzec jest konkretna klasa, która dodatkowo jest zdekomponowana na: zależność od innej klasy, modyfikator dostępu określający zasięg widoczności. Każdy element scharakteryzowany jest przez nazwę oraz możliwość zanegowania (dopuszczalne jest wtedy wszystko inne niż dany element). W szczególnym przypadku może występować wyłącznie jeden element danej cechy, co skraca hierarchię do poziomu cech. Encja *ElementDef* jest jednocześnie łącznikiem pomiędzy modelem definicji a opisem szczegółowej implementacji wzorców projektowych.



Rysunek 2. Model definicji

Zaproponowany model danych pozwala na bardzo dużą dowolność w opisie definicji wzorców projektowych. W zależności od wzorca możliwe jest opisanie ogólnej cechy, której implementacja możliwa jest na wiele różnorodnych sposobów, lub wyszczególnienie elementów danej cechy, gdy występuje stała implementacja.

4.2. Model Referencyjny

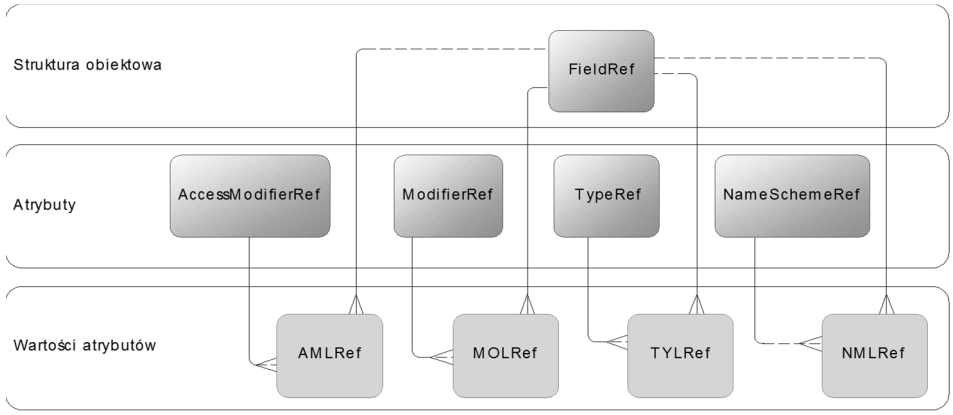
Model referencyjny jest jednym z rozwiązań przeznaczonych do opisu szczegółowej implementacji wzorców projektowych, inaczej danych implementacyjnych. Szczegółowa implementacja to uzupełnienie modelu definicji o zbiór drobnoziarnistych informacji (inaczej atrybutów) [3] związanych bezpośrednio z kodem programu. Drobnoziarnistość informacji pozwala na opis realizacji wzorców w konkretnym języku programowania. Jednocześnie możliwy jest opis generycznych implementacji (inaczej szablonowych). Oznacza to, że występujące dane określają ogólny charakter opisywanych atrybutów bez narzucania konkretnych wartości, np. określenie typu danych pola w klasie jako to dowolny typ numeryczny.

Proponowany model referencyjny jest przewidziany do opisu implementacji struktury tworzącej wzorce projektowe. Struktura tworząca wzorzec to na ogół zbiór klas i interfejsów połączonych ze sobą, np.: we wzorcu Strategia jest to interfejs deklarujący strategię oraz klasy, które implementują ten interfejs. Zgodnie z wcześniejszymi założeniami, badane oprogramowanie, dokładniej dane zawarte w reprezentacji formalnej, jest porównywane z zawartością modelu referencyjnego. Model referencyjny również spełnia wymagania zostały narzucone na formalną reprezentację oprogramowania. Aby zautomatyzować proces porównania, model referencyjny zawiera analogiczną dekompozycję struktury obiektowej na encje jak model formalnej reprezentacji oprogramowania, tj. *TypeRef* zawiera kolekcję *FieldRef* itd. (zgodnie z PascalCase został dodany przyrostek „Ref” do encji modelu referencyjnego w celu łatwiejszego odróżnienia encji pomiędzy modelami, Ref to skrót od Reference). Jednakże dane występujące w tych encjach zostały dodatkowo

zdekomponowane do osobnych encji, które umożliwiają opisanie różnych wariantów drobnoziarnistych atrybutów. Każdy z wariantów oznaczony jest odpowiednim poziomem dopasowania. Poziom dopasowania to określenie stopnia dostosowania danego atrybutu do konkretnego wariantu implementacji. W przyjętym opracowaniu jest to zakres od „0" do „2", gdzie wartość „2" oznacza najwyższy poziom dopasowania. Możliwe jest występowanie wielu atrybutów o jednakowym poziomie dopasowania.

Dane opisywane przez model referencyjny można podzielić na trzy grupy:

1. dane tworzące strukturę obiektową,
2. dane określające atrybuty występujące z danym elementem struktury obiektowej,
3. dane reprezentujące konkretne wartości atrybutów, w tej grupie występuje również określenie poziomu dopasowania.



Rysunek 3. Wybrane encje i relacje opisujące pole w modelu referencyjnym

Podział na wymienione wcześniej trzy grupy danych został zaznaczony na rysunku nr 3, który przedstawia fragment modelu referencyjnego. Na rysunku widoczne są wyłącznie encje opisujące pole. Dla uproszczenia nie widoczne są niektóre relacje, tj. każde pole może być zawarte w konkretnym typie (np. jako element składowy klasy), natomiast każdy typ może być scharakteryzowany przez różne atrybuty. Całość modelu referencyjnego przedstawia rysunek 4. Tabela 1 przedstawia encje tworzące grupę struktury obiektowej w odniesieniu do charakteryzujących je encji atrybutów, co jest zgodne z paradygmatem programowania obiektowego. Dekompozycja pozostałych encji należących do grupy struktury obiektowej została zrealizowana analogicznie jak przedstawiony przykład encji *Field*. Encje *KindOfTypeRef*, *KindOfRelRef*, *ModifierRef*, *AccesModifierRef* to słowniki stałych atrybutów, których wystąpienie wartości (dla danego elementu

struktury obiektowej) przechowywane jest odpowiednio w encjach: *KTLRef*, *KRLRef*, *MOLRef*, *AMLRef*. Znaczenie tych encji jest analogiczne jak w modelu formalnej reprezentacji oprogramowania. Nieco inaczej jest wykorzystywana encja *NameSchemeRef*, której wartości (encja *NMLRef*) pozwalają na opisanie konkretnych nazw zgodnie ze schematami: zaczyna się od, kończy się na, zawiera. Encja *TypeRef* pełni podwójną rolę, zawiera w sobie elementy struktury obiektowej (elementy składowe typów) oraz występuje jako słownik atrybutu typ. Ostatecznie grupa encji reprezentujących konkretne wartości atrybutów jest połączona encją asocjacyjną z *ElementDef* modelu definicji.

Tabela 1. Zestawienie atrybutów występujących z encjami („+" atrybut występuje w danej encji)

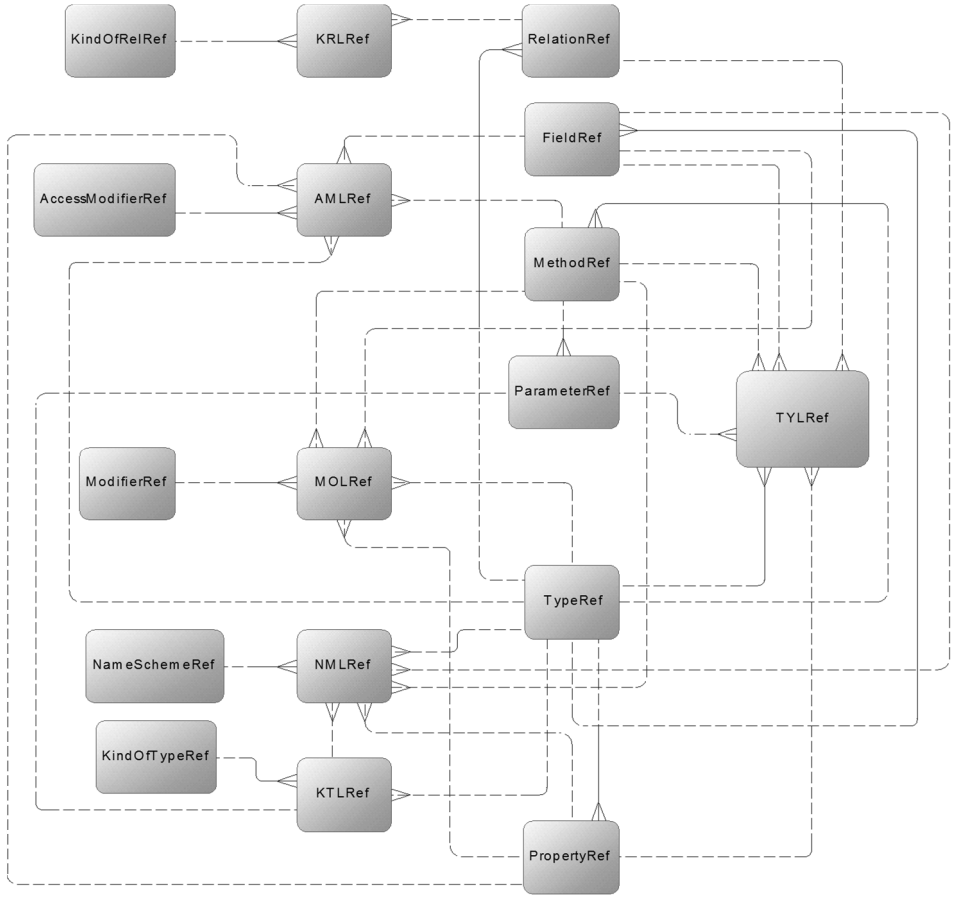
	KindOf-Type-Ref	*KindOf-Rel-Ref*	*Modifier-Ref*	*Acces Modifier-Ref*	*Name Scheme-Ref*	*Type-Ref*
TypeRef	+		+	+	+	
FieldRef			+	+	+	+
Property-Ref			+	+	+	+
MethodRef			+	+	+	+
Parameter-Ref					+	+
Relation-Ref		+				+

4.3. Weryfikacja w oparciu o proponowane modele

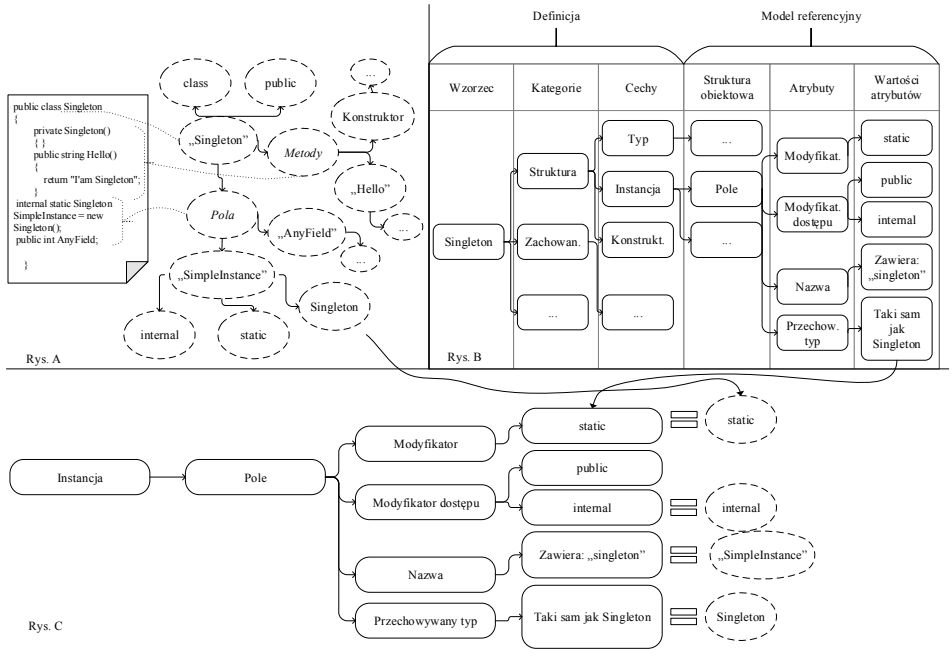
Weryfikacja wzorców projektowych jest złożonym procesem, stąd dążenie do automatyzacji. Nawet dla prostego wzorca, jakim jest Singleton, proces weryfikacji składa się z wielu kroków, których nie można zwięźle opisać. Dotychczasowe prace, tj. przykłady weryfikacji oraz oceny jakości implementacji wzorców projektowych wraz z interpretacją wyników zostały opisane w [24].

Rysunek 5 przedstawia diagram opisujący ideę weryfikacji w oparciu o model referencyjny. W części rys. 5a widoczny jest przykładowy kod źródłowy implementacji wzorca projektowego Singleton oraz wizualizacja kodu (kształty otoczone przerywaną linią) aby lepiej przybliżyć dalsze porównanie. W części rys. 5b widoczna jest uproszczona definicja wzorca projektowego Singleton, jest to wizualizacja danych z modelu referencyjnego. Dodatkowo zaznaczone zostały podziały na hierarchię modelu definicji oraz grupy danych modelu referencyjnego. W celu uproszczenia rys. 5b zostały pominięte mniej znaczące wartości atrybutów (np. brak modyfikatorów) oraz zerowe poziomy dopasowania (są niedozwolone, np.

modyfikator dostępu konstruktora inny niż prywatny). Diagram przedstawia wyłącznie wybrane elementy struktury tworzącej wzorzec, a w omówionym dalej przykładzie ogranicza się to do pola udostępniającego instancję Singleton. Miejsca wykropkowane oznaczają dalsze rozwinięcia danych [24]. W części rys. 5c przedstawiony został wynik porównania. Kształty z wizualizacji formalnej reprezentacji zostały zestawione z kształtami modelu referencyjnego. W przedstawionym przykładzie wystąpił wewnętrzny modyfikator dostępu (niższy poziom dopasowania), którego konsekwencje wyjaśnia tabela 2.



Rysunek 4. Model referencyjny



Rysunek 5. Diagram opisujący ideę weryfikacji w oparciu o model referencyjny

Tabela 2. Konsekwencje wystąpienia wybranych modyfikatorów dostępu w instancji wzorca Singleton

Modyfikator dostępu	Poziom dopasowania	Konsekwencje wystąpienia
public	2	Zalecane, udostępnia instancję Singletonu wszystkim klasom, nie ogranicza wykorzystania.
internal	1	Dopuszczalne, jednakże ogranicza zasięg widoczność wyłącznie do biblioteki. Może powodować problemy przy próbie integracji z badanym oprogramowaniem.
private	0	Niedopuszczalne, uniemożliwi dostęp do instancji, wzorzec nie będzie spełniał swojego przeznaczenia.

5. Podsumowanie

W pracy omówiono krótko potrzebę weryfikacji implementacji wzorców projektowych oraz przytoczono związane z tym problemy. Następnie opisano ogólne założenia autorskiego procesu weryfikacji oraz występujące w nim modele danych. Wynik weryfikacji jest niezbędny do realizacji dalszych prac w kontekście oceny jakości implementacji wzorców projektowych.

Pierwszy z zaprezentowanych modeli danych przeznaczony jest do przechowywania ekwiwalentu kodu źródłowego badanego oprogramowania, drugi przechowuje definicje wzorców projektowych. Oba modele oraz opisany proces zostały zrealizowane jako prototypowe narzędzie. Uzyskane wyniki potwierdziły słuszność przyjętych rozwiązań i stanowią podstawę do dalszych badań dotyczących rozbudowy repozytorium implementacji wzorców o możliwość weryfikacji aspektów behawioralnych, np. w oparciu o diagramy UML, a następnie zaproponowania kryteriów jakości implementacji wzorców projektowych.

Bibliografia

1. Binun A.: *High Accuracy Design Pattern Detection*. Dysertacja doktorska, Rheinischen Friedrich Wilhelms Universitat Bonn, 2012
2. Blewitt A.: *HEDGEHOG: Automatic Verification of Design Patterns in Java*. Dysertacja doktorska, University of Edinburgh, 2006
3. De Lucia A., i inni: *Design pattern recovery through visual language parsing and source code analysis*, Journal of Systems and Software archive, Vol.: 82, Issue 7, Elsevier Science Inc, New York, 2009
4. Dubielewicz I., Hnatkowska B., Huzar Z., Tuzinkiewicz L.: *Wykorzystanie analizy wielokryterialnej w ocenie modeli baz danych*, w: Bazy Danych: Nowe Technologie, Politechnika Śląska, WKŁ, 2007
5. Fowler M.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Wydanie III, Addison Wesley, 2003
6. Fowler M. i inni: *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, Helion, Gliwice, 2011
7. Gamma E. i inni: *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, Helion, Gliwice, 2010
8. Grzanek K.: *Realizacja systemu wyszukiwania wystąpień wzorców projektowych w oprogramowaniu przy zastosowaniu metod analizy statycznej kodu źródłowego*, Dysertacja doktorska, Politechnika Częstochowska, Łódź, 2008
9. Hernandez J. i inni: *Selection of Metrics for Predicting the Appropriate Application of design patterns*, 2nd Asian Conference on Pattern Languages of Programs, 2011

10. Kan S. H.: *Metryki i modele w inżynierii jakości oprogramowania*, PWN SA, Warszawa, 2006
11. Kerievsky J.: *Refaktoryzacja do wzorców projektowych*, Helion, Gliwice, 2005
12. Khaer Md. A. i inni: *An Empirical Analysis of Software Systems for Measurement of Design Quality Level Based on Design Patterns*, Computer and Information Technology, IEEE, 2007
13. Kirasić D., Basch D.: *Ontology-Based Design Pattern Recognition*, Knowledge-Based Intelligent Information and Engineering Systems, Zagreb, Croatia, 2008
14. Kornatka A.: *Projekt i konstrukcja systemu generującego elementy bazodanowych aplikacji biznesowych*, Studia Informatica Vol.: 33, No. 2B, s. 201-215, Wydawnictwo Politechniki Śląskiej, Gliwice, 2012
15. Martin R., Martin M: *Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#*, Helion, Gliwice, 2008
16. McConnell S.: *Kod Doskonały*, Helion, Gliwice, 2010
17. Rasool G.: *Customizable Feature based Design Pattern Recognition Integrating Multiple Techniques*, Dysertacja Doktorska, Technische Universitat Ilmenau, Ilmenau, 2010
18. Singh Rao R., Gupta M.: *Design Pattern Detection by Greedy Algorithm Using Inexact Graph Matching*, International Journal Of Engineering And Computer Science, Vol. 2, Issue 10, s. 3658-3664, 2013
19. Troelsen A.: *Język C# i platforma .NET 3.5*, PWN, Warszawa, 2009
20. Tsantalis N. i inni: *Design Pattern Detection Using Similarity Scoring.* IEEE Transactions on Software Engineering, Volume: 32, Issue: 11, s. 896-908, 2006
21. Wojszczyk R.: *Koncepcja hybrydowej metody do oceny jakości zaimplementowanych wzorców projektowych*, w: Zeszyty Naukowe Wydziału Elektroniki i Informatyki nr 7, s. 17-26, Wydawnictwo Uczelniane Politechniki Koszalińskiej, Koszalin, 2015
22. Wojszczyk R.: *Porównanie sposobów reprezentacji wzorców projektowych*, w: Modele inżynierii teleinformatyki 9, s. 133 - 145, Wydawnictwo Uczelniane Politechniki Koszalińskiej, Koszalin, 2014
23. Wojszczyk R.: *Pozyskiwanie struktury obiektowej z kodu zarządzanego przy wykorzystaniu metod inżynierii odwrotnej*, w: Inżynieria oprogramowania: badania i praktyka, s. 199-213, Zeszyty Rady Naukowej Polskiego Towarzystwa Informatycznego, Warszawa, 2014
24. Wojszczyk R.: *The model and function of quality assessment of implementation of design patterns*. Applied Computer Science, Vol. 11, No. 3, s 45-56, Institute of Technological Systems of Information, Lublin University of Technology, Lublin, 2015

Streszczenie

Wzorce projektowe to zagadnienie szeroko opisywane w uznanej literaturze i wykorzystywane przez wielu programistów, ale mimo to nie ma nad nimi formalnej kontroli. W artykule poruszony został problem weryfikacji implementacji wzorców projektowych stosowanych w programowaniu obiektowym. W procesie weryfikacji wyróżniono dwa modele danych: formalną reprezentację będącą ekwiwalentem badanego oprogramowania oraz repozytorium implementacji wzorców zawierające informacje opisujące implementację wzorców projektowych. Opracowane rozwiązanie pozwoli wykazać błędy i potencjalne problemy w implementacji.

Abstract

Although the design patterns constitute the issue that has been widely discussed in the literature and used by many software developers, there is no formal control over them. The article discussed the problem of verifying the implementation of design patterns applied in object-oriented programming. Two following data models were distinguished in the process of verification: a formal representation that is an equivalent of the analysed software, and a repository of implementation of patterns containing information describing the implementation of design patterns. The proposed solution will make it possible to show implementation errors and potential problems.

Keywords: design patterns, data model, ERD, verifying implementation